StageNet: A Reconfigurable Fabric for Constructing Dependable CMPs

Shantanu Gupta, Shuguang Feng,
Amin Ansari, *Student Member*, *IEEE*, and Scott Mahlke, *Member*, *IEEE*

Abstract—CMOS scaling has long been a source of dramatic performance gains. However, semiconductor feature size reduction has resulted in increasing levels of operating temperatures and current densities. Given that most wearout mechanisms are highly dependent on these parameters, significantly higher failure rates are projected for future technology generations. Consequently, fault tolerance, which has traditionally been a subject of interest for high-end server markets, is now getting emphasis in the mainstream computing systems space. The popular solution for this has been the use of redundancy at a coarse granularity, such as dual/triple modular redundancy. In this work, we challenge the practice of coarse-granularity redundancy by identifying its inability to scale to high failure rate scenarios and investigating the advantages of finer-grained configurations. To this end, this paper presents and evaluates a highly reconfigurable CMP architecture, named as StageNet (SN), that is designed with reliability as its first-class design criteria. SN relies on a reconfigurable network of replicated processor pipeline stages to maximize the useful lifetime of a chip, gracefully degrading performance toward the end of life. Our results show that the proposed SN architecture can perform 40 percent more cumulative work compared to a traditional CMP over 12 years of its lifetime.

Index Terms—Reliability, fault tolerance, multicore, CMP, wearout.

1 Introduction

Technological trends into the nanometer regime have lead to increasing current and power densities and rising on-chip temperatures, resulting in both increasing transient, as well as permanent failures rates. Leading technology experts have warned designers that device reliability will begin to deteriorate in future technology nodes [1]. 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 the performance (or even fail) over the expected lifetime of the processor [2]. In an effort to assuage power density concerns, industry has initiated a shift toward multi/many-core chips with simpler cores to limit their power and thermal envelope [3]. However, this paradigm shift also leads toward core designs that have little inherent redundancy and are, therefore, incapable of performing the self-repair possible in big superscalar cores [4]. Thus, in the near future, architects must directly address reliability in computer systems through innovative fault-tolerant techniques.

The sources of computer system failures are widespread, ranging from transient faults, due to energetic particle strikes [5] and electrical noise, to permanent errors, caused by wearout phenomenon such as electromigration [6] and time-dependent dielectric breakdown [7]. In recent years, industry designers and researchers have invested significant effort in building architectures resistant to transient faults [8]. 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. 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.

The challenge of tolerating permanent faults can be broadly divided into three requisite tasks: fault detection, fault diagnosis, and system reconfiguration/recovery. Fault detection mechanisms [9], [10] are used to identify the presence of a fault, while fault diagnosis techniques [11] are used to determine the source of the fault, i.e., the broken component(s). System reconfiguration needs to leverage some form of a spatial or temporal redundancy to keep the faulty component isolated from the design. As an example, many computer vendors provide the ability to repair faulty memory and cache cells through the inclusion of spare memory elements. Recently, researchers have begun to extend these techniques to support sparing for additional on-chip resources, such as branch predictors [12] and registers [4]. The granularity at which spares/redundancy is maintained determines the number of failures a system can tolerate. The focus of this work is to understand the issues associated with system reconfiguration and to design a fault-tolerant architecture that is capable of tolerating a large number of failures.

Traditionally, system reconfiguration in high-end servers and mission critical systems has been addressed by using mechanisms such as dual- and triple-modular redundancy (DMR and TMR) [13]. With the recent popularity of multicore systems, these traditional core-level approaches

• *The authors are with the University of Michigan, 2260 Hayward St, 4861 CSE Building, Ann Arbor, MI 48105.*
E-mail: {shangupt, shoe, ansary, mahlke}@umich.edu.

Manuscript received 25 Jan. 2010; revised 4 June 2010; accepted 11 Aug. 2010; published online 6 Oct. 2010.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2010-01-0058.
Digital Object Identifier no. 10.1109/TC.2010.205.

have been able to leverage the inherent redundancy present in large chip multiprocessors (CMPs) [14], [15]. 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, often times with the majority of the core still intact and functional.

In contrast, this paper argues the case for reconfiguration and redundancy at a finer granularity. To this end, this work presents the *StageNet* (SN) fabric, a highly reconfigurable and adaptable computing substrate. SN is a multicore architecture designed as a network of pipeline stages, rather than isolated cores in a CMP. The network is formed by replacing the direct connections at each pipeline stage boundary by a crossbar switch interconnection. Within the SN architecture, pipeline stages can be selected from the pool of available stages to act as logical processing cores. A logical core in the StageNet architecture is referred to as a *StageNetSlice* (SNS). An SNS can easily isolate failures by adaptively routing around faulty stages. The interconnection flexibility in the system allows SNSs to salvage healthy stages from adjacent cores and even makes it possible for different SNSs to time-multiplex a scarce pipeline resource. Because of this added flexibility, an SN system possesses inherent redundancy (through borrowing and sharing pipeline stages) and is, therefore, all else being equal, capable of maintaining higher throughput over the duration of a system's life compared to a conventional multicore design. Over time as more and more devices fail, such a system can gracefully degrade its performance capabilities, maximizing its useful lifetime.

The reconfiguration flexibility of the SN architecture has a cost associated with it. The introduction of 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 SN design is rethinking the organization of a basic processor pipeline to more effectively isolate the operation of individual stages. More specifically, interstage communication paths must either be removed, namely, by breaking loops in the design, or the volume of data transmitted must be reduced. This paper starts off with the design of an efficient SNS (a logical StageNet core) that attacks these problems and reduces the performance overhead from network switches to an acceptable level. Further, it presents the SN multicore that stitches together multiple such SNSs to form a highly reconfigurable architecture capable of tolerating a large number of failures. In this work, we take an in-order core design as the basis of the SN architecture to provide a proof of concept. The choice of an in-order is also motivated by the fact that thermal and power considerations are pushing designs toward simpler cores. In fact, simple cores have already been adopted by designs targeting massively multicore chips for low latency and high throughput applications, e.g., Sun UltraSparc T1/T2 [3], Tilera [16], and Intel Larrabee [17].

The contributions of this paper include:

1. a design space exploration of reconfiguration granularities for resilient systems;
2. design, evaluation, and performance optimization of StageNetSlice, a networked pipeline microarchitecture;
3. design and evaluation of StageNet, a resilient multicore architecture, composed using multiple SNSs; and
4. scalability analysis and system design of a large-scale StageNet chip.

2 Reconfiguration Granularity

For tolerating permanent faults, architectures must have the ability to reconfigure, where reconfiguration can refer to a variety of activities ranging from decommissioning nonfunctioning, noncritical processor structures to swapping in cold spare devices. Support for reconfiguration can be achieved at various granularities from ultrafine 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. This section shows experiments studying this trade-off and draws upon these results to motivate the SN architecture.

2.1 Experimental Setup

In order to effectively model the reliability of different designs, a Verilog model of the OpenRISC 1200 (OR1,200) core [18] was used in lifetime reliability experiments. The OR1200 is an open-source core with a conventional four-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 90 nm process. The final floorplan along with several attributes of the design is shown in Fig. 1.

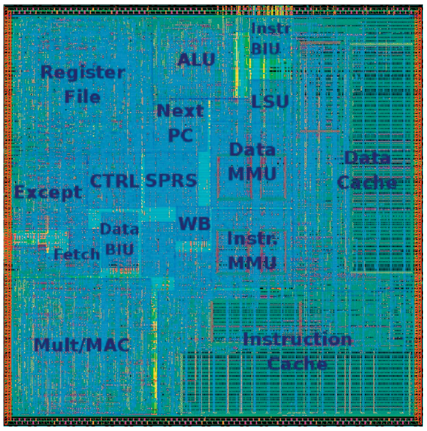
To study the impact of reconfiguration granularity on chip lifetimes, the mean-time-to-failure (MTTF) was calculated for each individual module in the OR1200. MTTF was determined by estimating the effects of a common wearout mechanism, time-dependent dielectric breakdown (TDDB) on an OR1200 core running a representative workload. Employing an empirical model similar to that found in [19], (1) presents the formula used to calculate per-module MTTFs. The temperature numbers for the modules were generated using HotSpot [20]. Given the MTTFs for individual modules, stage-level MTTFs in our experiment were 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:

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

where $V =$ operating voltage, $T =$ temperature, $k =$ Boltzmann's constant, and a, b, X, Y, and Z are all fitting parameters based on [19].

2.2 Granularity Trade-Offs

The granularity of reconfiguration is used to describe the unit of isolation/redundancy for modules within a chip. Various options for reconfiguration, in order of increasing granularity, are discussed below.



(a)

OR1200 Core	
Area	1.0 mm^2
Power	123.9 mW
Voltage	1.1 V
Clock Frequency	400 MHz
Data Cache Size	8 KB
Instruction Cache Size	8 KB
Technology Node	90nm

(b)

Fig. 1. OpenRisc 1200 embedded microprocessor. (a) Overlay of floorplan. (b) Implementation details.

