The StageNet Fabric for Constructing Resilient Multicore 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

Scaling of CMOS feature size has long been a source of dramatic performance gains. However, the reduction in voltage levels has not been able to match this rate of scaling, leading to increasing operating temperatures and current densities. 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. Consequently, high reliability and fault tolerance, which have traditionally been subjects of interest for high-end server markets, are now getting emphasis in the mainstream desktop and embedded 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 multicore architecture, named 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 towards the end of life. Our results show that the proposed SN architecture can perform nearly 50% more cumulative work compared to a traditional multicore.

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 from the 65nm node onward [8]. 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]. In an effort to assuage these concerns, industry has initiated a shift towards multicore and GPU inspired designs that employ simpler cores to limit the power and thermal envelope of the chips [42, 21, 23]. However, this paradigm shift also leads towards core designs that have little inherent redundancy and are therefore incapable of performing the self-repair possible in big superscalar cores [32, 38]. Thus, in the near future, architects 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 [47] and electrical noise [43], to permanent errors, caused by wearout phenomenon such as electromigration [13] and time dependent dielectric breakdown [46]. In recent years, industry designers and researchers have invested significant effort in building architectures resistant to transient faults [33, 33, 34, 44]. 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 recovery/reconfiguration. Fault detection mechanisms [9, 26, 5] are used to identify the presence of a fault, while fault diagnosis techniques [25, 15, 12] are used to determine the source of the fault, i.e. the broken component(s). System recovery 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 [36]. Recently, researchers have begun to extend these techniques to support sparing for additional on-chip resources [38], such as branch predictors [11] and registers [32]. 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 recovery and to design a fault tolerant architecture that is capable of tolerating a large number of failures.

Traditionally, system recovery in high-end servers and mission critical systems has been addressed by using mechanisms such as dual and triple-modular redundancy (DMR and TMR) [7, 36]. However, such approaches are too costly and therefore not applicable to desktop and embedded systems. 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 (CMPs) [35, 1, 39]. 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.

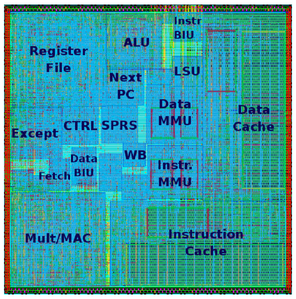
In contrast, this paper argues the case for reconfiguration and redundancy at a finer granularity. To this end, this work presents the *StageNet* (SN) fabric, a highly reconfigurable and adaptable multicore computing substrate. SN is a multicore architecture designed as a network of pipeline stages, 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). A 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, a 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 re-thinking 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 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. It further presents the complete SN architecture 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 a simple in-order core design as the basis of the SN architecture. This is motivated by the fact that thermal and power considerations are pushing designs towards simpler cores. Furthermore, they are being adopted by designs targeting massively multicore chips [42, 24] and are suitable for low-latency and high throughput applications [23, 21]. At the same time, we believe that the proposed design methodology of fine-grained reconfiguration can also be effectively applied to deeper and more aggressive pipeline designs.

The primary contributions of this paper include: 1) A design space exploration of reconfiguration granularities for resilient systems; 2) Design and evaluation of a StageNetSlice, a networked pipeline microarchitecture; and 3) Design and evaluation of StageNet, a resilient multicore architecture, composed using multiple SNSs.

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 non-functioning, non-critical processor structures to swapping in cold spare devices. In a reconfigurable architecture, recovery entails isolating defective module(s) and incorporating spare structures as needed. Support for reconfiguration can be achieved at various granularities, 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. This section shows experiments studying this trade-off and draws upon these results to motivate the design of the SN architecture.



(a) Overlay of floorplan

OR1200 Core	
Area	1.28 mm^2
Power	92.22 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.

2.1 Experimental Setup

In order to effectively model the reliability of different designs, a Verilog model of the OpenRISC 1200 (OR1200) core [27] 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. Employing an empirical model similar to that found in [37], Equation 1 presents the formula used to calculate per-module MTTFs. The temperature numbers for the modules were generated using HotSpot [19].

