Fault Tolerance Techniques for the Merrimac Streaming Supercomputer

Mattan Erez Nuwan Jayasena Timothy J. Knight
William J. Dally

Department of Electrical Engineering
Stanford University
Stanford, CA 94305

ABSTRACT

As device scales shrink, higher transistor counts are available while soft-errors, even in logic, become a major concern. A new class of architectures, such as Merrimac and the IBM Cell, take advantage of the higher transistor count by exposing control, communication, and a large number of functional-units at the architectural level, thus achieving high performance and efficiency. This paper explores soft-error fault tolerance in the context of these compute-intensive architectures, which differ significantly from their control-intensive CPU counterparts. The main goal of the proposed schemes for Merrimac is to conserve the critical and costly off-chip bandwidth and on-chip storage resources, while maintaining high peak and sustained performance. We achieve this by allowing for reconfigurability and relying on programmer input. The processor is either run at full peak performance employing software fault-tolerance methods, or reduced performance with hardware redundancy. We present several methods, their analysis, and detailed case studies.

1. Introduction

Advances in semiconductor technology allow much higher performance levels on a single chip. At the same time, the ever-shrinking device dimensions and voltages have given rise to an increased problem of *soft errors*, which are transient faults caused by noise or radiation [36]. These errors must be dealt with not only at a system level, for which a large body of work exists, but also at the single-processor level. In this paper we explore these trends of increased performance and the need for greater reliability in the case of the Merrimac streaming processor. Merrimac [9], along with the IBM/Sony/Toshiba Cell [19] and other academic and industry processors [12, 16, 34, 4, 28], belongs to a new class of architectures that are *compute-intensive* and achieve high performance by reducing dynamic control, providing a large number of programmable functional units, and exposing low-level hardware communication to the programming system. These features differ significantly from modern general-purpose CPUs and offer opportunities for novel fault-tolerance schemes.

In this paper we will focus on soft-error fault-tolerance techniques within the Merrimac processor, as the details for other fault types and full system reliability follow common practice and the large knowledge base developed over many years of dealing with reliable high-performance computers. For the most part Merrimac follows a dynamic redundancy approach to fault-tolerance, and relies on robust fault-detection followed by re-execution for fault-correction. Therefore, the discussion concentrates on fault-detection methods, and only briefly touches on the mechanism for recovery. Strong fault-detection capabilities are particularly important and difficult for compute-intensive processors. The reason is that a vast majority of instructions are data-manipulation instructions and do not affect program control or make off-chip memory references. These types of instructions are not likely to lead to a catastrophic failure, where a violation is easy to detect (e.g., a segmentation fault), but rather remain a *silent error* that corrupts the computed result without producing a warning [25]. This is in contrast to control-intensive high-performance CPUs where a fault will many times quickly lead to a catastrophic failure, and other times be squashed in the speculative pipeline without even affecting the architectural state of the machine [41].

In order to detect soft-errors in execution, some degree of redundancy must be introduced in either hardware, software, or both. Building redundant hardware increases cost which adversely affects peak performance as the additional units consume die area that could have been used otherwise. Redundant instruction execution, on the other hand, places greater demands on the execution and bandwidth resources possibly leading to degraded sustained performance when a large number of redundant instructions are introduced.

The Merrimac reliability scheme uses a careful balance of hardware and software redundancy to ensure that the most vital resources are not squandered. For compute-intensive architectures such as Merrimac, careful management of the bandwidth hierarchy is crucial for achieving high utilization of the large number of arithmetic units available. Therefore, the fault-tolerance mechanisms stress conservation of the critical resources in this respect – off-chip bandwidth and on-chip memory capacity. These two resources are often responsible for limiting application performance [9, 10],

This work was supported, in part, by the Department of Energy ASCI Alliances Program, Contract LLNL-B523583, with Stanford University.

as compute resources are relatively cheap in modern VLSI technology (discussion in Section 2).

