# StageNetSlice: A Reconfigurable Microarchitecture Building Block for Resilient CMP Systems

Shantanu Gupta Shuguang Feng Amin Ansari Jason Blome Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{shangupt, shoe, ansary, jblome, mahlke}@umich.edu

## ABSTRACT

Although CMOS feature size scaling has been the source of dramatic performance gains, it has lead to mounting reliability concerns due to increasing power densities and on-chip temperatures. Given that most wearout mechanisms that plague semiconductor devices are highly dependent on these parameters, significantly higher failure rates are projected for future technology generations. Traditional techniques for dealing with device failures have relied on coarse-grained redundancy to maintain service in the face of failed components. In this work, we challenge this practice by identifying its inability to scale to high failure rate scenarios and investigate the advantages of finer-grained configurations. We use this study to motivate the design of StageNet, an embedded CMP architecture designed from its inception with reliability as a first class design constraint. StageNet relies on a reconfigurable network of replicated processor pipeline stages to maximize the useful lifetime of the chip, gracefully degrading performance toward end of life. This paper addresses the microarchitecture of the basic building block of StageNet, named StageNetSlice, which is a processor core comprised of networked pipeline stages. A naive slice design results in approximately 4X slowdown verses a traditional processor due to longer communication delays in the pipeline. However, several small design changes that eliminate inter-stage communication paths and minimize communication bandwidth reduce this overhead to 11% on average while providing high levels of fine-grain adaptability.

## Categories and Subject Descriptors

B.8.1 [**Hardware**]: Reliability, Testing and Fault-Tolerance; C.1.0 [**Computer System Organization**]: Processor Architecture

## General Terms

Design, Reliability, Performance

## Keywords

Multicore, Reliability, Architecture, Pipeline

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*CASES'08,* October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-469-0/08/10 ...$5.00.

## 1. INTRODUCTION

Device scaling trends into the nanometer regime have lead to increasing current and power densities and rising on-chip temperatures, resulting in increasing device failure rates. Leading technology experts have begun to warn designers that device reliability will begin to deteriorate from the 65nm node onward [7]. Current projections indicate that future microprocessors will be composed of billions of transistors, many of which will be unusable at manufacture time, and many more which will degrade in performance (or even fail) over the expected lifetime of the processor [10]. To assuage these reliability concerns, computer designers must directly address reliability in computer systems through innovative fault-tolerance techniques.

The sources of computer system failures are widespread, ranging from transient faults, due to energetic particle strikes [40] and electrical noise [37], to permanent errors, caused by wearout phenomenon such as electromigration [13] and time dependent dielectric breakdown [39]. In recent years, industry designers and researchers have invested significant effort in building architectures resistant to transient faults and soft errors. Though there is significant evidence suggesting a growing rate of soft errors in future technology generations [10], this problem is actively being addressed in research [27, 27, 28, 38].

In contrast, much less attention has been paid to the problem of permanent faults, specifically transistor wearout due to the degradation of semiconductor materials over time. Concerns about wearout are primarily due to increasing power and current densities, both of which lead to increasing on-chip temperatures. All three of these parameters have been shown to heavily influence most wearout mechanisms [4]. In fact, most wearout mechanisms exhibit an exponential dependence on temperature [17] [13] [32]. Furthermore, device scaling increases the susceptibility to wearout by shrinking the thickness of the gate and inter-layer dielectrics and increasing interconnect current density. Traditional techniques for dealing with transistor wearout have involved extra provisioning in logic circuits, known as guard-banding, to account for the expected performance degradation of transistors over time. However, the increasing degradation rate projected for future technology generations implies that traditional margining techniques will be insufficient. This necessitates revolutionary new designs for systems that can identify and adapt to wearout through reconfiguration.

The challenge of tolerating permanent faults can be broadly divided into three requisite tasks: fault detection, fault diagnosis, and system recovery/reconfiguration. Fault detection mechanisms [8, 22, 5] are used to identify the presence of a fault, while fault diagnosis techniques [21, 16, 12] are used to determine the source and nature of the fault. System recovery can consist of a number

of different tasks, based on the nature of the fault. For example, if the fault is transient, the incorrect state may be corrected by simply flushing the processor pipeline [5]. However, if the fault is permanent, then a recovery mechanism which leverages system reconfiguration may be necessary to avoid propagating faults through the use of a failed component.

The third and the last task of system reconfiguration usually requires additional redundant resources, or the decommissioning of non-critical components. For example, many computer vendors provide the ability to repair faulty memory and cache cells, through the inclusion of spare memory elements [30]. Recently, researchers have begun to extend these techniques to support sparing for additional on-chip resources [33], such as branch predictors [11] and registers [26].

Classical mechanisms such as dual and triple-modular redundancy (DMR and TMR) have also been used to address the problem of system recovery [6, 30]. With the recent popularity of multicore systems these traditional core-level approaches have been able to leverage the inherent redundancy present in large chip multiprocessors (CMP) [29, 1, 34]. However, both the historical designs and their modern incarnations, because of their emphasis on core-level redundancy, incur high hardware overhead and can only tolerate a small number of defects. With the increasing defect rate in semiconductor technology, it will not be uncommon to see a rapid degradation in throughput for these systems as single device failures cause entire cores to be decommissioned (oftentimes with the majority of the core still intact and functional). While such solutions may be appropriate for mainframes and mission-critical systems, they are too costly in terms of area and power the embedded domain.

In contrast, this paper argues the case for reconfiguration and redundancy at a finer granularity. Providing finer-grain control enables isolation and/or replacement of individual structures within a processor core. This not only allows for more effective provisioning of spare parts (i.e., replicating only those components most susceptible to failure) but also minimizes waste by permitting a design, under the right circumstances, to salvage healthy components from dying cores. Over time as more and more devices fail, such a system will gracefully degrade its performance capabilities, maximizing its useful lifetime. The focus of this work is to propose an efficient solution for maintaining redundancy and enabling reconfiguration. The other requisite tasks of detecting and diagnosing the source of failure are assumed to be already in place and their discussion is beyond the scope of this paper.

This work presents *StageNetSlice* (SNS), the basic building block for *StageNet*, a highly reconfigurable and adaptable CMP computing substrate. Each SNS is a network of pipeline stages that collectively act as a logical processor. With flexible interconnects, an SNS can easily isolate failures by adaptively routing around faulty stages, either by leveraging spare components or in the case of a large CMP system time-multiplexing the same stage in an adjacent SNS. CMPs consisting of multiple SNS pipelines stitched together would possess inherent fine-grain redundancy and be capable of maintaining higher throughput over the duration of a system's life (even extending that lifetime) compared to conventional multicore designs. With a sea of pipeline stages at its disposal, an intelligent reliability management system can dynamically configure SNS-based CMPs to meet changing reliability and performance demands.