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

where V = operating voltage, T = temperature, k = Boltzmann's constant, and a,b,X,Y,and Z are all fitting parameters based on [37].

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.

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. Such designs have been active topics of research [16, 32]. The biggest downside of this reconfiguration level is that

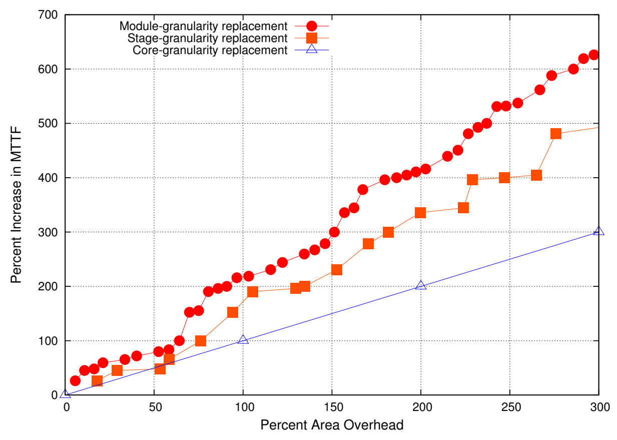


Figure 2: Gain in MTTF from the addition of cold spares at the granularity of micro-architectural modules, pipeline stages, and processor core. The gains shown are cumulative, and spare modules are added (denoted with markers) in the order they are expected to fail.

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: 1) pipeline stages are tightly coupled with each other (reconfiguration can cause performance loss), and 2) cold sparing pipeline stages is expensive (area overhead).
4. *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 [39, 1], 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.

While multiple levels of reconfiguration granularity could be utilized, Figure 2 demonstrates the effectiveness of each applied in isolation (gate-level reconfiguration was not included 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 a 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 [37]. 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 are 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.

2.3 Implications on StageNet

Stage level reconfiguration is positioned as a good candidate for system recovery as it scales well with the increase in area available for redundancy (Figure 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:

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 (Figure 8). 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 *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 performance, and a gradual decline in throughput.

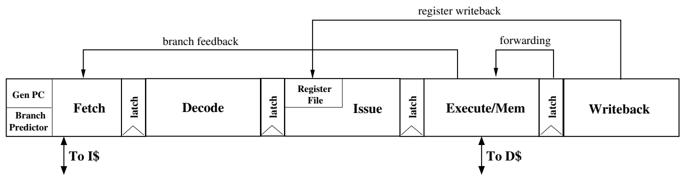
Along with its benefits, SN architecture has certain area and performance overheads associated with itself. Area overhead primarily arises from the switch interconnection network between the stages. And depending upon the switch bandwidth, a variable number of cycles will be required to transmit operations between stages, leading to performance penalties. The remainder of this paper investigates the design of a practical SN architecture starting with a single SNS.

3. SNS MICROARCHITECTURE

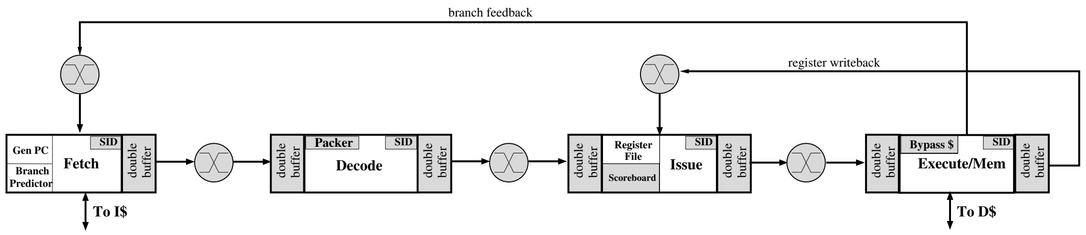
3.1 Overview

The 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 SNS design, a simple embedded processor core is used, consisting of five stages namely, fetch, decode, issue, execute/memory, and writeback [3, 4, 27] (see Figure 3(a)). Although the execute/memory block is sometimes separated into multiple stages, it is treated as a single stage in this work.

