Survey of Error and Fault Detection Mechanisms

Ikhwan Lee
ikhwan@mail.utexas.edu

Mehmet Basoglu
mbasoglu@mail.utexas.edu

Michael Sullivan
mbsullivan@mail.utexas.edu

Doe Hyun Yoon
doehyun.yoon@gmail.com

Larry Kaplan
lkaplan@cray.com

Mattan Erez
mattan.erez@mail.utexas.edu

Locality, Parallelism and Hierarchy Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
1 University Station (C0803)
Austin, Texas 78712-0240

TR-LPH-2011-002
April 2011

This research was, in part, funded by the U.S. Government . The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Survey of Error and Fault Detection Mechanisms

Abstract

This report describes diverse error detection mechanisms that can be utilized within a resilient system to protect applications against various types of errors and faults, both hard and soft. These detection mechanisms have different overhead costs in terms of energy, performance, and area, and also differ in their error coverage, complexity, and programmer effort.

In order to achieve the highest efficiency in designing and running a resilient computer system, one must understand the trade-offs among the aforementioned metrics for each detection mechanism and choose the most efficient option for a given running environment. To accomplish such a goal, we first enumerate many error detection techniques previously suggested in the literature.

1 Introduction

Error detection mechanisms form the basis of an error resilient system as any fault during operation needs to be detected first before the system can take a corrective action to tolerate it. Myriad error detection techniques have been proposed in the literature, where each option has different tradeoff options in terms of energy, performance, area, coverage, complexity, and programmer effort; however, there is no single technique that is optimal for all parts of a complex computer system, all conditions of a large variety of applications, or all operating scenarios. Thus, adaptability and tunability become crucial aspects of an error-resilient system with high efficiency. In that respect, we must fully understand each error detection technique, in the context of a specific system, to choose the best option for a given operating scenario and application.

Detection mechanisms proposed thus far can be classified in three different ways as shown in Table 1: based on type of redundancy, placement in the system hierarchy, or detection coverage. Type of redundancy can be space-redundant, where hardware is replicated, or time-redundant, where software code is replicated. On the other hand, not all techniques utilize redundancy; thus, type of redundancy does not provide a comprehensive coverage of all available error detection mechanisms. Whether redundant or not, all techniques, however, are fully covered by a categorization based on placement in the system hierarchy or detection coverage. Placement of detection mechanisms can be at the circuit, architecture, software system, or application levels or involve a combination of these levels in a hybrid approach. Finally, these detection techniques cover hard, soft or both types of errors.

In short, this report lists all the detection techniques that can be applied to the Echelon system and provides a qualitative trade-off analysis, which will help achieve a tunable and adaptable

Table 1: Classification of error detection mechanisms

Criterion	Category
Redundancy type	Space-redundant Time-redundant
System hierarchy	Circuit-level Architecture-level Software system Application-level Hybrid
Detection coverage	Hard errors Intermittent errors Transient errors

resiliency within the Echelon system. The rest of the paper is organized as follows: Section 2 explains the failure mechanisms we assume for the errors. Section 3, Section 4, and Section 5 explain and compare various error detection techniques for memory, compute, and system, respectively. Then concluding remarks will be given in Section 6. Note that this report includes tables summarizing and comparing the different techniques. These tables contain overhead numbers as reported in the research papers describing the mechanisms. The overhead numbers *are not in the context of Echelon*. We will evaluate the mechanisms for Echelon in the future.

2 Failure Mechanisms

In this report, we describe existing error detection techniques for both hard and soft errors. The failure mechanisms for hard errors are permanent stuck-at faults that occur in the field, undetected manufacturing or design flaws, or degradation-dependent faults that initially look like transient errors but become permanent under further degradation. This type of error causes permanent removal of a component and may trigger reconfiguration of the system. Note that we do not cover design errors that can be detected by traditional testing methods such as boundary scan chain or built-in self-test (BIST). We also exclude timing errors that can be detected by techniques like Razor [1] from our discussion.

The failure mechanisms for soft errors can be classified into two types. First, energetic particle strikes cause hole-electron pairs to be generated, effectively injecting a momentary ($<$ 1ns) pulse of current into a circuit node. This results in a single event upset (SEU), which we refer to as a transient error. This type of failure mechanism is also applicable in the case of supply noise briefly affecting a circuit's voltage level. Second, variations introduced during manufacture and runtime can cause temporal timing violations along the critical paths of the logic. They are referred to as intermittent errors and they are becoming more serious as we push the margin with techniques like dynamic voltage and frequency scaling (DVFS) to achieve higher efficiency. While intermittent errors are actually the result of hard faults, they are often treated as soft errors because of the difficultly of systematically reproducing the conditions that trigger an error and

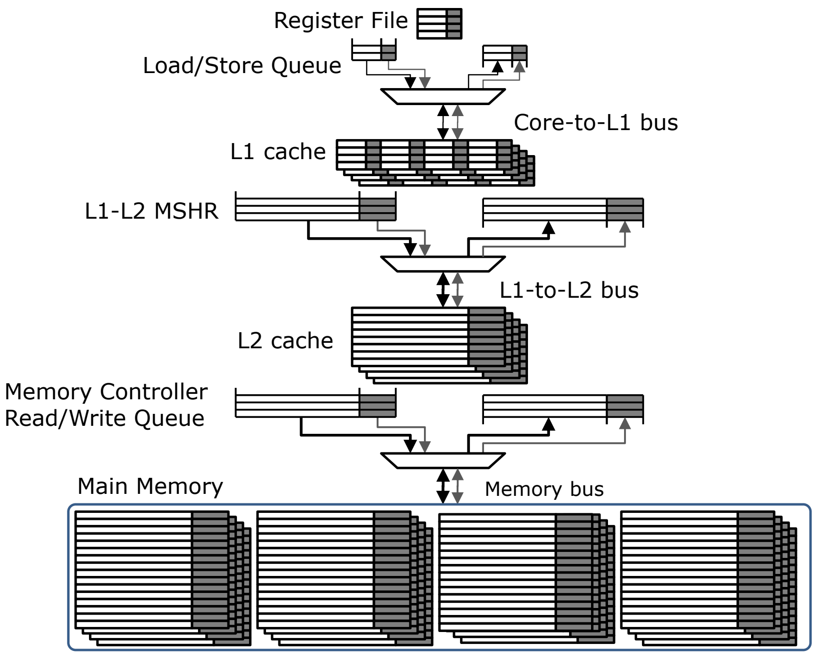

Figure 1: The memory hierarchy with uniform ECC; gray color denotes storage and interconnections dedicated to redundant information. Note that ECC is applied at a finer granularity in a register file and an L1 cache but that L2 and main memory have ECC per data line. Though not shown, lower storage levels such as Flash memory based disks or disk caches and hard-disk drives also have uniform ECC, but at a coarser granularity; e.g., 4kB data blocks in NAND Flash memory

their relatively low error rate.

