Resilient High-Performance Processors with Spare RIBs

This processor design incorporates bit-sliced redundancy along the data path, making it possible to tolerate defects without hurting performance, because the same bit offset is left unused throughout the execution core. This design enhances performance by avoiding slow critical paths created by random delay variations. Adding just one bit slice reduces the delay overhead of random process variations by 10 percent while providing fault tolerance for 15 percent of the execution core.

David J. Palframan
Nam Sung Kim
Mikko H. Lipasti
University of Wisconsin-Madison

Each successive technology generation brings new reliability challenges to overcome. These issues are manifested as hard defects or parametric variations across a die. Defects or variations impacting only a portion of the die are of particular concern, because such occurrences can cause the whole chip to be discarded or clocked at a lower frequency. Such scenarios in which a small amount of faulty logic determines the entire chip's fate are inherently inefficient, because the remaining circuitry might still be good.

Prior work suggests the use of redundant logic blocks that can be swapped in for a faulty one. Such redundancy is often relatively coarse grained (for example, a spare arithmetic unit) and requires multiplexing logic to select which blocks to use. Our observation is that these multiplexers can cause additional delays to critical paths and hurt performance. In this article, we analyze a novel fault-tolerant processor design based on bit-sliced redundancy, as we've previously proposed.[1] By creating a wider data path with extra bit slices, we can leave some intermediate slices unused. We keep these redundant intermediate bit slices (RIBs) at the same offset throughout the data path to avoid multiplexing.

This approach produces a versatile and resilient processor core. First, we can exploit the redundancy in the traditional manner to tolerate hard defects in the data path. To do this, we simply decommission the bit slice that contains the hard fault. Using redundant bit slices allows tolerance of hard faults that might not be covered by traditional redundancy, such as faults in pipeline latches. Second, because our redundancy scheme doesn't degrade performance, we can aggressively pursue high-frequency operation and use the redundant slices to counter extra delays incurred by random process variations. For this approach, we can simply leave the bit slices with the slowest logic unused. We particularly emphasize the delay increase from random process variations in this article.

Random process variations are a type of within-die variation that occurs at fabrication. This type of variation results from dopant

Published by the IEEE Computer Society

fluctuations and can therefore independently affect each gate's delay. Thus, this effect can result in a few slow logic gates that can limit the whole-unit performance.

In this work, we comprehensively evaluate a processor designed with Spare RIBs. Specifically, we analyze the hard fault coverage that this technique provides in the execution core and discuss performance enhancement in the presence of random process variation.

Background and motivation

Historically, designers have considered die-to-die variations more important and prominent than within-die variations.[2] To regain yield lost due to die-to-die variations, they often use binning. 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. Die-to-die variation no longer remains the sole concern, however, as subwavelength feature sizes increase the level of within-die variation. Process variation can be systematic in that it impacts a region of the die or it can randomly affect individual devices.

Although frequency binning can be applied to improve yield when systematic within-die variation is present, it's less efficient because variation is no longer constant across the chip. Unlike die-to-die variation, within-die variation decreases the performance of fabricated chips, because a chip can be clocked only as fast as its slowest path. For example, Figure 1 shows the increase in the average arithmetic logic unit (ALU) delay with different adder types and degrees of variability. In this case, random variations alone have caused the delay overheads, though we can observe a similar trend with systematic variations.

Techniques that target within-die variation range from the circuit to architecture level. In the circuit domain, recent work proposes modifying the body bias to compensate for varying threshold voltages.[3] Although we can apply this technique to subsections of the die, it's best suited to compensate for die-to-die or systematic within-die variations. Still other techniques propose clocking a processor at its intended frequency and detecting any errors that might occur. In this case, the error rate can be decreased by purposeful modification of certain circuit paths' latency.[4]

