Rescue: A Microarchitecture for Testability and Defect Tolerance

Ethan Schuchman and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{erys, vijay}@purdue.edu

Abstract

Scaling feature size improves processor performance but increases each device's susceptibility to defects (i.e., hard errors). As a result, fabrication technology must improve significantly to maintain yields. Redundancy techniques in memory have been successful at improving yield in the presence of defects. Apart from core sparing which disables faulty cores in a chip multiprocessor, little has been done to target the core logic. While previous work has proposed that either inherent or added redundancy in the core logic can be used to tolerate defects, the key issues of realistic testing and fault isolation have been ignored. This paper is the first to consider testability and fault isolation in designing modern high-performance, defect-tolerant microarchitectures.

We define intra-cycle logic independence (ICI) as the condition needed for conventional scan test to isolate faults quickly to the microarchitectural-block granularity. We propose logic transformations to redesign conventional superscalar microarchitecture to comply with ICI. We call our novel, testable, and defect-tolerant microarchitecture Rescue.

We build a verilog model of Rescue and verify that faults can be isolated to the required precision using only conventional scan test. Using performace simulations, we show that ICI transformations reduce IPC only by 4% on average for SPEC2000 programs. Taking yield improvement into account, Rescue improves average yield-adjusted instruction throughput over core sparing by 12% and 22% at 32nm and 18nm technology nodes, respectively.

1 Introduction

CMOS scaling trends allow for increased performance through faster devices and innovations in circuits and architectures. Despite the reduction in feature size, added innovations keep chip area roughly constant. The combination of a scaling feature size and a constant area results in greater vulnerability to *defects* at fabrication time. Increased vulnerability results in lower yields and decreased profitability.

To understand how defects affect yield and how yield scales, we turn to the common yield model: yield = $e^{-faults/chip}$. Note that yield is related to the average faults/chip, and not defects/chip. In yield analysis, faults are considered a subclass of defects. Defects include all imperfections, both those that cause malfunctions and those that have no effect on circuit operation. Faults include only the defects that cause circuit malfunctions [15]. These faults are hard, persistent faults, not to be confused with transient faults. The fact that an increase in chip area would increase faults/chip is partially responsible for the economic limits that keep chip size from growing.

The average faults/chip in the yield expression is calculated from a *fault density*. Because defect density increases under scaling, faults/chip increases *even if chip area remains constant*. A simple model explains this trend: assuming that defects are circular, only defects with diameter greater than the feature size can cause malfunctions. As feature size decreases, defects that were not large enough to be faults now become faults, increasing fault density.

For the most part, process improvements have been responsible for controlling fault density and maintaining yield. To maintain the economically-acceptable, random-defect-limited yield of 83% at constant area, the ITRS roadmap requires defect budgets to improve as the square of the scaling factor. It is not clear that such improvements will be both attainable and economical. By the 65nm node, there are process passes where manufacturable solutions that can meet the defect budget are not known [24]. Microarchitectures for managing faults are worth exploring.

There have been previous attempts to improve yield through means other than process control. For memories, Built-in Self Test (BIST) combined with redundancy techniques such as spare rows, columns, and sub-arrays have significantly improved DRAM yield. These techniques have moved into on-chip caches as well, but the processor has been left exposed. Single faults in the processor kill the chip.

Tolerating faults in the processor is hard because the processor is not as highly-regular and redundant as memory. More recently, chip multiprocessors (CMPs) have made it possible to begin considering CPU-core-level redundancy, called *core sparing*. [1] describes features in the Power4 that allow in-the-field diagnostic tools to isolate hard faults to single chips (each containing two cores) in a multi-chip-module CMP. [20] describes hardware and testing techniques needed to isolate faults to, and then disable, single cores in a CMP. This strategy makes it relatively easy for chips to be salvaged by enabling only fault-free cores. However, because a single fault kills an entire core and faults/chip grows with technology scaling, the number of faulty cores per chip increases. Therefore, we advocate a finer-grain approach where we disable individual components of a core so that the core can be used even if some of its components are faulty.

Physical clustering of faults has always resulted in higher yields than if faults were randomly distributed, but has never solved the defect problem. With clustering, multiple faults are more likely to hit the same chip killing only one chip as opposed to each fault hitting and killing a different chip. It may seem that if the clusters are the same size as cores, then exploiting core-level redundancy as done by core sparing would suffice and there

would be little opportunity for our finer-grained approach. However, clusters are not *exactly* the same size as cores and do not cause faults in *every* microarchitectural block eliminating any chance of salvaging cores. Consequently, despite clustering, core sparing disables many cores with only a few defects that could be tolerated with a finer-grain approach, allowing the core to be salvaged. In our results, we include clustering effects by using the ITRS clustering model [24] and show that our approach significantly improves over core sparing.

Previous fine-grain approaches include bit slicing in a non-pipelined processor [17], using spare functional units to be mapped-in during a post-fabrication phase [3], and exploiting inherent redundancy in superscalar processors [23]. Exploiting fine-grain redundancy adds the following extremely important testability requirement: *It must be possible to detect, isolate, and map out faults with sufficient precision and speed so that only faulty resources are disabled*. If faulty behavior can be isolated only to a group of components, the *entire* group must be assumed faulty and be disabled. Because testing time is so important, current testing techniques isolate faults only to the granularity of cores and not microarchitectural blocks. Unfortunately, the previous fine-grain approaches do not consider the testability requirement in modern processors.

In this paper, we focus on architecting modern processors for testability and defect tolerance. To propose realistic architectures that exploit microarchitectural redundancy for defect tolerance we start with four basic requirements that we place on the architecture and testing processes: 1) any defect-tolerance approach must be based on realistic and commonly-used testing methodology; 2) it must be possible to isolate faults to the precision of microarchitectural blocks; 3) testing time required to isolate faults in a core must be comparable to that needed today to detect faults in a chip; and 4) because extra logic for isolating and mapping out is added whether there are faults or not, the extra logic must be as little as possible to avoid degrading the cycle time and yield.

We satisfy these requirements using the following ideas which are the main contributions of this paper:

- Because scan chains are the choice testing methodology, our approach is based on conventional scan chains and Automated Test Pattern Generation (ATPG) [12].
- The second requirement translates to a new constraint on the allowed interconnections among microarchitectural blocks. We define *intra-cycle logic independence (ICI)* to formalize the constraint. Some pipeline stages (e.g., decode) already conform to the constraint, and we show why; others (e.g., issue) do not. We propose novel logic transformations, called *ICI transformations*, to redesign the non-conforming components to satisfy the constraint. The transformed components form our novel *microarchitecture* called ***Rescue***, which is the first microarchitecture to consider testability and fault isolation. Our transformations incur minimal performance degradation.
- Conventional scan test is fast because each scan chain can test for thousands of faults at the same time. Because conforming to ICI allows our microarchitecture to use the *same* scan chains and the standard test flow, our fault-isolation time is similar to conventional scan test time.
- Adding extra logic to map out individual faulty component (e.g., functional unit) in a multi-way-issue pipeline would incur prohibitive overhead. Instead, we map out at a coarser granularity and disable an entire pipeline way (e.g., faulty functional unit's backend way, or a faulty issue queue segment and its associated search/selection logic). Thus we exploit the inherent microarchitectural granularity to avoid overhead.

We build *a verilog model* of Rescue, and insert 6000 randomly chosen faults all of which can be isolated using just conventional scan chains and ATPG. We show that our ICI transformations reduce IPC by only 4% on average for SPEC2000 programs. We also evaluate performance and yield together through yield-adjusted throughput (YAT) [23], where throughput is measured in instructions per cycle. Our simulations show that Rescue improves average YAT over CPU sparing by 12% and 22% at 32nm and 18nm technology nodes, respectively.