1. *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.
2. *Module level:* In this scenario, a processor core can replace broken microarchitectural structures such as an ALU or branch predictor [21], [4]. The biggest drawback 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.
3. *Stage level:* Here, the entire pipeline stages are treated as single monolithic units that can be replaced. Reconfiguration at this level is challenging because: a) pipeline stages are tightly coupled with each other (reconfiguration can cause the performance loss), and b) cold sparing pipeline stages are expensive (area overhead).
4. *Core level:* This is the coarsest level of reconfiguration where an entire core is isolated after developing a failure. Core-level reconfiguration has also been an active area of research [15], [14], and from the perspective of a system designer, it is probably the easiest technique to implement. However, it has the poorest returns in terms of lifetime extension, and therefore, might not be able to keep up with increasing defect rates.

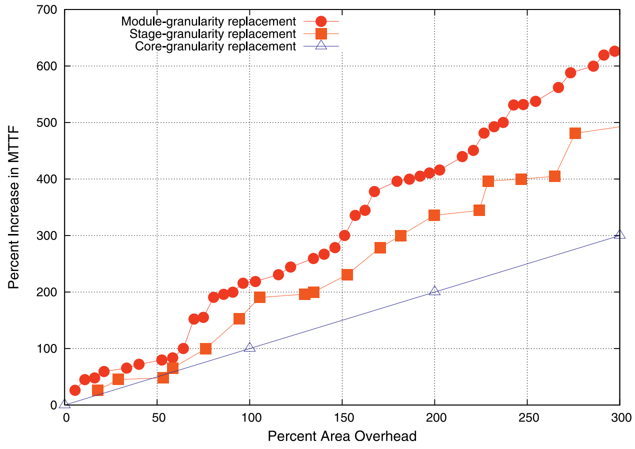


Fig. 2. Gain in MTTF from the addition of cold spares at the granularity of microarchitectural modules, pipeline stages, and processor core. The gains shown are cumulative, and spare modules are added (denoted with markers) in the order they are expected to fail.

While multiple levels of reconfiguration granularity could be utilized, Fig. 2 demonstrates the effectiveness of each applied in isolation (gate-level reconfiguration was excluded in this study). The figure shows the potential for lifetime enhancement (measured as MTTF) as a function of how much area a designer is willing to allocate to cold spares. The MTTF of an n-way redundant structure is taken to be n times its base MTTF. And the MTTF of the overall system is taken to be the MTTF of the fastest failing module in the design. This is similar to the serial model of failure used in [19]. The figure overlays three separate plots, one for each level of reconfiguration. The redundant spares were allowed to add as much as 300 percent area overhead.

The data shown in Fig. 2 demonstrate that going toward finer grain reconfiguration is categorically beneficial as far as gains in MTTF are concerned. But it overlooks the design complexity aspect of the problem. Finer grain reconfiguration tends to exacerbate the hardware challenges for diagnosing faults (needs better observability) and maintaining redundancy (muxing logic, wiring overhead, and circuit timing management). 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.

2.3 Harnessing Stage-Level Reconfiguration

Stage-level reconfiguration is positioned as a good candidate for system recovery as it scales well with the increase in area available for redundancy (Fig. 2). Logically, stages are a convenient boundary because pipeline architectures divide work at the level of stages (e.g., fetch, decode, etc.). Similarly, in terms of circuit implementation, stages are an intuitive boundary because data signals typically get latched at the end of every pipeline stage. Both these factors are helpful when reconfiguration is desired with a minimum impact on the performance. However, there are two major obstacles that must be overcome before stage-level reconfiguration is practical:

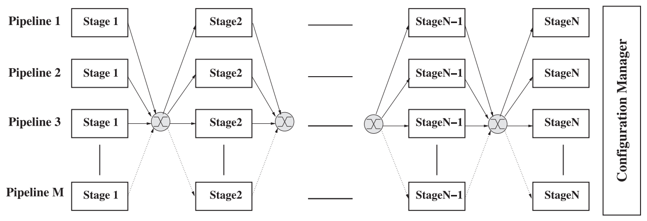


Fig. 3. A StageNet assembly: group of slices connected together. Each SNS is equivalent to a logical processing core. This figure shows M, N-stage slices. Broken stages can be easily isolated by routing around them, whereas crossbar failures can be tolerated using spares.

1. Pipeline stages are tightly coupled with each other and are, therefore, difficult to isolate/replace.
2. Maintaining spares at the pipeline stage granularity is very area-intensive.

One of the ways to allow stage-level reconfiguration 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 Fig. 3. We call this design **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 an **SNS**. The use of switches allows complete flexibility for a pipeline stage at depth N to communicate with any stage at depth $N+1$, even those from a different SNS. The SN architecture overcomes both of the major obstacles for stage-level reconfiguration. Pipeline stages are decoupled from each other, and hence, faulty ones can be easily isolated. Furthermore, there is no need to exclusively devote chip area for cold sparing. The SN architecture exploits the inherent redundancy present in a multicore by borrowing/sharing stages from adjacent cores. As nodes (stages) wearout and eventually fail, SN will exhibit a graceful degradation in the performance, and a gradual decline in throughput.

Along with its benefits, SN architecture has certain area and performance overheads associated with itself. Area overhead primarily arises from the switch interconnection network between the stages. And depending upon the switch bandwidth, a variable number of cycles will be required to transmit operations between stages, leading to performance penalties. The next section investigates the performance overheads when using an SNS and also presents our microarchitectural solutions to regain these losses. The remainder of the paper focuses on the design and evaluation of the SN architecture, and demonstrates its ability to maintain high lifetime throughput in the face of failures.

3 The StageNetSlice Architecture

3.1 Overview

SNS is a basic building block for the SN architecture. It consists of a decoupled pipeline microarchitecture that allows convenient reconfiguration at the granularity of stages. As a basis for the SNS design, a simple in-order core is used, consisting of five stages, namely, fetch, decode, issue, execute/memory, and writeback [22], [18]. Although the execute/memory block is sometimes separated into multiple stages, it is treated as a single stage in this work.

Starting with a basic in-order pipeline, we will go through the steps of its transformation into SNS. As the first step, pipeline latches are replaced with a combination of a crossbar switch and buffers. A graphical illustration of the resulting pipeline design is shown in Fig. 4. The shaded boxes inside the pipeline stages are microarchitectural additions that will be discussed in detail later in this section. To minimize the performance loss from interstage communications, we propose the use of full crossbar switches since 1) these allow nonblocking access to all of their inputs and 2) for a small number of inputs and outputs, they are not prohibitively expensive. The full crossbar switches have a fixed channel width and as a result, transfer of an instruction from one stage to the next can take a variable number of cycles. However, this channel width of the crossbar can be varied to trade-off performance with area. In addition to the forward data path connections, pipeline feedback loops in SNS (branch mispredict and register writeback) also need to go through similar switches. With the aid of these crossbars, different SNSs within an SN multicore can share their stages with each other. For instance, the result from, say, SNS A's execute stage, might need to be directed to SNS B's issue stage for the writeback. Due to the introduction of crossbar switches, 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 for

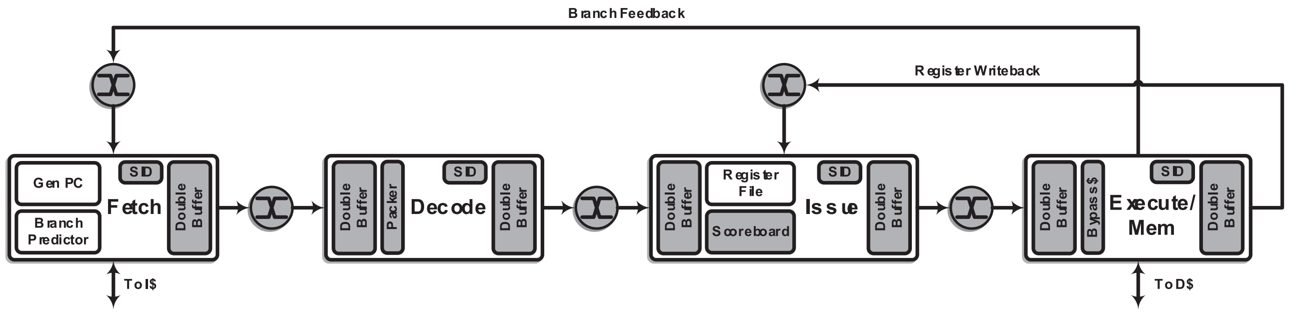


Fig. 4. An SNS pipeline. Stages are interconnected using a full crossbar switch. The shaded portions highlight modules that are not present in a regular in-order pipeline.

cases such as multicycle operations, memory access, and other hazards. Whereas flush signals are necessary to squash instructions that are fetched along mispredicted control paths. In SNS, all the stages are decoupled from each other, and global broadcast is infeasible.

2. *Forwarding:* Data forwarding is essential to prevent frequent stalls that would, otherwise, occur because of data dependencies in the instruction stream. 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 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 to create a functionally correct pipeline (Section 3.2) and propose techniques that can recover the expected loss in the performance (Section 3.3).