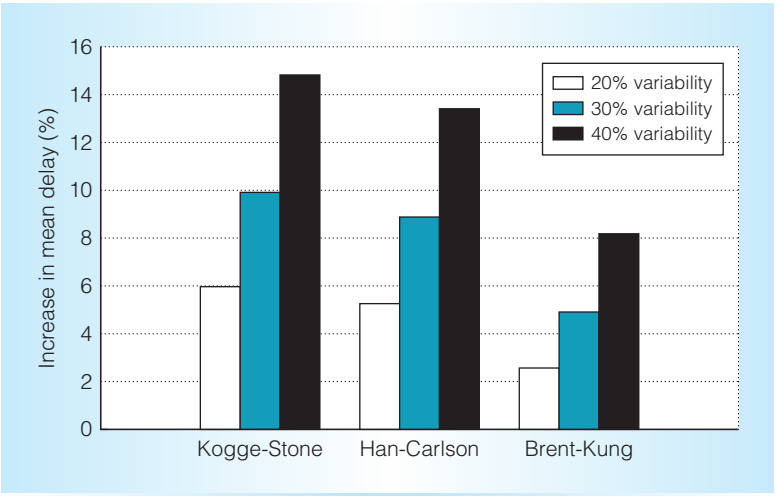

Figure 1. Increase in arithmetic logic unit (ALU) delays due to random variations. Each bar represents the mean delay of a distribution of 1,000 generated circuits.

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.[5] Another option is to reduce the processor frequency only when there's sufficient instruction-level parallelism to require both the fast and slower functional units.[6] Yet another alternative is time borrowing, in which specific clock signals can be skewed to let a slow stage use some of the time originally allocated to a faster stage.[7]

Just as frequency binning might not be optimal in the case of systematic variations, we suggest that techniques to address systematic within-die variations might not be ideal for random variations. Many of these techniques are applied uniformly to whole pipeline stages, but random variation isn't uniform throughout the logic.

Intermediate bit slices

Our proposed design uses a data path that includes more bit slices than there are data bits. This design lets us accommodate unused slices at purposefully chosen bit offsets.

That is, if the original design calls for a 64-bit-wide data path, we could implement a 65-bit data path, allowing a single bit to be left unused at the same location in data registers, latches, wiring, and logic. Keeping this redundant bit at the same offset throughout the data path has the benefit of simplicity and avoids adding multiplexers to the critical path. For instance, consider a naive implementation in which we add RIBs only to particular units. In this case, only the targeted logic would contain extra bit slices, and we would require multiplexers at the logic input and output to route around the unused slices. Keeping the unused slices throughout the data path eschews the need for such input and output multiplexers and their potential to counterproductively increase delays. Additionally, maintaining redundancy throughout the data path lets our design tolerate some hard defects in logic that typically isn't protected, such as defects in the bypass network and pipeline latches.

There are two ways that RIBs can decrease the maximum stage delay when delay variation is present. Most obviously, once offsets are chosen for the unused slices, the output of any unit at these bit offsets can be ignored while correctness is preserved. Therefore, any critical paths that fan out only to RIB offsets no longer contribute to the maximum delay. This behavior is also what lets us tolerate hard defects. We also consider that because the input to any unit can be ignored at these offsets, these input values can be overridden. Assuming 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 could render some gates static and remove them from the critical path.

Applying this concept to much of the data path is relatively straightforward because many resources are dedicated to simply moving data around. Likewise, bitwise operations such as AND and OR require no special consideration when including RIBs. We can extend our scheme to the cache or constrain it to a smaller part of the processor. Execution logic brings additional challenges, however, because data is being manipulated.

Application to prefix adders

The ALU and bypass loop is a critical path in most processors; because of dependency chains, pipelining it usually isn't considered. Because addition requires carry precomputation or propagation, it is typically the ALU operation with the highest latency. Of the adders we study, the fastest (Kogge-Stone) has a delay that is $O(\log N)$, while a log shifter's delay is also $O(\log N)$. In this case, if the shifter is on the critical path, we could make a microarchitectural modification to disallow completing wide shifts in a single cycle in order to reduce the shift latency. Blocks other than the ALU, such as issue or floating-point logic, could also be on the critical path for certain designs. Unlike the ALU, however, these structures have more flexibility in that we can change the instruction window size or pipeline the floating-point logic differently. Based on this insight, we perform delay analysis of the ALU considering only the adder and bypass network.

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 doesn't matter. One way to enforce carry propagation is to insert multiplexers between each bit slice. Although this approach might be effective for simplistic carry-propagation adders, it would be expensive in terms of area, complexity, and delay when applied to higher performance prefix adders that compute carries in parallel.