Section 2 gives background on scan chains and scan testing. Section 3 defines ICI and describes ICI transformations. Section 4 describes our microarchitecture. Section 5 describes our methodology and Section 6 presents results. Section 7 discusses related work, and Section 8 concludes.

2 Scan Test

Testing and design for test (DFT) techniques [12] have been in use for decades to simplify and speed up the complex problem of determining if a manufactured processor is free of hard faults. One of the most prevalent DFT techniques is scan test which allows state to be inserted and collected from devices under test (DUT) with minimal additional external pins [12]. Scan chains are inserted in a logic circuit before layout. The process of insertion replaces memory elements (e.g., latches and registers) with scan-equivalent elements. There are many scan cell styles used, but we assume the simple and common multiplexed flip-flop variety of scan cells. (Because scan techniques can handle both edge-triggered flip-flops and level-sensitive latches, we use latches and flip-flops interchangeably). This scan cell contains two additional ports, a second data input port and a control signal port for a mux to select between the two input ports. The additional data input port is connected to adjacent scan cell output ports so that all scan elements are connected in series and form a shift register. After a state is shifted in, the DUT can be taken out of test mode and *cycled* for one or more cycles of regular operation, and then returned to test mode. With every memory element replaced by its scan equivalent (full-scan) which is common today, one cycle of operation between scan-in and scan-out is sufficient for detection of all detectable stuck-at faults (a net that is either stuck always at 1 or always at 0) [12]. We assume full-scan, stuck-at faults and single cycle tests. After the cycle of operation, the state of the DUT can be scanned out through the scan-out pin. Figure 1 shows a circuit during scan-in, after scan-in has completed, and after one cycle of execution. The scan chain input is all ones and the primary inputs are all zeros. A tester can read the output state along with the chips primary output ports and compare them against a gold standard output. A mismatch signifies a faulty chip, and the chip is discarded. We will show that with single cycle scan tests, faults can be isolated quickly to the microarchitectural block

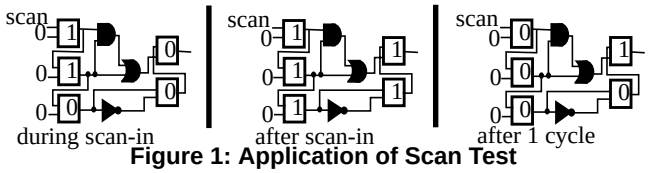


Figure 1: Application of Scan Test

granularity.

Automated Test Pattern Generation (ATPG) [12] software is responsible for generating the test patterns that are scanned in during test. ATPG software works by targeting each possible fault. To test for a stuck-at-1 fault at a particular node, the ATPG software finds register state and input pin values that cause the circuit to attempt to drive the node to 0 and propagate the resulting value to a scan register or output pin. The tester then observes that the output is incorrect and flags a fault.

Diagnosis [12] is used to improve yield in future lots by pinpointing defective tools and steps in the fabrication process. Because diagnosis pinpoints faults as precisely as possible—usually at the gate level or lower—it is a time-consuming process (on the order of hours) that usually requires physical circuit inspection with scanning electron microscopes [26]. Consequently, diagnosis is performed only on a small sample of failing chips. We show this level of precision is not needed for defect tolerance.

3 Intra-cycle Logic Independence

We first describe how ATPG and scan testing can be used to isolate faults in a simple case. We then generalize and define *intra-cycle logic independence* as the condition required to enable fault isolation to specific logic components.

3.1 Simple Fault Isolation

Figure 2a shows four logic components arranged into a larger logic system. The logic component X (LCX) and Y (LCY) are both driven by logic component M (LCM) and share one common input (B). LCX and LCY jointly drive LCN. LCM is driven by input pins (controlled by the tester), and LCN drives output pins (observable by a tester). Assume that there are no memory elements (i.e., flip flops or latches) inside this circuit and also that the system can still function if one of LCX or LCY is known faulty and disabled. If logic components were designed well, good ATPG software should be able to find test patterns that would detect any faults in any of the logic components simply by driving the input pins and observing the output pins.

Despite detecting the presence of the error, it would be difficult to isolate the fault—i.e., pinpoint which logic component contains the fault—from looking only at the pin inputs and outputs. The problem is that an incorrect output observed by the tester at the output pins could be caused by any of the logic components. It is a computationally complex procedure (diagnosis) to trace back the observed output to the input figuring out all combinations of faults that could have caused the observed output. Without such a procedure, *we can assume only that a fault occurred somewhere between where the test vector is input and where the output is collected*. In this example, a single fault in LCX would be detected, but the test would only reveal that there are one or more faults somewhere in LCX, LCY, LCM, and LCN. The tester would not know if disabling LCX would make the system fault-free and the whole system would have to be discarded as faulty.

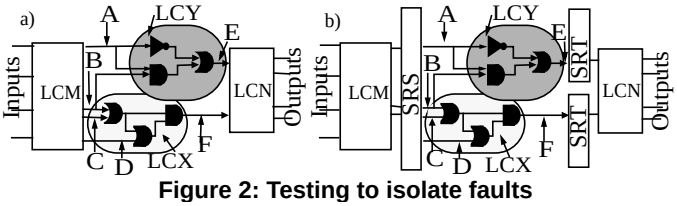


Figure 2: Testing to isolate faults

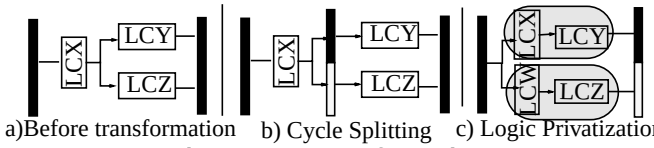
We next consider what happens when the circuit is pipelined and the pipeline latches are integrated into a scan chain to allow scan testing. Figure 2b shows the new circuit broken into 3 stages. Input test data (generated from normal ATPG software) can now be inserted into and collected from the two scan registers (SRS and SRT). That is, test inputs can be applied to points A, B, C, D, and outputs can be collected from E and F. In addition, SRS collects outputs from LCM, and SRT drives inputs to LCN. With this configuration, the granularity of logic between test vector application and collection is now much finer. A fault detected in SRS must be in LCM and a fault detected in SRT must be in either LCX or LCY. Finally, a fault detected in the outputs of LCN must be in LCN. Thus, in pipelined circuits, a fault can be quickly isolated to a pipeline stage (using only conventional ATPG) by checking only where the fault is observed!

In our example system, it is not enough to know that a fault is located in either LCX or LCY. We want to know which one is faulty so that if only one is faulty it can be disabled and the system can still be used. Notice that in Figure 2b SRT has been broken in two to show that some bits of the register collect data only from LCY and the other bits collect data only from LCX (this is constant for a design, and determined once when scan cells are inserted). Now, any faulty data in the top part of SRT must have been caused by LCY and any faulty data in the bottom part of SRT must have been caused by LCX. By a single lookup, faults can be mapped from a specific register bit index (scan chain index number) and can be isolated to LCM, LCX, LCY, or LCN.

Generalizing these examples gives us the *intra-cycle logic independence (ICI) rule* which states that any scan detectable fault can be caused by one and only one element in a set of logical components if and only if there is no communication *within a cycle* among the logical components making up the set.

As an example violation of ICI, assume that in Figure 2b LCY reads the output of LCX as an additional input. In this case an incorrect output is detected in the top half of SRT after 1 cycle. But it is no longer clear that the incorrect output was caused by a fault in LCY. LCX could be sending faulty output to LCY, causing LCY to output faulty data despite LCY being fault free (LCX is faulty), or LCX could be sending correct output to LCY, but LCY is outputting incorrect data (LCY is faulty). Without complicated diagnosis, it is not possible to determine precisely which of the two components is faulty.

