Mitigating Random Variation with Spare RIBs: Redundant Intermediate Bitslices

David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
palframan@wisc.edu, nskim3@wisc.edu, mikko@engr.wisc.edu

***Abstract*—Delay variation due to dopant fluctuation is expected to become more prominent in future technology generations. To regain performance lost due to within-die variations, many architectural techniques propose modified timing schemes such as time borrowing or variable latency execution. As an alternative that specifically targets random variation, we propose introducing redundancy along the processor datapath in the form of one or more extra bitslices. This approach allows us to leave dummy slices in the datapath unused to avoid excessively slow critical paths created by delay variations. We examine the benefits of applying this technique to potential critical paths such as the ALU and register file, and demonstrate that our technique can significantly reduce the delay penalty due to variation. By adding a single bitslice, for instance, we can reduce this delay penalty by 10%. Finally, we discuss heuristics for configuring our redundant design after fabrication.**

***Index Terms*—process variation; doping; bitsliced design; performance; reliability;**

I. Introduction

With each successive technology generation, the impact of process variation becomes increasingly significant. In addition to die-to-die (D2D) variations in device characteristics, current and future technologies are increasingly susceptible to within-die (WID) variations [1]. The latter effect is of particular concern, since it can shift the distribution of fabricated chips, decreasing the mean operating frequency [2]. It has been suggested that the impact of WID variation could even counteract the potential gains of technology generations to come [3]. In this paper, we focus on random variation, a type of WID variation that arises primarily due to dopant fluctuations [4]. Random variations become particularly prominent at small device sizes, with the International Technology Roadmap for Semiconductors (ITRS) currently predicting as much as 81% variability in V_t for minimum-sized devices due to doping effects in the near future [5].

Many previous proposals to address increased delay due to systematic variations are also applicable to random variations, though they are arguably less efficient in this case. Many of these techniques are applied at the scope of execution units or pipeline stages. We note, however, that a single slow gate or delay fault in one of these units can limit a processor's maximum operating frequency. With this in mind, we propose a fundamentally different architectural technique intended to regain the performance lost due to random delay variations. Instead of operating at the granularity of pipeline stages, our approach is applied to datapath bitslices. We create a wider datapath with extra bitslices, allowing us to leave some intermediate slices unused, thereby hiding the delay of the slowest paths. For the remainder of the paper, we will refer to these extra slices as Redundant Intermediate Bitslices (RIBs). In addition to avoiding complicated clocking or scheduling issues, this approach is not difficult to implement, since it involves widening datapath elements that we are already adept at designing. Note that unlike previous bitsliced architectures that primarily target hard defects [6], our technique does not use fuses or multiplexers between slices since this would increase delays. Using RIBs does offer significant opportunity for hard defect tolerance, though in this paper we focus on the performance implications.

Although we emphasize performance improvement at a fixed supply voltage, it is worth mentioning that our technique could also be used to save power through voltage scaling. This is possible because in a processor without RIBs, it is necessary to compensate for process variation by reducing the clock frequency to accommodate slower logic or by increasing the supply voltage to speed up the slower logic. Since using RIBs will reintroduce some slack along the critical path, we can then fill this slack by decreasing the supply voltage or increasing the clock frequency so that the chip operates closer to the design point. This paper includes the following contributions:

1) Description of our technique to regain performance lost due to random delay variations;
2) Discussion of the addition of RIBs to the ALU, register file, and other datapath logic;
3) Analysis of the benefits of this technique with varying levels of redundancy;
4) Discussion of a simplified heuristic for post-silicon RIB configuration.

The remainder of this paper is organized as follows. Section II discusses related work and motivates our proposal. Section III presents our idea for incorporating RIBs throughout the datapath, with particular focus on the ALU and register file. Section IV includes results from simulating our redundant design. Finally, Section V concludes the paper.

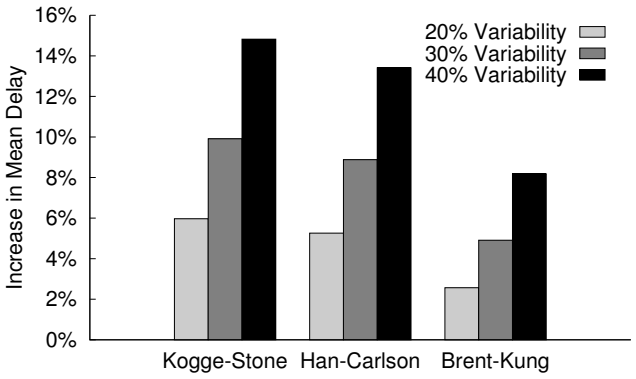


Fig. 1. Increase in ALU delays due to random variations. Each bar represents the mean delay of a distribution of 1000 generated circuits

II. Background and Motivation

Historically, D2D variations were considered more important and prominent than WID variations [3]. To regain yield lost due to D2D variations, binning can be applied. In this approach, constraints that affect the entire chip are relaxed. For instance, operating voltage can be increased or clock frequency decreased to compensate for slower logic. D2D variation no longer remains the sole concern, however, as sub-wavelength feature sizes increase the level of WID variation. Variation due to lithographic effects, for instance, can affect gate channel lengths and impact regions of the die. This type of WID variation that affects multiple devices is termed systematic.

Though frequency binning can be applied to improve yield when systematic WID variation is present, it is less efficient since variation is no longer constant across the chip. Unlike D2D variation, WID variation decreases the mean performance of fabricated chips, since a chip can only be clocked as fast as its slowest path. As an example of this effect, Figure 1 shows the increase in the average ALU delay (with different adder types) that we discuss in detail in Section III-A. In this case, the delay overheads are caused by random variations alone, though a similar trend can be observed with systematic variations. This phenomenon has motivated a host of techniques that specifically aim to counter the effects of WID variation, including both systematic and random fluctuations in delays [7]–[13].

Techniques that target WID variation range from circuit-level to architectural. In the circuit domain, recent work proposes modifying the body bias to compensate for varying threshold voltages [7]. Even though this technique can be applied to subsections of the die, it is best suited to compensate for D2D or systematic WID variations. Still other techniques propose clocking a processor at its intended frequency and detecting any errors that may occur. In this case, the error rate can be decreased by purposefully modifying the latency of certain circuit paths [8].

As an alternative to disabling slow units, most architecture-level techniques compensate for slower stages or units with modified timing schemes. One possibility is to allow variable-latency functional units so that slow units can take additional clock cycles to complete [9], [10]. Relaxing constraints on operation latency, however, can potentially complicate scheduling and decrease the number of instructions executed per cycle (IPC). To avoid these issues, recent work proposes reducing the processor frequency only when there is sufficient instruction-level parallelism to require both the fast and slower functional units [11]. Yet another alternative is time borrowing, in which specific clock signals can be skewed to allow a slow stage to use some of the time originally allocated to a faster stage [12].