Soft and hard errors can also cause more coarse-grained failures at the system level. Rare errors may be detected by the interconnection network fabric but which cannot be hidden from other layers of the system. Additionally, entire nodes may become non-responsive because of power failures or intermittent errors at the interface or runtime system. Note that we do not discuss file system failures or higher level network end-to-end schemes in this report.

3 Detection Mechanisms for Memory

A common solution to address memory errors is to apply error checking and correcting (ECC) codes uniformly across all memory locations; *uniform ECC*. Figure 1 illustrates an example memory hierarchy with uniform ECC. In uniform ECC, additional storage and interconnection wires are dedicated to storing and transferring redundant information at every level, and even intermediate buffers such as MSHR (miss status handling register) and read/write queues in a memory controller have uniform ECC codes. We start by describing commonly used ECC codes in Section 3.1, then briefly review cache memory protection in Section 3.2 and main memory protection in Section 3.3.

Table 2: ECC storage array overheads [11].

	SEC-DED		SNC-DND		DEC-TED	
Data bits	check bits	overhead	check bits	overhead	check bits	overhead
16	6	38%	12	75%	11	69%
32	7	22%	12	38%	13	41%
64	8	13%	14	22%	15	23%
128	9	7%	16	13%	17	13%

3.1 Information Redundancy

Typically, error-detection only codes are simple parity codes, while the most common ECCs use Hamming [2] or Hsiao [3] codes that provide single-bit-error-correction and double-bit-error-detection (SEC-DED).

When greater error detection is necessary, double-bit-error-correcting and triple-bit-error-detecting (DEC-TED) codes [4], single-nibble-error-correcting and double-nibble-error-detecting (SNC-DND) codes [5], and Reed Solomon (RS) codes [6] have also been proposed. DEC-TED and SEC-DED are a special case of BCH (Bose-Chaudhuri-Hocquenghem) code [7, 8] that detects and corrects random bit errors, and SNC-DND and RS codes are symbol based error codes. Such complex error codes, however, increase the overheads of ECC circuits and storage rapidly as correction capability is increased [9, 10]. Hence, parity and SEC-DED codes are used in cache memories for low-latency decoding while symbol-based error codes are mostly used in main memory and disk systems. Table 2 compares the overhead of various ECC schemes. Note that the relative ECC overhead decreases as data size increases.

3.2 Cache Memory Error Protection

In cache memory, different error codes are used based on cache levels and write-policy (write through or write-back). If the first-level cache (L1) is write through and the LLC (Last Level Cache; for example, L2 cache) is inclusive, it is sufficient to provide only error detection on the L1 data array because the data is replicated in L2. Then, if an error is detected in L1, error correction is done by invalidating the erroneous L1 cache line and re-fetching the cache line from L2. Such an approach is used in the SUN UltraSPARC-T2 [12] and IBM Power 4 [13] processors. The L2 cache is protected by ECC, and because L1 is write-through, the granularity of updating the ECC in L2 must be as small as a single word. For instance, the UltraSPARC-T2 uses a 7-bit SEC-DED code for every 32 bits of data in L2, an ECC overhead of 22%.

If L1 is write-back (WB), then L2 accesses are at the granularity of a full L1 cache line. Hence, the granularity of ECC can be much larger, reducing ECC overhead. The Intel Itanium processor, for example, uses a 10-bit SEC-DED code that protects 256 bits of data [14] with an ECC overhead of only 5%. Other processors, however, use smaller ECC granularity even with L1 write-back caches to provide higher error correction capabilities. The AMD Athlon [15] and Opteron [16] processors, as well as the DEC Alpha 21264 [17], interleave eight 8-bit SEC-DED codes for every 64-byte cache line to tolerate more errors per line at a cost of 12.5% additional overhead.

Recent research on low-power caches uses strong multi-bit error correction capabilities to tolerate failures due to reduced margin. This includes low-V_{CC} caches as well as reduced-refresh-rate embedded DRAM caches. Word disabling and bit fix [18] tradeoff cache capacity for reliability in low-V_{CC} operation. These techniques result in 50% and 25% capacity reductions, respectively. Multi-bit Segmented ECC (MS-ECC) [19] uses Orthogonal Latin Square Codes (OLSC) [20] that can tolerate both faulty bits in low-V_{CC} and soft errors, sacrificing 50% of cache capacity. Abella et al. [21] study performance predictability of low-V_{CC} cache designs using subblock disabling. Wilkerson et al. [22] suggest Hi-ECC, a technique that incorporates multi-bit error-correcting codes to reduce refresh rate of embedded DRAM caches. Hi-ECC implements a fast decoder for common-case single-bit-error correction and a slow decoder for uncommon-case multi-bit-error correction.

3.3 Main Memory Error Protection

Today's computer systems opt to use commodity DRAM devices and modules in main memory. Hence, main memory error protection uses DRAM modules that can store redundant information and apply ECC to detect and correct errors. This *ECC DIMM* (dual in-line memory module) requires a larger number of DRAM chips and I/O pins than a non-ECC DIMM.

Typically, an ECC DIMM is used to provide SEC-DED for each DRAM rank, and do so without impacting memory system performance. The SEC-DED code [2, 3] uses 8 bits of ECC to protect 64 bits of data. To do so, an ECC DIMM with a 72-bit wide data path is used, where the additional DRAM chips are used to store both the data and the redundant information. An ECC DIMM is constructed using 18 $\times 4$ chips ($\times 4$ ECC DIMM) or 9 $\times 8$ chips ($\times 8$ ECC DIMM). Note that an ECC DIMM only provides additional storage for redundant information, but that actual error detection/correction takes place at the memory controller, yielding the decision of error protection mechanism to system designers.