An important corollary of ICI is that multiple faults can be tested and isolated at the same time. In standard scan chain testing, each scan chain tests for many possible faults at the same time. This is necessary so that test time is manageable. With ICI, a fault in one component can not influence the testing of faults in other components. If there are multiple faults in different components and all are detectable by one input scan vector, then each



a)Before transformation b) Cycle Splitting c) Logic Privatization

Figure 3: ICI transformations

faulty output scan bit will map to one of the faulty components and all faulty components will be isolated with only one scan vector. Consequently, if a design obeys ICI, it takes no additional scan vectors to isolate faults than would be needed for traditional fault detection in the same design.

3.2 ICI transformations

While some of the stages in an out-of-order superscalar pipeline already conform to ICI, other do not. Those that do not must be modified to conform to ICI. We conform to ICI by turning intra-cycle communication into inter-cycle communication. We make this transformation by a combination of three different methods: (1) *cycle splitting*, (2) *logic privatization*, and (3) *dependence rotation*.

3.2.1 Cycle Splitting

Cycle splitting splits dependent logic into two cycles separated by a pipeline latch. Figure 3a shows an example logic diagram where no ICI is present inside the pipeline stage because both LCY and LCZ read from LCX. Applying cycle splitting results in Figure 3b, which now satisfies ICI in both stages. Any fault can be isolated uniquely to LCX, LCY, or LCZ. Cycle splitting comes at the cost of increased latency because one cycle is split into two, while the clock period remains constant. When there is not much performance penalty for increased latency, cycle splitting is the favored technique. For instance, cycle splitting logic in rename may be acceptable because that would increase branch misprediction penalty whereas cycle splitting select logic may not be acceptable because that would prevent back-to-back issue.

3.2.2 Logic Privatization

Logic privatization replicates logic to avoid two or more blocks depending on the same block. Figure 3c shows the result of transforming Figure 3a through logic privatization instead of cycle splitting. In Figure 3c LCX is duplicated so that LCY and LCZ each read from a unique copy of LCX. Faulty output from LCY can be caused by a fault only in LCX or LCY (we can not know which one), but can not be caused by a fault in LCW or LCZ. For fault isolation purposes LCX and LCY become one super-component (shaded ovals), and LCW and LCZ become another. ICI exists between the two super-components. Logic privatization may be preferred in some cases because it consumes extra area instead of the extra latency needed for cycle splitting. If the duplicated logic block is small, the area penalty may be acceptable.

It is possible to use partial logic privatization to achieve larger grain fault isolation at less additional area cost. As an example of partial logic privatization, assume 4 logic blocks (LCC, LCD, LCE, LCF) reading from one logic block, LCA. Full

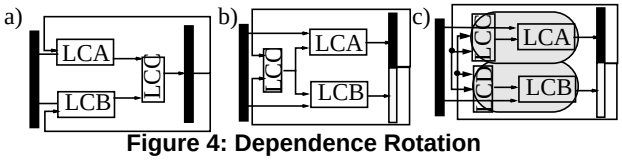


Figure 4: Dependence Rotation

privatization would create 3 additional copies of LCA, one for each of LCC through LCF, resulting in 4 super-components. Instead, partial privatization only creates 1 additional copy of LCA, called LCB. LCC and LCD read from LCA. LCD and LCE read from LCB. Now there are 2 super-components. LCA, LCC, and LCD constitute one. LCB, LCE, and LCF constitute the other.

3.2.3 Dependence Rotation

Privatization through duplication may not be enough to ensure ICI. Figure 4a and b depict another transformation, dependence rotation. In Figure 4a, ICI is violated because LCC reads from both LCA and LCB. Dependence rotation is particularly applicable to single-stage pipeline loops (where the outputs of a stage are inputs to the same stage in the next cycle). Although cycle splitting could create ICI in Figure 4a, cycle splitting may not be favorable if the length of this loop is critical in determining system performance (e.g., issue-wakeup loop). Privatizing only LCC would not help because LCC and its duplicate would still each read from both LCA and LCB. Additionally, duplicating LCB and LCA would provide ICI, but only in the trivial sense by duplicating the entire stage.

Instead we use dependence rotation first which recognizes that not only are LCA and LCB reading from LCC but also LCC is reading from LCA and LCB. The placement of the pipeline latch is somewhat arbitrary in this loop. Dependence rotation rotates the logic around the pipeline latch so that the pipeline latch is in a more favorable location. Because dependence rotation only rotates the logic already within the cycle (does not add new logic), the logic complexity and delay within the cycle stays roughly the same. Dependence rotation transforms Figure 4a into Figure 4b. In Figure 4b, LCC still gets input signals from LCA and LCB, but it reads the signals from a pipeline latch instead of directly from the logic components (now obeying ICI). Conversely, LCA and LCB now read directly from LCC (now violating ICI) while before rotation they read the signals from the pipeline latch. Although dependence rotation has transformed one ICI violation into another, this new violation is easier to handle. Note that Figure 4b resembles Figure 3a. Privatization through duplication of LCC can now be applied, resulting in Figure 4c. In Figure 4c, ICI exists between the 2 super-components shaded in gray.

3.3 Mapping out faults

Once faults are isolated, they have to be mapped out. Map-out involves three issues: (1) Faulty blocks have to be disabled and not counted as available resources, which is trivially achieved. (2) Non-faulty blocks should mask out inputs from faulty blocks. For example, in Figure 4c, if LCA is faulty then the LCA input to the bottom copy of LCC should be masked out so that LCA's fault does not propagate to the non-faulty LCB. (3) To allow degraded operation, instructions need to be routed around faulty blocks. We explain how we do this routing later.

4 Testable & Defect-Tolerant Microarchitecture

Our microarchitecture, Rescue, is based on an out-of-order, multi-issue superscalar which may be thought of as two in-order half-pipelines (frontend and backend) connected by issue. As mentioned in Section 1, adding extra logic to map out individual faulty components (e.g., functional unit or rename port) in a multi-issue pipeline would incur prohibitive overhead. Instead, we map out at a coarser granularity and disable *the entire half-pipeline way* that is affected by the fault (e.g., faulty functional unit's issue/backend way or faulty rename port's fetch/frontend way). Figure 5 shows the half pipeline granularity for two processor ways.

In our approach, an n-way-issue processor can be degraded due to faults as follows: (1) The frontend supports degraded fetch, decode and rename of n-1 down to 1 instructions to support faults in 1 up to n-1 of the frontend ways. (2) The issue queue and the load/store queue can be degraded to half their original size to support a fault in the other half of the queue or the selection/search logic for that half. (3) The backend supports degraded issue of n-1 down to 1 instructions to support faults in 1 up to n-1 backend ways of register read, execute, memory, or writeback. The processor in Figure 5 is operational as long as one frontend way, one backend way, half of the issue queue, and half of the load/store queue (not shown) are functional.

To route around faulty frontend ways, we insert a shifter stage after fetch so that the fetched instructions can be shifted around and directed to the non-faulty ways. In the backend, we add a shifter stage after issue to route issued instruction to functional ways. We describe both shifter stages later.

Each processor would have a one-hot *fault-map register* that specifies the faulty components. For an n-wide-issue machine, there would be 2*n+4 bits to represent the frontend and backend of each of the n ways (2 bits each) and the two halves of both the issue and load/store queue (2 bits each). During test, the register is part of the scan chain. SUN has proposed a similar strategy to support core sparing in CMP [20]. Because we require only 2*n+4 bits, we assume that after test, inputs to the fault-map register can be fixed permanently by fuses as used in [28].

In the rest of the section we proceed as follows. We examine each pipeline stage starting from the most complex stage for ICI compliance. For those stages that already conform to ICI we explain why. To the non-compliant stages, we apply ICI transformations and redesign the stage to be ICI-compliant. We also describe fault map-out to allow degraded operation with limited additional logic.