Starting with the basic pipeline design (Figure 3(a)), we will go through the steps of its transformation into a 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 Figure 3(b). 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 inter-stage communications, we propose the use of full



(a) A 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 3: Traditional in-order pipeline design and SNS.

crossbar switches [28] 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. 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 the SNS also need to go through similar switches. These are needed because, in the context of the complete SN architecture, different SNSs can share their stages with each other and thus require an exchange of branch mispredict and writeback results. 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 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 for cases such as multi-cycle operations, memory access, and other hazards. Similarly, flush signals are necessary to squash instructions that are fetched along mispredicted control paths. In a SNS, all the stages are decoupled from each other, and global broadcast 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 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 a 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 (Section 3.2) and also propose techniques that can recover the expected loss in performance (Section 3.3).

3.2 Functional Needs

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 Figure 3(b) 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 (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 decode does not match the stream id of an incoming instruction, a branch mispredict is implied and decode flushes its instruction buffer.
- *Issue:* It 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:* 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.

Scoreboard: The second 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 a 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 7(c)), 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 mis-predict, the scoreboard needs to be wiped clean since it gets polluted 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 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.

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

3.3 Performance Enhancement

The additions to the SNS discussed in the previous section brings 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 (Figure 3(a)), we conducted some experiments using a cy-

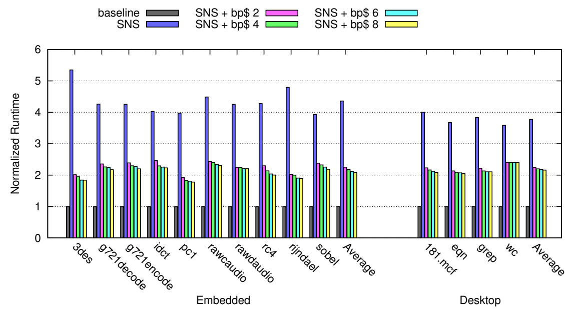


Figure 4: SNS pipeline compared to the baseline pipeline with and without the bypass cache. The slowdown, seen because of the issue stage stalls, reduces as the size of bypass cache is increased.

cle accurate simulator developed in the Liberty Simulation Environment [41]. *Basic* here implies a SNS configured with the stream identification logic, scoreboard, and double buffering. Interested readers can find the details of our simulation setup and benchmarks in Section 5.1. The performance of the SNS in comparison to the baseline is shown as a comparison between the first and second bars in Figure 4. 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 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.

Bypass Cache: Due to the lack of forwarding logic in a 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 7(d)). 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 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.

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 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 (Figure 4), the addition of the bypass cache results in dramatic improvements in the overall performance of the SNS. The biggest improvement comes between the SNS configuration without any bypass cache (second bar) to the one with a bypass cache of size 2 (third bar). This improvement diminishes after a while, and saturates beyond 8 entries. The average slowdown hovers around 2.1X with the addition of the bypass cache, which is still high.

Crossbar Width: The crossbar channel width is the number of

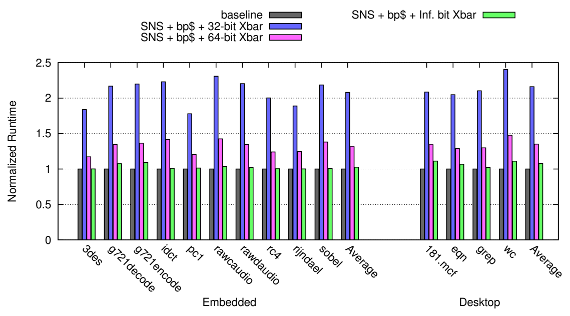


Figure 5: SNS pipeline, with variation in the transmission bandwidth. As expected the performance improves with higher transmission bandwidth.