Just as frequency binning may not be optimal in the case of systematic variations, we suggest that techniques to address systematic WID variations may not be ideal for random variations. Many of these techniques are applied uniformly to whole pipeline stages, but random variation is not uniform throughout the logic. Also, though fine-grained body biasing techniques have been proposed as in [13], maintaining numerous body-biasing domains that also require multiple voltage domains can be complicated.

We note that our proposed technique is orthogonal to other techniques that involve clock manipulation or variable latency execution. Other work demonstrates that multiple techniques can be effectively combined to reduce performance loss due to delay variations [9], [10]. RIBs can be implemented across the majority of the processor datapath, potentially alongside a relatively more coarse-grained scheme. Since the RIB technique takes a quite different approach, it does not share many of the limitations of other design modifications. For instance, RIBs does not require a single "fast" copy of a unit (e.g. it can be applied to simple cores), and unlike time borrowing, it is not limited by pipeline loops [11].

III. Intermediate Bitslices

Addressing random delay variations with techniques that uniformly impact particular logic stages may not be the most efficient approach, since the increased stage delay may be caused by only a subset of gates. Instead, we propose a bitsliced technique similar to those targeting hard defects [6], [14]. We create a wider datapath that can accommodate unused slices at purposefully chosen offsets. That is, if the original design calls for a 64-bit wide datapath, we could implement a 65-bit datapath, allowing a single bit to be left unused at the same location in data registers, latches, wiring, and logic. Though the delay reduction benefits of this technique are derived from applying it to a critical path, keeping this redundant bit at the same offset throughout the datapath has the benefit of simplicity and avoids adding multiplexers to the critical path. For instance, consider a naïve implementation in which RIBs are only added to targeted critical paths. In this case, only the targeted logic would contain extra bitslices and we would require multiplexers at the logic input and output to route around the unused slices. Keeping the unused slices throughout the datapath eschews the need for such input and output multiplexers and their potential to counterproductively increase delays.

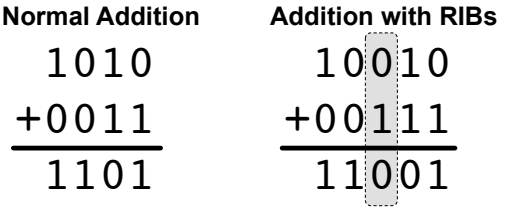


Fig. 2. Conceptual example of binary addition with RIBs. Note that the carry traverses the extra slice.

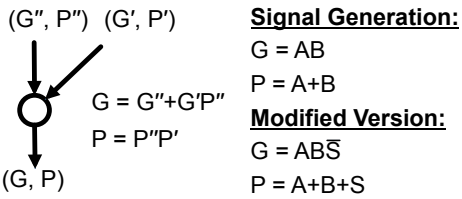


Fig. 3. Basic prefix adder operation. Our proposal adds an additional skip input S to the initial propagate and generate logic for each bit position. A and B are operand bits at a given offset.

Using RIBs produces two different effects that can combine to decrease the maximum stage delay. Most obviously, once offsets are chosen for the unused slices, the output of any unit at these bit offsets can be ignored while preserving correctness. Any critical paths that fan out only to RIB offsets therefore no longer contribute to the maximum delay. We also consider that since the input to any unit can be ignored at these offsets, these input values can be overridden. Assuming that these inputs are on a critical path that fans out to one or more valid output bits, a secondary effect is that these forced inputs may render some gates static and remove them from the critical path. Of course, these signals must be overridden in a way that preserves or promotes the correct functionality of the unit. This effect is particularly relevant in adders, which we will discuss in the next section.

There are a number of potential critical paths in a modern processor, including but not limited to the integer ALU, register file, scheduler, load-store queue, and load path. We place particular emphasis on the integer ALU, since it imposes a fundamental limit on cycle time. Many other structures such as the scheduler and load-store queues are more flexible in that it is possible to change the number of entries they hold or otherwise resize them. For a particular datapath width, however, the ALU latency is more constrained.

Application of this concept to much of the datapath is relatively straightforward, since many resources are dedicated to simply moving data around. Likewise, for bitwise operations such as AND and OR, no special consideration is required when inserting RIBs.

We must of course enforce some bound on where data is stored with RIBs and where it is stored in the traditional compacted form. For instance, our scheme could be extended to the cache or constrained to a smaller part of the processor. Assuming that not every cycle in a multi-cycle cache access is critical, there may be enough slack for the logic to make this transition between the datapath and the L1 cache. We note that existing techniques already include redundant columns in SRAM arrays to protect against hard faults in bit cells [15]. When this type of redundancy is employed in a design, a level of multiplexing is required to select the desired columns. Use of this spare column redundancy in the cache fits nicely with our concept of RIBs. As long as RIBs are placed at the unused or defective SRAM column offsets, the multiplexing stage becomes unnecessary.

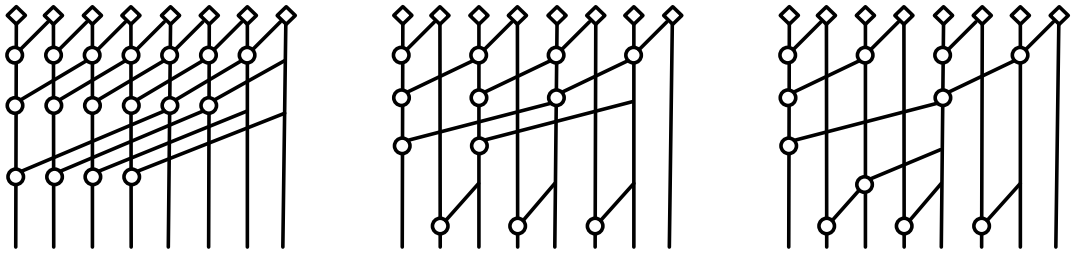
Execution logic brings additional challenges since data is actually being manipulated. Also, it should be noted that although the analysis in this paper focuses on the integer datapath, floating point arithmetic plays an important role in many applications. Floating point logic introduces considerable additional complexity, though at its core it contains similar arithmetic circuits. For this reason, we defer consideration of floating point logic to future work. The remainder of this section discusses how we address the challenges and implications of adding RIBs to critical paths in the execution core and register file.

A. Application to Prefix Adders

The ALU and bypass loop is a critical path in most processors, since pipelining it is usually not considered due to dependency chains. Since it requires carry precomputation or propagation, addition is typically the ALU operation with the highest latency. Under this assumption, we consider only the adder and bypass network in our ALU critical path analysis.

Should a RIB be inserted into the datapath to compensate for delay variation in another unit or the adder itself, it is important that the adder function properly without imposing additional delay due to our modifications. Correct addition of two numbers that contain an unused bit at the same offset is conceptually simple. The sole requirement is that the carry signals propagate across the gap to produce the correct sum—the output value at the RIB offset does not matter. One way to enforce carry propagation is to insert multiplexers between each bitslice. Although this approach might be effective for simplistic carry propagate adders, it would be expensive in terms of area, complexity, and delay when applied to higher performance prefix adders that compute carries in parallel. Fortunately, the insertion of multiplexers in this manner would only be necessary to isolate a hard fault in the adder.