Assuming we can manipulate the bits in the operand RIBs, we can make carries propagate by setting 1s in the unused bits of one operand and 0s in the other, as in Figure 2a. 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 bit slice, we assert the propagate signal while clearing the generate signal. To avoid adding extra delay, we propose augmenting the gates that generate these signals with an extra control signal. Figure 2b shows the logic used in the building blocks of parallel prefix adders, as well as our proposed logic modification. In this work, we consider the Kogge-Stone,

Han-Carlson, and Brent-Kung adder architectures to explore tradeoffs between complexity, delay, and the number of critical paths.

In the case of a hard defect, appropriately configuring our redundant bit slices makes it possible for the processor to tolerate the majority of hard faults in the adder. Unlike bitwise logic, the adder's carry signals fan out and make it impossible to correct 100 percent of the possible hard faults in the adder. Figure 3 shows an example of an 8-bit Han-Carlson adder with two RIBs. Because bits at the RIB offsets contain no useful data, any logic that fans out only to the RIBs does not affect correctness. Thus, in the case of a hard defect, we can configure the RIBs such that the fault does not affect correctness. Also, remember that we are setting some propagate signals to 1 and some generate signals to 0. Should, for instance, a propagate signal already be stuck at 1 near the adder input, we can configure the RIBs such that this value stays at 1.

Similarly, RIBs also makes it possible to create a processor that can tolerate a subset of possible delay faults within the adder and its critical path. Because some propagate and generate bits that correspond to a RIB remain fixed, the delay to generate these signals no longer contributes to the critical path, as Figure 3 shows. Additionally, if we align a RIB with an output that is on a critical path, any logic on the critical path no longer contributes to the maximum delay. Because we ignore the value at the redundant offset, it doesn't matter if this signal switches late.

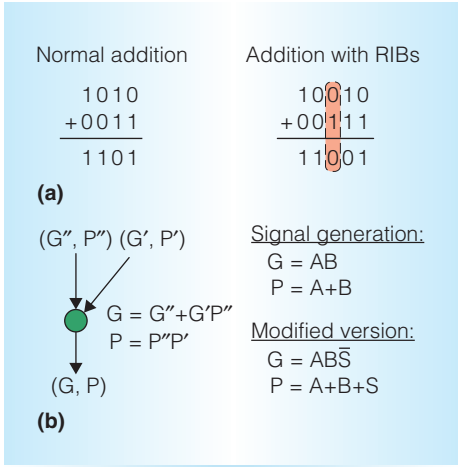

Figure 2. Adder operation with Spare RIBs (redundant intermediate bit slices) and required modifications. Conceptual example of binary addition with RIBs (a). Note that the carry traverses the extra slice. Basic prefix adder operation (b). Our proposal adds an additional skip input *S* to the initial propagate and generate logic for each bit position. *A* and *B* are operand bits at a given offset.

Execution logic modifications

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 data path. Multiplication introduces a similar problem, because the partial products to be added are essentially shifted versions of an operand. To allow for correct operation, we propose sandwiching the shifter and multiplier between two additional logic stages. The first stage compacts the input operand by removing all redundant bits. The compressed value is then shifted (or multiplied) as usual, and the last stage reintroduces redundancy at the appropriate offsets.

Like the ALU, the register file is another unit that can be on a critical path and typically isn't pipelined. Application to the register file is conceptually simple, because each bit slice is independent. Unlike logic, static RAM (SRAM) has many more critical paths with a relatively low logic depth. As Liang and Brooks note,[5] this makes the register file particularly susceptible to random variations. RIBs in the register file (and other data path SRAM structures) allow hard faults or delay variation to be tolerated in column circuitry such as storage cells, bitlines, and sense amplifiers. Notably, coverage doesn't naturally extend to the address decode logic, because this is shared across columns. Aside from wider registers, the register file requires no special modification to be compatible with RIBs.

Defect tolerance analysis

Using Spare RIBs enables the tolerance of hard faults in data path pipeline latches, execution units, and SRAM structures.