A recent study, however, shows memory chip failures, possibly due to packaging and global circuit issues, cause significant downtime in datacenters [23]. Hence, business critical servers and datacenters demand *chipkill-correct* level reliability, where a DIMM is required to function even when an entire chip in it fails. Chipkill-correct "spreads" a DRAM access across multiple chips and uses a wide ECC to allow strong error tolerance [24, 12, 25]. The error code for chipkill-correct is a single-symbol-error-correcting and double-symbol-error-detecting (SSC-DSD) code. It uses *Galois Field* (GF) arithmetic [4] with b-bit symbols to tolerate up to an entire chip failing in a memory system. The 3-check-symbol error code [5] is a special case of RS code and the most efficient SSC-DSD code in terms of redundancy overhead. The code-word length of the 3-check-symbol code is,

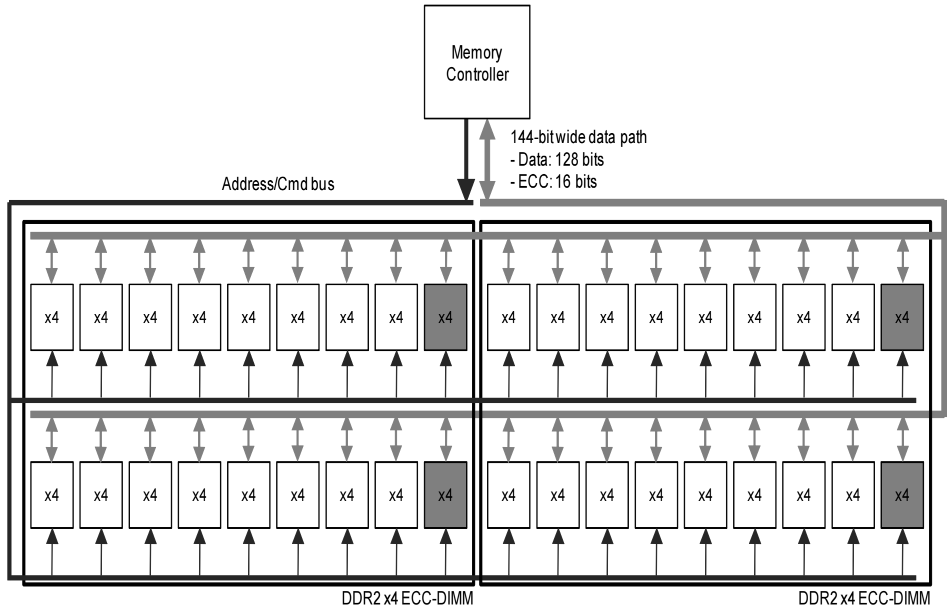

Figure 2: Baseline chipkill correct DRAM configuration (gray DRAMs are dedicated to ECC storage).

however, limited to $2^b + 2$ symbols, so it is a poor match for $\times 4$ configuration; using $\times 4$ DRAMs leads to a granularity mismatch that results in data words that are 60 bits long - a non power of two (3 4-bit check symbols can correct a symbol error in 15 data symbols). Instead, SNC-DND code [26] with 4 check symbols is used for $\times 4$ DRAMs, where the 4^{th} check symbol allows a longer code-word. Four 4-bit check symbols provide SSC-DSD protection for 32 4-bit data symbols, resulting in an access granularity of 128 bits of data with 16 bits of redundant information; this wide data-path is implemented using two $\times 4$ ECC DIMMs in parallel as shown in Figure 2. This organization is used by the Sun UltraSPARC-T1/T2 [12] and the AMD Opteron [25]. This chipkill memory system works well with DDR2 using minimum burst of 4; the minimum access granularity is 64B (4 transfers of 128bits). It is, however, problematic with DDR3 or future memory systems; longer burst combined with a wide data path for chipkill-correct ECC codes leads to larger access granularity [27].

4 Detection Mechanisms for Compute

Error detection mechanisms can be categorized in number of ways depending on the criteria. They can be classified into space-redundant or time-redundant techniques, or they can be classified based on the type of errors they can detect. Here, we will differentiate them based on the system level hierarchy they are implemented at. At the lowest level, there are circuit-level techniques; then, a technique can be implemented one level up in architecture. Next, a software system can be introduced to handle the detection, and application itself can be in charge of detection. Finally, there are hybrid techniques, which mix multiple of these to achieve higher efficiency. We will

Table 3: Comparison of circuit-level detection techniques. Overheads quoted from papers and not yet brought into Echelon's context.

	Hardening		**Hardening heuristics**			**Circuit monitoring**	
	Hazucha et al. [29]	Lunardini et al. [30]	Mohanram et al. [31]	Rao et al. [32]	Zoellin et al. [33]	Ndai et al. [34]	Narsale et al. [35]
Mechanism	Redundant transistors	Gate resizing	Partial duplication	Gate resizing and flip-flip selection	Selective gate resizing	Current mirror	Supply rail monitor
Error coverage	High	High	Configurable	Low	Configurable	High	High
Performance overhead	None	-50%, Faster due to bigger transistors	None	None	None	Configurable	Negligible
Power overhead	30-40%	400%	Depends on error coverage	Minimal	Depends on error coverage	Depends on performance overhead	20%
Area overhead	40%	100%	Depends on error coverage	5%	Depends on error coverage (10-50%)	Unknown	20%

discuss each technique with respect to its cost and types of errors it covers.

4.1 Circuit-level Techniques

Fault-tolerance and redundancy can be introduced at design-time with little effect on the overall architecture. These techniques are attractive when dealing with small parts of the design, or when some amount of redundancy is already present for other reasons (one example is scan-chains that are used for testing) [28]. This type of circuit is sometimes referred to as a *hardened circuit* as many were originally designed for high-radiation environments. Most commonly, these designs use latches based on multiple flip-flops and possibly special logic circuits with built-in verification. To the best of our knowledge, hardened designs typically require roughly twice the area and a longer clock-cycle than an equivalent conventional circuit [29, 30]. Because of this high and fixed overhead, there has been some work in providing sufficient error coverage through the logic delay timing slack and vulnerability-proportional hardening of components for the cost-effective error resiliency of mainstream designs [31, 32, 33]. These heuristic approaches report very low area and delay overheads given a target error coverage of less than 100%. However, no heuristic approach is able to provide complete error coverage while requiring less than twice the area.