Regardless of adder architecture and assuming that the bits in the operand RIBs can be manipulated, carries can be made to propagate by setting ones in the unused bits of one operand and zeros in the other, as demonstrated in Figure 2. In a prefix adder, overriding these bits is equivalent to fixing the carry computation signals that correspond to the RIB offset. For carries to traverse the extra bitslice, the *propagate* signal is asserted while the *generate* signal is cleared. To avoid adding extra delay, we propose the addition of an extra control signal input to the gates that generate these signals from the operands. This is a simple modification, since *propagate*



(a) Kogge-Stone adder (b) Han-Carlson adder (c) Brent-Kung adder

Fig. 4. Prefix graphs for the adders studied in this paper.

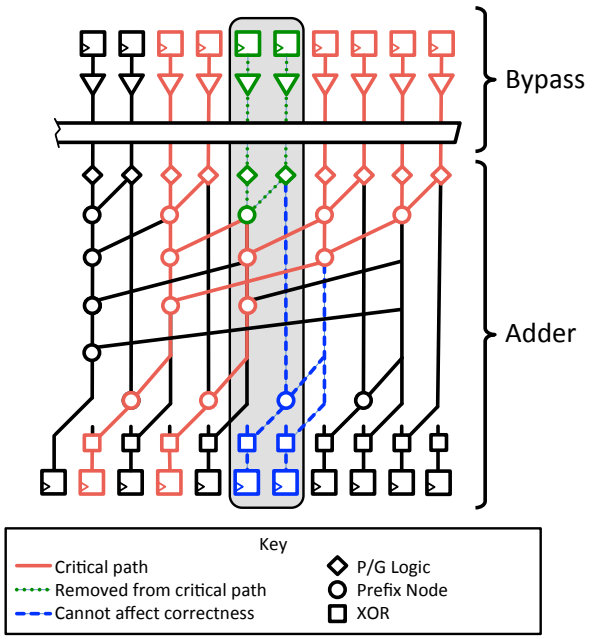


Fig. 5. Example of adding two extra bits to an 8-bit Han-Carlson adder. The shaded region indicates the location of the two RIBs.

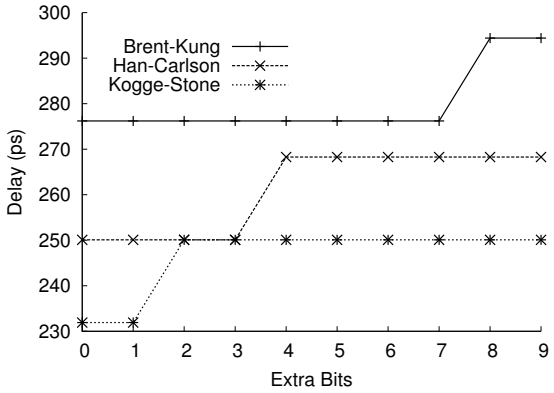


Fig. 6. Change in ideal ALU delay with wider datapaths. The baseline design is 16 bits wide. The point at which the delay increases depends on the adder type.

is initially calculated with an OR while *generate* uses an AND gate. Figure 3 shows the logic used in the building blocks of parallel prefix adders as well as our proposed logic modification. In this paper, we consider the Kogge-Stone [16], Han-Carlson [17], and Brent-Kung [18] adder architectures to explore trade-offs between complexity, delay, and the number of critical paths. Prefix graphs for these adders are shown in Figure 4.

Even if the adder is not on the critical path, this modification allows functionally correct addition when RIBs in the datapath are used to compensate for slow critical paths in other units. It is also possible to amortize a subset of delay faults within the adder and its critical path. As previously mentioned, since the *propagate* and *generate* bits that correspond to a RIB remain fixed, the delay to generate these signals no longer contributes to the critical path. In addition, slow logic that does not fan out to multiple bits, including some prefix nodes, final sum-calculating XORs, and bypass buffers can be ignored with appropriate RIB placement. Figure 5 shows an example of an 8-bit Han-Carlson adder with two RIBs. Since we are overriding the *propagate* and *generate* signals at the input of the adder, any excessive delay in the logic indicated will no longer contribute to the critical path. The figure also highlights paths that fan out only to the redundant bits. If these paths were critical, they would also have no impact on the maximum delay.

In widening the datapath, it is not possible to add an arbitrary number of extra bits without increasing delay. We note that for prefix adders, the number of extra bits that can be added for "free" in terms of delay varies with each adder type. For instance, a (power of two wide) Kogge-Stone adder can be widened by only a single bit before delay begins to increase. This is possible because although a wider Kogge-Stone adder requires more prefix nodes to calculate the carry-out bit, the carry-out does not require an XOR, as shown in Figure 5. Figure 6 illustrates how the variation-free ALU delay increases as extra bits are added to the datapath width. Before the maximum delay increases, the Kogge-Stone adder can accommodate one extra bit, the Han-Carlson adder three bits, and the Brent-Kung adder can be widened by nearly half of its original width.

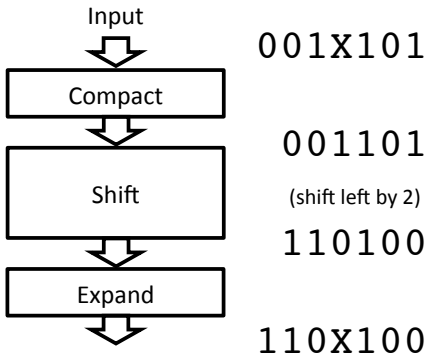


Fig. 7. Modified shifter with compact and expand stages. An example showing the output of each block is also shown. A bit value of X means 'don't know' or 'don't care,' due to a RIB in the datapath.

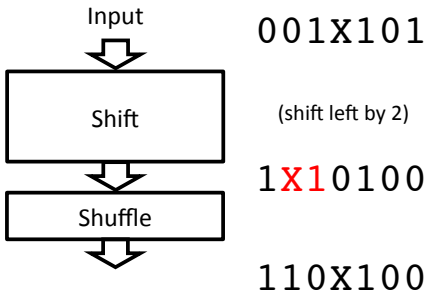


Fig. 8. Modified shifter with shuffle stage. In this example, the values of two bits in the shifted result are corrected by the shuffle stage.

B. Shifter and Multiplier Modification

Shifts, unlike addition or bitwise operations, introduce additional complications. Without modification, any RIB in the shifted result would not be aligned with RIBs in the rest of the datapath. To remedy this, we revisit our previous assumption that addition is the slowest ALU operation. The slack present in the shifter allows the insertion of additional logic to address this issue while leaving the shifter logic intact.

We present two possible modified shifter designs. In the simplest case, the shifter is sandwiched between two additional logic stages, as shown in Figure 7. The figure also shows an example performing a shift when a RIB is present. The RIB offset is indicated by a value of X at the input and output since this value is either unknown or does not matter. The first stage compacts the input operand by removing all redundant bits. The compressed value is then shifted as usual, and the last stage reintroduces redundancy at the appropriate offsets.