4.1 Issue

Issue illustrates a complex stage where there is fine-grained redundancy (at the issue-queue-entry level) but no coarse-grain redundancy that can be used easily for defect tolerance. Unfortunately, there is no ICI among the entries. A fault in one entry can propagate to almost any other entry through the selection logic *within* one cycle. The overhead to detect faults at this granularity would be large. Instead, we choose to isolate faults only between halves of the queue and selection logic, and show the necessary modifications to create ICI and allow degraded operation.

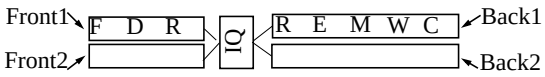


Figure 5: Half-pipeline map out

4.1.1 Baseline Issue Queue

We choose a compacting queue [19,6] as our baseline because it represents the most common implementation today. In a compacting issue queue, instructions are inserted into the tail. Compaction occurs each cycle, counting free slots and choosing which newer instruction should move into each free slot. In parallel with compaction is wakeup and select. At the beginning of the cycle, the previous cycle's selected instructions broadcast to the entire issue queue, waking up instructions that have all their operands ready. After wakeup, instructions are selected for issue so that the maximum amount of issue bandwidth is used while observing resource constraints and issue priority.

The compacting issue queue violates ICI making it impossible to isolate faults to either half in our design. The *old half* contains the older instructions and the *new half* contains the newer instructions. There are three ICI violations that must be considered: (1) Compaction of the new half is dependent on how many free slots are available in the old half. (2) Compaction of the old half is dependent on the state, number and position of instructions in the new half. (3) Selection in each half is dependent on the type and number of (post-wakeup) ready instructions in the other half.

4.1.2 Creating ICI

Handling violations (1) and (2) is best done by cycle splitting of inter-segment compaction. We still allow compaction *within* each segment to occur in parallel in a single cycle, but we split compaction *between* segments into more than one cycle. Note that this does not increase the pipeline depth of the architecture, and consequently, cycle splitting this logic comes with little performance penalty.

In our ICI compliant inter-segment compaction policy, each half does not communicate directly with the other half. All inter-segment communication gets written to a temporary latch at the end of the cycle, and read from the latch in the next cycle. Each cycle, the old half compacts its own entries, and if free slots open up, the old half requests instructions from the new half. In the next cycle, while the new half compacts its own entries, it moves its oldest entries to the temporary latch if the old half had made a request. In the last cycle, the content of the temporary latch is moved into the old half. The new half can insert new instructions from rename in the second cycle of the compaction process because it is in this cycle that the new half moves its entries into the temporary latch. Consequently, we have increased the time between the entries becoming free and new instructions being inserted into the issue queue by only one cycle.

Although compaction now obeys ICI, a further modification is necessary for correctness of our extended compaction policy. Instructions are in the temporary latch for only one cycle, but they must be able to see all wakeup broadcasts. As such, the entries in the latch must have wakeup logic. Because this new wakeup logic reads only from the temporary latch and writes only to the old half, it does not provide a communication pathway between the two halves *within a cycle* and ICI is not broken. Although instruc-

tions can be woken up in the temporary latch, they can not be selected for issue until they reach the old half.

The remaining ICI violation is the inter-segment communication that occurs during selection. Cycle splitting inter-segment communication during selection (as we did in compaction), would prevent back-to-back issue. Instead, we employ dependence rotation and logic privatization to enforce ICI. We now describe how dependence rotation and logic privatization apply to superscalar issue.

Superscalar selection is usually implemented with multiple selection trees. Each selection tree first selects from each half, and then the root node of the selection tree chooses between the two halves. Therefore, the root node is dependent on both halves of the queue. In typical selection logic, the root node selection is the last step in the cycle, where the instructions are selected and latched at the end of the cycle. These selected instructions are then used for broadcast at the beginning of the next cycle.

This data flow is analogous to Figure 4a; LCA corresponds to the new half of the issue queue combined with the selection logic that chooses instruction from that half, LCB is similar but for the old half, and LCC is the roots of the selection trees that choose among instructions presented from the two halves. Accordingly, we apply dependence rotation which rotates the root nodes of the selection tree around the issue queue and sub-trees of the selection logic, breaking all communication between selection of the issue queue halves, producing a data flow analogous to Figure 4b.

However, this dependence rotation causes a problem: we must issue the selected instructions in the cycle immediately following select for back-to-back issue. But the selection process is incomplete in that we have selected only from each issue queue half but not combined the selections of the halves (at the selection tree root). Dependence rotation has eliminated communication between the halves, and has moved this combining at the root to the next cycle (LCC in Figure 4b is fed from the latches in the next cycle).

Breaking communication between the halves causes a significant departure from typical issue policy. Now, there is no way to determine how many and which instructions should be selected by each half to maximize issue bandwidth while not exceeding resource constraints. Instead of trying to limit conservatively or predict the number of instructions selected by each half, we allow each half to select instructions as if the other half will wakeup no additional instruction. Each half still obeys resource constraints, though together their sum may not. In the rare case that more instructions are selected than can be issued, we force a replay of *all* instructions from the half that selected fewer instructions (we replay all instead of some subset for simplicity). Because each half obeys resource constraints in its selection, the non-replayed half will conform to resource constraints. This replay is similar to that on an L1 miss; the issued instructions have not been removed from the issue queue yet. Replay simply clears the issued bit of all instructions issued in the last cycle from the replayed half.

The replay signals, along with wakeup broadcasts, are generated by a logic block that corresponds to LCC in Figure 4b. The replay signals and the broadcasts go to both halves of the issue queue. To maintain ICI, this logic must be privatized (one copy for each issue queue half). The resulting data flow is shown in Figure 6 and is analogous to Figure 4c. In Figure 6, the lower select unit and the lower queue segment are analogous to LCB in Figure 4c and the lower broadcast/replay logic unit is analogous to LCD in Figure 4c.

After issue completes, each issued (not replayed) instruction needs to be routed to a backend way for execution. We use an extra cycle after issue to do this routing. If there are too many selected instructions, there is a replay, and all instruction from the non-replayed half are chosen and routed in their selection order to the backend ways. If there is not a replay then each selected instruction (from both halves) is routed to a backend way. In either case, this routing is simple because there are never more instructions that need to be routed than backend ways. This routing is done by muxes and each mux controller must be privatized to maintain ICI.

4.1.3 Map out and Degraded Operation

Our ICI transformed issue stage easily supports degraded operation after faults have been isolated. Each half (new and old) has three components, the queue half (including wakeup logic), the selection logic for that half, and the wakeup/replay logic for that half. A fault in any of the components is detected as a fault in that half, and the entire half (all three components) is assumed faulty.

With a faulty old half, the new half simply masks out any compaction requests from the old half (Section 3.3). With a faulty new half, the old half compacts as normal, but instead of compacting from the temporary latch, it compacts from the newly renamed instructions, bypassing the new half.

Faulty backend ways are accounted for by reduced resource counts. If only n-1 backend ways are functional, each half's selection logic selects only up to n-1 instructions. The replay signal logic similarly adjusts to replay when greater than n-1 instructions are selected. Furthermore, the routing logic immediately after issue avoids the faulty way.

4.2 Fetch

Fetch includes the i-cache and logic to select the fetch PC among the outputs of the BTB, return-address-stack, and PC increment. The i-cache is covered by BIST with repair. There is no redundancy in the fetch PC logic and therefore no opportunity for defect tolerance. As such, we treat this small logic as chipkill.