3.2 Functional Needs

3.2.1 Stream Identification

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

The SNS design shown in Fig. 4 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 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 mispredict 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 mispredict is detected (using the branch update from execute/memory stage), it toggles the `sid` register and flushes the program counter. From this point onward, the instructions fetched are stamped with the updated stream-id.

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

Issue. This maintains the `sid` register along with an additional 1-bit `last-sid` register. The `sid` register is updated using 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 becomes important. 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. The complete significance of `last-sid` will be made clear later in this section.

Execute/Memory. This compares the stream-id of the incoming instructions to the `sid` register. In the event of a mismatch, the instruction is squashed. A mispredicted branch instruction toggles its own stream-id along with the `sid` register value stored here. This branch resolution information is sent back to the fetch stage, initiating a change in its `sid` register value. The mispredicted branch instruction also updates the `sid` in the issue stage during 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 component required for proper functionality of 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 Fig. 5c): the first to maintain a *valid* bit for each register and second to store the *id* of the last modifying instruction. In case of a branch mispredict, the scoreboard needs to be wiped clean since it gets polluted by instructions on the wrong path of execution. To recognize a mispredict, 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 mispredict has taken place. At this point, the scoreboard

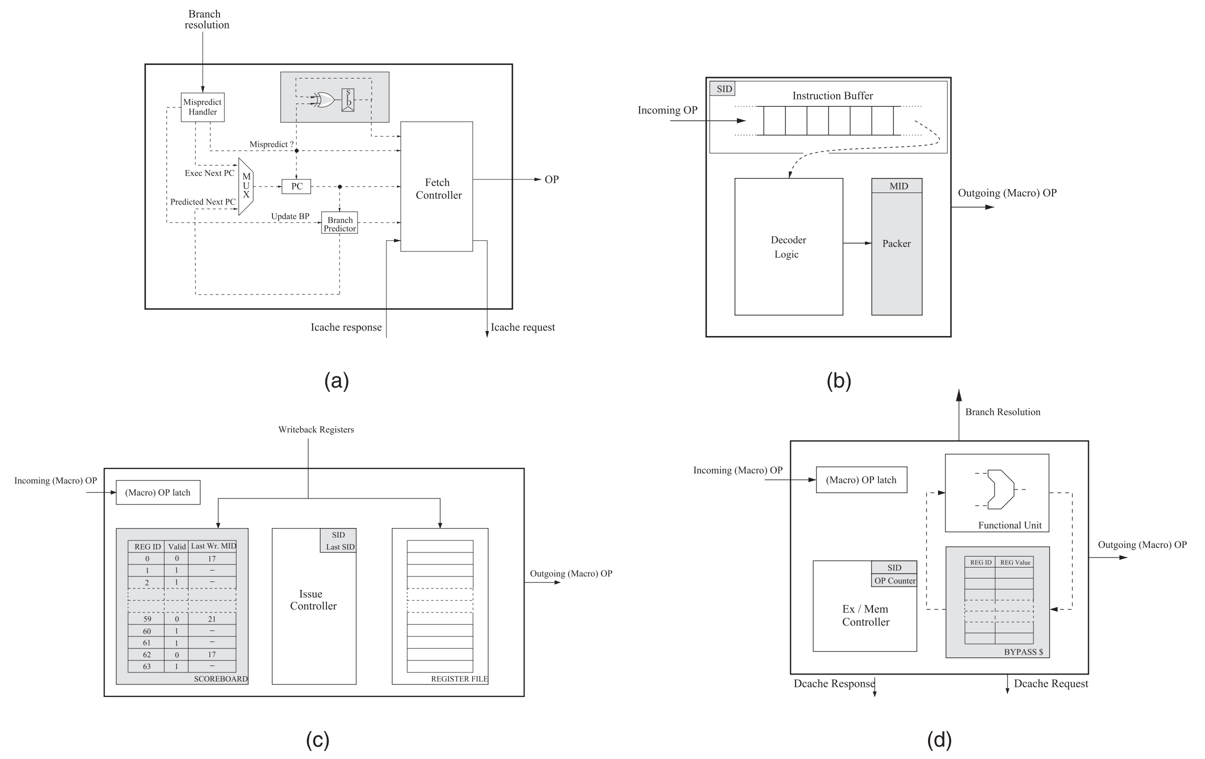


Fig. 5. Pipeline stages of SNS. Gray blocks highlight the modules added for transforming a traditional pipeline into SNS. (a) Fetch. (b) Decode. (c) Issue. (d) Execute/Memory.

waits to receive the writeback, if it hasn't received it already, for the branch instruction that was the cause of the mispredict. This branch instruction can be easily identified because it will bear the same stream-id as the new incoming instruction. Finally, after this waiting period, the scoreboard is cleared and the new instruction is issued.

3.2.3 Network Flow Issues

In 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 or consumer. In SNS, all stages have their input and output latches double buffered to enable this optimization.

3.3 Performance Enhancement

The additions to SNS discussed in the previous section bring the design to a point where it is functionally correct. In order to compare the performance of this *basic* SNS design to an in-order pipeline, we conducted some experiments using a cycle accurate simulator developed in the Liberty Simulation Environment [23]. *Basic* here implies an SNS pipeline that is configured with the stream identification logic, scoreboard, and double buffering. The details of our simulation setup and benchmarks are provided in Section 6.1. The performance of a basic SNS pipeline (first bar) in comparison to the baseline is shown in Fig. 6. The results are normalized to the runtime of the baseline in-order processor. On average, a 4X slowdown was observed, which is a significant price to pay in return for the reconfiguration flexibility. However, in this version of the SNS design, much is left on the table in terms of the performance. Most of this performance is lost in the stalls due to 1) the absence of forwarding paths and 2) transmission delay through the switches.

3.3.1 Bypass Cache

Due to the lack of forwarding logic in 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 Fig. 5d). This cache stores 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 issue waiting

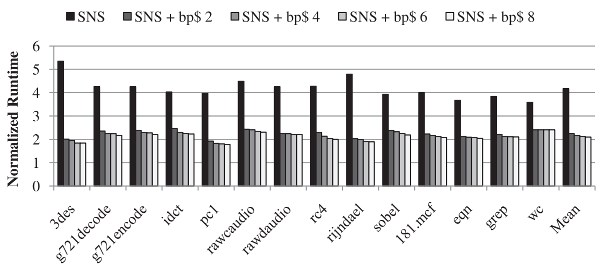


Fig. 6. SNS performance normalized to the baseline. Different configurations of SNS are evaluated, both with and without the bypass cache. The slowdown reduces as the bypass cache size is increased (fewer issue stage stalls).

for writeback. In fact, if this 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 emulating forwarding logic.

An 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 in the issue stage is made aware of the bypass cache size when the system is first configured. Whenever the number of outstanding registers in the scoreboard becomes equal to this cache size, instruction issue is stalled. In all other cases, the instruction can be issued as all of its input dependencies are guaranteed to be present within the bypass cache. Hence, the scoreboard can accurately predict whether or not the bypass cache will have a vacancy to store the output from the current instruction. Furthermore, the issue stage can perform selective register operand fetch for only those values that are not going to be available in the bypass cache. By doing this, the issue stage can reduce the number of bits that it needs to transfer to the execute/memory stage.

As evident from the experimental results (Fig. 6), the addition of the bypass cache results in dramatic improvements in the overall performance of SNS. The biggest improvement comes between the SNS configuration without any bypass cache (first bar) to the one with a bypass cache of size two (second bar). This improvement diminishes after a while and saturates around six entries. The average slowdown hovers around 2.1X with the addition of the bypass cache. In terms of its cost, a six-entry bypass cache is about 20 percent in size relative to the register file, which has 32 entries.

3.3.2 Interconnection Network

Within an SNS, pipeline stages are connected to each other using nonblocking, narrow-width crossbars. The width of a crossbar determines the number of bits that can be transferred across it in a single cycle. The SNS stages maintain latches at their inputs as well as outputs, isolating the interconnection network into a separate stage. The interconnection network does not contain any internal buffers. Thus, the transfer latency for an instruction across this network is determined by the crossbar width. For instance, a 32-bit crossbar would take three cycles to transfer 96-bits of data.

The results presented so far in this section have been with a crossbar width of 32-bits. Fig. 8 illustrates the impact of varying this width on the performance. Three data points are presented for every benchmark: a 32-bit channel width, a 64-bit channel width, and infinite channel width. A large performance gain is seen when going from 32-bit width to 64-bit width. Infinite bandwidth essentially means that any amount of information can be transferred between the stages in a single cycle, resulting in the performance comparable to the baseline (however, at a tremendous area cost). With a 64-bit crossbar switch, SNS has an average slowdown of about 1.35X. The crossbar width discussion is revisited after the next performance enhancement.