The main drawback of this approach is that it adds two multiplexer delays to the shift delay. This could be problematic depending on the delay gap between the shifter and the adder in the ALU. While similar to a standard shift, the compact and expand logic differs in that the multiplexers required do not share the same control signal. However, since each multiplexer select signal depends only on the RIB offset, the select signals are static since RIB offsets are predetermined. Each additional RIB that we add requires a higher degree of multiplexing.

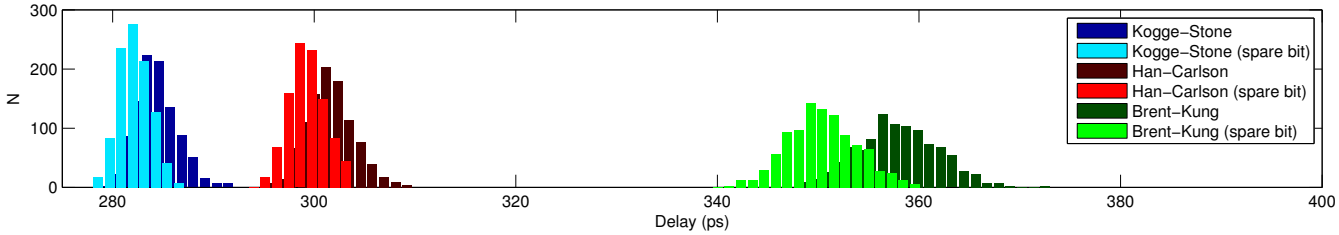
An alternative design is shown in Figure 8. This case requires only a single additional stage instead of two stages. In this design, the operand is shifted while containing redundant bits, and an extra "shuffle" fixes any incorrect values after the shift. As the example shows, only a subset of bits require extra shifting. The "shuffle" logic is similar to the compact and expand logic, with some additional complexity. Unlike the previous case, the multiplexer select signals depend on the shift amount as well as the shift direction. The latency of this control logic can be hidden by the shift latency, since the shuffle stage is placed after the shift.

These modifications are intended to allow the shifter logic to function correctly in a processor with RIBs, and do not directly provide any performance or reliability benefits. Though the modified shifter logic is potentially slower than an unmodified design, we assume a delay gap between the adder and shifter to hide the additional latency. In case the extra multiplexer stages introduce too much latency, we suggest a possible microarchitectural modification: the latency of the shift phase can be reduced by removing support for wide shifts if a log shifter is used. Assuming that wide shifts are rare, the restriction would have minimal impact on performance.

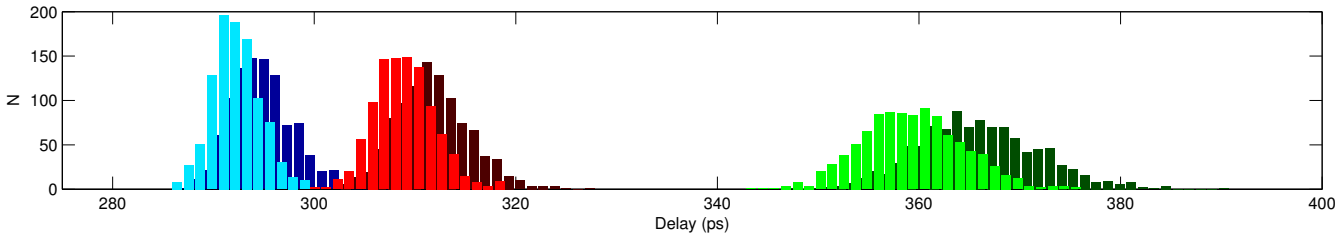
Multiplication introduces a similar problem, since the partial products to be added are essentially shifted versions of an operand. Even if RIBs in all partial products could be aligned, there would be additional complications when summing. Unlike the ALU adder, in which either a single carry or no carries propagate across a RIB, the carry select addition in the multiplier would need to allow multiple carries to traverse a RIB. Thus, our ALU adder modification is ineffective in this case. To avoid these issues, we can leverage the "compact" and "expand" technique shown in Figure 7. This approach has minimal impact on performance, since the multiplier is already a pipelined unit and the extra delay required is minimal. We investigate the performance implications of higher latency multiplication in Section IV-E. Alternatively, dedicated "compact" and "expand" logic in the multiplier can be avoided by using the already-modified shifter unit. Custom "compact" and "expand" instructions could be executed prior to and following multiplication. The advantage of utilizing the modified shifter in this manner is that there is no additional multiplication latency if all RIBs are in the MSBs, but otherwise multiplication will require at least two extra clock cycles and potentially tie up execution resources. For these reasons, we do not further analyze this option.

C. Application to the Register File

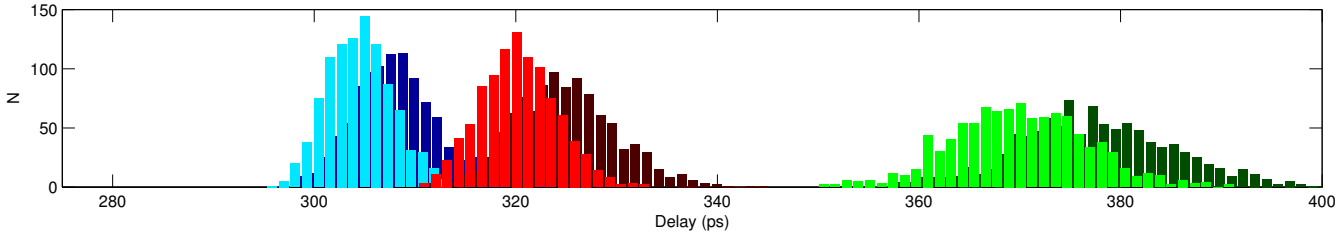
The register file is another unit that may be on a critical path and is typically not pipelined. Application to the register file is conceptually simple, since each bitslice is independent. It is important to note that as an SRAM structure, the register file has very different properties from pure logic structures like the ALU. In particular, SRAM has many more critical paths with a relatively low logic depth. As noted in [9], this makes the register file particularly susceptible to random variations.



(a) Threshold voltage variability (3σ) set to 20%



(b) Threshold voltage variability (3σ) set to 30%



(c) Threshold voltage variability (3σ) set to 40%

Fig. 9. ALU critical path delay histograms. Each distribution is comprised of 1000 circuits, each with randomized delays. Our scheme with one spare RIB consistently improves the delay distribution.

Unlike the ALU, increasing the width of registers in the register file does increase the overall delay of the structure. This is due to an increase in wire lengths. Wordline wires, for instance, run across the width of the structure, and are lengthened as register widths are increased.

As we increase register widths, we therefore observe a linear increase in the modeled register file delay. This trend is in sharp contrast to the ALU delays shown in Figure 6. In this paper, we use a modified version CACTI 6.5 [19] to model the register file. Though this trend of increasing latency with register width could be counterproductive, we consider that as SRAM structures, register files may soon be designed with the same redundant columns employed in caches for reliability reasons.