The main modification needed is to route around faulty frontend ways. Because fetch simply maps one-to-one the fetched instructions to the frontend ways, it does not already have the routing ability. Therefore, we add a routing stage after fetch like the routing logic immediately after issue. Normally instructions are fetched in parallel and passed, in program order, to the decode stage. If one or more of the frontend ways are faulty, we must ensure that the instructions are still decoded and, in particular, renamed in program order. Accordingly, the routing stage has two functions: (1) Assign the earliest instruction to the first fault-free frontend way, the second instruction to the second fault-free way, and so on, until all the non-faulty ways have been given instructions. (2) Stall fetch and assign any remaining instructions in the same manner until all fetched instructions are processed.

The routing stage is composed of muxes (one for each frontend way) that choose an instruction for that frontend way. We

privatize the control for each mux so that there is ICI in the routing stage and faults can be isolated easily to the way of the shift stage. A faulty way in the routing stage is equivalent to a fault in the corresponding frontend way and results in that way being disabled.

4.3 Decode

Decode obeys ICI without any modifications. Multiple instructions feed into the stage from the fetch-decode latch. Each instruction is decoded in parallel without any intra-cycle communication. Output from the stage is collected in the decode-rename latch.

4.4 Rename

In the rename stage, the single register map table and the free list break ICI. These structures are read by each renamer and therefore cause an ICI violation similar to Figure 3a. Fixing the ICI violation by privatization of the tables is not realistic because that requires fully replicating the relatively large tables once or more. After the tables are read, the mappings are fixed to reflect any hazards. RAW and WAW hazards among the instructions being renamed require source register maps and previous destination-register maps to be fixed. Consequently, we use cycle splitting to separate the table reads from the rename logic at the cost of an additional frontend pipeline stage. In one cycle, register mappings and free registers are looked up in the tables and latched. In the next cycle, the mappings are fixed. There may be dependencies between instructions in the two stages of rename that need to be accounted for. Allowing writes in the second stage to pass through the tables and be read in the same clock cycle by the first stage would violate ICI. Instead, we forward back results from the previous renamed instructions. Since we have separated rename in to two cycles, there is ample time to perform the forwarding. The only cost is a small area penalty.

It may seem that the map-fixing logic would violate ICI. If there is a RAW or WAW hazard between instructions A and B, then it may seem that B's map-fixing logic would read the output of A's map-fixing logic and violate ICI. However, real designs avoid making B's rename depend on A's because doing so would totally serialize renaming among the instructions being renamed. Instead, real designs operate in parallel by redundantly computing the hazards. For each instruction, the hazards of all its previous instructions are computed in parallel. If an instruction appears ahead of three others then the instructions' hazards are computed redundantly by each of the three. Therefore, each map-fixing logic reads all previous instructions' architectural and physical register numbers from the cycle-splitting pipeline latch at the end of rename tables, and does not read anything from the other map-fixing logic and ICI is maintained.

The above cycle splitting makes rename completely testable even when there are faults in the rename tables. This ability is important because then the tables can be tested (using BIST or any other method) while faults in the rest of the processor are being isolated with conventional ATPG.

Although we can isolate faults in the rest of the processor while other faults may exist in the rename tables or the free list, our processor would not be able to execute with faults in the structures because there is no redundancy in them (though there is BIST, there are no spare rows or columns because spares are used for larger structures like caches). To create redundancy, we use partial privatization, but because we already enforced ICI, we have some freedom in how we make the copies; we can avoid wasting ports on redundant reads from the map table and free list.

We create two copies of the tables, each of which has half as many read ports as the baseline design (as done in Alpha 21264 register file for speed reasons [14]). The first copy does the look-ups for the first half instructions, and the second copy for the last half instructions. Data read from the table copies is latched and is potentially used by all renamers in the next cycle. A fault in the first (second) table copy disables the frontend ways for the first (second) half instructions.

Because the rename stage modifies state in the tables at the end of renaming, a few extra precautions are necessary for map out and degraded operation. First, we must ensure that faulty ways do not affect the state of the free list and rename tables. We require that the all free list and rename table write ports have the ability to be disabled selectively based on the fault-map register. Second, we must ensure that a fault-free rename way does not use data from a faulty way. To that end, we use a mask on the matches for RAW and WAW hazards, and ignore matches from faulty ways.

4.5 Register Read

As with rename, we assume that BIST detects faults in the register file and that there is no redundancy. We therefore use multiple copies, each with fewer ports. No additional modifications are necessary as there is no state modification in the stage. Each register file copy is analogous to the logic block in Figure 2. The copies obey ICI. Register specifiers come directly from input latches and go into the copies. There is no communication among the copies, and each copy outputs directly to its output latch.

4.6 Execute

Execute obeys ICI without any modifications. Register forwarding does *not* violate ICI because forwarded data comes from pipeline latches (and therefore implies inter-cycle communication and not intra-cycle communication). However, for map out and degraded operation, we need to ensure that fault-free ways will never try to forward from faulty ways. To that end, we mask out the register dependence match from being signaled if the matching way is specified to be faulty in the fault-map register.

4.7 Memory

We assume that the cache hierarchy uses BIST and row and column sparing, and focus on the load/store queue (LSQ). Surprisingly, searching the LSQ obeys ICI without modification. Insertion requires only minor modifications to support ICI.

We assume search for memory disambiguation is implemented in a tree structure similar to select. Like the issue queue, we segment the LSQ into halves. Two concurrent accesses to the LSQ requires two trees (each tree has two sub-trees, each sub-tree searches half of the queue). There are *two* modes of degraded operation in the LSQ: faults in a half or the sub-tree searching the

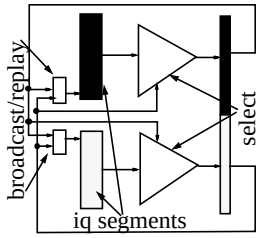


Figure 6: Issue logic flow

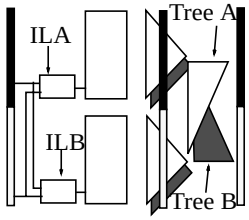


Figure 7: LSQ logic flow

half cause the half to be disabled. Faults in a tree (either of the sub-trees) cause the tree to be disabled allowing only one access but use of both halves.

Because the LSQ is a non-compacting structure, there is little communication between the halves. Because the search can take as long as L1 access, the search tree is usually pipelined into two or more cycles. Therefore, the trees are already cycle split. Figure 7 shows the two LSQ search trees pipelined into two cycles (note that the pipeline latch can fall in the middle of the sub-trees, as shown). Tree A is light, Tree B is dark. In the first cycle, each of the sub-trees searches its half in parallel. Because there are two sub-trees (one light and one dark) reading from each half, the half and its two sub-trees make one super-component. Because there is no communication between these super-components, ICI is maintained between these super-components in the first cycle. In the second cycle, the rest of the trees read from the pipeline latch to generate final results. Though the root node causes communication between its two sub-trees, there is no communication between the two search trees (light and dark) in the second cycle. Consequently, ICI is maintained between the trees (but not between the sub-trees of a tree) in the second cycle. Thus, ICI holds for different components in the first and second cycles, resulting in the two modes of degraded operation, as we show below.

To enforce ICI for insertion of entries into the LSQ we privatize the insertion logic for that half (ILA and ILB in Figure 7). Each half must keep redundant copies of the head and tail pointers. When the tail pointer lies in a half, that half is responsible for inserting instructions in the correct position. For ICI, each copy of the insertion logic gets lumped into the super-component comprising the half it is controlling.

Now we explain the modes of degraded operation: A faulty LSQ half (or search logic in the first cycle) is avoided by the insertion logic and the search logic in the second cycle of search. When the faulty half is disabled (specified in the fault-map register), the fault-free half uses the reduced LSQ size to calculate head and tail pointers, and consequently is responsible for all insertions. Each search tree root ignores (a simple mask based on the fault-map register) results from its sub-tree that searches a faulty half.