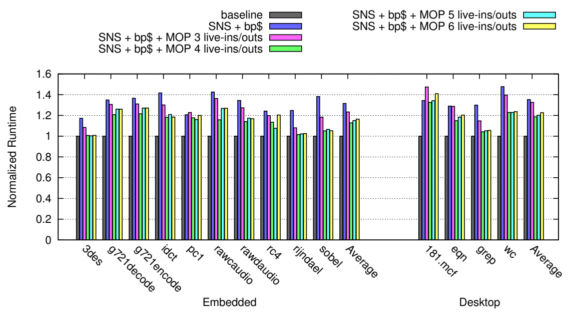


Figure 6: A SNS with a bypass cache and the capability to handle MOPs, compared to the baseline in-order pipeline. The second bars are for MOP sizes fixed at 1, while the other bars have constraint on the number of live-ins and live-outs.

bits that can be transferred to/from the crossbar in a single cycle. In the context of a SNS, it determines the number of cycles it will take to transfer an instruction between the stages. The results presented so far in this section have been with a crossbar channel width of 32-bits. Figure 5 illustrates the impact of varying this width on 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 eliminating all transfer latency between the stages, resulting in performance comparable to the baseline. With a 64-bit crossbar switch, SNS has an average slowdown of about 1.35X.

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. Here, a natural optimization would be to increase the granularity of communication to a bundle of multiple operations (a *MOP*). There are two advantages of doing this:

1. More work (multiple instructions) is available for the stages to work on while the next MOP is being transmitted.
2. Macro-ops 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 [14], 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.

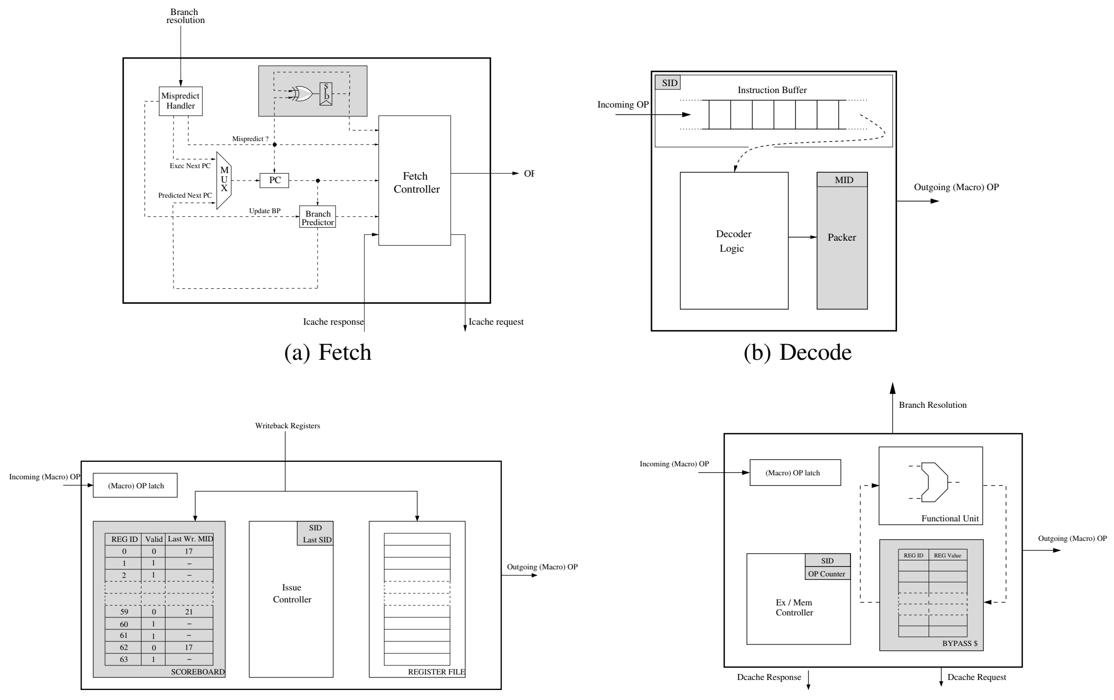
The compiler embeds the MOP boundaries, internal data flow, and live-in/live-out information in the program binary. During runtime, the decode stage's *Packer* structure is responsible for identifying and assembling MOPs. Leveraging hints for the boundaries that are embedded in the program binary, the 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 a MOP, executing them in sequence.