IV. Analysis

In this section, we present results from our simulations of random variations in ALU critical loops with different adder types. We begin by discussing our simulation methodology and framework. Finally, we present and discuss the results obtained with and without RIBs.

A. RIBs in the ALU

Since random variation affects each device separately, it is potentially very time consuming to simulate at a low level. To address this challenge, we characterize the delay of each gate type at a certain level of V_t variability using HSPICE Monte Carlo simulations. This initial step allows us to obtain a probability density function for the delay of each type of gate. All of our simulations use the 22nm predictive technology model [20]. We then synthesize a logic netlist for the desired adder and bypass network width. Adders are implemented with inverting CMOS gates, and our bypass network model includes latches, buffers, and multiplexers as shown in Figure 5. We choose the bypass wire capacitance such that bypass delay is roughly one third of the cycle time when a 64-bit Kogge-Stone adder is used.

To generate a circuit with randomized delays, we use the previously-obtained delay PDFs for each gate in the circuit. Note that since the ALU is relatively small and likely to be uniformly affected by D2D and systematic WID variations, we simulate only random variations. After all gate delays have been generated, we calculate the maximum delay of the circuit. If RIBs are present, the offsets are found that will minimize the circuit's worst-case delay. We begin our analysis by examining the case with only a single RIB present. This allows comparison across all of the adder organizations, since no extra delay will be introduced through widening the adder. The addition of a single bitslice requires the least complexity and hardware overhead, and allows us to easily perform an exhaustive search for the optimal RIB offset.

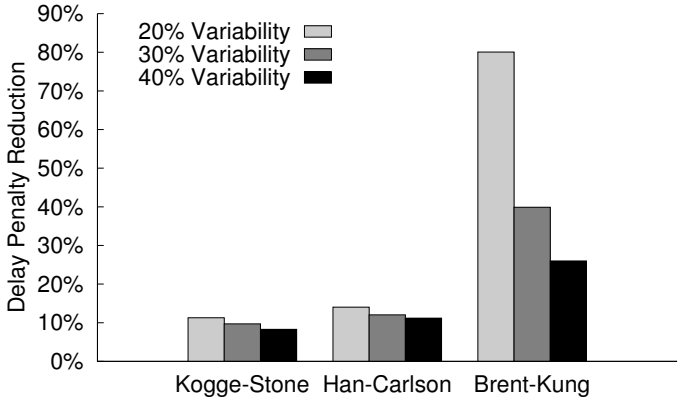


Fig. 10. Reduction in the delay penalty imposed by random threshold voltage variations when one RIB is introduced.

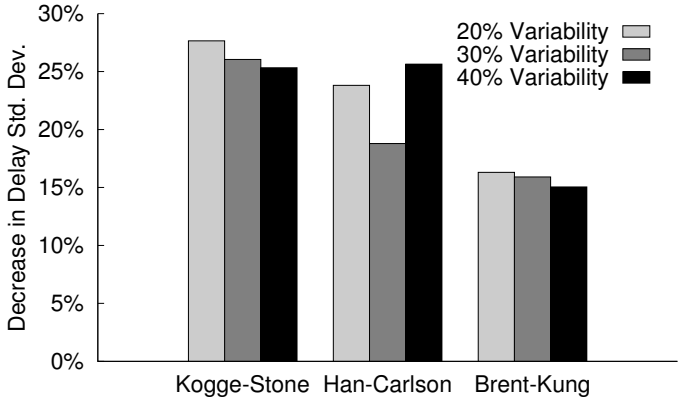


Fig. 11. Decrease in ALU delay distribution standard deviations when one RIB is introduced.

Figure 9 shows the delay distributions when 1000 ALU critical path circuits are generated for each configuration. The baseline cases use a 64-bit wide datapath, and we add a single unused bitslice when applying our scheme. We repeat the experiment for 20%, 30%, and 40% variability in threshold voltages. We consider these levels of variability to be reasonable, since although the ITRS predicts quite high variability for minimum-sized devices, not all devices will realistically be minimum-sized. We first observe the expected shift in the delay distributions with increasing variability, validating our initial motivation. We also see that with the addition of a spare bitslice, we are able to consistently regain some of the performance lost. In addition to decreasing the mean delay, adding a RIB also reduces the standard deviation of the delay distribution. This is beneficial, since with a narrower performance distribution, the worst case cycle time is also decreased.

When analyzing circuits with RIBs, we are not concerned with the overall reduction in delay. Instead, we focus on countering the delay overheads introduced by random variations. These "delay penalties" are shown in Figure 1 and correspond to the mean delays of the baseline distributions in Figure 9. The benefits of applying our technique in the ALU with one RIB are summarized in Figure 10. Using circuit delays in the absence of variation for reference, we see that we can reduce the delay overhead of random variation by about 10%. The ALU with the Brent-Kung adder dominates the plot because it is actually a special case in this experiment. Since there is only a single critical path at the adder output, it is possible to decrease the adder delay by placing a RIB at this offset even in the absence of delay variations. As Figure 11 shows, we are able to reduce the standard deviation of the delay distribution by up to 25%.

Next, we consider the scalability of our technique by examining the theoretical benefits of introducing multiple spare RIBs. There are two alternatives that can be considered in such a design. In the simplest approach, a constraint can be imposed that all RIBs must be adjacent. This constraint simplifies the control logic, but restricts the potential benefits. The second option is to allow arbitrary RIB offsets. While potentially the most ideal and flexible option, the added flexibility introduces complexity in the peripheral logic. In particular, the compact logic discussed in Section III-B requires a higher degree of multiplexing with more RIBs. Allowing arbitrary RIB offsets also significantly increases the number of possible configurations, and a brute-force algorithm for finding the ideal RIB columns quickly becomes infeasible. To simplify this analysis, we employ a greedy heuristic. For each circuit, we begin by placing all RIBs so that they fill the most significant bits. We then find the offset for the first RIB that improves the delay the most, and fix it at this position. The algorithm then continues to place subsequent RIBs in this manner. While effective, this heuristic can overlook certain cases. For instance, as shown in Figure 5, adjacent RIBs can "disable" logic deeper in the adder. For this reason, we consider the results of the greedy heuristic as well as another scenario in which RIBs are adjacent, and choose placement

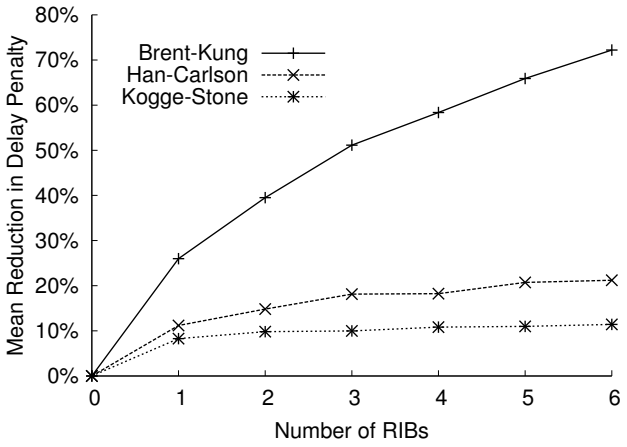


Fig. 12. The reduction in the delay penalty incurred by random variations in the ALU can be increased by adding more RIBs. In this experiment, Vt variability is set to 40% and there are no constraints on RIB placement.