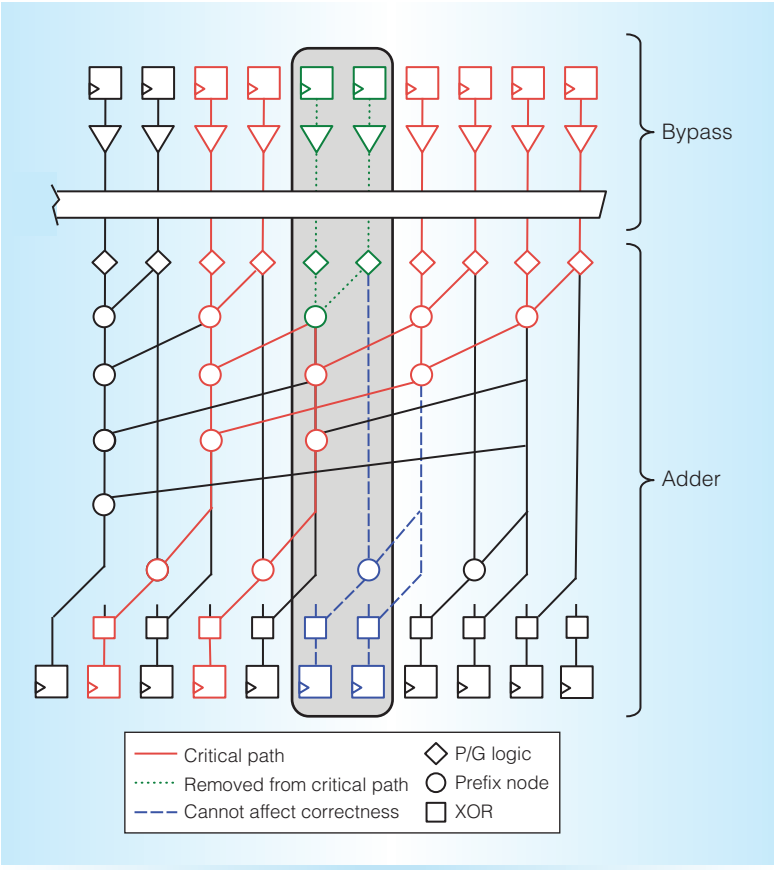

Figure 3. Example of adding two extra bits to an 8-bit Han-Carlson adder. The shaded region indicates the location of the two RIBs.

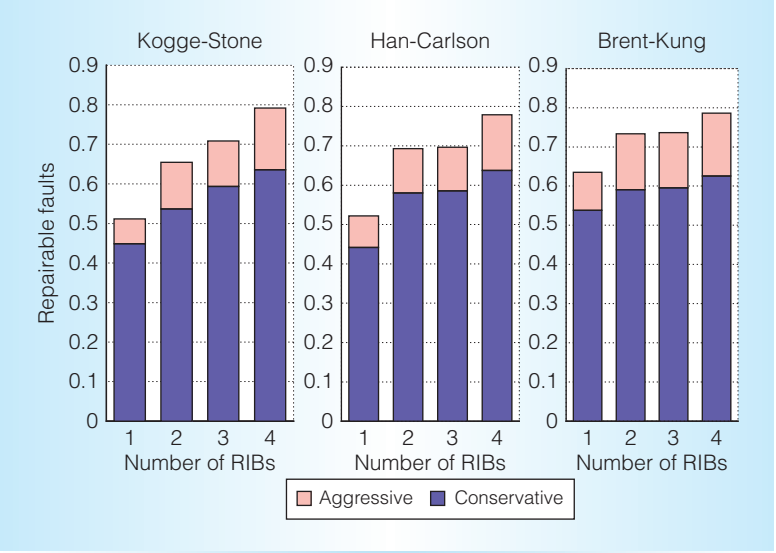

Figure 4. Fraction of repairable adder faults with different numbers of RIBs. Adding more redundancy increases the fault coverage.

To evaluate hard fault coverage, we look specifically at the integer adder. As a starting point, we use netlists for 64-bit Kogge-Stone, Han-Carlson, and Brent-Kung adders. Next, we widen each of these circuits depending on the desired number of RIBs. Assuming a single stuck-at fault model, we then iterate through all nodes in the netlist and consider the scenario in which each is stuck at 0 or 1.

In each case, we determine if there is a RIB configuration that would make the adder usable. Most of the time when we can repair the adder, we don't use the faulty node's value in the computation. We place these scenarios in the conservative-repair category. As previously mentioned, we can occasionally exploit the fact that a propagate node is stuck at 1 or a generate node is stuck at 0. We classify these cases as aggressive repairs.