3.3.3 Macro Operations

The performance of the SNS design suffers significantly from the overhead of transferring instructions between stages, as every instruction has to go through a switched network with a variable amount of delay. Here, a natural optimization would be to increase the granularity of communication to a bundle of multiple operations that we call a macro-op (*MOP*). This has two advantages:

1. More work (multiple instructions) is available for the stages to work on while the next MOP is being transmitted.
2. MOPs 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.

These collections of operations can be identified 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 bounding two parameters: 1) the number of live-ins and live-outs and 2) the number of instructions. We use a simple greedy policy, similar to [24], that maximizes the number of instructions, while minimizing the number of live-ins and live-outs. When forming MOPs, as long as the computation time in the stages can be brought closer to the transfer time over the interconnection, it is a win.

The compiler embeds the MOP boundaries, internal data flow, and live-in/live-out information in the program binary. During runtime, the decode stage's *Packer* structure is responsible for identifying and assembling MOPs. Leveraging hints for the boundaries that are embedded in the program binary, the Packer assigns a unique MOP id (MID) to every MOP flowing through the pipeline. All other stages in the SNS are also slightly modified in order to work with these MOPs instead of simple instructions. This is particularly true of the execute/memory stage where a controller cycles across the individual instructions that comprise an MOP, 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 use of MOPs also impacts the way the exceptions are handled by an SNS. The exceptions that occur at MOP boundaries do not need special attention, as the program execution can resume at the next MOP. On the other hand, exceptions in the middle of an MOP need a more involved response. At the time of exception, the remaining instructions in the MOP are thrown away and don't get executed. When the program execution resumes, the *Packer* starts precisely from the instruction following the excepting instruction, forming only singleton operations (not MOPs). This is continued until an MOP boundary is reached, at which point regular MOPs can be formed again.

The performance results shown in Fig. 7 are for an SNS pipeline with the bypass cache, 64-bit switch channel width and MOPs. The various bars in the plot are for different configurations of the MOP selection algorithm. The results show that beyond a certain limit, relaxing the MOP selection constraints (live-ins and live-outs) does not result in the performance improvement. Prior to reaching this limit, relaxing constraints helps in forming longer MOPs,

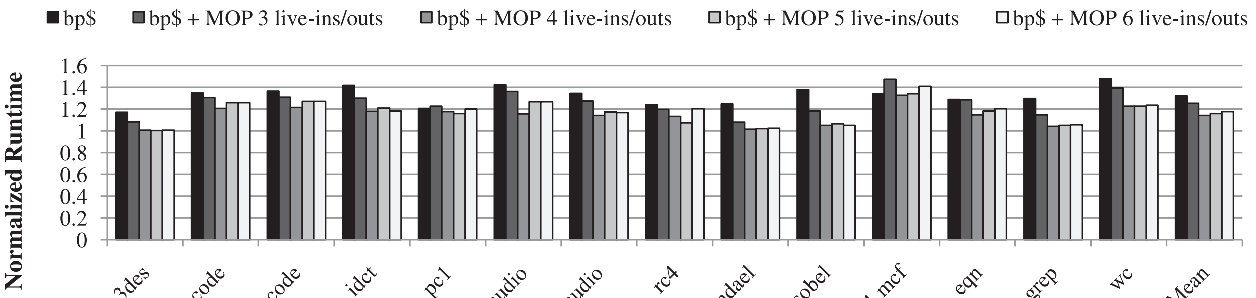


Fig. 7. SNS with a bypass cache and the capability to handle MOPs, compared to the baseline in-order pipeline. The first bars are for MOP sizes fixed at 1, while the other bars have constraint on the number of live-ins and live-outs.

thereby balancing transfer time with computation time. Beyond this limit, relaxing constraints does not result in longer MOPs. Instead, it produces wider MOPs that have more live-ins/outs, which increases transfer time without actually increasing the number of distinct computations that are encoded. On average, the best performance was observed for live-ins/outs constraint of four. This yielded 1.14X slowdown for an SNS pipeline over the baseline. The worst performers were the benchmarks that had very poor branch prediction rates. In fact, the performance on SNS was found to be strongly correlated with the number of mispredicts per thousand instructions. This is expected because the use of MOPs, and the additional cycles spent for data transfer between stages, causes the SNS pipeline to behave like a very deep pipeline.

3.3.4 *Crossbar Width Optimization*

The bandwidth requirement at each SNS switch interface is not the same. For instance, macro-ops that are transmitted from decode to issue stage do not have any operand values. But the ones that go from issue to execute/memory stage hold the operand values read from the register file, making them larger. This observation can be leveraged to optimize the crossbar widths between every pair of stages, resulting in an overall area saving.

A series of experiments was conducted to track the number of bits transmitted over each crossbar interface (fetch-decode, decode-issue, issue-execute, execute-issue, and execute-fetch) for every MOP. The average number of bits transmitted varied from 32 to 87. Given a fixed budget of total crossbar width (across all interfaces), a good strategy is to allocate width to each interface in proportion to the number of bits it transfers. The result of applying this optimization to the SNS pipeline is shown in Fig. 9. For nearly the same crossbar area (budget of 300-bits), the optimized assignment of crossbar widths is able to deliver three percent performance improvement over uniform usage of 64-bit crossbars (equivalent to 320-bits in total). With this final performance enhancement, the SNS pipeline slowdown stands at about 1.11X of the baseline.

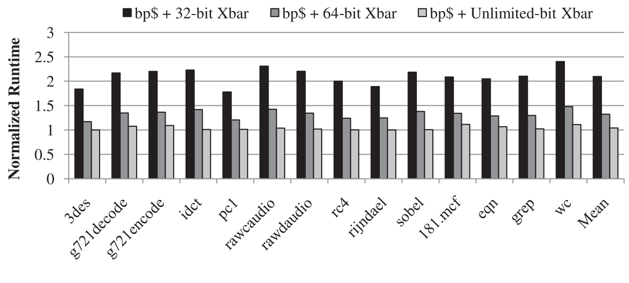


Fig. 8. An SNS pipeline, with variation in the transmission bandwidth. The performance improves with the increasing transmission bandwidth and almost matches the base pipeline at unlimited bandwidth.

3.4 Stage Modifications

This section summarizes the modules added to each stage in the pipeline.

Fetch. The `sid` register and a small amount of logic to toggle it upon branch mispredicts (Fig. 5a).

Decode. An instruction buffer, augmented with a `sid` register to identify branch mispredicts and flush the buffer. The decode stage (Fig. 5b) is also augmented with the Packer. The Packer logic reads instructions from the buffer, identifies the MOP boundaries, assigns them an MID, and fills out the MOP structure attributes such as length, number of operations, and live-in/out register names.

Issue. The issue stage (Fig. 5c) is modified to include a Scoreboard that tracks register dependencies. For an MOP that is ready for issue, the register file is read to populate the live-ins. The issue stage also maintains two 1-bit registers: `sid` and `last-sid`, in order to identify branch mispredicts and flush the Scoreboard at appropriate times.

Execute/Memory. The execute/memory stage (Fig. 5d) houses the bypass cache that emulates the job of forwarding logic. This stage is also the first to update its `sid` register upon a branch mispredict. In order to handle MOP execution, the execute/memory controller is modified to walk the MOP instructions one at a time (one execution per cycle).

4 The Stagenet Multicore

The SNS presented in the last section is in itself a complete microarchitectural solution to allow pipeline-stage-level reconfiguration. By maintaining cold spares for stages that are most likely to fail, an SNS-based design can achieve the lifetime enhancement targets projected in Fig. 2. However, these gains can be greatly amplified, without the cold sparing cost, by using multiple SNSs as building blocks to form an SN multicore.

The high-level abstraction of SN (Fig. 3), in combination with the SNS design, forms the basis of the SN multicore

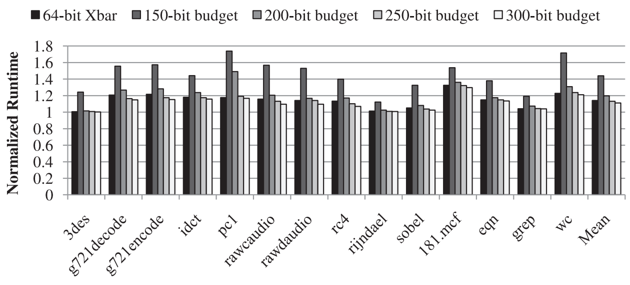


Fig. 9. Performance comparison with different budgets for crossbar widths. The first bar is for static assignment of 64-bit crossbars at all interfaces, which is equivalent to a 320-bit (64×5) budget. Optimized assignment of 300-bits is able to deliver better performance than 320-bit static assignment.

(Fig. 10). The resources within this are not bound to any particular slice and can be connected in any arbitrary fashion to form logical pipelines. The SN multicore has two prominent components to glue SNSs together:

1. *Interconnection Crossbars:* The role of the crossbar switch is to direct the incoming MOP to the correct destination stage. For this task, it maintains a static routing table that is addressed using the thread-id of the MOP. The thread-id uniquely determines the destination stage for each thread. To circumvent the danger of having them as single points of failure, multiple crossbars can be maintained by the SN multicore.
2. *Configuration Manager:* Given a pool of stage resources, the configuration manager divides them into logical SNSs. The configuration manager logic is better suited for a software implementation since: a) it is accessed very infrequently (only when new faults occur), and b) more flexibility is available in software to experiment with resource allocation policies. The configuration manager can be designed as a firmware/kernel module. When failures occur, a trap can be sent to the virtualization/OS interface, which can then initiate updates for the switch routing tables.