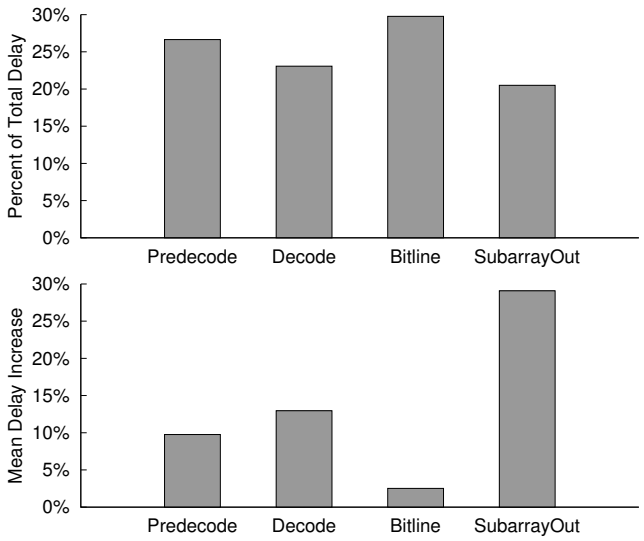


Fig. 13. Register file delay breakdown without variation and delay overheads with 40% variability.

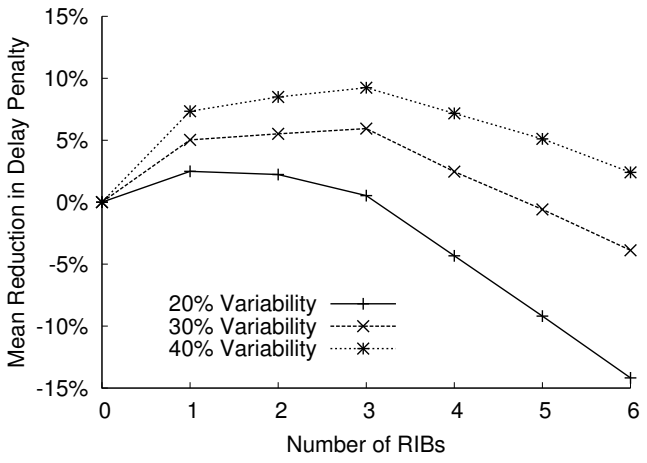


Fig. 14. Reduction in the random variation delay penalty with RIBs in the register file. We see an optimal number of RIBs in terms of the observed benefits.

with better results. We employ this more exhaustive search to investigate the theoretical benefits of using Spare RIBs and discuss a more practical approach in Section IV-D.

Figure 12 shows the results of including multiple RIBs in the ALU and allowing arbitrary placement. As expected, we do not observe significant benefits from adding RIBs beyond the point where the adder delay increases as shown in Figure 6. The slight additional gains that we observe beyond this point appear when occasionally, strong variations along a normally non-critical path make this path become critical. Note that by allowing arbitrary RIB placement, we do not see any increase in the mean ALU delay, even when we have widened the adder beyond the point where its maximum delay would increase if all bits were being used. Since extra stages are required by only some of the outputs in the MSBs, these paths become critical, and we must place RIBs at these offsets. Since these RIBs are now fixed at the locations with increased logic depth, they cannot be moved to counteract delay variations, but keep performance from degrading by masking the delay of the longer paths.

Although this analysis focuses on a single ALU, many processors contain multiple execution units. Considering a superscalar processor, we must take into account systematic within-die variations. Since the ALUs are discrete units, it is possible for them to have different delay characteristics. If one ALU is slower than others due to local variation, the RIB placement can be tailored to primarily benefit this unit. If ALU delays are comparable, on the other hand, RIBs must be placed considering the slow paths in multiple ALUs.

B. RIBs in the Register File

We model the effect of random variation on the register file by using a modified version of HP CACTI 6.5 [19]. CACTI is a tool that can model the area, access time, and power consumption of DRAM and SRAM structures, including caches and register files. CACTI models delays by using device resistances and capacitances to first compute a time constant for each gate. Equation 1 is then used to estimate the gate delay for a rising input [21], with an analogous equation for falling inputs. In this equation, t_f is the time constant, t_{rise} is the input rise time, and b is a constant. Delay is computed as the time between the input voltage reaching the gate switching threshold and output voltage reaching the switching threshold of the next gate. These switching thresholds are indicated by v_{s1} and v_{s2}, respectively. Once the gate delay is calculated, the input rise time to the fan out gate can be computed using Equation 2.

$$delay = t_f \cdot \sqrt{\left[\ln\left(\frac{v_{s1}}{V_{dd}}\right)\right]^2 + 2t_{rise}b\left(1-\frac{v_{s1}}{V_{dd}}\right)}/t_f + \left[\ln\left(\frac{v_{s1}}{V_{dd}}\right) - \ln\left(\frac{v_{s2}}{V_{dd}}\right)\right] \quad (1)$$

$$t_{fall} = \frac{delay}{1 - v_{s1}/V_{dd}} \quad (2)$$

In the case of a rising input, the switching threshold in the model roughly corresponds to the threshold voltage of the pull-down transistor. We use HSPICE simulations to correlate deviations in device threshold voltages with the change in this gate switching threshold parameter. As in the case of the ALU, we use Monte Carlo simulations for our analysis of random variation in the register file. For each register file simulated, we choose a random switching threshold for each gate based on the threshold voltage variability. In its original implementation, CACTI must only calculate the delay of the critical path for a particular configuration. Our modified version expands on this functionality and tracks the cumulative worst-case delay of each gate.

When choosing an optimal memory configuration, CACTI varies a number of parameters including the number of wordlines and the degree of bitline multiplexing. These configuration parameters can be chosen to meet particular design constraints. CACTI allows weights to be assigned to certain constraints so that an optimal configuration can be chosen. In this paper, we use a balanced configuration with weights divided between access time, power consumption, and area. Furthermore, we assume a register file with 6 read ports and 3 write ports and 128 64-bit physical registers as a baseline.

Figure 13 shows the contribution of each component to the delay of the ideal register file. In CACTI, memory arrays are divided into *mats*. The address predecode logic is shared among all mats in a bank, and each mat has its own decode logic that drives the wordlines. In the figure, we include the wordline delay with the "Decode" delay. The "SubarrayOut" delay includes the sense amplifier delay, sense amplifier multiplexing delay, and the data output driver delay. When we introduce random variations in our Monte Carlo simulation, we observe an increase in the mean delay of up to 17% for 40% V_t variability. Figure 13 also shows the mean delay increases in each logic component when our 40% variability case is applied. We note that since the predecode and decode logic is shared across bitlines, RIBs cannot completely compensate for increased delays in this logic. We observe the least delay overhead in the bitlines because their delay is wire-dominated.