Conceptually, the design of an SNS is relatively straight-forward. However, introducing network switches into the heart of a processor pipeline will inevitably lead to poor performance due to high communication latencies and low communication bandwidth between stages. The key to creating an efficient SNS design is rethinking the organization of a basic processor pipeline to more effectively isolate the operation of individual stages. More specifically, inter-stage communication paths must either be removed, namely by breaking loops in the design, or the volume of data transmitted must be reduced. This paper presents the design of an efficient SNS that attacks these problems and reduces the reconfigurability overhead to an acceptable level. The primary contributions of this paper include:

- A design space exploration of reconfiguration granularities for resilient systems
- A highly adaptive microarchitecture (SNS) suitable for robust CMPs design
- An examination of the trade-offs and optimizations that helped realize the SNS design
- A detailed evaluation of SNS performance compared to a conventional embedded processor

# 2. RECONFIGURATION GRANULARITY

An architecture for tolerating permanent faults must have the ability to reconfigure, where reconfiguration can refer to a variety of activities ranging from decommissioning non-functioning, non-critical processor structures to swapping in cold spare devices. In a reconfigurable architecture, recovery entails isolating defective component(s) and incorporating spare structures as needed. Support for reconfiguration can be achieved at various levels of granularity, from ultra-fine grain systems that have the ability to replace individual logic gates to coarser designs that focus on isolating entire processor cores. This choice presents a trade-off between complexity of implementation and potential lifetime enhancement, where finer grain solutions provide greater lifetime extensions, at significantly more cost, than their coarser counterparts. Generally speaking, the law of diminishing returns places a lower limit on the granularity of reconfiguration. This section presents experiments studying this trade-off and draws upon these results to motivate the design of SNS.

## 2.1 Experimental Setup

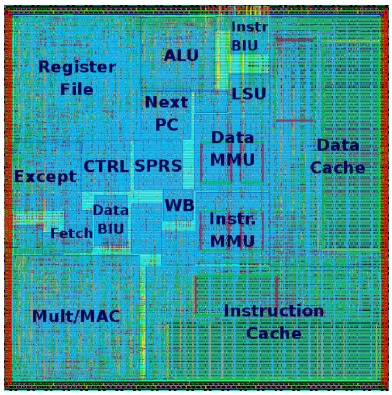
In order to effectively model the reliability of different designs, a Verilog model of the OpenRISC 1200 (OR1200) core [**?**] was used in lifetime reliability experiments. The OR1200 is an open-source core with a conventional 5-stage pipeline design, representative of commercially available embedded processors. The core was synthesized, placed and routed using industry standard CAD tools with a library characterized for a 130nm process. The final floorplan along with several attributes of the design is shown in Figure 1.

To study the impact of reconfiguration granularity on chip lifetimes the mean-time-to-failure (MTTF) was calculated for each individual module in the OR1200. MTTF was determined by estimating the effects of a common wearout mechanism, time-dependent-dielectric-breakdown (TDDB) on a OR1200 core running a representative workload.[1] Employing an empirical model similar to that found in [31], Equation 1 presents the formula used to calculate per-module MTTFs. More details about these calculations can be obtained from [9], which uses a similar experimental setup.

$$MTTF_{TDDB} \propto (\frac{1}{V})^{(a-bT)} e^{\frac{(X+\frac{Y}{T}+ZT)}{kT}} \quad (1)$$

where V = operating voltage, T = temperature, k = Boltzmann's constant, and a,b,X,Y,and Z are all fitting parameters based on [31].

[1]A similar analysis can be done for other wearout mechanisms including negative bias threshold inversion (NBTI), hot carrier injection (HCI) and electromigration (EM).




(a) Overlay of the OR1200 floorplan on top of the placed and routed implementation of the CPU core

| OR1200 Core | |
|---|---|
| Area | 1.28 $mm^2$ |
| Power | 93.4 mW |
| Clock Frequency | 200 MHz |
| Data Cache Size | 8 KB |
| Instruction Cache Size | 8 KB |
| Technology Node | 130 nm |

(b) Implementation details

Figure 1: OpenRisc 1200 embedded microprocessor.

For the purposes of this study, it is assumed that the fastest failing component in the design (the one with the smallest MTTF) determines the operational lifetime of the core. Using this MTTF data, the next subsection will discuss the advantages and disadvantages of reconfiguring hardware at different levels of granularity.

## 2.2 Granularity Tradeoffs

The granularity of reconfiguration is used to describe the unit of isolation for components within the processor core. Implicitly it also defines the granularity at which redundancy can be leveraged by the system. It is important to note that it is not strictly necessary to use cold spares to replace failed components, in certain situations the isolation of non-critical, faulty components suffices. Various options for reconfiguration, in order of increasing granularity, are as follows:

*Gate level:* At this level of reconfiguration, a system can replace individual logic gates in the design as they fail. Unfortunately, such designs are typically impractical because they require both precise fault diagnosis and tremendous overhead due to redundant components and wire routing area.

*Module level:* In this scenario, a processor core can replace/isolate broken microarchitectural structures such as an ALU or branch predictor. Such designs have been active topics of research [15, 26]. The biggest downside of this reconfiguration level is that maintaining redundancy for full coverage is almost impractical. Additionally, for the case of simple cores, even fewer opportunities exist for isolation since almost all modules are unique in the design.

*Stage level:* Here, the entire pipeline stages are treated as single monolithic units that can be replaced. Reconfiguration at this level is challenging because: 1) pipeline stages are tightly coupled with each other (reconfiguration can cause performance loss), and 2) cold sparing pipeline stages is expensive (area overhead).

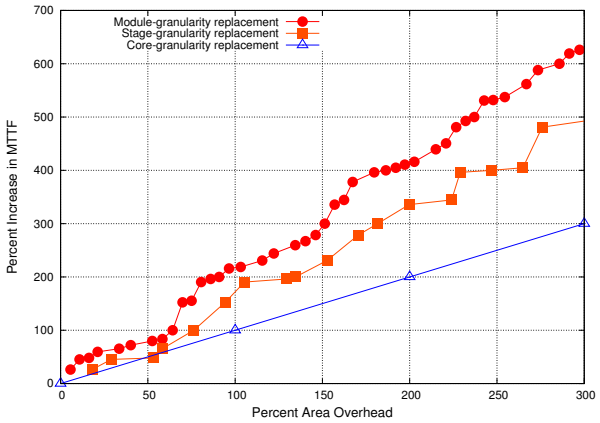



Figure 2: Gain in MTTF from the addition of cold spares at the granularity of micro-architectural modules, pipeline stages, and processor cores. The base system is a single core machine. The gains shown are cumulative, and spare modules are added in the order they are expected to fail (the markers indicate the times when a cold spare is added to the system). A higher slope indicates better returns on the area investment but at the same time involves more design complexity.