Figure 4 shows the fraction of repairable faults for each adder type with different redundancy levels. Our single-fault model wouldn't benefit from arbitrary RIB placement, so we always place RIBs at adjacent bit positions. Adding redundancy also increases the total number of possible faults. As shown, including more redundancy produces a net increase in fault coverage because more nodes that fan out to multiple outputs can be covered. In the aggressive case, more redundancy lets us tolerate faults that are deeper in the adder.

We also estimated overall fault coverage for a sample processor. We based this experiment on the Alpha 21264 design and confined the RIBs to the integer data path, excluding caches. To determine the area of the repairable logic, we used the area breakdown in Shivakumar et al.,[8] in addition to the Alpha 21264 floorplan. Repairable data path units include the register file, integer ALUs, load and store queues, and the translation look-aside buffer. We modeled the SRAM structures using a modified version of HP Cacti to estimate the area of unrepairable decoder logic present in each structure. We also accounted for metadata bits such as flags and considered these to be unrepairable.

The 21264 also has several integer execution units, including four integer ALUs, two

Table 1. Fault coverage for execution logic.

Structure	Core area (%)	Unit fault coverage (%)	Core fault coverage (%)
Integer execution units	19.2	22.5	4.3
Register file	3.4	79.7	2.7
Load-store queue	4.1	60.0	2.5
Translation look-aside buffer	7.5	71.3	5.3
Total	34.2	N/A	14.8

of which have shifting functionality, and a pipelined integer multiplier. We synthesized these execution logic blocks to estimate the percentage of repairable versus unrepairable ALU area, because RIBs can't repair the shifters or the multiplier.

Table 1 shows the area and fault coverage for the different structures. By multiplying each structure's area with its fault coverage, we obtained the fault coverage relative to the execution core (excluding caches). We then summed the fault coverage contribution from each structure, giving us an estimate of about 15 percent fault coverage from using Spare RIBs. This is just under half of the integer execution logic, which comprises 34.2 percent of the execution core.

Delay variation analysis

For our random-variation analysis, we used HSpice to generate a probability distribution for each gate type's delay. We then synthesized a logic netlist for the desired adder and bypass network width.

To generate a circuit with randomized delays, we used the previously obtained delay probability distributions for each gate in the circuit. Because the ALU is relatively small and likely to be uniformly affected by die-to-die and systematic within-die variations, we simulated only random variations. After all gate delays were generated, we calculated the circuit's maximum delay. If RIBs were present, we found the offsets that would minimize the circuit's worst-case delay.

Figure 5 shows the delay distributions when 1,000 ALU critical-path circuits are generated for each configuration. The baseline cases use a 64-bit-wide data path. As the figure shows, the Kogge-Stone adder is inherently the fastest, whereas the Brent-Kung adder has the highest delay. In each case, the addition of a single redundant bit slice at the optimal offset shifts the delay distribution and improves performance.

When analyzing circuits with RIBs, we were unconcerned with the overall reduction in delay. Instead, we focused 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 5. Using circuit delays in the absence of variation for reference, we reduced the delay overhead of random variation by about 10 percent by adding one RIB.

We also considered our technique's scalability by examining the benefits of introducing multiple spare RIBs. Using multiple spare RIBs can be useful if there are multiple slow paths in the same unit or spread across units. When slow paths exist in two different units, for instance, having multiple RIBs lets us target both units, because each RIB's bit offset is fixed throughout the data path, and it is unlikely that multiple slow slices would exist at the same bit offset. To simplify this analysis, we employed a greedy heuristic to determine the best configuration. Figure 6a shows the results of including multiple RIBs in the ALU. As expected, we didn't observe significant benefits from adding RIBs beyond the point where each (full-width) adder delay increased due to widening it. For a detailed discussion of this behavior, see our previous work.[1]

In the register file, we modeled the effect of random variation using a modified version of HP Cacti.[9] The Cacti tool can model the area, access time, and power consumption of DRAM and SRAM structures, including caches and register files. As with the ALU, we used Monte Carlo simulations for our