The performance results shown in Figure 6 are for a SNS 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 performance improvement. Prior to reaching this limit, relaxing constraints helps in forming longer MOPs, 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. The best case result for embedded applications has about 1.1X slowdown, and the desktop equivalent has about 1.2X slowdown. The worst performers were the benchmarks that had very poor branch prediction rates. In fact, the performance on a 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 to behave like a very deep pipeline.

3.4 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 here are restricted to the addition of `sid` register and a small amount of logic to toggle it upon branch mis-predicts (Figure 7(a)).
- *Decode:* The decode stage (Figure 7(b)) collects the fetched instructions in a buffer. An instruction buffer is a common structure found in most pipeline designs, and to that we add our `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 MOP boundaries, assigns them a MID, and fills out the MOP structure attributes such as length, number of operations and live-in/out register names.
- *Issue:* The issue stage (Figure 7(c)) is modified to include a Scoreboard that tracks register dependencies. For a 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 mis-predicts and flush the Scoreboard at appropriate times.
- *Execute/Memory:* The execute/memory stage (Figure 7(d)) 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 mis-predict. 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). At



(a) Fetch (b) Decode

(c) Issue (d) Execute/Memory

Figure 7: Pipeline stages of the SNS. Gray blocks highlight the modules added for transforming a traditional pipeline into a SNS.

the same time, the computed results are saved into the bypass cache for later use.

4. STAGENET MULTICORE FABRIC

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, a SNS-based design can achieve the lifetime enhancement targets projected in Figure 2. However, these gains can be greatly amplified, without the cold sparing cost, by using multiple SNSs as building blocks for the SN architecture.

The high level abstraction of the SN, in combination with the SNS design, forms the basis of the SN fabric (Figure 8). This final SN design is a highly reconfigurable multicore fabric capable of arbitrarily using pipeline stages to form logical SNSs. The two prominent hardware structures within the SN architecture are:

1. *Interconnection Switch:* The role of the 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 because, in its current form, SN does not allow a thread to use two stages of the same type. To circumvent the danger of having them as single points of failure, multiple switches are maintained by the SN fabric.
2. *Configuration Manager:* Given a pool of stage resources, the configuration manager divides them into logical SNSs. The logic for the configuration manager is better suited to a software implementation since: 1) it will be accessed infrequently, and 2) 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 *issue* stage, can still run separate threads while sharing the single live *issue* 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 a 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 Figure 8, the top three SNSs have a minimum of two stages alive of each type, and, thus, two logical SNSs are formed. The logical slices are highlighted using the shaded path indicating the flow of the instruction streams.

It is noteworthy that all top three pipelines in Figure 8 have at least one failed stage, and therefore, a multicore system in a similar situation would have lost all three pipelines. Hence, SN's ability to efficiently borrow stages from different slices, gives it the competitive edge over a traditional multicore.

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. Figure 8 has the bottom two

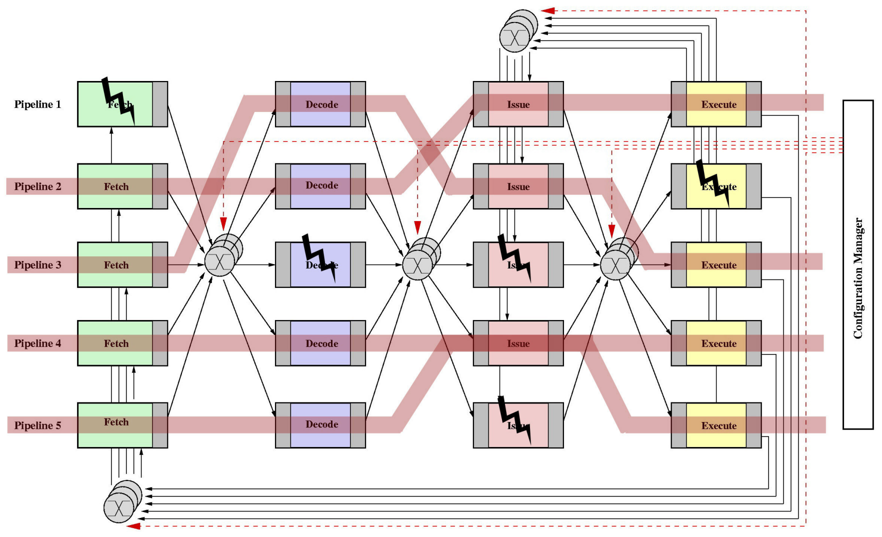


Figure 8: A SN architecture formed using five SNSs. As an example scenario where lightning bolts indicate broken stages, four operational SNSs are extracted as shown by the transparent paths.