There are other classes of error detecting circuitry aiming at reducing the overhead of hardening or replicating. They usually monitor either switching current [34] or supply voltage [35] to tell if there is an unexpected event. Circuit monitoring techniques provide high error coverage with modest overhead; however, real-world problems like process variation and supply voltage droop complicate the actual implementation of such mechanisms. Table 3 compares various circuit-level error detection techniques. Note that some of the techniques are configurable so that trade-offs can be made between different metrics.

4.2 Architecture-level Techniques

In order to detect errors, some degree of redundancy must be introduced in the architecture. Code-based techniques operate by providing a redundant representation of numbers with the property that certain errors can be detected and sometimes corrected through the analysis and handling of the resulting erroneous number. Code-based techniques offer several distinct advantages over alternative strategies for protecting computation—they run *concurrently*, generally detecting and reporting errors online with minimal latency, and they operate through *selective redundancy*, requiring only a fractional increase in area to provide error coverage. The amount of error coverage provided by a code-based technique is rarely complete, but is often quantifiable and may be tuned to the system requirements to provide low-cost error detection for a target failure rate. When code-based techniques are not applicable, or require too much custom design, execution redundancy is the most common architectural alternative. Through the use of redundant execution at the module level, errors can be detected with very high coverage and little design cost. Execution redundancy usually has high fixed overhead close to 100%, either in space or time.

4.2.1 Code-based Techniques

While the regular structure of memory arrays enable the efficient protection through parity-based codes or communication codes, these error-correcting codes are not ideally suited for arithmetic operations. AN codes and residue codes are the most well known examples of error codes which are designed to detect and correct errors which occur during the processing of integer arithmetic operations. *AN codes*, also known as *linear residue codes*, *product codes*, and *residue-class codes* [36], represent a given integer N by its product with a constant A. Therefore, the addition of two numbers $N_1 + N_2$ can be checked by testing the equality of Equation 1. Variants which work under other operations of interest exist [37]; error detection (and perhaps correction) is applied at the functional unit granularity, as there is no separability between the coded circuitry and the circuitry which performs the original operation.

$$A * N_1 + A * N_2 \stackrel{?}{=} A * (N_1 + N_2) \tag{1}$$

A class of arithmetic error codes called *residue codes* is largely equivalent to AN codes, but has significant practical implementation advantages [36]. Figure 3 shows an overview of the error detection process using residue codes. Most arithmetic operations can be checked by testing the equality of Equation 2, where $|N|_A = N \bmod A$ and $\oplus$ is the operation of interest. If both sides of Equation 2 are equal, it is likely that no error has occurred. If both sides are not equal, then some error *has* occurred. Residue codes are more flexible than AN codes–a single residue checker can detect errors in numerous operations–and provide separability between the circuitry that performs the original computation and that checks the computation. This separation simplifies implementation, reduces the intrusiveness of designs, and can make it easier to detect errors without impacting the delay of the original circuit.

$$|N_1 \oplus N_2|_A \overset{?}{=} ||N_1|_A \oplus |N_2|_A|_A \tag{2}$$

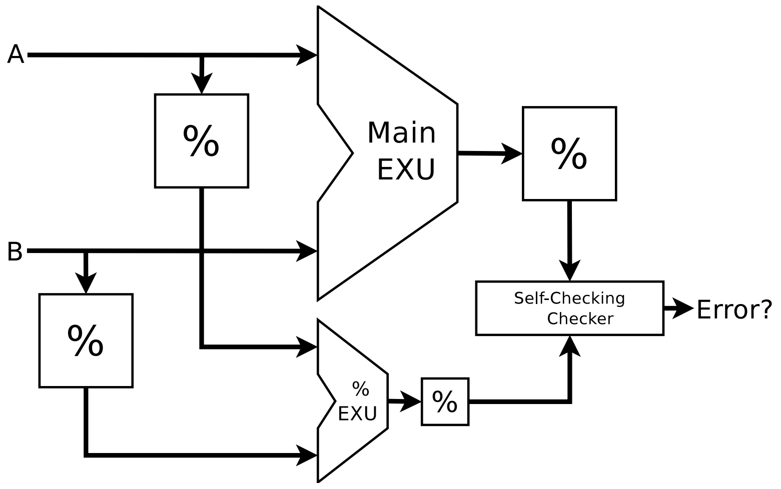

Figure 3: An overview of the residue code error detection process.

The arithmetic error control codes are especially useful because they are preserved under arithmetic operations. While non-arithmetic error codes are not, they may also be applied to integer operations through a process called *check prediction*. Parity codes [38], checksum codes [39], and Berger codes [40] have all been successfully applied to protect computer arithmetic.

There is no known direct check prediction or error coding method for floating-point arithmetic. However, more intrusive methods of error detection exist which use residue checking or Burger check prediction in a piecewise fashion within the floating-point unit to provide resiliency [39]. These methods, while intrusive and requiring custom design, report low area overheads.

4.2.2 Execution Redundancy

Code-based techniques are cost-effective; however, they require custom design and are relatively inflexible for covering errors in a variety of hardware structures. In the general case, a viable option is to replicate the execution of some logic and compare the results. At the architecture level, hardware components can be replicated at varying granularity from a single module to an entire core. [41] suggests that a simple *checker* module can be used to detect errors in execution. Further analyses show that the hardware costs are modest and that performance degradation is low. While being a promising design point for complex modern control-intensive superscalar processors, this method is not applicable to compute-intensive architectures. The reason is that the main computational engine is in essence as simple as the suggested checker, and the overall scheme closely resembles full hardware replication of a large portion of the processor as done in the lockstepped IBM G5 processor [42].

Table 4: Comparison of architecture-level detection techniques. Overheads quoted from papers and not yet brought into Echelon's context.