A fault in one of the sub-trees in the second cycle is avoided by disabling the corresponding tree entirely (the LSQ remains full size). The faulty search tree is avoided by disabling the backend way that uses the tree, as described in Section 4.1.3.

4.8 Writeback

Writeback modifies register state. Because the register file already uses multiple copies (e.g., 21264), the only additional requirement is that data from faulty ways not be written to the register file (a faulty write port can be treated equivalently to a faulty backend way that writes incorrect data). As with rename tables, we require that the register write ports be disabled selectively by the fault-map register.

4.9 Commit

Commit consists of writes to the free list and the active list. Disabling the write ports prevents faulty backend ways from writing to these tables.

5 Methodology

Our methodology quantifies the value of our intra-cycle-independent architecture, Rescue, in three aspects: testability, fault-free performance degradation, and expected (average) throughput.

To demonstrate the feasibility of fault isolation in Rescue, we create a verilog model of the processor described above. The model includes details of fetch, decode, rename, issue, register read, execute, memory, writeback and commit and all the described logical connections and fault propagation pathways. We map the design into a gate-level verilog description and insert one scan chain using Synopsys Design Compiler. We use Synopsys TetraMax Pro, which is an ATPG generator and fault simulator, to simulate faults and we report the results in Section 6.

We next evaluate the performance aspects of Rescue through performance (IPC) and yield-adjusted throughput (YAT) [23], where throughput is measured in instructions per cycle (IPC). YAT is the sum of the products of the IPC of a degraded configuration times the probability of the occurrence of the configuration (or the average chip throughput when a large number of chips are fabricated).

To measure performance of Rescue we modify SimpleScalar [4] in the following ways: (1) we separate the issue queues and active list; (2) we add two cycles to the branch misprediction cycle penalty to account for the shift stages in the front and back-end of the processor (Section 4.1 and Section 4.2); (3) we cycle-split the inter-segment issue queue compaction, using four entries of each issue queue as the compaction buffers (Section 4.1) (baseline and Rescue issue queues have the same total resources); (4) we hold the issue queue entries for an extra cycle and squash an extra cycle of issued instructions on L1 misses to account for the additional shift stage between issue and register read (Section 4.1); (5) we implement the issue/replay policy described in Section 4.1.

We use our simulator to model a 4-way-issue superscalar processor with the baseline parameters listed in Table 1. Increasing issue width beyond four ways would only increase redundancy and improve our results. When we vary technology generations, we increase memory latency by 50% and add 2 cycles to the misprediction penalty each time transistor area decreases by a factor of two. We simulate 23 of the SPEC2000 benchmarks using SimPoints [21] to determine the execution sample location and run for 100 million instructions. We leave out *ammp*, *galgel*, and *gap* because of simulation time.

Estimating each degraded configuration's probability as

required by YAT is more involved. It is equal to the probability of a faulty circuit that results in the degraded configuration. The relative area of a circuit determines the probability of faults in the circuit, given an average fault density. To compute fault density, we use ITRS's Equation 1 which extrapolates allowable particles per wafer pass (PWP) from faults per mask (F) [24]. S is the critical defect size equal to half the minimum feature size. This equation dictates the necessary improvement in fabrication technology needed to keep yield rates constant. We use this equation in reverse to compute faults per chip at a technology node but assume that PWP remains constant after a specific technology node. We vary the technology node at which PWP rates stop improving. Preventing PWP from scaling causes faults per mask to scale as $1/s^2$ after the technology node, where s is the feature size scaling factor. This trend is common [15, 27].

The remaining task is to compute the relative areas of the processor's circuit blocks. Because area data from die photographs are available only at a coarse granularity and because we wish to reduce the number of required simulations by reducing the number of possible degraded configurations, we divide the processor components into groups of fault equivalent components. A fault in any component in a group causes the entire group to be disabled. Note that this grouping makes our results conservative.

The groups are defined as follows. The frontend is composed of two groups. Each group decodes and renames two instructions (the baseline width is four). A fault in one group halves the frontend bandwidth. A fault in both groups kills the processor. The integer backend is composed of two groups. Each group has two integer ALUs, one integer multiplier/divider, and one memory port. The floating-point backend is modeled similarly, except each group contains only one floating-point adder and one floating-point multiplier/divider. The two issue queues and the load/store queue are independent. A fault in a queue's segment halves the queue size.

We estimate the relative area of each group from the area model provided by HotSpot [25]. Each frontend group area includes decode logic and a copy of the rename tables with half the read ports (Section 4.4). Because rename tables are not usually large enough to justify using copies, and because copies may incur some overhead, we conservatively assume that two reduced-ported copies consume 50% more area than the single fully-ported table. The integer backend group area includes half of the functional unit area and half the integer register file. Because the HotSpot area model already has two copies of the integer register file (from Alpha) [14], we do not increase area estimates for the integer register file. We similarly estimate the floating-point backend group area, but we do assume a 50% increase in floating point register file size due to implementing it as two reduced-port copies. For the two issue queues and load/store queues, we divide the respective areas between the halves of the queues.

We do not address branch prediction or the TLBs, so we add these to chipkill. We also do not focus on caches because there are other redundancy techniques available for the cache data arrays. Cacti 3.2 [22] reports the data array area to be 78% for the cache

$$PWP_n = PWP_{n-1} \times \frac{F_n}{F_{n-1}\langle \frac{S_{n-1}}{S_n} \rangle^2} \qquad \text{(EQ 1)}$$

Table 1: System Parameters

issue width	4
iq, lsq	36, 36
int alu, mult/div	4, 2
fp alu, mult/div	2, 2
branch pred.	8kB hybrid, 1kB 4 way BTB, 15 cycle misprediction penalty
L1 caches	64kB, 2 way, 32 B blocks, 2 cycle, 2 port data cache
L2 cache	2Mb, 8 way, 64 B blocks, 15 cycle
memory latency	250 cycle

Table 2: Total areas and component relative areas

Baseline Total Area	96 mm^2
Rescue Total Area	107 mm^2
frontend	12%
int IQ	4%
fp IQ	4%
int backend	15%
fp backend	21%
LSQ	4%
Chipkill	40%

configuration given in Table 1. We remove this area from our area model, and include only the control/routing area as chipkill.

Because defects in the scan cells make the design untestable we need to count scan cell area as chipkill. To estimate this area, we *compile and map our verilog model* to components in the CMU standard cell library [11]. We then insert scan chains and obtain the pre-layout area breakdown of the design (based on the area and number of each gate-level component used). The queue stage (LSQ and IQ) contain a high proportion (25%) of scan cells because the queues themselves are included in the scan-chain. Among the remaining stages, scan cells account for 12% of the area. Correspondingly, for our fault model we count 25% of the LSQ and IQ as chipkill and 12% of the frontend, integer backend and floating point backend as chipkill.

We then scale the frontend and backend components to account for the additional shift stages (including pipeline latches). From our verilog model, we estimate these as a 2% increase for the integer and floating point backends and a 6% increase for the frontend. Finally, we add an additional 5% area overhead to all redundant components to account for any additional area overhead incurred by our tranformations.

Table 2 shows the resulting relative component area, total area for Rescue, and total area for the baseline core with only scan. We do not calculate component areas for the baseline core because a fault in any part of the core has the same effect and disables the core.

To extrapolate for future technologies, we acknowledge that the area of each core will decrease, as technology scales and CMPs become common architectures. However, we assume that there will be microarchitectural innovation which will add hardware to the core. Consequently, the area of each core will decrease each generation but not as fast as s^2. We assume core growth at a steady pace so that a fixed percentage of new functionality is added when the device area shrinks by half. We show results for different growth rates. Total area of all cores together (without defect-tolerance modifications, but with L1 caches) remains at 140mm^2, as ITRS specifies [24].