Next, we experiment with different numbers of RIBs in the register file. Choosing the optimal RIB offsets in the register file is more straightforward than in the ALU, since we do not have to consider overriding any inputs—all register file address bits are used, and input RIBs fan out only to output RIBs. Thus, we simply choose the outputs with the largest maximum delay and place RIBs at these locations. As Figure 14 shows, RIBs have a greater benefit in the register file when more variation is present. However, depending on the expected level of variation, there may be an optimal number of RIBs. We see that as the register size continues to increase as more redundant bits are added, there is a point where the improvement from adding spare slices cannot counter the extra delay from enlarging the structure.

C. Full System Implications

Considering systematic process variation as well as random variation, there may be a significant delay gap between the register file and the ALU. If either the ALU or the register file becomes the critical path, RIB placement is determined by the slower unit. In the case where delays are comparable, path delays must be compared between the two units, and a hybrid solution can be found.

Often times when there is a large gap between the latency of two pipeline stages, time borrowing can be applied so that the slower unit borrows some execution time from the faster unit. A processor that uses spare RIBs can also employ time borrowing. When time borrowing is employed between two units, a single unit no longer limits the cycle time. Instead, the cycle time is determined by the total latency of the units.

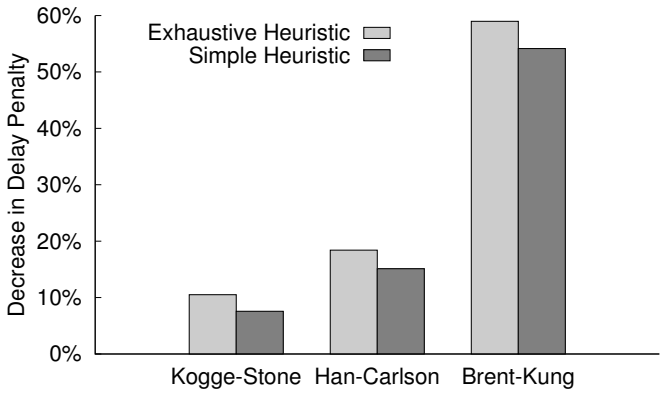


Fig. 15. Comparison of the greedy heuristic for RIB placement in the ALU and the simplified heuristic with 4 RIBs. The simplified heuristic slightly underperforms the greedy heuristic, but falls within 5%.

In this case, we must consider the slowest path delays in both units when determining RIB placement. If the register file is slower than the ALU, for instance, but we see minimal benefit from introducing RIBs in the register file, RIB placement that targets the ALU can "create" more time to borrow for the register file.

D. RIB Configuration

In a processor with spare RIBs, an efficient diagnosis methodology is required to determine beneficial RIB placement. Determining the optimal RIB configuration when targeting the ALU is more complex than for the register file. This is due to the side effects that arise when we fix adder prefix signals that fan out to multiple outputs. In our previous analysis, we employed an exhaustive search heuristic to determine RIB offsets in the ALU, while for the register file we simply place RIBs at the output offsets with the highest delays. To increase the efficiency of diagnosis, this simplified methodology can also be applied to the ALU.

To demonstrate the efficiency of this simplified analysis, we examine a scenario with four RIBs that can be placed arbitrarily to improve the ALU delay. Figure 15 compares the mean improvement in delay of the more time-consuming heuristic with the simplified approach. We see that though the simplified version performs slightly worse than the original heuristic, it comes within 5%.

This result has interesting implications for the method by which RIB offsets could be determined. Since only the output delays must be considered with the simplified approach, it is possible to look for timing failures at the outputs. Naïvely, this could be done using conventional scan latches while scaling the frequency (or voltage) until a failure is observed. Such an approach may be prohibitively expensive in terms of testing time, however, so we suggest an alternative. Li et al. propose a method for performing fine-grained at-speed delay characterization that could be applied [22]. This approach is conceptually similar to the well-known Razor technique, in

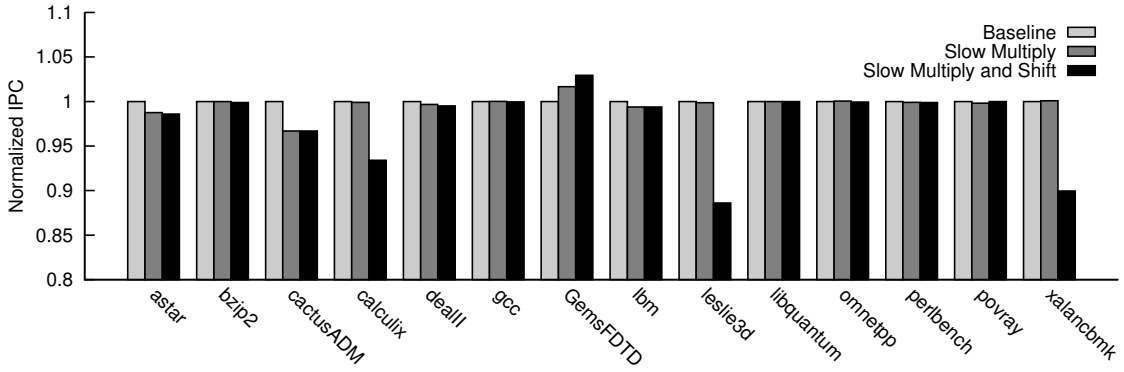


Fig. 16. IPC penalties for SPEC2006 benchmarks with increased integer multiply and shift latencies. Increasing the multiplication latency has minimal impact on IPC.

TABLE I
SIMULATOR CONFIGURATION

Category	Configuration
Superscalar/OoO Components	Physical registers: 128 Fetch/commit: 4-wide Issue: 6-wide Reorder buffer: 128 entries Instruction queue: 36 entries Load queue: 48 entries Store queue: 32 entries Pipeline stages: 11
Execution Units	Integer ALUs: 3 (1 cycle, 4 cycle multiply) Memory: 2 (2 cycles) FP adder/multiplier: 2 (6 cycles) FP div/square-root: 1 (12 cycles)
Memory	Instruction cache: 32KB/2-way L1 data cache: 32KB/4-way L2 data cache: 2MB/8-way Off-chip memory: 168 cycles

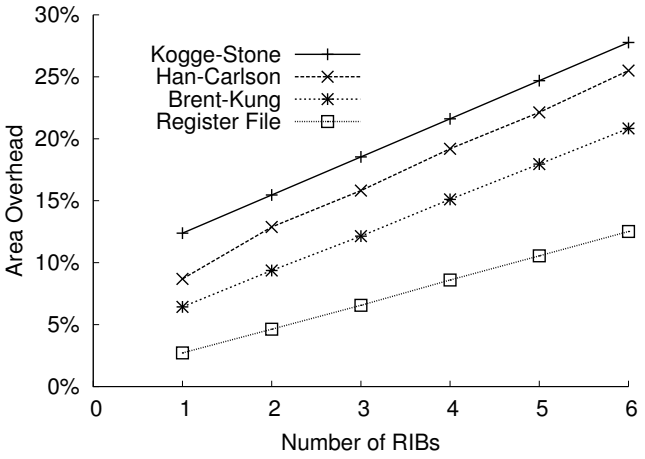


Fig. 17. Estimated area overheads when adding spare slices to ALUs with different adder types and the register file.

which a shadow latch samples the circuit output at a different time than the primary latch [23]. In Li's implementation, the clock for the shadow latches has a negative skew, meaning that they sample earlier than the primary latches. While the circuit is being clocked at its operating frequency, the negative skew on the shadow latch clock is increased. Through the use of comparators and peripheral circuitry, the point at which each output fails can be determined, as indicated by a mismatch between the value captured by the shadow and primary latches. From this information, RIBs can be placed at the offsets of the outputs with the highest delays.