	Pipeline manipulaion		**Hardware replication**		**Redundant multithreading (RMT)**	**Chip Multiprocessor RMT**
	SITR [43]	RazorII [44]	DIVA [41]	Lockstepped pipeline [42]	[45, 46, 47, 48]	[50, 51]
Mechanism	Time redundancy	Detecting circuitry	Partial replication	Full replication	Time redundancy	Space and time redundancy
Error types	Transient, intermittent	Transient, intermittent	All	Hard, transient	Transient	Hard, transient
Performance Overhead	Low	Low	Low	Very high (> 2.0X)	High ($1.5 - 2.0$X)	Modest (< 1.5X)
Energy Overhead	Low	Low	Low	Very high (> 2.0X)	High ($1.5 - 2.0$X)	Modest (< 1.5X)
Area Overhead	Low	Low	Low	High	None	None

Instead of replicating an architectural module, small augmentations can be made to the processor pipeline [43, 44]. In [43], pipelined functional units perform two redundant waves of computation consecutively. This is done by holding the value of input latch for two cycles. By comparing the results at the end of the pipeline, both transient and intermittent errors are detected with relatively low overhead. [44] presents a specially designed flip-flop for pipeline registers. The proposed flip-flop detects spurious transitions and generates error signals that trigger the architectural replay mechanism for recovery.[1]

A different set of techniques relies on the fact that control-intensive processor execution resources are often idle and utilizes them for time-redundant execution that can be initiated by the microarchitecture [45, 46, 47, 48, 49]. In compute-intensive processors however, resources are rarely idle, and therefore these schemes are not directly applicable. Unlike the time redundant multithreading techniques mentioned above, similar approaches can be taken in the space domain. In [50, 51], two (or more) copies of the program/thread run concurrently on a chip multi-processor. Again, hardware can be introduced to reduce software overhead for initiation and comparison [50], and efficient comparison is an important issue as discussed in [52]. Table 4 summarizes various architecture-level techniques discussed so far.

Most of the techniques described above are designed for control-intensive processors, and thus their efficiency in compute-intensive architectures like Echelon is limited. We are currently evaluating an alternative solution for compute-intensive processors which we call duplex execution. Duplex execution uses redundant execution units to run the same computation twice. The difference between this and previous approaches is that in duplex execution only the execution units are replicated rather than the entire pipeline. Given the trend that the data movement is consuming more power than the actual computation, duplex execution can save energy by only reading and writing data once while computing twice with them.

[1]Even though the detection is done at circuit-level, the technique in its entirety is discussed in the context of processor pipeline, hence belonging to this section.

Table 5: Comparison of detection techniques in software system. Overheads quoted from papers and not yet brought into Echelon's context.

	TIMA [55]	**ED^{4}I [54]**	**SWIFT [56]**	**SWAT [61]**	**Shoestring [62]**
Replication	Instruction	Instruction	Instruction	None	Some instruction
Control flow check	Yes	No	Yes	Implicit	Implicit
Error types	Hard, transient	All	Hard, transient	All	All
Error coverage	High	High	High	Low	Low
Performance Overhead	200%	80%	41%	5-14%	16%
Energy Overhead	Roughly the same as performance overhead				

4.3 Software Systems

The main advantage of implementing error detection mechanisms in software is that it is not intrusive to the design and can be applied to most systems without modifying the underlying hardware structures. The most straightforward approaches in software have been replication and re-execution. Several automated frameworks have been developed in this context ranging from intelligent full re-execution [53] to compiler insertion of replicated instructions and checks [54, 55, 56]. There exist a number of mechanisms for the low-cost detection of control errors [57, 58, 59, 60]; the aforementioned software-only resiliency frameworks devote much of their attention to control flow checking, which is not a high priority for compute-intensive architectures.

Recently, another class of error detection techniques has been proposed that relies on software level symptoms to detect errors. These techniques impose very little performance overhead as compared to the replication based techniques. However, this advantage comes at the cost of lower error coverage. In [61], symptoms such as fatal traps or application aborts are used to identify both hardware faults and transient errors, and compiler-inserted range-based invariants are used to detect silent data corruption that escapes those symptom checks. A similar symptom based approach is presented in [62]. To increase the error coverage, however, compiler analysis is performed to identify more vulnerable instructions, and these instructions are further protected with instruction duplication. A comparison of the different techniques discussed above is given in Table 5. The symptom based nature of SWAT and Shoestring gives protection against all types of errors whereas other instruction replication techniques do not detect intermittent errors except for ED^{4}I. ED^{4}I introduces data diversity in redundant copies of the program providing limited protection against intermittent errors.

4.4 Application-level Techniques

The most comprehensive information about the application is available at this level, enabling a much more efficient detection than other techniques. However, it might not always be possible for the programmer to find a good application-level technique for a given application. Also, error coverage is usually lower than other techniques based on more aggressive redundancy.

4.4.1 Algorithmic Based Fault Tolerance (ABFT)

Algorithmic-based checking allows for cost-effective fault tolerance by embedding a tailored checking, and possibly correcting, scheme within the algorithm to be performed. It relies on a modified form of the algorithm that operates on redundantly encoded data, and that can decode the results to check for errors which might have occurred during execution. Since the redundancy coding is tailored to a specific algorithm, various trade-offs between accuracy and cost can be made by the user [63, 64]. Therein also lies this technique's main weakness as it is not applicable to arbitrary programs and requires time-consuming algorithm development. In the case of linear algorithms amenable to compiler analysis, an automatic technique for ABFT synthesis was introduced in [63]. ABFT enabled algorithms have been developed for various applications including linear algebra operations such as matrix multiplication [65, 66] and QR decomposition [67] as well as the compiler synthesis approach mentioned above, FFT [68], and multi-grid methods [69]. A full description of the actual ABFT techniques is beyond the scope of this paper. It should be mentioned that the finite precision of actual computations adds some complication to these algorithms but can be dealt with in the majority of cases.

4.4.2 Assertion and Sanity-Based Fault Tolerance

A less systematic approach to software fault detection, which still relies on specific knowledge of the algorithm and program, is to have the programmer annotate the code with assertions and invariants [70, 71, 72]. Although it is difficult to analyze the effectiveness of this technique in the general case, it has been shown to provide high error-coverage at very low cost.