In the event of any stage failure, the SN architecture can initiate recovery by combining live stages from different slices, i.e., salvaging healthy modules to form logical SNSs. We refer to this as the *stage borrowing* (Section 4.1). In addition to this, if the underlying stage design permits, stages can be time-multiplexed by two distinct SNSs. For instance, a pair of SNSs, even if one of them loses its *execute* stage, can still run separate threads while sharing the single live *execute* stage. We refer to this as *stage sharing* (Section 4.2).

4.1 Stage Borrowing

A pipeline stage failure in the system calls upon the configuration manager to determine the maximum number of *full* logical SNSs that can be formed using the pool of live stages. *Full* SNS here implies an SNS with exclusive access to exactly one stage of each type. The number of such SNSs that can be formed by the configuration manager is determined by the stage with the fewest live instances. For example, in Fig. 10, the bottom two SNSs have a minimum of one stage alive of each type, and thus, one logical SNS is formed. The logical slices are highlighted using the shaded path, indicating the flow of the instruction streams.

It is noteworthy that all four slices in Fig. 10 have at least one failed stage, and therefore, a multicore system in a similar situation would have lost all working resources. Hence, SN's ability to efficiently borrow stages from different slices gives it the competitive edge over a traditional multicore.

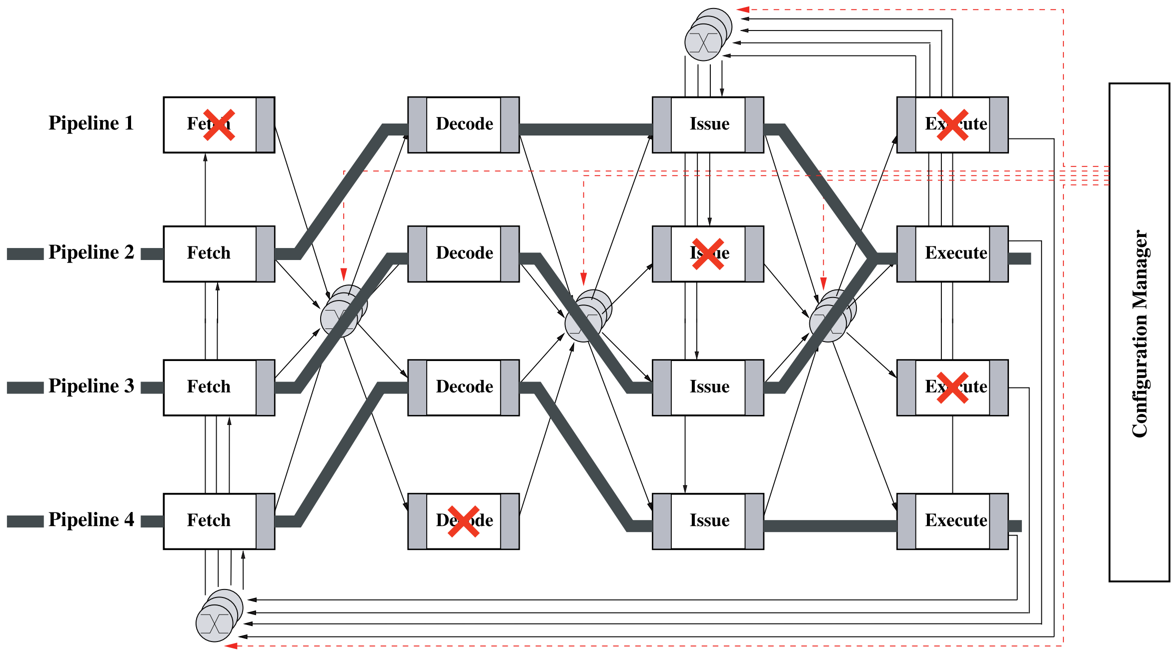


Fig. 10. An SN multicore formed using four SNSs. As an example, a scenario with five broken stages is shown (crosses indicate broken stages). Faced with a similar situation, a regular CMP will lose all its cores. However, SN is able to salvage three operational SNSs, as highlighted by the bold lines (note that these bold lines are not actual connections). The configuration manager is shown for illustrative purposes, and is not an actual hardware block.

4.2 Stage Sharing

Stage borrowing is good, but it is not enough in certain failure situations. For example, the first-stage failure in the SN fabric reduces the number of logical SNSs by one. However, if the stages can be time-multiplexed by multiple SNSs, then the same number of logical SNSs can be maintained. Fig. 10 has the top two logical SNSs sharing an *execute* stage. The number of logical SNSs that can share a single stage can be tuned in our implementation.

The sharing is beneficial only when the threads involved present opportunities to interleave their execution. Therefore, threads with very high instructions per cycle (IPC) are expected to derive lesser benefit compared to low IPC threads. Furthermore, as the degree of stage sharing is increased, the benefits are expected to shrink since more and more threads will contend for the available stage. In order for the stages to be shared, hardware modifications are required in each of them.

The fetch stage needs to maintain a separate program counter for each thread and has to time-multiplex the memory accesses. The instruction cache, in turn, will also be shared implicitly by the executing threads. In decode, the instruction buffer has to be partitioned between different threads. The scoreboard and the register file are populated with state values specific to a thread, and it is not trivial to share them. There are two ways to handle the sharing for these structures: 1) compile the thread with fewer registers or 2) use a hardware structure for register caching [25]. In our evaluation, we implement the register caching in hardware and share it across multiple threads. Finally, in the execute stage, bypass cache is statically partitioned between the threads.

4.3 Fault Tolerance and Reconfiguration

SN relies on a fault detection mechanism to identify broken stages and trigger reconfiguration. There are two possible solutions for detection of permanent failures: 1) continuous monitoring using sensors [9], [26] or 2) periodic testing for faults. The discussion of exact mechanism for detection is beyond the scope of this paper. The configuration manager is invoked whenever any stage or crossbar switch is identified to be defective. Depending upon the availability of working resources, configuration manager determines the number of logical SNSs that can be formed. It also configures the stages that need to be shared and partitions their resources accordingly between threads. While working with higher degrees of sharing, the configuration manager employs a fairness policy for resource allocation so that the work (threads) gets evenly divided among the stages. For example, if there are five threads that need to share three live stages of same type, the fairness policy prefers a 2-2-1 configuration (two threads each to stages 1 and 2 and remaining one to stage 3) over a 3-1-1 configuration (three threads to stage 1, one each to stages 2 and 3).

5 System-Level Design

The SN design, as presented in the previous section, fails to cover two important system-level aspects. First, for a large-scale multicore (10-100s of cores), how does the SN concept scale? The crossbars that were used to connect the slices together are notorious for steep growth in area and delay overheads as the number of ports is increased [27]. Second, how important is the interconnection reliability, and what are the ways to improve it? SN's robustness hinges on the crossbar reliability, and a failure in the same can render all of the working stages useless. The purpose of this section is to find answer to these questions.

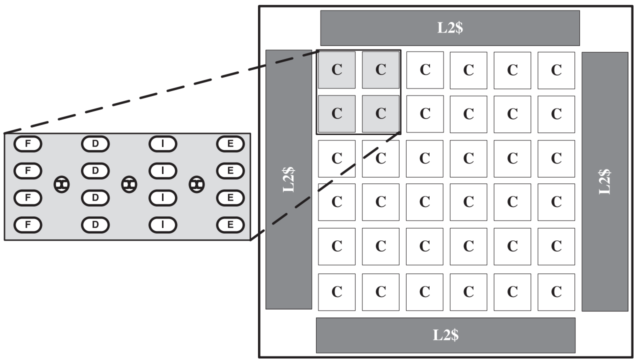


Fig. 11. A many core SN chip. The shaded group of cores constitutes an SN island. The island size (four in this example) is a design parameter.

The reliability advantages of SN stem from the ability of neighboring slices (or pipelines) to share their resources with one another. Thus, a direct approach for scaling would be to allow as many slices to connect together as possible. However, practical constraints such as area, delay, and layout-related issues would constitute reasons that limit the connectivity between the slices. Thus, a large many core system would need to be logically divided into smaller SN islands. Each such island would offer full connectivity within itself. Fig. 11 shows a conceptual floorplan of a large-scale SN chip that can be divided into nine SN islands, each containing four slices/cores (C). The memory hierarchy within SN is same as that of a typical many core chip, with a shared L2 and private L1 I/D cache for each slice.