E. Overheads

As previously discussed, correct operation of some execution units with RIBs such as shifters or multipliers may require logic incurring additional delay. Ideally, there exists sufficient cycle time slack to absorb this extra latency. We recognize, however, that this may not be the case. In some situations, it may be necessary to increase the number of execution cycles required by some operations to avoid changing the clock frequency. To examine this scenario, we use a cycle-accurate x86 simulator along with a representative set of the SPEC2006 benchmarks [24]. The simulator is configured as an out-of-order processor with three integer ALUs. More configuration details are shown in Table I. After obtaining the baseline performance, we evaluate the impact of increasing the integer multiplication latency from 4 cycles to 5 cycles. We also experiment with increasing the shift latency from 1 cycle to 2 cycles. Figure 16 shows the impact of these modifications on the IPC for each benchmark. As shown, increasing the multiplication latency by one cycle ("Slow Multiply") has minimal impact on IPC. Also increasing the shift latency by one cycle ("Slow Multiply and Shift"), however, can reduce the IPC of some benchmarks by around 10%. Thus, we conclude that increasing the shift latency should be avoided. Note that in this experiment, GemsFDTD shows a counterintuitive improvement in performance because our changes result in less misspeculation. Similarly, we observed a quite significant performance improvement for mcf, and therefore have omitted it from the figure.

Figure 17 shows the estimated area overheads for the three ALU types studied as well as the register file. For the ALU, our estimate is based on the transistor count. For the register file, we use the area figures reported by CACTI as well as standard cell areas for additional logic. In addition to the overhead incurred by widening each unit, we also include the overhead of the required testing logic as well as the shifter compact/expand logic in the ALU. The difference in overheads for the ALUs is primarily due to the difference in the number of adder critical paths. This is because it is only necessary to add testing logic to characterize the delays of the most critical paths. Note that considering the whole datapath, the extra testing logic is only needed for the critical path logic that our technique is targeting. The majority of the datapath logic can simply be widened, and does not require special logic. For most of the datapath, the overhead is around 1/64 or 1.6% for one RIB in a 64-bit processor. The testing logic cost is therefore amortized over the rest of the design.

V. Conclusions

In this paper, we have presented a bitsliced technique to improve processor performance in the presence of within-die random variations. Specifically, we have performed analysis of this technique when applied to the ALU and register file, which are both potential critical paths. Our technique is capable of reducing delay and has a low overhead. We have also demonstrated that a simple heuristic for placing redundant slices can provide nearly the same benefits as a more complex semi-exhaustive search. In addition to this, we note that while outside the scope of this paper, our technique also has potential to increase yield by tolerating hard defects.

Acknowledgment

This work was supported in part by National Science Foundation awards (CCF-1116450, CCF-095360, and CCF-1016262) as well as generous grants from AMD, the Samsung Advanced Institute of Technology Global Research Outreach Program, and the Wisconsin Alumni Research Foundation.

References

[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proc. 40th Annual Design Automation Conference (DAC '03)*. New York, NY, USA: ACM, 2003, pp. 338–342.

[2] M. Orshansky, L. Milor, P. Chen, K. Keutzer, and C. Hu, "Impact of spatial intrachip gate length variability on the performance of high-speed digital circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 5, pp. 544–553, May 2002.

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

[4] H. Mahmoodi, S. Mukhopadhyay, and K. Roy, "Estimation of delay variations due to random-dopant fluctuations in nanoscale cmos circuits," *IEEE J. Solid-State Circuits*, vol. 40, no. 9, pp. 1787–1796, September 2005.

[5] "International technology roadmap for semiconductors, 2009 edition," *Semiconductor Industry Association*, 2009.

[6] R. Leveugle, Z. Koren, I. Koren, G. Saucier, and N. Wehn, "The hyeti defect tolerant microprocessor: A practical experiment and its cost-effectiveness analysis," *IEEE Trans. Comput.*, vol. 43, pp. 1398–1406, December 1994.

[7] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," in *Proc. IEEE Int. Solid-State Circuits Conf.*, ser. ISSCC 2002, vol. 1, 2002, pp. 422–478.

[8] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "EVAL: utilizing processors with variation-induced timing errors," in *Proc. 41st Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 423–434.

[9] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *Proc. 39th Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 504–514.

[10] X. Liang, G.-Y. Wei, and D. Brooks, "ReVIVaL: A variation-tolerant architecture using voltage interpolation and variable latency," in *Proc. 35th Annual Int. Symp. on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 191–202.

[11] E. Chun, Z. Chishti, and T. N. Vijaykumar, "Shapeshifter: Dynamically changing pipeline width and speed to address process variations," in *Proc. 41st Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 411–422.

[12] A. Tiwari, S. R. Sarangi, and J. Torrellas, "ReCycle: pipeline adaptation to tolerate process variation," in *Proc. 34th Annual Int. Symp. on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 323–334.

[13] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Mitigating parameter variation with dynamic fine-grain body biasing," in *Proc. 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–42.

[14] Z. Chen and I. Koren, "Techniques for yield enhancement of vlsi adders," in *Proc. IEEE Int. Conf. on Application Specific Array Processors*, ser. ASAP '95. Washington, DC, USA: IEEE Computer Society, 1995.

[15] I. Kim, Y. Zorian, G. Komoriya, H. Pham, F. Higgins, and J. Lewandowski, "Built in self repair for embedded high density SRAM," in *Proc. Int. Test Conf.*, Oct. 1998, pp. 1112 –1119.

[16] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. 22, pp. 786–793, August 1973.

[17] T. Han and D. Carlson, "Fast area-efficient VLSI adders," in *Proc. 8th Computer Arithmetic Symp*, 1987, pp. 49–56.

[18] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. 31, pp. 260–264, March 1982.

[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A tool to model large caches," *HP Laboratories online*, 2009.

[20] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm design exploration," in *Proc. 7th Int. Symp. on Quality Electronic Design*, ser. ISQED '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 585–590.

[21] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[22] J. Li and J. Lach, "Negative-skewed shadow registers for at-speed delay variation characterization," in *Proc. 25th Int. Conf. Comput. Design (ICCD '07)*, Oct. 2007, pp. 354–359.

[23] D. Ernst *et al.*, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Proc. Int. Symp. on Microarchitecture*, 2003, pp. 7–18.

[24] A. Phansalkar, A. Joshi, and L. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 412–423.