*Core level:* This is the coarsest level of reconfiguration where entire processor cores are isolated from the system in the event of a failure. Core level reconfiguration has also been an active area of research [34, 1], and from the perspective of a system designer, it is probably the easiest technique to implement. However, it has poorest returns in terms of lifetime extension, and therefore might not be able to keep up with the increasing defect rate.

Figure 2 demonstrates the effectiveness of the above granularities of reconfiguration (gate-level reconfiguration is not included in this study due to the complexity of implementation). MTTFs for individual modules in the OR1200 core were computed as described previously. Stage-level MTTFs were then defined as the minimum MTTF of any module belonging to the stage. Similarly core level MTTFs were defined as the minimum MTTF across all the modules. The figure shows the potential for lifetime enhancement as a function of how much area a designer is willing to allocate to cold spares. While this study evaluates processors with cold spares, it can also be viewed as investigating how efficiently the natural redundancy in a CMP can be exploited for reliability. The figure overlays three separate plots, one for each level of reconfiguration. The redundant spares were allowed to add as much as 300% area overhead.

The data shown in Figure 2 demonstrates that going towards finer-grain reconfiguration is categorically beneficial as far as gains in MTTF is concerned. But, it overlooks the design complexity aspect of the problem. Finer-grain reconfiguration tends to exacerbate the hardware challenges for supporting redundancy, e.g. muxing logic, wiring overhead, circuit timing management, etc. At the same time, very coarse grained reconfiguration is also not an ideal candidate since MTTF scales poorly with the area overhead. Therefore, a compromise solution is desirable, one that has manageable reconfiguration hardware and a better life expectancy.

Stage level reconfiguration is positioned as a good candidate because of the following reasons:

1. Stages provide clean natural boundaries. Logically stages are a convenient boundary because pipeline architectures divide work at the level of stages (like fetch, decode, etc.). Simi-

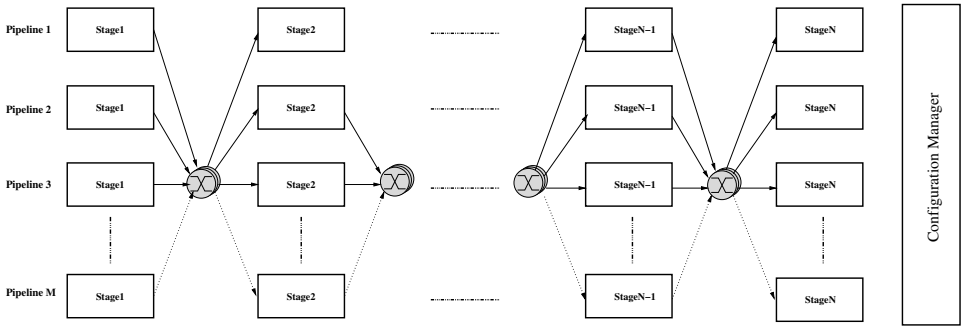



Figure 3: StageNet architecture. Each StageNet Slice (SNS) is equivalent to a logical processing core. This figure shows M, N-stage slices. More than one crossbar switches can be kept at the pipeline stage boundaries in order to tolerate rare, albeit possible, failure of the switch itself.

larly, in terms of circuit implementation stages are a intuitive boundary because data signals typically get latched at the end of every pipeline stage.

2. Stage-based reconfiguration scales well with the increase in available redundant spares (see Figure 2).
3. Lastly, in the vision of SNS-based CMPs, the ability to share stages between neighboring SNSs makes the system inherently redundant.

The next section introduces the SNS architecture, a scalable fault tolerant pipeline designed to allow stage level reconfiguration.

# 3. STAGENET: A RECONFIGURABLE MICROARCHITECTURE

As shown in the previous section, fine-grain reconfigurability at the pipeline stage granularity is an effective tool to combat future reliability challenges. One of the ways to allow reconfiguration at this granularity is to decouple the pipeline stages from each other. In other words, remove all direct point-to-point communication between the stages and replace them by a switch based interconnection network. A conceptual picture of a chip multiprocessor using this philosophy is presented in Figure 3. We call this design *StageNet* (SN). Processor cores within SN are designed as part of a high speed network-on-a-chip, where each stage in the processor pipeline corresponds to a node in the network. A horizontal slice of this architecture is equivalent to a logical processor core, and we call it a *StageNetSlice* (SNS). The use of switches allows complete flexibility for a pipeline stage at depth *X* to communicate with any stage at depth *X+1*, even those from a different SNS. Such a system can isolate nodes (pipeline stages) that are defective and configure pipelines to share (time-multiplex) certain stages. As nodes wearout and eventually fail, SN will exhibit a graceful degradation in performance, and a gradual decline in throughput.

The advantages of the SN architecture do not come for free, and there are certain overheads associated with this design, namely area and performance overheads. Area overhead arises from the switch interconnection network between stages. Depending upon the switch design, a variable number of cycles will be required to transmit operations between stages, leading to performance penalties.

## 3.1 SNS: Overview

A conventional embedded processor pipeline [2, 3, 23] usually consists of five stages namely, fetch, decode, issue, execute/memory, and writeback (see Figure 4a). Although the execute/memory block is sometimes separated into multiple stages, for the sake of simplicity (and generality) it is treated as a single stage in this work.

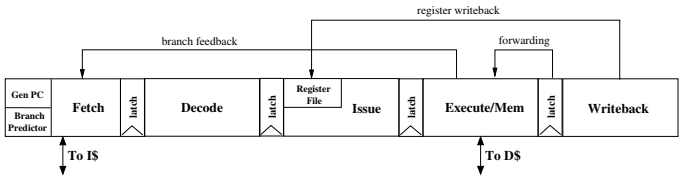
Starting with the basic pipeline design in Figure 4a, we will show how it's transformed into an SNS. We begin by replacing the pipeline latches with a combination of crossbar switches and buffers. A graphical illustration of such a pipeline design is shown in Figure 4b. For now, ignore the shaded boxes inside the pipeline stages, the significance of these portions will be addressed later in this section. To maximize performance, we propose the use of full crossbar switches [24] since a) these allow non-blocking access to all of their inputs and b) for a small number of inputs and outputs they are not prohibitively expensive. Furthermore, the channel width of the crossbar can be varied to trade off performance with area. In addition to the forward connections, feedback loops in the SN architecture also need to go through similar switches. This is because different SNSs can share their resources with each other and thus require an exchange of results. For instance, the result from, say SNS A's execute stage, might need to be directed to SNS B's issue stage for writeback. Due to the introduction of the crossbar switches, the SNS has three fundamental challenges to overcome:

*1. Global Communication:* Global pipeline stall/flush signals are fundamental to the functionality of a pipeline. Stall signals are sent to all the stages in the pipe for cases such as multi-cycle operations, memory access, etc. Similarly, flush signals are necessary to squash instructions that are fetched along mis-predicted control paths. In an SNS, all the stages are decoupled from each other, and global broadcast of a signal is infeasible.