5.1 Island Size

In order to ascertain the right number of slices that can be efficiently grouped together to form an island, we conducted lifetime reliability experiments for different island sizes. The total number of slices in these experiments was fixed, but they were grouped together at a range of values. For instance, 16 slices can be interwoven at granularity of two slices (leading to eight SN islands), four slices (leading to four SN islands), eight slices, or all 16 slices together. Fig. 12 shows the cumulative work done (left Y-axis) by a large number of slices grouped at a range of island sizes. The result is normalized to an equally provisioned CMP. Cumulative work metric, as defined in Section 6.2, measures the amount of useful work done by a system over its entire lifetime. Note that the interconnection fabric here is kept fault-free for the sake of estimating the upper bound on the advantage offered by SN.

The experiment shows that a bulk of reliability benefits are garnered by sharing among a small group of stages. The returns diminish with the increasing number of pipelines, and beyond 8-10 pipelines, there is only a marginal impact. This is so because, as an island spans more and more slices, the variation in time to failure of its components gets smaller and smaller. Thus, in a larger set of stages, most fail

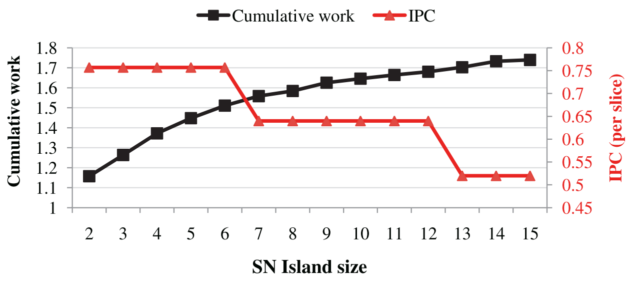


Fig. 12. Increase in cumulative work for a fixed size SN system as the islands are made larger. Interconnection faults were ignored for this experiment. The second Y-axis shows the decline in IPC for each slice as the crossbar latency increases (with island size).

around the same time anyway, making the option to borrow from neighbors less meaningful. A second, and more important, factor in determining SN island size is the crossbar wire delay. Due to the longer wires, connecting up more than six slices (computed using intermediate wiring delay from [28]) results in addition of a cycle to crossbar latency. This causes significant IPC loss, as shown in Fig. 12 (right Y-axis). Considering both factors mentioned above, the SN island size of six is used for system-level experiments in the rest of this paper.

5.2 Spare Crossbars

The functionality of interconnection is as important as that of stages for maintaining proper operation within an SN island. The interconnection wire links are typically reliable, and hardly develop faults (because: 1) they aren't scaled as badly as transistors, and 2) signal wires have low vulnerability to Electromigration). On the other hand, crossbar switches do need fault tolerance. This can be incorporated by maintaining cold spares. Fig. 13 shows improvement in cumulative work for a six-slice SN island with a varying number spares maintained per crossbar in the design. With no spares, the cumulative work is actually worse than the baseline (six-core CMP). The gains, however, are negligible beyond two spare crossbars.

In summary, a large-scale SN system has to be designed hierarchically, with each SN island containing six slices. Further, each crossbar should be allocated two spares for operating reliably.

6 Results and Discussion

6.1 Simulation Setup

The evaluation infrastructure for the SN architecture consisted of three major components: 1) a compilation framework, 2) an architectural simulator, and 3) a Monte Carlo simulator for lifetime throughput estimations. A total of 14 benchmarks were selected from the embedded and desktop application domains: encryption (3des, pc1, rc4, and rijndael), audio processing (g721encode, g721decode, rawcaudio, and rawdaudio), image/video processing (idct and sobel), Unix utilities (grep and wc), and SPECint benchmarks (mcf and eqn).

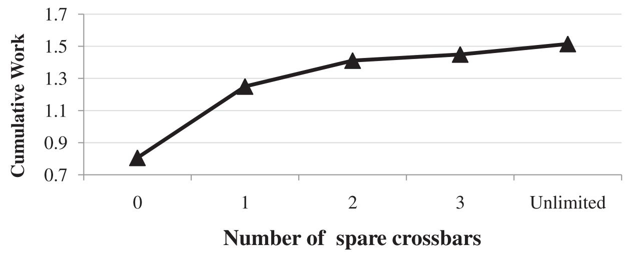


Fig. 13. Increase in cumulative work for an SN island (six slices) as more cold spares are added for crossbar switches.

TABLE 1
Architectural Attributes

Baseline	5-stage in-order pipeline
SNS	4-stage in-order, with double buffering and all other performance enhancements
Branch pred.	global, 16-bit history, gshare predictor BTB size of 2KB
L1 I$, D$	4-way, 16 KB, 1 cycle hit latency
L2 $ unified	8-way, 64 KB, 5 cycle hit latency
Memory	40 cycle hit latency

We use the Trimaran compilation system [29] as our first component. The MOP selection algorithm is implemented as a compiler pass on the intermediate code representation. During this pass, the code is augmented with the MOP boundaries. The final code generated by the compiler uses the HPL-PD ISA [30].

The architectural simulator for the SN evaluation was developed using the Liberty Simulation Environment (LSE) [23]. 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 experiments. The second simulator implemented the SN architecture with all the proposed enhancements. Table 1 lists the common attributes for our simulations.

The third component of our simulation setup is the Monte Carlo engine that we employ for lifetime throughput study. Each iteration of the Monte Carlo process simulates the lifetime of the SN architecture. The configuration of the SN architecture is specified in Table 1. The MTTF for the various stages and switches in the system was calculated using (1), with the fetch qualified to have an MTTF of 10 years. The crossbar switch peak temperature was taken from [31] that performs interconnection modeling for the RAW multicore chip [32]. The stage temperatures were extracted from HotSpot simulations of the OR1200 core with the ambient temperature normalized to the one used in [31]. The calculated MTTFs are used as the mean of the Weibull distributions for generating time to failure (TTF) for each module (stage/switch) in the system. For each iteration of the Monte Carlo, the system gets reconfigured over its lifetime whenever a failure is introduced. The instantaneous throughput of the system is computed for each new configuration using the architectural simulator on multiple random benchmark subsets. From this, we obtain the system throughput over the lifetime. The Monte Carlo study required about 1,000 iterations before arriving at *reasonably* accurate results. Final numbers from a pair of these Monte Carlo studies differed by less than 0.5 percent.

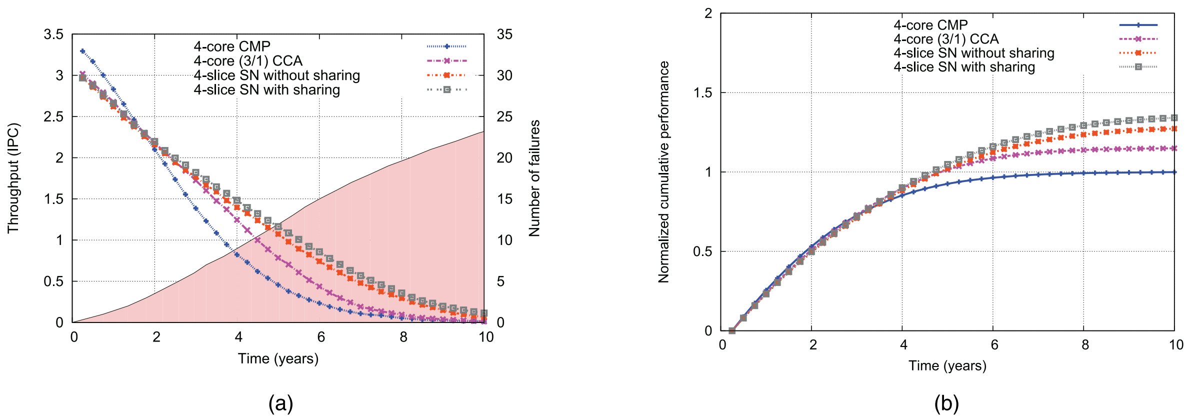


Fig. 14. Throughput and cumulative performance results for four-core CMP, four-core CCA (3/1) configuration, four-slice SN, and four-slice SN with sharing. (a) Throughput over time. This plot shows (shaded portion) the expected number of failed modules (stages/switch) until that point in the lifetime. (b) Cumulative work done. These curves are time integral of throughput over time plots.

6.2 Simulation Results

6.2.1 Lifetime Performance (Four-Slice SN Chip)

Fig. 14a shows the lifetime throughput results for a four-core CMP compared against four-core CCA (3/1) configuration [33], and two equally provisioned configurations of the SN architecture. The four-core CCA (3/1) configuration, a concurrent effort for stage-level reconfiguration, allows three cores to borrow resources of the fourth core in the event of failure(s). This is strictly less flexible than SN where all cores can borrow resources from each other.

The CMP system starts with a performance advantage over all other configurations due to a slightly higher single-thread performance. However, as failures accumulate, the throughput of SN overtakes other configurations and remains dominant thereafter. For instance, at year five, the throughput of SN is more than double the throughput of CMP. The CCA configuration is able to keep up with SN until the middle of third year, but it falls behind as the number of failures rises beyond six. The shaded portion in this figure depicts the number of failures accumulated in the system by a given point in the life. For instance, this plot shows that after eight years, on average, there are 20 failed structures in the system. The difference between SN configuration with and without sharing was found to be almost negligible. Remaining results in the paper are for SN configuration without sharing.