SNSs sharing the *issue* 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 IPC (instructions per cycle) 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 shareable, certain hardware modifications are also required:

- *Fetch:* It 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
- *Decode:* The instruction buffer has to be partitioned between different threads.
- *Issue:* 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 [29]. In our evaluation, we implement the register caching in hardware and share it across multiple threads.
- *Execute/Memory:* The bypass cache is statically partitioned between the threads. Similarly, the data cache gets shared by the threads.

The configuration manager is invoked whenever any stage resource develops a defect in the SN system. Depending upon the degree of stage sharing, it determines the number of logical SNSs that can be formed. The configuration manager 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). Further details of regarding the resource allocation are omitted for space reasons.

5. RESULTS AND DISCUSSION

5.1 Simulation Setup

The evaluation infrastructure for the SN fabric consisted of three major components: 1) the 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. For these evaluations, the emphasis was on the embedded benchmarks because the SNS is based on an in-order embedded core. A variety of these were used including several encryption (3des, pc1, rc4, rijndael), audio processing (g721encode, g721decode, rawcaudio and rawdaudio), and image/video processing (idct, sobel) benchmarks. In addition, four desktop benchmarks (181.mcf, eqn, grep, wc) were also included in order to exhibit the potential of this architecture for other domains.

We use the Trimaran compilation system [40] 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 and other miscellaneous attributes. The final code generated by the compiler uses the HPL-PD ISA [20].

The architectural simulator for the SN evaluation was developed using the Liberty Simulation Environment (LSE) [41]. 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 Equation 1[1]. The crossbar switch peak temperature was taken from [31] that performs interconnection modeling for the RAW multicore chip [24]. 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. Nearly 200 such iterations are run for conducting the Monte Carlo study.

Base core	5-stage in-order pipeline
SNS	4-stage in-order, with double buffering and 64-bit 5×5 crossbar switches
Base multicore	5 baseline cores
SN	5 SNSs, five 2-way redundant 64-bit crossbar switches, and a configuration manager
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

Table 1: Architectural attributes.

5.2 Simulation Results

Lifetime performance benefits: The simulation results were gathered separately for the embedded and desktop benchmarks. Figure 9(a) shows the lifetime throughput results for a baseline multicore compared against various configurations of the SN architecture running the embedded benchmarks. The baseline system starts with a performance advantage over the SN architecture. However, as failures accumulate over time, the throughput of SN overtakes the baseline performance and thereafter remains dominant. For instance, at year 7, the IPC of the SN is nearly 4X the baseline IPC. The shaded portion in this figure depicts the cumulative distribution function (CDF) of the combined MTTF Weibull distributions. For instance, this plot shows that after 10 years, on an average, there are 12 failed modules in the system. Similar throughput improvements were seen for the desktop benchmarks (Figure 9(c)).

Figure 9(b) shows the cumulative performance (total work done) for various SN configuration compared against the baseline. By the end of the lifetime, we achieve as much as 50% improvement in the work done for the SN fabric. About 40% of this is achieved by *stage borrowing* only, and the additional 10% benefit is a result of *stage sharing*. Furthermore, going from 2-way sharing to 3-way and higher sharing yields negligible improvement. This is an expected result because the opportunities to overlap executions diminish as the contention for a stage increases. For the desktop benchmarks (Figure 9(d)), the cumulative performance improvements stand at about 40%. Overall, the desktop benchmarks exhibit lesser benefits than their embedded counterparts. This result follows the trend seen in Figure 6.