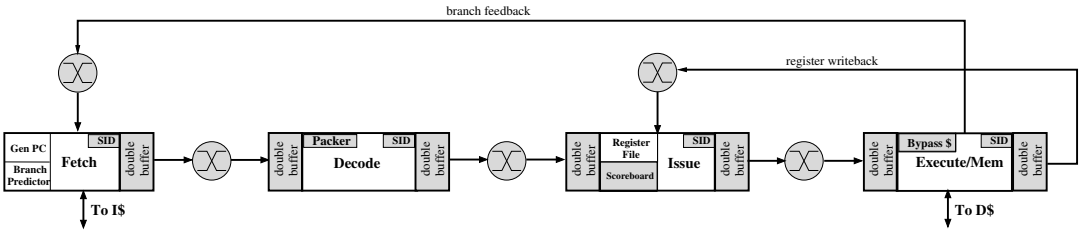
*2. Forwarding:* Data forwarding is a crucial technique used in a pipeline for avoiding frequent stalls that would otherwise occur because of data dependencies between consecutive instructions. The data forwarding logic relies on precisely timed (in an architectural sense) communication between execute and later stages using combinational links. With variable amounts of delay through the switches, and the presence of intermediate buffers, forwarding logic within an SNS is not feasible.

*3. Performance:* Lastly, even if the above two problems are solved, communication delay between stages is still expected to result in a hefty performance penalty.

The rest of this section will discuss how the SNS design overcomes these challenges and will also propose techniques that can recover lost performance.




(a) A simple 5-stage in-order pipeline




(b) The SNS pipeline. Stages are interconnected using a full crossbar switch. The shaded portions highlight modules that are not present in a regular pipeline.

Figure 4: Traditional in-order pipeline design and SNS.

## 3.2 SNS: Functional needs

### 3.2.1 Stream Identification

The basic SNS pipeline lacks global communication signals. Without global stall/flush signals traditional approaches to flushing instructions upon a branch mis-predict are not applicable. The first addition to the basic pipeline, a stream identification register, targets this problem.

The SNS design shown in Figure 4b has certain components that are shaded in order to distinguish the ones that are not found in a traditional pipeline. One of these additional components is a stream identification (`sid`) register in all the stages. This is a single bit register and can be arbitrarily (but consistently across stages) initialized to 0 or 1. Over the course of program execution this value changes whenever a branch mispredict takes place. Every in-flight instruction in a SNS carries a stream id, and this is used by the stages to distinguish the instructions on the correctly predicted path from those on the incorrect path. The former are processed and allowed to proceed, and the latter are squashed. A single bit suffices because the pipeline model is in-order and it can have only one resolved branch mispredict outstanding at any given time. All other instructions following this mispredicted branch can be squashed. In other words, the stream id works as a cheap and efficient mechanism to replace the global branch mis-predict signal. The details of how and when the `sid` register value is modified are discussed below on a stage-by-stage basis:

*Fetch:* Every new instruction is stamped with the current value stored in the `sid` register. When a branch mis-predict is detected by fetch (using the branch update from execute/memory stage) it toggles the `sid` register and flushes the program counter. From this point onwards, the instructions fetched are stamped with the updated stream id.

*Decode:* The `sid` register is updated from the stream ids of the incoming instructions. If at any cycle, the old stream id stored in the decode does not match the stream id of an incoming instruction, a branch mispredict is detected and decode flushes the instruction buffer.

*Issue:* It maintains the `sid` register along with an addition 1-bit `last-sid` register. The `sid` register is updated by the stream id of the instruction that performs register writeback. And the `last-sid` value is updated from the stream id of the last successfully issued instruction. For an instruction reaching the issue stage, its stream id is compared with the `sid` register. If the values match, then it is eligible for issue. A mismatch implies that some branch was mispredicted, in the recent past, and further knowledge is required to determine whether this new incoming instruction is on the correct path or the incorrect path. This is where the `last-sid` register gathers importance. A mismatch of the new instruction's stream id with the `last-sid` indicates that the new instruction is on the corrected path of execution and hence it is eligible for issue. A match implies the otherwise and the new instruction is squashed. Complete significance of `last-sid` will be made clear in the next sub-section.

*Execute/Memory:* It compares the stream id of the incoming instructions to the `sid` register. In the event of a mismatch, the instruction is squashed, otherwise the instruction is executed. A mispredicted branch instruction toggles the `sid` register value stored in this stage along with its own stream id. This branch resolution information is sent back to the fetch stage, initiating a change in the `sid` register value stored there. The mispredicted branch instruction also updates the `sid` in the issue stage during the writeback. Thus, the cycle of updates is completed.

To summarize, under normal operating conditions (i.e. no mispredicts), instructions go through the switched interconnection fabric, get issued, executed and write back computed results. When a mispredict occurs, using the stream id mechanism, instructions on the incorrect execution path can be systematically squashed in time.

### 3.2.2 Scoreboard

The second important component required for proper functionality of the SNS is a *scoreboard* that resides in the issue stage. A scoreboard is essential in this design because a forwarding unit (that normally handles register value dependencies) is not feasible. More often than not, a scoreboard is already present in a pipeline's issue stage for hazard detection. In such a scenario, only minor modifications are needed to tailor a conventional scoreboard to the needs of an SNS pipeline.

The SNS pipeline needs a scoreboard in order to keep track of the registers that have results outstanding and are therefore invalid in the register file. Instructions for which one or more input registers are invalid can be stalled in the issue stage. The SNS scoreboard table has two columns (see Figure 6c), one to maintain a `valid`

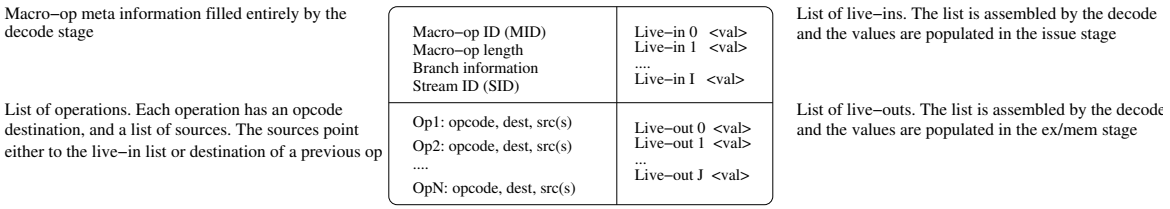



Figure 5: Structure of a macro-op.

bit for a register, and a second to store the `id` of the last modifying instruction. In case of a branch mis-predict, the scoreboard needs to be wiped clean since it gets updated by instructions on the wrong path of execution. To recognize a mis-predict, the issue stage maintains a `last-sid` register that stores the stream id of the last issued instruction. Whenever the issue stage finds out that the new incoming instruction's stream id differs from `last-sid`, it knows that a branch mis-predict has taken place. At this point, the scoreboard waits to receive the writeback, if it hasn't received it already, for the branch instruction that was the cause of the mis-predict. Finally, after this waiting period, the scoreboard is cleared and the new instruction is issued.