As evident from this result, the SN system is able to sustain a higher throughput, relative to a traditional design, for a longer duration. In the context of technology scaling, there are two ways to benefit from this result. First, in the future, marketing models can be expected for mass consumer chips where a fraction of actual resources are exposed to the end user (similar to IBM zSeries processor modules [34]). The remaining resources are either dead on arrival, or are used as spares for in-field failures. In this scenario, SN chips can be placed into higher bins, as they provide stronger throughput guarantees. Second, architectural solutions like SN can help the semiconductor manufacturers take an extra step in silicon scaling, beyond what can be achieved by process engineers alone.

The throughput differential between SN and other designs can also be compared by integrating the respective throughputs over the lifetime. We call this *cumulative work*, and it is roughly the total amount of work performed by a chip. Fig. 14b shows the cumulative work for SN configurations compared against the baseline. By the end of the lifetime, we achieve as much as 37 percent improvement in the work done for the SN fabric. About 30 percent of this is achieved by *stage borrowing* only, and the additional seven percent benefit is a result of *stage sharing*. The sharing was not found to be very effective as the opportunities to time-multiplex stages were very few and far between. The CCA configuration achieves about 15 percent improvement in the cumulative work done.

6.2.2 Lifetime Performance (64-Slice SN Chip)

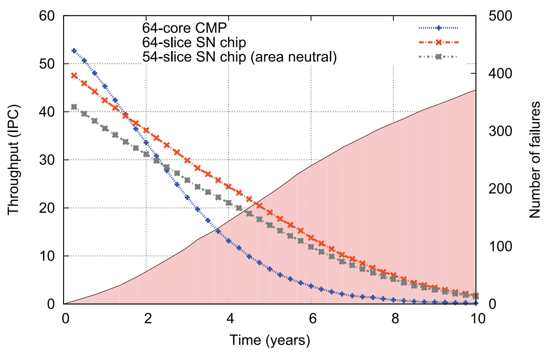
The above experiments were also repeated for a larger many core system. Typically, the die sizes today are about $100\ \text{mm}^2$, out of which nearly 60 percent is devoted to processing cores. Given that estimate, one such die can hold 64 OR1200 cores (see Fig. 1b).

Fig. 15a shows the lifetime throughput results for a 64-core CMP, 64-slice SN chip, and 54-slice SN chip. Both SN chips use an island size of six slices and employ two spare crossbars per interface. The 54-slice SN chip configuration was added to have an area neutral comparison with the baseline CMP (area overhead of SN is discussed later in more detail). As in the case above, the CMP system starts with a performance advantage over the SN architecture, but falls below both SN configurations around the two-year mark.

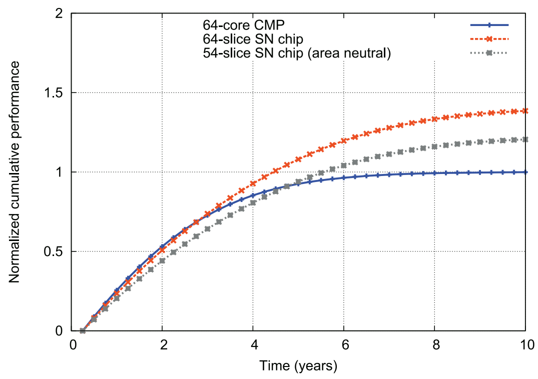
Fig. 15b shows the cumulative performance (total work done) for SN configurations compared against the baseline. By the end of the lifetime, 64-slice SN chip achieves 40 percent improvement. Even the area neutral SN configuration with 54-slices dominates baseline CMP by 22.5 percent. Thus, at no additional silicon cost, SN solution delivers a significant reliability advantage.

6.2.3 Area Overhead

The area overhead in the SN arises from the additional microarchitectural structures that were added and the interconnection fabric composed of the crossbar switches. Area overhead is shown using an OR1200 core as the baseline (see Section 2.1). The area numbers for the bypass cache and register cache are estimated by taking similar



(a)



(b)

Fig. 15. Throughput and cumulative performance results for 64-core CMP, 64-slice SN chip, and 54-slice SN chip (area neutral). (a) Throughput over time results. This plot also shows (shaded portion) the expected number of failed modules (stages/switch) until that point in the lifetime. (b) Cumulative work done. These curves are time integral of throughput over time plots.

structures from the OR1200 core and resizing them appropriately. More specifically, bypass cache and register cache areas are based on the TLB area, which is also an associative lookup structure. And finally, the area of double buffers is based on the maximum macro-op size they have to store. The sizing of all these structure was done in accordance with the SNS configuration that achieved the best performance. The crossbar switch area is based on the Verilog model from [27]. The total area overhead for the SN design (no sharing) is ~15 percent (Table 2). This was computed assuming that six slices share a crossbars, and each crossbar maintains two cold spares. Note that the scoreboard area is ignored in this discussion, as the introduction of sufficiently sized bypass cache eliminates the need for them.

All the design blocks were synthesized using Synopsys Design Compiler and placed and routed using Cadence Encounter with a TSMC standard cell library characterized for a 90 nm process. The area overhead for separate modules, crossbar switches, and SN configurations is shown in Table 2.

6.2.4 Power Overheads

Power dissipation for various modules in the design is simulated using Synopsys Primepower on an execution trace of OR1200 running media kernels. The crossbar power dissipation was simulated separately using a representative activity trace. The stage to crossbar interconnection power was calculated using standard power equations [35] with capacitance from Predictive Technology Model [36] and intermediate wiring pitch from 90 nm node (ITRS [28]).

The power overhead in SN comes from three sources: crossbars, stage/crossbar interconnection, and miscellaneous logic (extra latches and new modules). Table 3 shows the breakdown, with total power overhead at 16.4 percent. A majority of this power overhead comes from the interconnection network (crossbars and links).

6.2.5 Timing Overhead

Although we have not investigated the impact of our microarchitectural changes to the circuit critical paths, a measurable influence on the cycle time is not expected in SNS, because: 1) our changes primarily impact the pipeline depth (due to the additional buffers), and 2) all logic changes are local to the stages and do not introduce any direct (wire) communication between them.

7 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. This section focuses on the most relevant subset of work, those that propose architectures that tolerate and/or adapt to the presence of faults.

High-end server systems, like Tandem NonStop and IBM zSeries [34], typically rely on coarse-grained spatial redundancy to provide a high degree of reliability. However, such dual and triple modular redundant systems incur significant overheads in terms of area and power, and cannot tolerate a high failure rate. More recently,

TABLE 2
Area Overhead of the SN Architecture

Design Blocks		
Block name	**Area** (mm ^{2})	**Percent overhead**
Bypass cache	0.044	3.4%
Register cache	0.028	2.2%
Double buffers	0.067	5.3%
Miscellaneous logic	0.012	0.9%
64-bit crossbar switch	0.028	2.1%
SN Configurations		
Configuration		**Percent overhead**
SN without sharing		15.1%
SN with sharing		17.3%

TABLE 3
Power Overhead for a Single SNS: The Percentages Are with Respect to OR1200's Power Consumption

Component	Power overhead (mW)	Percent overhead
Crossbars	5.3	4.26%
Interconnection links	7.6	6.19%
Other design blocks	7.9	6.38%
Total power overhead		**16.4%**

ElastIC [15], Configurable Isolation [14], and Architectural Core Salvaging [37] are high-level architectural proposals for multiprocessor fault tolerance. Although good in a limited failure rate scenario, all of these proposals need a massive number of redundant cores, without which they face the prospect of rapidly declining processing throughput as faults lead to core disabling.

Much work has also been done in fine-grained redundancy maintenance such as Bulletproof [21], sparing in array structures [12], and other such microarchitectural structures [4]. These schemes typically rely on inherent redundancy of superscalar cores, and it is also extremely hard to achieve good coverage with them.

SN differs dramatically from solutions previously proposed in that our goal is to minimize the amount of hardware used solely for redundancy. More specifically, we enable reconfiguration at the granularity of a pipeline stage, and allow pipelines to share their stages, making it possible for a single core to tolerate multiple failures at a much lower cost. In parallel to our efforts, Romanescu and Sorin [33] have proposed a multicore architecture, 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 SN, 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 SN.

8 Conclusion

Technology scaling is driven by a need to keep improving computation capabilities by packing more and more resources onto a single chip, and can be sustained as long as the benefits outweigh the associated costs. With the growing reliability concerns, this scaling is under a threat. Therefore, as the CMOS technology evolves, so 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 paper contributes to the area of hardware reconfiguration by proposing a radical architectural shift in processor design. The end goal is to extend the life of silicon technology, keeping the scaling beneficial even in the face of high defect rates.

The proposed architecture, named as SN, is motivated by a need for fine-grained reconfiguration, which enables a better resource utilization in the presence of faults. In SN's design, networked pipeline stages were identified as the effective trade-off between cost and reliability enhancement. Although the performance suffered at first as a result of changes to the basic pipeline, a few well-placed microarchitectural enhancements were able to reclaim much of what was lost. Ultimately, the SN architecture exchanged a modest amount of area overhead (15 percent) in return for a highly resilient fabric with only 10 percent degradation in the single-thread performance.