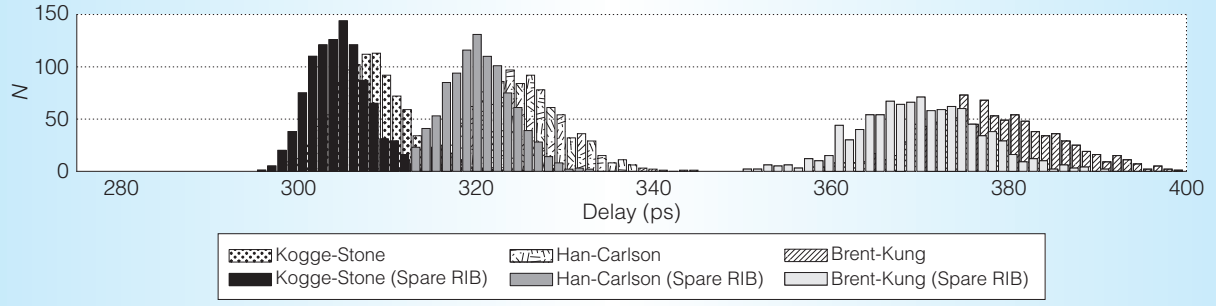

Figure 5. Histograms of ALU critical-path delay. Every distribution comprises 1,000 circuits, each with randomized delays and 40 percent threshold voltage variability. Our scheme with one spare RIB consistently improves the delay distribution.

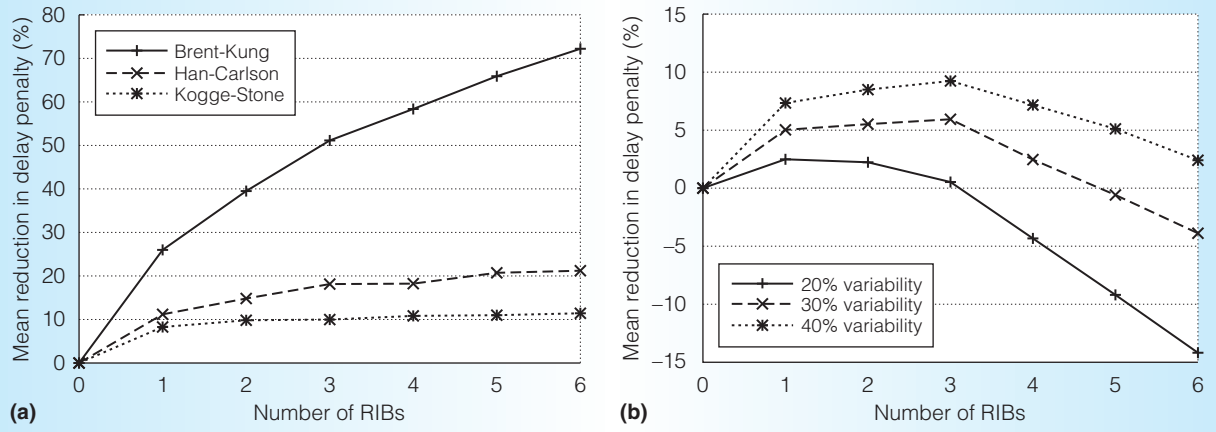

Figure 6. Benefit of RIBs in the ALU (a) and the register file (b). We can reduce the delay penalty incurred by random variations by adding more RIBs. In these experiments, threshold voltage variability is set to 40 percent. For the register file, we see an optimal number of RIBs in terms of the delay reduction.

analysis of random variation in the register file. For each register file simulated, we chose a random delay for each gate on the basis of the threshold voltage variability. Our modified version of Cacti tracks the cumulative worst-case delay of all gates. In this work, we modeled a register file with six read ports, three write ports, and 128 64-bit physical registers as a baseline.

When we introduced random variations in our Monte Carlo simulation, we observed an increase in the mean delay of up to 17 percent for 40 percent V_{TH} variability. To counter this increase, we experimented with different numbers of RIBs in the register file. Choosing the optimal RIB offsets in the register file is straightforward: we simply choose the outputs with the largest maximum delay. As Figure 6b shows, RIBs have a greater benefit in the register file when more variation is present. As the register size continues to increase as more redundant bits are added, there's a point where the improvement from adding spare slices can't counter the extra delay from enlarging the structure.