An interesting specific case of an assertion is to specify a few sanity checks and make sure the result of the computation is reasonable. An example might be to check whether energy is conserved in a physical system simulation. This technique is very simple to implement, does not degrade performance, and is often extremely effective. In fact, it is probably the most common technique employed by users when running programs on cluster machines and grids [73].

As in the case of ABFT, when the programmer knows these techniques will be effective, they are most likely the least costly and can be used without employing any hardware methods.

4.5 Hybrid Techniques

Most resiliency schemes focus on one or more of the aforementioned hardware or software techniques to achieve a target error coverage. However, despite the number and variety of existing techniques, the resiliency design space remains relatively sparse—achieving high error coverage either takes prohibitively much area and power, or incurs a heavy performance overhead. In addition, not all workloads demand the same amount of error tolerance. Hybrid software-hardware resiliency schemes, where software mechanisms operate with some architectural support, offer a cost effective way to explore the spatial and time-based dimensions of the design space. Hybrid techniques also can give the flexibility to dynamically tradeoff reliability and performance to best suit an application's needs.

Relax [74] uses try/catch like semantics to provide reliability though a cooperative hardware-software approach. Relax relies on low-latency hardware error detection capabilities while software handles state preservation and restoration. The programmer uses the Relax framework to declare a block of instructions as "relaxed". It is the obligation of the compiler to ensure that a relaxed code block can be re-executed or discarded upon a failure. As a result, hardware can relax the safety margin (e.g., frequency or voltage) to improve performance or save energy, and the programmer can tune which block of codes are relaxed and how the recovery is done.

FaulTM [75] is another research project which uses transactional semantics for reliability. FaulTM uses hardware transactional memory with lazy conflict detection and lazy data versioning to provide hybrid hardware-software fault-tolerance. While the programmer declares a *vulnerable* block (similar to transactional memories and Relax), lazy transactional memory (in hardware) enables state preservation and restoration of a user-defined-block. FaulTM duplicates a vulnerable block across two different cores for reliable execution.

CRAFT [76] is a hybrid approach which combines the software-only approach of replicated instructions and checks [56] with some time redundant multithreading-style hardware support in order to achieve higher error coverage and slightly improved performance [46, 51]. By taking a hybrid approach, CRAFT achieves better reliability and performance than the software-only approach while requiring less additional area than time redundant multithreading. Performance is still degraded to a large degree compared to aggressive hardware-based resiliency approaches, however.

Argus [77] also takes a hybrid approach for control protection. Argus compiler generates static control/data flow graph, and this information is inserted into the instruction stream as basic block signatures. At runtime, hardware modules generate dynamic control/data flow graph and perform comparisons against the static information passed from the compiler. While this provides an economic way of protecting control, computation must also be protected in order to avoid silent data corruption. Argus employs previously suggested techniques such as modulo checker for protecting the ALU and Multiplier/Divider.

5 System-Level Detection Mechanisms

Detection mechanisms at higher levels in the system hierarchy encapsulate those at the single core level and the entire system level. Potentially, another layer of hierarchy can be inserted by grouping a certain number of cores together for management purposes, but we will touch upon such an organization when we discuss detection mechanisms at the system level.

5.1 Detection at the Core Level

One implementation of detection at the core level is utilized in the IBM z990 processor [78]. It integrates multiple hardware techniques discussed thus far in Sections 3 and 4 together to create a fault-tolerant core with the most efficient detection mechanisms for different parts of the system. Overall, the techniques used in IBM z990 are ECC, parity, retry (re-execution), mirroring (hardware duplication), checkpointing, and rollback.

In IBM z990, ECC and parity are the main choice of detection mechanisms for the components of the memory hierarchy. The main memory is protected by 2-bit symbols. Furthermore, there is extra ECC to detect hard and soft errors on the address lines. L2 caches are again protected by ECC, which allows purging, cleaning, and/or invalidation of data as necessary. Moreover, the system keeps track of persistent errors in the cache, and if one exists, it shuts down the cache line causing the error. Simultaneously, the L2 pipeline is checked against errors by parity bits placed in each stage of the pipeline. In case of repeating errors, the system turns off the entire core. Other SRAMs and register files are similarly protected by ECC and parity. Finally, the memory address and command interface is covered by parity with re-execution of the memory command in cases of failure.

The datapath and the surrounding logic also benefit from ECC and parity; however, other more suitable techniques exist for these parts of the IBM z990. Logic in the pipeline is mirrored and checkpointed. The results of the duplicated hardware are compared against each other, and the core returns to the checkpointed state if the results do not match. Similarly, fetch data bus, I/O buses, and the store address stack are protected by parity with recovery through checkpointing and rollback. If the error persists, the entire core is turned off. Furthermore, to create an even more robust system, the checkpoint arrays themselves are protected by ECC as a second layer of protection. Control signals in the pipeline are protected by ECC in each stage, and the progatation of this ECC data is checked with parity bits. I/O operations are also covered by parity with re-execution on errors. Finally, ECC is recommended for off-chip address and control signals in SMPs.

5.2 Detection at the System Level

5.2.1 Detecting Network Failures

In [79, 80], network communications are protected by having strong error detection on all data paths and structures. ECC is used to protect memories and data paths. Network packet transfers are protected with cycle redundancy checks (CRC). The network provides a 16-bit packet CRC, which protects up to 64-byte of data and the associated headers (768 bits max). The receiving link checks the CRC as a packet arrives, returning an error if it is incorrect. The CRC is also checked as a packet leaves each device, and as it transitions from the router to the NIC, enabling detection of errors occurring within the router core. Furthermore, many of these paths and structures also have error correction.

When an unrecoverable hardware error is detected, some form of notification is always generated. For errors in data payloads, the errors are reported directly to the client that requested the communication. This is usually done in the form of completion events where the error indication is included as part of that event. For severe errors that might affect the operation of the network, the operating system is also informed via an interrupt.

Errors in control information are more problematic in that the client information cannot be trusted when this occurs. So a direct report to the client is not always possible. Instead, the communication is usually dropped at the point of this kind of error. However, every communication on the network is tracked in various ways that always include hardware timeout mechanisms to report if the communication has been lost. These timeouts are also reported via the completion events. Again, severe errors are reported to the operating system via an interrupt if they are detected in hardware directly associated with a node. If the error is detected in hardware not associated with a particular node, the error is reported to the independent supervisory system (which uses a separate network and processors).