SN's ability to salvage working pipeline stages, from otherwise broken cores, enables it to give stronger throughput guarantees. This translates into a higher number of available hardware contexts for a longer period of time. For instance, a 64-core SN chip was able to provide an IPC of 20 until the five year mark. A similarly provisioned traditional CMP chip (with core disabling) dipped below 20 IPC just after three years. Accumulated over the entire lifetime, this throughput differential results in 40 percent more cumulative work for the SN chip.

For marketing chips with such high defect rates, future chip vendors might expose only 50-75 percent of the resources that are actually present on a chip. This marketing model is already employed in IBM zSeries servers, where every processor module has two spare cores. In such a scenario, SN chips can be rated at a significantly higher throughput (relative to a CMP), for a given service lifetime. In addition to wearout fault tolerance, the SN concept is similarly beneficial for failures at manufacture time, resulting in yield improvements. Hence, the SN fabric is well positioned to withstand the rapidly increasing device failure rates, permitting the aggressive scaling of technology.

Acknowledgments

The authors gratitude goes to the anonymous referees who provided excellent feedback on this work. The authors also thank David Penry for his assistance with the Liberty Simulation Environment and Visvesh Sathe for his help with Cadence Encounter. This research was supported by ARM, Ltd., the US 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.

References

[1] K. Bernstein, "Nano-Meter Scale cmos Devices (Tutorial Presentation)," 2004.

[2] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro,* vol. 25, no. 6, pp. 10-16, Nov./Dec. 2005.

[3] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor," *IEEE Micro,* vol. 25, no. 2, pp. 21-29, Feb. 2005.

[4] P. Shivakumar, S. Keckler, C. Moore, and D. Burger, "Exploiting Microarchitectural Redundancy for Defect Tolerance," *Proc. 2003 Int'l Conf. Computer Design,* pp. 481-488, Oct. 2003.

[5] J. Zeigler, "Terrestrial Cosmic Ray Intensities," *IBM J. Research and Development,* vol. 42, no. 1, pp. 117-139, 1998.

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

[7] E. Wu, J.M. McKenna, W. Lai, E. Nowak, and A. Vayshenker, "Interplay of Voltage and Temperature Acceleration of Oxide Breakdown for Ultra-Thin Gate Oxides," *Solid-State Electronics,* vol. 46, pp. 1787-1798, 2002.

[8] C. Weaver and T.M. Austin, "A Fault Tolerant Approach to Microprocessor Design," *Proc. 2001 Int'l Conf. Dependable Systems and Networks,* pp. 411-420, 2001.

[9] J.A. Blome, S. Feng, S. Gupta, and S. Mahlke, "Self-Calibrating Online Wearout Detection," *Proc. 40th Ann. Int'l Symp. Microarchitecture,* pp. 109-120, 2007.

[10] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," *IEEE Micro,* vol. 28, no. 1, pp. 52-59, Jan. 2008.

[11] F.A. Bower, D.J. Sorin, and S. Ozev, "A Mechanism for Online Diagnosis of Hard Faults in Microprocessors," *Proc. 38th Ann. Int'l Symp. Microarchitecture,* pp. 197-208, 2005.

[12] F.A. Bower, P.G. Shealy, S. Ozev, and D.J. Sorin, "Tolerating Hard Faults in Microprocessor Array Structures," *Proc. 2004 Int'l Conf. Dependable Systems and Networks,* pp. 51-60, 2004.

[13] D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "Nonstop Advanced Architecture," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 12-21, June 2005.

[14] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J.E. Smith, "Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors," *Proc. 34th Ann. Int'l Symp. Computer Architecture,* pp. 470-481, 2007.

[15] D. Sylvester, D. Blaauw, and E. Karl, "Elastic: An Adaptive Self-Healing Architecture for Unpredictable Silicon," *IEEE J. Design and Test,* vol. 23, no. 6, pp. 484-490, June 2006.

[16] Tilera "Tile64 Processor—Product Brief," http://www.tilera.com/pdf/, 2008.

[17] L. Seiler et al., "Larrabee: A Many-Core $\times 86$ Architecture for Visual Computing," *ACM Trans. Graphics,* vol. 27, no. 3, pp. 1-15, 2008.

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

[19] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers, "The Case for Lifetime Reliability-Aware Microprocessors," *Proc. 31st Ann. Int'l Symp. Computer Architecture,* pp. 276-287, June 2004.

[20] W. Huang, M.R. Stan, K. Skadron, K. Sankaranarayanan, and S. Ghosh, "Hotspot: A Compact Thermal Modeling Method for cmos vlsi Systems," *IEEE Trans. Very Large Scale Integration Systems,* vol. 14, no. 5, pp. 501-513, May 2006.

[21] K. Constantinides, S. Plaza, J.A. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, "Bulletproof: A Defect-Tolerant CMP Switch Architecture," *Proc. 12th Int'l Symp. High-Performance Computer Architecture,* pp. 3-14, Feb. 2006.

[22] ARM "Arm11," http://www.arm.com/products/CPUs/families/ARM11Family.html, 2010.

[23] 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 Trans. Computer Systems,* vol. 24, no. 3, pp. 211-249, 2006.

[24] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, "Scalable Subgraph Mapping for Acyclic Computation Accelerators," *Proc. 2006 Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems,* pp. 147-157, Oct. 2006.

[25] M. Postiff, D. Greene, S. Raasch, and T. Mudge, "Integrating Superscalar Processor Components to Implement Register Caching," *Proc. 2001 Int'l Conf. Supercomputing,* pp. 348-357, 2001.

[26] E. Karl, P. Singh, D. Blaauw, and D. Sylvester, "Compact In Situ Sensors for Monitoring nbti and Oxide Degradation," *Proc. 2008 IEEE Int'l Solid-State Circuits Conf.,* Feb. 2008.

[27] L.-S. Peh and W. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* pp. 255-266, Jan. 2001.

[28] ITRS "Int'l Technology Roadmap for Semiconductors 2008," http://www.itrs.net/, 2008.

[29] Trimaran "An Infrastructure for Research in ILP," http://www.trimaran.org/, 2000.

[30] 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.

[31] L. Shang, L. Peh, A. Kumar, and N.K. Jha, "Temperature-Aware On-Chip Networks," *IEEE Micro,* vol. 26, no. 1, pp. 130-139, Jan./Feb. 2006.

[32] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 46-57, Oct. 1998.

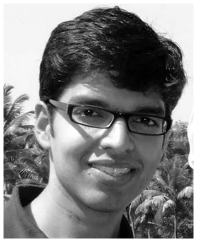
[33] B.F. Romanescu and D.J. Sorin, "Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processor in the Presence of Hard Faults," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques,* 2008.

[34] W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 87-96, Jan.-Mar. 2004.

[35] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits,* second ed. Prentice Hall, 2003.

[36] PTM "Predictive Technology Model," http://ptm.asu.edu/, 2010.

[37] M.D. Powell, A. Biswas, S. Gupta, and S.S. Mukherjee, "Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance," *Proc. 36th Ann. Int'l Symp. Computer Architecture,* June 2009.



Shantanu Gupta received the BTech degree in computer science and engineering from the Indian Institute of Technology, Guwahati, in 2005, and the MSE degree in computer engineering in 2007 from the University of Michigan, where he is currently working toward the PhD degree at the Electrical Engineering and Computer Science Department. His research interests span various aspects of compilers and architectures with a focus on fault tolerance, power efficiency, and single-thread performance.

Shuguang Feng received the bachelor's degree in computer engineering from the University of Florida in 2005, and the MSE degree in computer engineering in 2006 from the University of Michigan, Ann Arbor, where he is currently working toward the PhD degree at the Electrical Engineering and Computer Science Department. His research interests include fault tolerant, reconfigurable computer architectures, and investigating techniques that can exploit the interaction between software and hardware to enhance the system reliability.

Amin Ansari received the BS degree in computer engineering from Sharif University of Technology, Iran, in 2007, and the MSE degree in computer science and engineering in 2008 from the University of Michigan, Ann Arbor, where he is currently working toward the PhD degree at the Department of Electrical Engineering and Computer Science. His research interests include designing architectural and microarchitectural techniques for enhancing reliability of high-performance microprocessors in deep submicron technologies. He is a student member of the IEEE and the ACM.

Scott Mahlke received the PhD degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1997. He is an associate professor in the Electrical Engineering and Computer Science Department at the University of Michigan, where he leads the Compilers Creating Custom Processors Research Group. The CCCP group delivers technologies in the areas of compilers for multicore processors, application-specific processors for mobile computing, and reliable system design. His achievements were recognized by being named the Morris Wellman assistant professor in 2004 and being awarded the Most Influential Paper Award from the International Symposium on Computer Architecture in 2007. He is a member of the IEEE.