Overhead

To facilitate RIB configuration for hard faults, we relied on functional-testing mechanisms already employed after fabrication. To configure RIBs for optimal performance

in the case of delay variations, we must introduce some additional circuitry to detect critical-path delays. We used negative-skewed shadow latches, which can be used at speed to detect path delays.[1]

In our previous work, we estimated the area overhead of adding Spare RIBs and the required testing logic on a per-unit basis.[1] We computed per-unit area overhead to be in the range of 10 to 20 percent due to the required testing and control logic. The extra testing logic is needed only for the critical-path logic, however, whereas the majority of the data path logic can simply be widened and doesn't require special logic. This puts the overall data path overhead at around 1/64, or 1.6 percent for one RIB in a 64-bit processor. Considering the Alpha 21264 example used in our hard-defect analysis, the overhead is significantly less, because the integer execution logic comprises roughly one-third of the core.

RIB configuration information can be stored in on-chip fuses that are scanned into latches when the chip is powered on. For a 64-bit processor, we required 7 bits of configuration data per RIB to indicate which bit slice to decommission.

Technology scaling and the ensuing reliability challenges will continue to drive designers to create increasingly resilient processors that can tolerate process variations and defects. To this end, it will likely be necessary to combine a number of robust design techniques. Spare RIBs is orthogonal to many techniques that aim to counter the performance implications of process variation. Our fine-grained redundancy can also be integrated with a more coarse-grained approach. Determining the ideal combination and integrating fine-grained redundancy into additional logic is left to future work. MICRO

Acknowledgments

This work was supported in part by generous grants from Advanced Micro Devices, IBM, and the National Science Foundation (CCF-1116450, CCF-095360, and CCF-1016262). Nam Sung Kim has financial interest in AMD.

References

1. D.J. Palframan, N.S. Kim, and M.H. Lipasti, ''Mitigating Random Variation with Spare RIBs: Redundant Intermediate Bitslices,'' *Proc. 42nd Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 12), IEEE CS, 2012, pp. 1-11.
2. 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,* Feb. 2002, pp. 183-190.
3. J. Tschanz et al., ''Adaptive Body Bias for Reducing Impacts of Die-to-Die and Within-Die Parameter Variations on Microprocessor Frequency and Leakage,'' *Proc. IEEE Int'l Solid-State Circuits Conf.,* IEEE CS, 2002, pp. 422-478.
4. S. Sarangi et al., ''EVAL: Utilizing Processors with Variation-Induced Timing Errors,'' *Proc. 41st IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2008, pp. 423-434.
5. X. Liang and D. Brooks, ''Mitigating the Impact of Process Variations on Processor Register Files and Execution Units,'' *Proc. 39th IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2006, pp. 504-514.
6. E. Chun et al., ''Shapeshifter: Dynamically Changing Pipeline Width and Speed to Address Process Variations,'' *Proc. 41st IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2008, pp. 411-422.
7. A. Tiwari, S.R. Sarangi, and J. Torrellas, ''ReCycle: Pipeline Adaptation to Tolerate Process Variation,'' *Proc. 34th Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM, 2007, pp. 323-334.
8. P. Shivakumar et al., ''Exploiting Microarchitectural Redundancy for Defect Tolerance,'' *Proc. 21st Int'l Conf. Computer Design,* IEEE CS, 2003, pp. 481-488.
9. N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi, *CACTI 6.0: A Tool to Model Large Caches,* tech. report HPL-2009-85, HP Laboratories, 2009.

David J. Palframan is a research assistant in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include low-cost error mitigation and reliable computer

architectures. Palframan has an MS in computer engineering from the University of Wisconsin-Madison. He's a student member of IEEE.

Nam Sung Kim is an associate professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include low-power and high-performance circuits, circuit-microarchitecture codesigns, CAD algorithms, and bio-inspired computing systems. Kim has a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He's a senior member of IEEE.

Mikko H. Lipasti is the Philip Dunham Reed Professor of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include high-performance, low-power, and reliable processor cores; networks on chips (NoCs) for many-core processors; and fundamentally new, biologically inspired computation models. Lipasti has a PhD in computer engineering from Carnegie Mellon University. He's a fellow of IEEE.

Direct questions and comments about this article to David Palframan, 1415 Engineering Drive, Madison, WI, 53705; palframan@wisc.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.