### *3.2.3 Network Flow Issues*

In a SNS, the stalls are automatically handled by maintaining network back pressure through the switched interconnection. A crossbar does not forward values to the buffer of a subsequent stage if the stage is stalled. This is similar to the way network queues handle stalls. In our implementation, we guarantee that an instruction is never dropped (thrown away) by a buffer.

For a producer consumer based system, where the transfer latency is variable, double buffering is a standard technique used to make the transfer latency overlap with the job cycles of a producer / consumer. In a SNS, all stages have their input and output latches double buffered to enable this optimization.

## 3.3 SNS: Performance Enhancement

The functional additions to the SNS discussed in the previous section brings the design to a point where it is functionally correct. However, in its present state, much is left on the table in terms of performance. This section introduces techniques to address this shortcoming.

### *3.3.1 Bypass cache*

Due to the lack of forwarding logic in an SNS, frequent stalls are expected for instructions with register dependencies. To alleviate the performance loss we add a *bypass cache* in the execute/memory stage (see Figure 6d). The job of this cache is to store values generated by recently executed instructions within the execute/memory stage. The instructions that follow, can use these cached values and need not stall in the issue. In fact, if the cache is large enough, results from every instruction that has been issued, but has not written back, can be retained. This would completely eliminate the stalls arising from register dependencies. In this manner, the bypass cache emulates the work of forwarding logic.

A FIFO replacement policy is used for this cache because older instructions are less likely to have produced a result for an incoming instruction. The scoreboard unit of the issue stage is made aware of the bypass cache size when the system is first configured. As discussed earlier, the scoreboard maintains the *id* of the last instruction which wrote to a particular register. Knowing the size of the bypass cache, and the absolute difference between the *id* of a given instruction *X* and the *id(s)* of the last writing instruction(s) for input register(s) of *X*, the scoreboard can decide whether to issue or stall. If this difference is less than the depth of bypass cache, then the required input register is guaranteed to be present in the bypass cache (assuming all instructions have a maximum of 1 destination). Therefore, the instruction can be issued when all the outstanding input dependencies are guaranteed to be present in the bypass cache, and stalled otherwise. Furthermore, the issue stage can also perform selective register operand fetch for only those values that are not going to be available in the bypass cache. By this optimization, we can reduce the number of bits that are transferred from the issue stage to the execute/memory stage.

### *3.3.2 Macro Operations*

The performance of the SNS design suffers significantly from the overhead of transferring instructions between stages, since every instruction has to go through a switched network with a variable amount of delay. For instance, if an instruction takes three cycles for transfer between the stages, and one cycle to execute, the CPI of the system (with perfect cache, assuming no branches) would be 4. Another way to look at this is that since each of the stages usually takes a single cycle to execute the instruction, they are idle the majority of the time waiting for instructions to arrive. A natural optimization would be to increase the granularity of communication to a bundle of multiple instructions/operations (*macro-op*). There are two advantages of doing this:

1. More work (multiple instructions) is available for the stages to work on while waiting for the next macro-op to arrive.

2. This can eliminate the temporary intermediate values generated within small sequences of instructions, and therefore give an illusion of data compression to the underlying interconnection fabric.

This collection of operations can be done both statically (at compile time) or dynamically (in the hardware). To keep the overall hardware overhead low, we form these statically in the compiler. Our approach involves selecting a subset of instructions belonging to a basic block, while keeping two sets of constraints: 1) the number of live-ins and live-outs of the macro-op, and 2) the number of instructions. We use a simple greedy policy, similar to [14], that maximizes the number of instructions, while minimizing the number of live-ins and live-outs. The ideal macro-op is one where the computational cost is roughly equivalent to the communication costs. Effectively, the idea is to pipeline the transfer with the computation.

The complete structure of a macro-op is shown in the Figure 5. The compiler embeds the macro-op boundaries, internal data flow, and live-in/live-out information in the program binary. During runtime, decode's *Packer* structure is responsible for identifying and assembling macro-ops. Leveraging hints for the macro-op boundaries that are embedded in the program binary, the *Packer* assigns