Using the above estimates for area and the PWP-based model for fault density, we determine random defect-limited yield using the negative binomial yield model as used by ITRS for all defect budgeting through the end of the road map. The negative binomial yield model extends the simple Poisson yield model by accounting for yield improvements due to clustering of faults. We extend this model as described in [15] to calculate the distribution of pos-

sible chip configurations. For each possible working configuration, the model gives the probability of that configuration occurring. In the negative binomial models, clustering is accounted for by averaging the expected yield results across a mixing function whose parameter, alpha, specifies the amount of clustering. We set alpha to be 2, as projected by ITRS. Because we are interested in YAT instead of yield, we average the expected value of YAT across the mixing function instead of the yield, resulting in Equation 2 and Equation 3. In these equations, *lambda* is the fault density, and *gamma* is the gamma function.

6 Experimental Results

6.1 ATPG results

We use TetraMax to generate basic scan patterns for both the baseline and Rescue. Table 3 lists the scan chain length, the number of scan chains required to test the two designs, and the number of cycles to scan in and apply all scan vectors. There are three important observations that can be made from these data. (1) The scan chain for Rescue is longer than the baseline design because cycle splitting has increased the number of pipeline registers. (2) ICI compliance has reduced the number of scan test patterns required because each component can more efficiently be tested in parallel than without ICI. (3) Fault isolation on Rescue requires only a 13% increase in testing time (cycles) over traditional defect detection.

Table 3: Scan Chain data

	Base	Rescue
faults	111294	113490
cells	2768	3334
vectors	1911	1787
cycles	5272449	5959645

Finally, to verify that the ATPG produced scan chains actually isolate faults in Rescue, we rename each register component to reflect the ICI component that writes to it (ICI ensures there is only one). We randomly insert 1000 faults from each of the following stages: fetch, decode, rename, issue, execute and memory. We leave out register read, writeback and commit because they do not contain significant logic other than RAM tables. The fault simulator, TetraMax, makes each fault active, one at a time, and simulates the application of all generated test patterns. TetraMax reports each failing bit position in the scan-out state, and keeps a table of each cell's name and position. We access this table and verify that the register name matches the name of the ICI component where the fault was inserted. All 6000 faults were isolated correctly.

6.2 IPC degradation

Enforcing ICI eliminates some communication pathways, and delays others across multiple cycles. Consequently, enforcing ICI comes at some cost to IPC. Figure 8 shows IPC results for a typical superscalar processor as described in Table 1 (black bars) and Rescue (white bars). IPC degradation ranges from 0% (swim) to 10% (bzip) with the average degradation across all benchmarks at 4%. It is not necessarily the highest and lowest IPC benchmarks that are affected by modifications, but the benchmarks that are particularly sensitive to issue queue size and selection policy.

$$YAT = \int_0^\infty YAT_E(l)\frac{\alpha^\alpha}{\lambda^\alpha \Gamma(\alpha)} l^{\alpha-1} e^{\frac{-\alpha}{\lambda}l}\, dl \qquad \text{(EQ 2)}$$

$$YAT_E(l) = \sum_{c \in C} Yield_c(l) \bullet IPC_c \qquad \text{(EQ 3)}$$

where C is all possible configurations

6.3 YAT improvement over core sparing

In this section, we discuss the average performance improvement achieved by adding Rescue on top of simple core sparing (CS) where faulty CPU cores in a CMP are disabled. We assume that more cores will be added as technology scales. We first present throughput improvements due to CS alone. We then show how Rescue can significantly improve upon CS.

We evaluate CS and Rescue by calculating yield-adjusted throughput (YAT), where throughput is measured in IPC, for 23 of the SPEC2000 benchmarks individually. We normalize these YATs relative to YAT with 100% yield and no degraded cores, and report the relative YAT averaged across all the benchmarks for 4 technology nodes between 90 and 18 nm and plot them in Figure 9. The hatched bars show YAT of chips with no redundancy techniques. The white bars show additional YAT obtained by adding core sparing. The black bars show additional YAT gained with Rescue. We do not present results for individual benchmarks because of space limitations and because they add little additional information.

Recall from Section 5 that we assume that economic or physical constraints will prevent PWP from improving, causing defect density to increase. We show two scenarios: PWP stagnating at 90nm (Figure 9a), and scaling until 65nm and then stagnating (Figure 9b). Also recall that we assume that microarchitectural innovations will contribute to core growth beyond the usual shrink due to technology scaling. We assume a 20%, 30%, 40%, and 50% growth starting from one core per chip at the 90nm node, and from two cores per chip at the 65nm node. The core complexity has grown steadily in the past. We envision that the growth will continue. We choose 20-50% growth based on the fact that (1) adding SMT to a superscalar pipeline results in 24% growth [13] and the reasoning that adding SMT is a relatively modest change and (2) we use only a 4-wide core while current machines are already 6-wide and greater. Figure 9 shows four bars for each node (labeled a-d), each representing YAT for one growth rate at that node. The number of cores fabricated (maximum number of functional cores) for each growth rate at each node is also given in the table under the bars for each node.

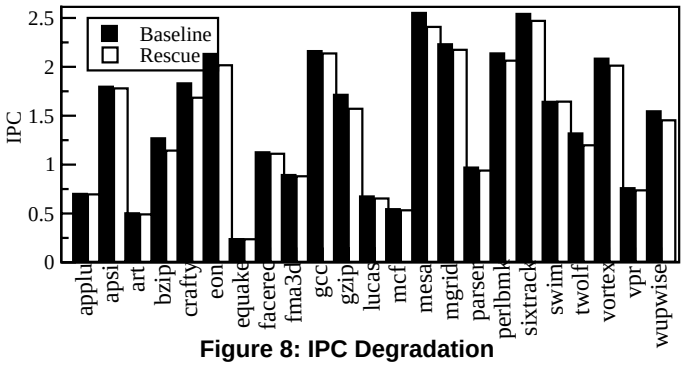


Figure 8: IPC Degradation

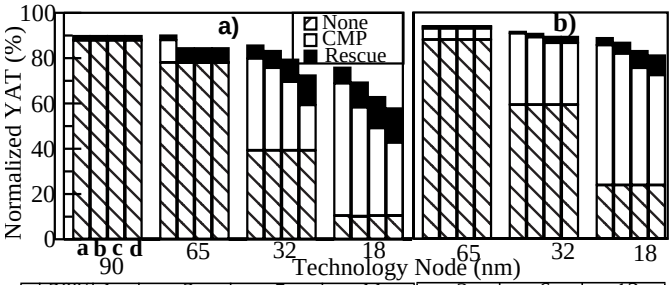
From this figure, we see three trends in the YAT results for CS. First, as technology scales, CS's improvements increase. With no defect tolerance of any kind, a single fault makes the entire chip unusable. With CS, each fault disables at most one core on the chip; the remaining cores still contribute to the chip throughput. As devices shrink, each core becomes more susceptible to defects, but the number of cores per chip increases. With a greater number of cores, the loss of a core leads to a smaller part of the chip being disabled, and hence greater improvement over the baseline.

Second, comparing Figure 9 a and b, we see that if process improvements allow defect densities to stagnate only at as late as 65nm, the opportunity for redundancy techniques such as CS (and Rescue) are reduced.

Finally, although CS is better than no sparing, average chip throughput still significantly decreases at each node**.** Under high fault densities, disabling entire cores when most of the components are still fault-free costs significant throughput.