In addition, any such errors, either in payload or control information, are always reported at the point of occurrence, usually to the supervisory system. This reporting channel is intended to be used for maintenance purposes.

5.2.2 Detecting Node Failures

Node failures are usually detected by closely monitoring them for health [81]. The monitoring is accomplished by requiring an operating system thread on every node to increment a heartbeat counter that is checked by the independent supervisory system. This thread also verifies that all of the cores of a node are functional, at least with the ability to schedule and run the heartbeat thread. When the supervisory system detects a lack of heartbeat, a failure event is generated. Other nodes in the system may subscribe for that event so that they are notified of any particular node failure.

In addition, the job launch and control system maintains a control tree of communication connections between the nodes in a job. If any of these connections fail, the nodes at the far end of the connection are considered down. This causes the entire job to be torn down in [81]. However, it has been suggested that we could optionally trigger a notification to the job and a reconstruction of the control tree. The receipt of the above node failure notifications by the job launch and control system from the supervisory system can also optionally trigger this notification and reconstruction.

6 Conclusion

In this report, we enumerate diverse existing error detection mechanisms for memory, compute, and system. The error detection mechanisms are further classified based on their redundancy type, placement in the system hierarchy, and error type coverage. As a qualitative trade-off analysis, techniques in each category are explained in detail and compared to one another where applicable. It is shown that different techniques have different trade-offs in terms of performance, energy and area. This analysis should provide an important insight in achieving efficient resiliency within the Echelon system.

Acknowledgements

This research was funded in part by DARPA contract HR0011-10-9-0008. The authors also wish to greatfully acknowledge the input of Jinsuk Chung, Evgeni Krimer, Karthik Shankar, and Jongwook Sohn for helping formulate the ideas contained in this report.

References

[1] D. Ernst, N. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," 2003. [Online]. Available: http://www.eecs.umich.edu/~taustin/papers/MICRO36-Razor.pdf

[2] R. W. Hamming, "Error correcting and error detecting codes," *Bell System Technical J.*, vol. 29, pp. 147–160, Apr. 1950.

[3] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Research and Development*, vol. 14, pp. 395–301, 1970.

[4] S. Lin and D. J. C. Jr., *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[5] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Research and Development*, vol. 28, no. 2, pp. 124–134, Mar. 1984.

[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. for Industrial and Applied Math.*, vol. 8, pp. 300–304, Jun. 1960.

[7] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres (Paris)*, vol. 2, pp. 147–156, 1959.

[8] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, pp. 68–79, 1960.

[9] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proc. the 40th IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, Dec. 2007.

[10] D. Strukov, "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories," in *Proc. Asilomar Conf. Signals Systems and Computers*, October 2006.

[11] C. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Trans. Device and Materials Reliability*, vol. 5, pp. 397– 404, Sep. 2005. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1545899

[12] *OpenSPARC T2 System-On-Chip (SOC) Microarchitecture Specification*, Sun Microsystems Inc., May 2008.

[13] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. Research and Development*, vol. 46, no. 1, pp. 5–25, Jan. 2002.

[14] J. Wuu, D. Weiss, C. Morganti, and M. Dreesen, "The asynchronous 24MB on-chip level-3 cache for a dual-core Itanium®-family processor," in *Proc. the Int'l Solid-State Circuits Conf. (ISSCC)*, Feb. 2005.

[15] J. Huynh, *White Paper: The AMD Athlon MP Processor with 512KB L2 Cache*, May 2003.

[16] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, Mar.-Apr. 2003.

[17] D. E. Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, Jul. 1999.

[18] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Proc. the 35th Ann. Int'l Symp. Computer Architecture (ISCA)*, Jun. 2008. [Online]. Available: http://pages.cs.wisc.edu/~alaa/papers/isca08_vccmin.pdf

[19] Z. Chisti, A. R. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *Proc. the 42nd IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, Dec. 2009.

[20] M. Y. Hsiao, D. C. Bossen, and R. T. Chien, "Orthogonal latic square codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 390–394, Jul. 1970.

[21] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. Gonzalez, "Low V_{ccmin} fault-tolerant cache with highly predictable performance," in *Proc. the 42nd IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, Dec. 2009.

[22] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-L. Lu, "Reducing cache power with low-cost, multi-bit error correcting codes," in *Proc. the Ann. Int'l Symp. Computer Architecture (ISCA)*, Jun. 2010.

[23] B. Schroeder, E. Pinheiro, and W. Weber, "DRAM errors in the wild: a large-scale field study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. ACM, 2009, pp. 193–204. [Online]. Available: http://research.google.com/pubs/archive/35162.pdf

[24] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," IBM Microelectronics Division, Nov. 1997.

[25] AMD, "BIOS and kernel developer's guide for AMD NPT family 0Fh processors," Jul. 2007. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/32559.pdf

[26] C. L. Chen, "Symbol error correcting codes for memory applications," in *Proc. the 26th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS)*, Jun. 1996.

[27] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memmory interfaces," in *Proc. the Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2009.

[28] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *Computer*, vol. 38, no. 2, pp. 43–52, 2005.

[29] P. Hazucha, T. Karnik, S. W. B. Bloechel, J. T. J. Maiz, K. Soumyanath, G. Dermer, S. Narendra, V. De, and S. Borkar, "Measurements and analysis of SER tolerant latch in a 90 nm dual-Vt CMOS process," in *2003 IEEE Custom Integrated Circuits Conference*, September 2003, pp. 617–620.

[30] D. Lunardini, B. Narasimham, V. Ramachandran, V. Srinivasan, R. D. Schrimpf, and W. H. Robinson, "A performance comparison between hardened-by-design and conventional-design standard cells," in *2004 Workshop on Radiation Effects on Components and Systems, Radiation Hardening Techniques and New Developments*, September 2004.

[31] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," *International Test Conference, 2003. Proceedings. ITC 2003.*, pp. 893–901, 2003.

[32] R. Rao, D. Blaauw, and D. Sylvester, "Soft error reduction in combinational logic using gate resizing and flipflop selection," in *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, 2006, pp. 502 –509.