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 scoreboard, bypass cache and the register cache are estimated by taking similar structures from the OR1200 core and resizing them appropriately. The scoreboard area estimate is based on the register file. Bypass cache and register cache areas are based on the TLB area, which is also an associative look-up 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 [28]. The total area overhead for the SN design is 15%.

All the design blocks were synthesized, placed and routed using industry standard CAD tools with a library characterized for a 130nm process. The area overhead for separate modules and the crossbar switches is shown in Figure 10(a). The total area overhead of the SN system compared to the multicore baseline is shown in Figure 10(b).

Although, we have not investigated the impact of our microarchitectural changes to the circuit critical paths, our changes primarily impact the pipeline depth (due to the additional buffers). Hence, we don't expect a measurable influence on the cycle time in the SNS as compared to the baseline.

Design block	Area (mm^2)	Percent overhead
Scoreboard	0.018	1.4%
Bypass cache	0.044	3.4%
Register cache	0.028	2.2%
Double buffers	0.067	5.3%
64-bit crossbar switch	0.028	2.1%

(a) Area for SNS modules and crossbar switch

Configuration	Percent overhead
SN without sharing	14.1%
SN with sharing	16.3%

(b) Total area overhead for SN

Figure 10: Area overhead for the SN architecture.

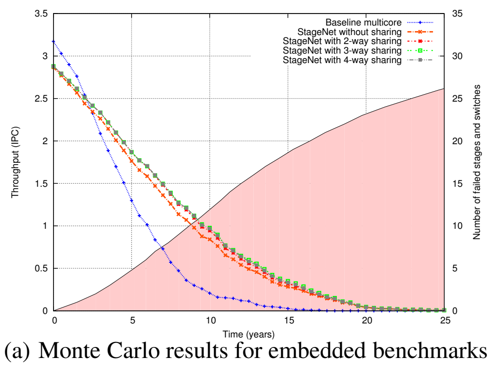
6. RELATED WORK

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 recent work, those that propose architectures that tolerate and/or adapt to the presence of faults.

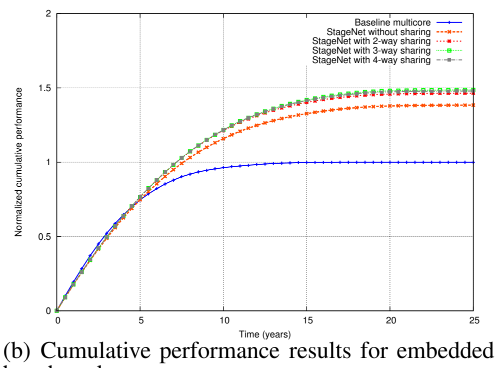
High-end server systems designed with reliability as a first-order design constraint have been around for decades but have typically relied on coarse grain replication to provide a high degree of reliability [7, 36], such as Tandem NonStop [6], Teramac [17, 2], Stratus [45], and the IBM zSeries [6]. 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 they cannot tolerate a high failure rate.

ElastIC [39] 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 a dynamic reliability management that can throttle and even-

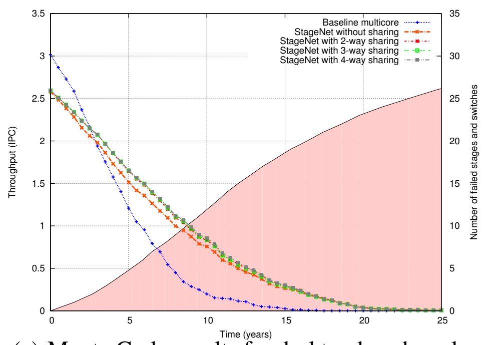
[1]The fetch stage was qualified to have a MTTF of 10 years. This is a conservative estimate and no actual module level MTTF values are available from any public source.