We next consider improvements when Rescue is applied on top of core sparing. At near-term technology nodes, low fault densities mean little opportunity for either redundancy technique (Rescue or CS). As fault density increases, more cores in CS are disabled. In Rescue, only some of those cores are disabled because of faults in the non-redundant chipkill area. Most faults affect redundant components. In these cases, faults may reduce the per-core IPC, but the core still contributes to chip throughput. Similar to CS, Rescue also achieves better improvements as technology scales. PWP stagnating at the 90nm node with high core growth (50%), by the 32nm node Rescue improves upon CS by 25% while at the 18nm node Rescue improves upon CS by an average of 40%. With medium core growth (30%), at the 18nm node Rescue improves an average of 22% over CS, and at low growth (20%) Rescue still improves upon CS by 13%. With PWP stagnating at the 65nm node, Rescue is still able to improve upon CS by 8% for medium core growth (30%), and 14% for high growth (50%) at the 18nm node.

At each node, Rescue shows greater improvement under larger core growth. We see this trend because under larger core growth, the chip has fewer cores at each node. Scaling from 1 core at the 90nm node we reach 11, 7, 5, 4 cores for core growths of 20%, 30%, 40% and 50%, respectively. With fewer cores, each disabled core disables a larger portion of the chip. Because Rescue avoids disabling entire cores, when there are fewer cores Rescue shows better improvement over CS. Thus, Rescue prevents faults from limiting core growth, and encourages continued microarchitectural innovation.



growth		90	65	32	18	65	32	18
a	20%	1	2	5	11	2	6	13
b	30%	1	1	4	7	2	5	10
c	40%	1	1	3	5	2	4	7
d	50%	1	1	2	4	2	4	6

Figure 9: YAT improvement from redundancy

7 Related Work

[23] proposed using inherent superscalar redundancy to improve yield, and evaluated results with YAT. The paper assumes advances in logic BIST without describing fault isolation and that sub-microarchitectural-block-level redundancy can be exposed without discussing microarchitectural modifications or cycle time penalties to support the intricate indirection required. We focus on microarchitectural-block-level redundancy which needs less indirection. We show how current scan test techniques can be used for fault isolation at this granularity.

Goldstein proposed FPGA-based techniques as a solution to defect tolerance for molecular electronics with high fault densities [7,8]. Although FPGAs can provide good defect tolerance [16], current FPGA implementations have significant cycle time and complexity limitations [9]. We exploit architectural-block granularity where the use of gate arrays is not necessary. Teramac [10] is another FPGA-based approach to achieve defect tolerance. The configurability of Teramac comes from a fat tree interconnect. Such high-connectivity interconnect may not scale to nanoscale integration. We exploit the existing redundancy and connectivity to provide defect tolerance without relying on special interconnects. Another work [18] evaluates defect-tolerance techniques and concludes that reconfiguration combined with redundancy of the order of 10^3-10^5 is needed to cope with high defect rates. We propose to exploit the existing redundancy in processor cores without needing such high redundancy.

[2] proposes self-healing arrays as an alternative to BIST with repair that detects and avoids defective entries in RAM arrays at run-time. [2] applies their techniques to the BTB and RUU. Self-healing arrays could be used along with Rescue to augment the coverage of Rescue. Self-healing arrays could provide coverage of the BTB and active list structures that we currently do not cover, and allow errors in the entries of a register file or rename table copy to be tolerated without disabling the entire copy as is currently required by Rescue.

There are too many papers to list that address testing and DFT techniques for modern non-defect-tolerant processors. Instead we refer the reader to [12, 5] for a thorough survey of modern testing and DFT.

8 Conclusions

As defect densities increase with scaling, exploiting core-level redundancy, as done in core sparing, is not enough to maintain yield. There is a need to exploit finer-grain redundancy at the microarchitectural-block level. To address this need, faults must be detected with enough precision to isolate faulty microarchitectural blocks.

We defined intra-cycle logic independence (ICI) as the condition needed for conventional scan test to isolate faults quickly to the required precision. We showed that some pipeline stages in an out-of-order multi-issue processor are already ICI-compliant. We proposed logic transformations for the non-compliant stages resulting in Rescue, the first microarchitecture to consider testability and defect isolation. We showed that faulty microarchitec-

tural blocks can be mapped-out without much overhead.

We built *a verilog model* of Rescue, and inserted 6000 faults, all of which were isolated using just conventional scan chains. We showed that our ICI transformations reduce IPC by only 4% on average for SPEC2000 programs. We also evaluated performance and yield together through yield-adjusted throughput (YAT) [23], where throughput is measured in instructions per cycle. Our simulations showed that Rescue improves average YAT over CPU sparing by 12% and 22% at 32nm and 18nm, respectively. Our improvements increase both as technology scales and under larger core growth, preventing faults from limiting the performance benefits of technology scaling and microarchitectural innovation.

Acknowledgments

We would like to thank the anonymous reviewers and Mark Hill for their comments. We would also like to thank Purdue University Rosen Center for Advanced Computing for providing technical support and maintenance of the computation resources used in this project.

References

[1] D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd. Fault-tolerant design of the IBM pSeries 690 system using the POWER4 processor technology. *IBM Journal of Research and Development*, 46(1), 2002.

[2] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[3] M. Broglia, G. Buonanno, M. G. Sami, and M. Selvini. Designing for yield: A defect-tolerant approach to high-level synthesis. In *Proceedings of the IEEE Internatioanl Symposium on Defect Tolerance in VLSI Systems*, November 1998.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, 1996.

[5] M. L. Bushnell and V. D. Agrawal. *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*. Kluwer Academic, 2000.

[6] J. A. Farrell and T. C. Fisher. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, 1998.

[7] S. Goldstein and M. Budiu. Nanofabrics: spatial computing using molecular electronics. In *Proceedings of the 28th annual international symposium on Computer architecture*, 2001.

[8] S. Goldstein, H. Schmit, M. Bidiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4), April 2000.

[9] R. W. Hartenstein. Coarse grain reconfigurable architectures. In *Proceedings of the ASP-DAC 2001 Design Automation Conference*, February 2001.

[10] J. R. Heath, P. J. Kuekes, G. S. Snider, and S. Williams. A defect-tolerant computer architecture: opportunities for nanotechnology. *Science*, 280:1716–1721, 1998.

[11] C. Inacio. CMU DSP: The Carnegie Mellon synthesizable signal processor core. http://www.ece.cmu.edu/ lowpower/benchmarks.html.

[12] N. Jha and S. Gupta. *Testing of Digital Systems*. Cambridge University Press, 2003.

[13] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *Proceedings of Fifteenth Symposium of IEEE Hot Chips*, August 2003.

[14] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, October 1998.

[15] I. Koren and Z. Koren. Defect tolerance in vlsi circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838, 1998.

[16] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Low-overhead fault-tolerant fpga systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6:212–221, 1998.

[17] 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 Transactions on Computers*, 43(8):880–891, 1994.

[18] K. Nikolic, A. Sadek, and M. Forshaw. Fault-tolerant techniques for nanocomputers. *Nanotechnology*, 13:357–362, 2002.

[19] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin, 1998.

[20] I. Parulkar, T. Ziaja, R. Pendurkar, A. D'Souza, and A. Majumdar. A scalable, low cost design-for-test architecture for UltaSPARC(TM) chip multi-processors. In *Proceedings of the International Test Conference*, October 2002.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, August 2001.

[23] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design (ICCD)*, October 2003.

[24] SIA. The international technology roadmap for semiconductors. http://public.itrs.net/Files/2003ITRS/Home2003.htm, 2003.

[25] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[26] R. Stanojevic, H. Balachandran, D. M. H. Walker, F. Lakhani, S. Jandyala, J. Saxena, and K. M. Butler. Computer-aided fault to defect mapping (CAFDM) for defect diagnosis. In *Proceedings of the IEEE International Test Conference*, October 2000.

[27] C. H. Stapper and R. J. Rosner. Integrated circuit yield management and yield analysis: Development and implementation. *IEEE Transactions on Semiconductor Manufacturing*, 8(2):95–102, 1995.

[28] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3-MB subarray-based third-level cache on an Itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1523–1529, 2002.