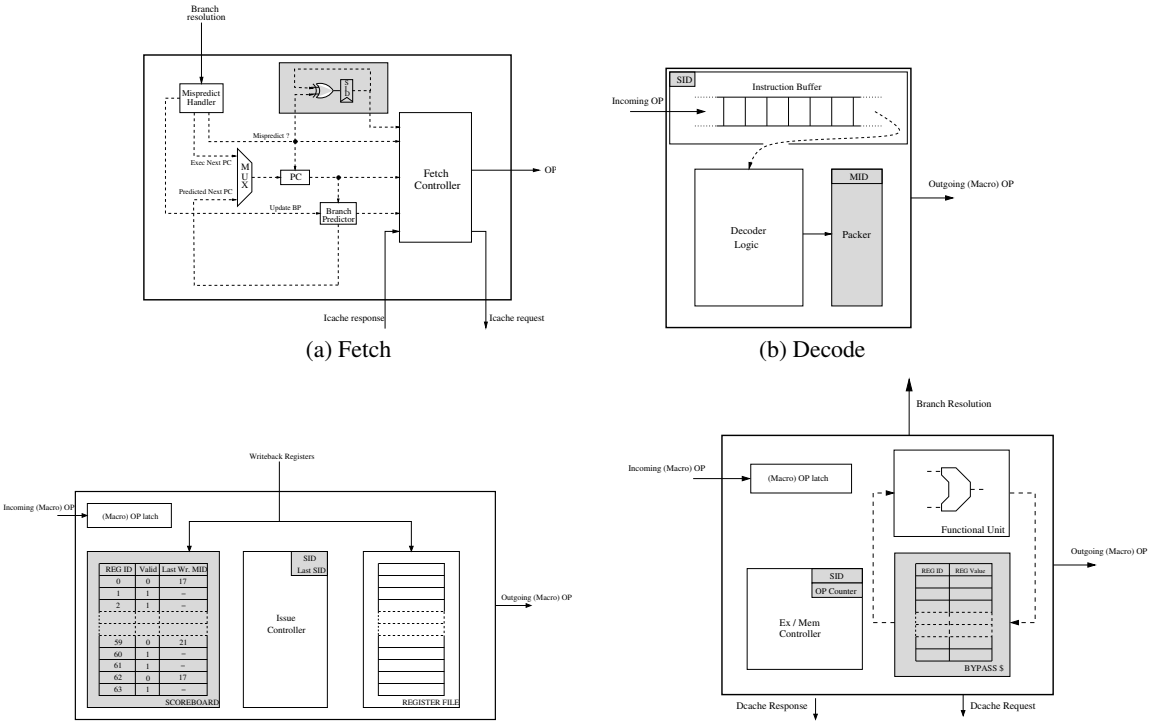



Figure 6: Pipeline stages of the SNS. Gray blocks highlight the modules added for transforming a traditional pipeline into a SNS.

a unique *macro-op ID* (MID) to every macro-op flowing through the pipeline. All other stages in the SNS are also modified to work with these macro-ops instead of simple instructions. This is particularly true of the execute/memory stage where a controller cycles across the individual instructions that comprise a macro-op, executing them in sequence. However, the bandwidth of the stages is not modified, and they continue to process one instruction per cycle. This implies that register file ports, execution units, memory ports etc. are not increased in their number or capability. The design overhead primarily arises from the new structures that we add to every stage as elaborated in the next subsection.

## 3.4 SNS: Stage Modifications

This section goes over the pipeline stages in the SNS, and summarizes the modules added to each of them.

**Fetch:** The modifications made in the fetch stage (Figure 6a) are restricted to the addition of an `sid` register and a small amount of logic to toggle it upon branch mis-predicts.

**Decode:** Decode stage (Figure 6b) collects the fetched instructions in an instruction buffer. An instruction buffer is a common structure found in most pipeline designs, and to that we add an `sid` register. For an incoming instruction with a different stream id, this register is toggled and the instruction buffer is flushed. The decode stage is also augmented with the *Packer*. The Packer logic reads instructions from the buffer, identifies the macro-op boundaries, assigns them a macro-op id, and fills out the macro-op structure attributes.

**Issue:** Issue stage (Figure 6c) is modified to include a Scoreboard that tracks register dependencies. For a macro-op that is ready for issue, the register file is read to populated the live-ins. The issue stage also maintains two 1-bit registers: `sid` and `last-sid`, in order to identify branch mis-predicts and flush the Scoreboard at appropriate times.

**Execute/Memory:** Execute/Memory stage (Figure 6d) houses the bypass cache that partially emulates the forwarding logic. This stage is also the first to update its `sid` register upon a branch mis-predict. In order to handle macro-op execution, the execute/memory controller is modified to go over the macro-op instructions one at a time. While doing this, the computed results are saved into the bypass cache for later use.

# 4. RESULTS AND DISCUSSION

## 4.1 Evaluation Infrastructure

The evaluation infrastructure for the SNS design consisted of two major components, 1) a compilation framework and 2) an architectural simulator. Ten benchmarks were selected from a variety of domains to evaluate SNS under differing conditions, including media kernels (idct, sobel), encryption (3des, rijndael, rc4, pc1), audio encoding (rawcaudio, g721encode), and audio decoding (rawdaudio, g721decode).

We use the Trimaran compilation system [35] as our first component. The macro-op selection algorithm is implemented as a compiler pass on the intermediate code representation. During this pass, the code is augmented with the macro-op boundaries and other miscellaneous attributes (Figure 5). The final code generated by the compiler uses the HPL-PD ISA [19].

The architectural simulator was developed using the Liberty Simulation Environment (LSE) [36]. A functional emulator was also developed for the HPL-PD ISA within the LSE system. Two flavors of the microarchitectural simulator were implemented in sufficient detail to provide cycle accurate results. The first simulator modeled a simple five stage pipeline, which is also the baseline for our

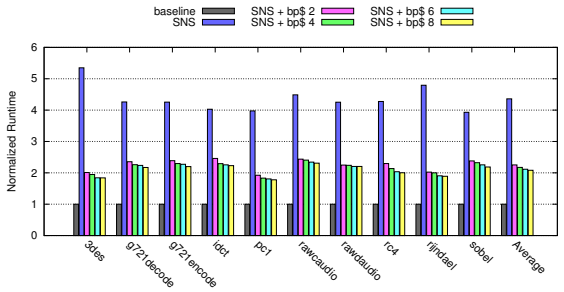



Figure 7: SNS pipeline, with a bypass cache, compared to the baseline pipeline. The slowdown, seen because of the issue stage stalls, reduces as the size of bypass cache is increased.

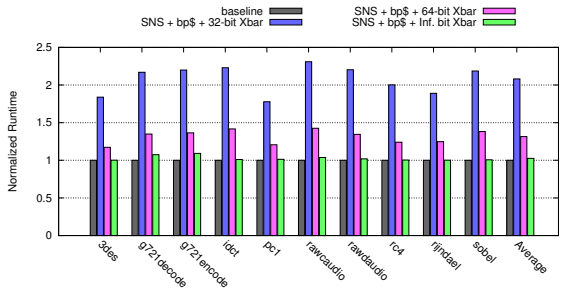



Figure 8: SNS performance with variations in the crossbar channel width. A 64-bit crossbar was found to present a good enough trade-off between the area and the performance.

experiments. The second simulator implemented the SNS architecture with all the proposed enhancements. Table 1 lists the common attributes for our architectural simulations.

| | |
|---|---|
| Base pipeline | 5-stage in-order |
| SNS | 4-stage in-order, with double buffering and 32/64 bit 5×5 crossbar switches |
| Branch predictor | global, 16-bit history, gshare predictor |
| Level 1 I/D cache | 4-way, 16 KB, 1 cycle hit latency |
| Level 2 unified cache | 8-way, 64 KB, 5 cycle hit latency |
| Memory | 40 cycle latency |

Table 1: Architectural attributes

## 4.2 Simulation Results

To evaluate SNS, we present the baseline performance results followed by successive addition of features to increase efficiency.

**Base SNS performance:** The performance of a *basic* SNS in comparison to the baseline (bars 1 and 2) is shown in Figure 7. *Basic* here implies an SNS configured with the stream identification logic, scoreboard and double buffering. The macro-op size is kept fixed at 1 and there is no bypass cache. The results are normalized to the runtime of the baseline processor. An average slowdown of over 4X was seen, which is a significant price to pay for the reconfiguration flexibility.