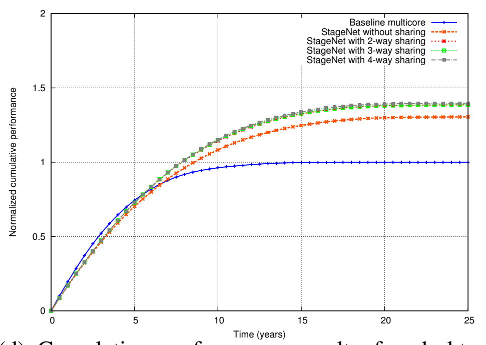
(a) Monte Carlo results for embedded benchmarks



(b) Cumulative performance results for embedded benchmarks



(c) Monte Carlo results for desktop benchmarks



(d) Cumulative performance results for desktop benchmarks

Figure 9: Throughput and cumulative performance results for the SN architecture. Plots (a) and (c) also show (shaded portion) the expected number of failed modules (stages/switch) until that point in the lifetime.

tually turn off cores as they age over time. 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 [16], sparing in array structures [11], and other such microarchitectural structures [32]. These schemes typically rely on inherent redundancy of superscalar cores, and it is also extremely hard to achieve full coverage with them. Nevertheless, ideas from such approaches can be leveraged in our system for additional benefits as they apply redundancy management at a finer granularity.

Other research has focused on building reliable substrates out of future nanotechnologies that are expected to be inherently fault-prone. The NanoBox Processor Grid [22] 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 the context of a single SNS. In parallel to our efforts, Romanescu et al. [30] 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 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 the CCA as compared to SN.

7. 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 the effective trade-off between cost and reliability enhancement. Although performance suffered at first as a result of the changes to the basic pipeline, a few well-placed microarchitectural enhancements were able to reclaim much of what was lost. Ultimately, the SN fabric exchanged a modest amount of area overhead (15%) in return for a highly resilient multicore fabric that yielded about 50% more work during its lifetime than a traditional multicore.

For mainstream process technologies (like 45nm), SN can be used to combat wearout failures over time. For upcoming technology generations with high defect rates at manufacture time (32nm and beyond), SN can also be employed for yield improvements. Finally, for more distant future technologies (such as carbon nanotubes), SN will likely need to be used in conjunction with other methods to combat high failure rates. Hence, the SN fabric is well positioned to withstand the rapidly increasing device failure rates expected in future technology nodes.

8. ACKNOWLEDGEMENTS

We thank David Penry for his assistance with 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.

9. 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] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proc. of the 1995 International Symposium on FPGA's for Custom Computing Machines*, pages 32–38, 1995.

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

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

[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] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two sytems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[7] 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.

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

[9] 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.

[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, A. Hormati, S. Mahlke, and S. Yehia. 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, 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.

[16] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: A defect-tolerant CMP switch architecture. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 3–14, Feb. 2006.

[17] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the teramac custom computer. In *Proc. of the 1997 International Symposium on FPGA's for Custom Computing Machines*, pages 116–123, 1997.

[18] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. Stagenetslice: A reconfigurable microarchitecture building block for resilient cmp systems. In *Proc. of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2008.

[19] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, and S. Ghosh. Hotspot: A compact thermal modeling method for cmos vlsi systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, May 2006.

[20] 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.

[21] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. Picoserver: using 3d stacking technology to enable a compact energy efficient chip multiprocessor. *ACM SIGPLAN Notices*, 41(11):117–128, 2006.

[22] A. KleinOsowski, K. KleinOsowski, and V. Rangarajan. 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.

[23] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Feb. 2005.

[24] 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. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998.

[25] 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.

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

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

[28] 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.

[29] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. of the 2001 International Conference on Supercomputing*, pages 348–357, 2001.

[30] 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.

[31] L. Shang, L. Peh, A. Kumar, and N. K. Jha. Temperature-aware on-chip networks. *IEEE Micro*, 2006.

[32] 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.

[33] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation, 3rd Edition*. AK Peters, Ltd., 1998.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. E. C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. In *2008 IEEE International Solid-State Circuits Conference*, pages 29–41, Jan. 2008.

[43] 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.

[44] 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.

[45] D. Wilson. The stratus computer system. *Resilient Computing Systems*, 1:208–231, 1986.

[46] 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*, 46:1787–1798, 2002.

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