[33] C. G. Zoellin, H.-J. Wunderlich, I. Polian, and B. Becker, "Selective hardening in early design steps," *European Test Symposium, IEEE*, vol. 0, pp. 185–190, 2008.

[34] P. Ndai, A. Agarwal, Q. Chen, and K. Roy, "A soft error monitor using switching current detection," in *2005 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings*, 2005, pp. 185–190.

[35] A. Narsale and M. Huang, "Variation-tolerant hierarchical voltage monitoring circuit for soft error detection," in *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*. IEEE, 2009, pp. 799–805.

[36] T. R. N. Rao, *Error Coding for Arithmetic Processors*. Orlando, FL, USA: Academic Press, Inc., 1974.

[37] I. Proudler, "Idempotent an codes," in *Signal Processing Applications of Finite Field Mathematics, IEE Colloquium on*, Jun. 1989, pp. 8/1 –8/5.

[38] A. Avizienis, "Arithmetic algorithms for error-coded operands," *IEEE Transactions on Computers*, vol. C-22, no. 6, pp. 567 –572, 1973.

[39] J.-C. Lo, "Reliable floating-point arithmetic algorithms for error-coded operands," *Computers, IEEE Transactions on*, vol. 43, no. 4, pp. 400 –412, apr. 1994.

[40] J. Lo, S. Thanawastien, and T. Rao, "Concurrent error detection in arithmetic and logical operations using berger codes," in *Proceedings of 9th Symposium on Computer Arithmetic*, Sep. 1989, pp. 233 –240.

[41] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, 1999, pp. 196–207.

[42] T. J. Slegel, R. M. A. III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, 1999.

[43] E. Mizan, T. Amimeur, and M. Jacome, "Self-Imposed Temporal Redundancy: An Efficient Technique to Enhance the Reliability of Pipelined Functional Units," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 45–53.

[44] S. Das, C. Tokunaga, S. Pant, W. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw, "Razorii: In situ error detection and correction for pvt and ser tolerance," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 32–48, 2009. [Online]. Available: http://www.ece.ncsu.edu/asic/ece733/2009/docs/RazorII.pdf

[45] N. Saxena and E. McCluskey, "Dependable adaptive computing systems-the roar project," in *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, vol. 3. IEEE, 2002, pp. 2172–2177.

[46] S. Reinhardt and S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 25–36, 2000.

[47] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, 1999, p. 84.

[48] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 214–224.

[49] M. K. Qureshi, O. Mutlu, and Y. N. Patt, "Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, ser. DSN '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 434–443. [Online]. Available: http://dx.doi.org/10.1109/DSN.2005.62

[50] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 98–109.

[51] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. the Ann. Int'l Symp. Computer Architecture (ISCA)*. Published by the IEEE Computer Society, 2002, p. 0099. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2679&rep=rep1&type=pdf

[52] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Fingerprinting: bounding soft-error detection latency and bandwidth," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004, pp. 224–234.

[53] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak, "The design, analysis, and verification of the SIFT fault tolerant system," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 458–469.

[54] N. Oh, S. Mitra, and E. J. McCluskey, "ED^{4}I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, 2002.

[55] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, and M. Violante, "A software fault tolerance method for safety-critical systems: Effectiveness and drawbacks," in *SBCCI '02: Proceedings of the 15th symposium on Integrated circuits and systems design*, 2002.

[56] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," in *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, 2005, pp. 243–254.

[57] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, Jun. 1995, pp. 218 –227.

[58] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 111 –122, Mar. 2002.

[59] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, ser. DFT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 581–.

[60] R. Venkatasubramanian, J. Hayes, and B. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, 2003, pp. 137 – 143.

[61] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "SWAT: An Error Resilient System," in *the Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE-IV)*. Citeseer, 2008. [Online]. Available: http://rsim.cs.illinois.edu/Pubs/08SELSE-Li.pdf

[62] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010. [Online]. Available: http://www.eecs.umich.edu/~shoe/papers/sfeng-asplos10.pdf

[63] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 436–446, 1990.

[64] A. Al-Yamani, N. Oh, and E. McCluskey, "Performance evaluation of checksum-based ABFT," in *16th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01)*, San Francisco, California, USA, October 2001.

[65] K. H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518–528, 1984.

[66] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham, "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1132–1145, 1990.

[67] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, 1990.

[68] J.-Y. Jou and J. A. Abraham, "Fault-tolerant FFT networks," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 548–561, 1988.

[69] A. Mishra and P. Banerjee, "An algorithm-based error detection scheme for the multigrid method," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1089–1099, 2003.

[70] D. M. Andrews, "Using executable assertions for testing and fault tolerance," in *9th Fault-Tolerance Computing Symposium*, Madison, Wisconsin, USA, June 1979.

[71] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *1983 International Test Conference*, Philadelphia, Pennsylvania, USA, October 1983, pp. 622–628.

[72] M. Z. Rela, H. Madeira, and J. G. Silva, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," in *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, 1996, p. 394.

[73] J. M. Wozniak, A. Striegel, D. Salyers, and J. A. Izaguirre, "GIPSE: Streamlining the management of simulation on the grid," in *ANSS '05: Proceedings of the 38th Annual Symposium on Simulation*, 2005, pp. 130–137.

[74] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *Proc. the Ann. Int'l Symp. Computer Architecture (ISCA)*, 2010.

[75] G. Yalcin, O. Unsal, I. Hur, A. Cristal, and M. Valero, "FaulTM: Fault-tolerant using hardware transactional memory," in *Proc. the Workshop on Parallel Execution of Sequential Programs on Multi-Core Architecture (PESPMA)*, 2010.

[76] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*. Madison, Wisconsin, USA: IEEE Computer Society, 2005, pp. 148–159.

[77] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 2007, pp. 210–222.

[78] P. Meaney, S. Swaney, P. Sanda, and L. Spainhower, "IBM z990 soft error detection and recovery," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 419 – 427, sep. 2005.

[79] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. IEEE, 2010, pp. 83–87.

[80] F. Godfrey, "Resiliency features in the next generation Cray Gemini network," in *Cray User Group (CUG) 2010*, 2010.

[81] "ASCI Red Storm system overview and design specification," Internal Report, Sandia National Labs, 2003.