**SNS with bypass cache:** The addition of the bypass cache results in drastic improvements in the overall performance of the SNS (Figure 7). For this experiment, macro-op size is kept fixed at 1, and the bypass cache depth is varied from 2 to 8. The depth of the bypass cache indicates the number of destinations that it can store. In Figure 7, for bypass cache sizes beyond 6, there is hardly any performance improvement. This follows from the fact that as the bypass cache size is increased, almost all the outstanding register values get cached into it, and the performance returns are progressively diminished. The average slowdown hovers around 2.1X with the addition of the bypass cache, which is still quite high.

**Varying crossbar channel width:** The channel width of the crossbar determines the number of cycles it takes to transfer an instruction between two stages. A bigger value is going to improve performance but at the same time present a higher area overhead. Figure 8 illustrates the impact of varying the crossbar channel width on the performance. Three data points are presented for every benchmark: 32-bit channel width, 64-bit channel width, and infinite channel width (added to show the maximum potential gain in the performance). A crossbar with 64-bit channel width is used

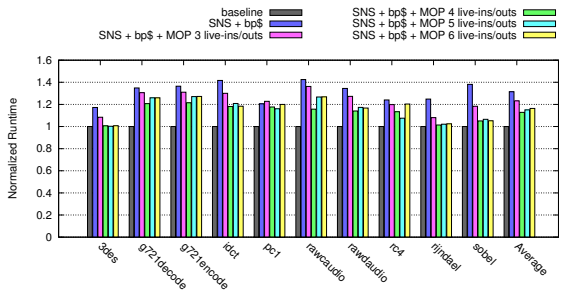



Figure 9: SNS pipeline, with a bypass cache and the capability to handle macro-ops, compared to the baseline in-order pipeline. An optimal resource constraint, which varied between benchmarks, was found to give the best performance.

for all subsequent experiments. The average slowdown of 1.3X was seen with the 64-bit crossbar. This might still be considered a costly trade for the reconfiguration flexibility achieved.

**SNS with bypass cache and macro-ops:** The performance results shown in Figure 9 are for an SNS with the bypass cache and the statically selected bundles of instructions (MOPs). The bypass cache size (6) is large enough to accommodate most of the outstanding values. Each bar in the plot (except 1 and 2) is for a different configuration of the MOP selection algorithm. The results show that beyond a certain limit, relaxing the macro-op selection constraints (live-ins and live-outs) does not result in performance improvement. Prior to reaching this limit, relaxing constraints helps in forming longer macro-ops, thereby balancing transfer time with computation time. Beyond this limit, relaxing constraints does not result in longer macro-ops, merely wider macro-ops (more live-ins/outs). This increases transfer time without actually increasing the number of distinct computations that are encoded.

The best result, an average slowdown of 11%, was achieved for the live-in/live-out constraint of 4 registers. The benchmarks with poor branch prediction rates performed the worst, see Figure 10. In fact, performance is strongly correlated with the number of mispredicts per thousand instructions. This is expected because the use of macro-ops, and the additional cycles spent for data transfer between stages, causes the SNS design to behave like a very deep pipeline. Future work will look into solutions that could eliminate hard-to-predict branches from the code by using if-conversion or similar schemes.

**Area overhead:** The area overhead in the SNS design arises from the additional microarchitectural structures that were added

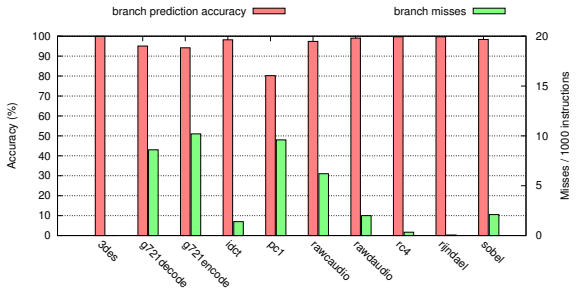



Figure 10: Branch predictor accuracy and number of misses per thousand instructions.

and the interconnection fabric composed of crossbar switches. Area overheads are shown using OR1200 core as the baseline (see Section 2.1) in Figure 4.2. The area for the scoreboard and bypass cache is estimated by taking similar structures from the OR1200 core and resizing them appropriately. The scoreboard area estimate is based on the register file. And the bypass cache is based on the TLB, as it requires associative look-up. Finally, the area of double buffers is based on their length and the maximum macro-op size they have to store. The sizing of all these structures was done according to the SNS configuration that achieved the best performance. The crossbar switch area is based on the Verilog model from [24]. The interconnection area for connecting the pipeline stages to the crossbar switches is not discussed here since it is challenging to estimate that without a complete VLSI layout. Furthermore, the area of a 64-bit bus link is expected to be an order of magnitude smaller than that of a 5×5 64-bit crossbar fabric.

All the design blocks were synthesized, placed and routed using industry standard CAD tools with a library characterized for a 130nm process. The overall area overhead is approximately 10.1% for the SNS modules, and 2.1% for each of the 64-bit 5×5 crossbar switches. An SNS has a total of five such crossbars, making the total area overhead 20%. However, the actual area overhead will be slightly lower because 1) this is a conservative estimate for the modules, and 2) the crossbar switches are shared resources in a SN CMP system. If a five SNS StageNet CMP is constructed, each SNS will see an area overhead of 12%.

# 5. RELATED WORK

Concern over reliability issues in future technology generations has spawned a new wave of research in reliability-aware microarchitectures. Recent work has addressed the entire spectrum of reliability topics, from fault detection and diagnosis to system repair and recovery. The following section focuses on the most relevant subset of recent work, those that propose architectures that tolerate and/or adapt to the presence of faults.

High-end server systems designed with reliability as a first-order design constraint have been around for decades but have typically relied on coarse grain replication to provide a high degree of reliability [6, 30]. However, dual and triple modular redundant systems incur significant overheads in terms of area and power. Furthermore, these systems still remain susceptible to wearout-induced failures since TMR is based on the premise that faults are unlikely to impact redundant modules simultaneously, a condition that is not obeyed given that all the redundant components are executing identical workloads and experiencing similar aging effects. Without additional cold-spare devices, traditional TMR, although effective at detecting faults, cannot significantly extend the lifetime of a processor. Given the cost, energy, and complexity overhead of these approaches, they are not appropriate for low-end server, desktop, or embedded computer systems.

ElastIC [34] is a slightly different high-level architectural vision for multiprocessor fault tolerance. Exploiting low-level circuit sensors for monitoring the health of individual cores, the authors propose dynamic reliability management that can throttle and eventually turn off cores as they age over time. Yet as with TMR, ElastIC relies upon redundancy management at the core level similar to the Configurable Isolation [1]. Within such a framework, the system must be provisioned with a massive number of redundant cores or face the prospect of rapidly declining processing throughput as single faults disable entire cores.

Much work has also been done in fine-grained redundancy maintenance such as Bulletproof [15], sparing in array structures [11], and other such microarchitectural structures [26]. These schemes typically rely on the inherent redundancy of superscalar cores, and can be difficult to leverage for full coverage. Nevertheless, ideas from such approaches can be incorporated in our system for additional benefits since they too apply redundancy management at a finer granularity.

Other research has focused on building reliable substrates out of future nanotechnologies which are expected to be inherently fault-prone. The NanoBox Processor Grid [20] was designed as a recursive system of black box devices, each employing their own unique fault tolerance mechanisms. While this project does boast a significant amount of defect tolerance, it comes at a 9x overhead in terms of redundant structures.

SNS differs dramatically from solutions previously proposed in that our goal is to minimize the amount of hardware used solely for redundancy. More importantly we enable reconfiguration at the granularity of a pipeline stage, making it possible for a single core to tolerate multiple failures at a much lower cost. The work here is an extension of [18] where we explored the potential of pipeline stage level reconfigurability. In parallel to our efforts, Romanescu et al. [25] have proposed a multicore architecture, named as Core Cannibalization Architecture (CCA), that also exploits stage level reconfigurability. CCA allows only a subset of pipelines to lend their stages to other broken pipelines, thereby avoiding full crossbar interconnection. Unlike SNS, CCA pipelines maintain all feedback links and avoid any major changes to the microarchitecture. Although these design choices reduce the overall complexity, fewer opportunities of reconfiguration exist for CCA as compared to a SNS based CMP.

# 6. CONCLUSION

As CMOS technology continues to evolve so too must the techniques that are employed to counter the effects of ever more demanding reliability challenges. Efforts in fault detection, diagnosis, and recovery/reconfiguration must all be leveraged together to form a comprehensive solution to the problem of unreliable silicon. This work contributes to the area of recovery and reconfiguration by proposing a radical architectural shift in processor design. Motivated by the need for finer-grain reconfiguration, networked pipeline stages were identified as an effective trade-off between cost and reliability enhancement. Although performance suffered at first as a result of the dramatic changes to the basic pipeline, a few well-placed microarchitectural enhancements were able to reclaim much of what was lost. Ultimately the SNS design exchanged a modest amount of performance (11%) and area (12%) overhead in return for a highly adaptable pipeline robust enough to withstand the rapidly increasing device failure rates expected in future technology nodes.

| Router width | Area ($mm^2$) | Percent overhead |
|---|---|---|
| 32 bits | 0.0158 | 1.2% |
| 64 bits | 0.028 | 2.1% |

(a) Router area for different channel width

| Design block | Area ($mm^2$) | Percent overhead |
|---|---|---|
| Scoreboard | 0.018 | 1.4% |
| Bypass cache | 0.044 | 3.4% |
| Buffers | 0.067 | 5.3% |

(b) Area for SNS specific modules

Figure 11: Area overhead for SNS modules and the crossbar switch

# 7. ACKNOWLEDGEMENTS

We graciously thank David Penry for his many answers to questions on the Liberty Simulation Environment and Visvesh Sathe for his help with Cadence Encounter. Also, our gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Ltd., the National Science Foundation grant CCF-0347411, and the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. We also kindly thank Hewlett-Packard and Intel Corporation for the equipment used to carry out this research.

# 8. REFERENCES

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 470–481, 2007.

[2] ARM. Arm11. http://www.arm.com/products/CPUs/families/ARM11Family.html.

[3] ARM. Arm9. http://www.arm.com/products/CPUs/families/ARM9Family.html.

[4] J. S. S. T. Association. Failure mechanisms and models for semiconductor devices. Technical Report JEP122C, JEDEC Solid State Technology Association, Mar. 2006.

[5] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999.

[6] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[7] K. Bernstein. Nano-meter scale cmos devices (tutorial presentation), 2004.

[8] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 109–120, 2007.

[9] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke. Online timing analysis for wearout detection. In *Proc. of the 2nd Workshop on Architectural Reliability*, pages 51–60, 2006.

[10] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[11] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, page 51, 2004.

[12] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 197–208, 2005.

[13] A. Christou. *Electromigration and Electronic Device Degradation*. John Wiley and Sons, Inc., 1994.

[14] N. Clark et al. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–157, Oct. 2006.

[15] K. Constantinides et al. Bulletproof: A defect-tolerant CMP switch architecture. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 3–14, Feb. 2006.

[16] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 97–108, 2008.

[17] D. Dumin. *Oxide Reliability: A Summary of Silicon Oxide Wearout, Breakdown, and Reliability*. World Scientific Publishing Co. Pte. Ltd., 2002.

[18] S. Gupta, S. Feng, J. Blome, and S. Mahlke. Stagenet: A reconfigurable cmp fabric for resilient systems. In *Proc. of the 2nd Reconfigurable and Adaptive Architecture Workshop*, 2007.

[19] V. Kathail, M. Schlansker, and B. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, Feb. 2000.

[20] A. KleinOsowski et al. The recursive nanobox processor grid: A reliable system architecture for unreliable nanotechnology devices. In *International Conference on Dependable Systems and Networks*, page 167, June 2004.

[21] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proc. of the 2008 International Conference on Dependable Systems and Networks*, June 2008.

[22] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[23] OpenCores. OpenRISC 1200, 2006. http://www.opencores.org/projects.cgi/web/ or1k/openrisc_1200.

[24] L.-S. Peh and W. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 255–266, Jan. 2001.

[25] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processor in the presence of hard faults. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[26] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of the 2003 International Conference on Computer Design*, page 481, Oct. 2003.

[27] D. Siewiorek et al. *Reliable Computer Systems: Design and Evaluation, 3rd Edition*. AK Peters, Ltd., 1998.

[28] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 256–268, Dec. 2004.

[29] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 223–234, 2006.

[30] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999.

[31] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 276–287, June 2004.

[32] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, pages 177–186, June 2004.

[33] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 520–531, June 2005.

[34] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Journal of Design and Test*, 23(6):484–490, 2006.

[35] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[36] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*, 24(3):211–249, 2006.

[37] S. Vrudhula, D. Blaauw, and S. Sirichotiyakul. Estimation of the likelihood of capacitive coupling noise. In *Proc. of the 39th Design Automation Conference*, pages 653–658, 2002.

[38] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *Proc. of the 2001 International Conference on Dependable Systems and Networks*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[39] E. Wu et al. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides. *Solid-State Electronics*, 46:1787–1798, 2002.

[40] J. Zeigler. Terrestrial cosmic ray intensities. *IBM Journal of Research and Development*, 42(1):117–139, 1998.