Careful attention is also paid to the amount of hardware overheads incurred in supporting fault-tolerance. Certain application classes contain information that allows sanity checks or algorithmic based techniques to be used to ensure correct execution with minimal performance impact and no additional hardware. In such cases, all available execution resources may be applied towards improving performance without loss of reliability. For other application classes, however, transient fault detection requires redundant execution. Therefore, the Merrimac system allows the programmer and/or the programming system to reconfigure the degree of hardware redundancy support depending on the availability of application-specific, efficient, software fault-detection methods.

Our analysis shows that in current 90nm technology, ECC protection of large arrays is enough to achieve reasonable failure rates and requires a small increase of only about 4% in chip area. However, in a future 50nm technology, using the same architectural unit mix, the computational clusters and interfaces will require coverage as well. Our proposed schemes will need up to 30% area overheads, but the ability to reconfigure the level of redundancy allows applications which can effectively be protected in software to suffer only small performance degradations.

Section 2 describes the Merrimac architecture in general, and is followed by a discussion of existing fault-detection methods and the degree in which they are suitable for Merrimac in Section 3. Section 4 explores fault-tolerance specifically for Merrimac and provides a general analysis of costs and performance, while a detailed treatment of case-studies highlighting the interplay between software and hardware techniques is presented in Section 5. We conclude the paper and suggest future work in Section 6.

2. Merrimac Processor Architecture

The Merrimac stream processor exploits the capabilities and constraints of modern VLSI and signaling technologies and matches them with application characteristics to achieve high performance at lower power and cost than today's state of the art systems. In a 90nm modern semiconductor fabrication process a 64-bit floating-point functional unit (FPU) requires an area of roughly 0.5mm^2 and consumes less than 50pJ per computation. Hundreds of such units can be placed on an economically sized chip making arithmetic almost free. Device scaling of future fabrication processes will make the relative cost of arithmetic even lower. On the other hand, the number of input/output pins available on a chip does not scale with fabrication technology making bandwidth the critical resource. Thus, the problem faced by architects is supplying a large number of functional units with data to perform useful computation.

In this section we present a detailed description of the base-line design of the Merrimac processor, as a good understanding of how it differs from conventional CPUs is critical to understand the new reliability trade-offs afforded. The full reliability details are deferred to Section 4. In addition, we will not describe the full Merrimac supercomputer system and refer the reader to [9].

Some of the key differences between Merrimac and a conventional CPU detailed in this section are listed below.

1. A large portion of the Merrimac die is devoted to functional unit logic and register-files.

2. Merrimac is tuned for arithmetic intensive scientific applications and does not support speculative execution. In addition all functional units are tightly scheduled by an optimizing compiler that is aware of all machine details. As a result a vast majority of instructions are actual floating-point computation instructions that can contribute to silent errors (in more technical terms, the *architectural vulnerability factor* (AVF) is close to 1 as explained in Subsection 4.1).

3. Merrimac contains no prediction tables and has very little micro-architectural state. That is, all storage arrays are architecturally exposed and must be protected against silent errors.

4. Most execution resources are rarely idle and never operate on speculative state, and as a result many of the recently suggested opportunistic fault-detection techniques cannot be applied.

2.1 Merrimac processor

The Merrimac processor is a stream processor that is specifically designed to take advantage of the high arithmetic-intensity and parallelism of many scientific applications. It contains a scalar core for performing control code and issuing stream instructions to the stream processing unit, as well as the DRAM and network interfaces on a single chip. Most of the chip area is devoted to the stream execution unit whose main components are a collection of arithmetic clusters and a deep *register hierarchy*. The compute clusters contain 64 64-bit FPUs and provide high compute performance, relying on the applications' parallelism. The register hierarchy exploits the applications' locality in order to reduce the distance an operand must travel thus reducing global bandwidth demands. The register hierarchy also serves as a data staging area for memory in order to hide the long memory and interconnection network latencies.

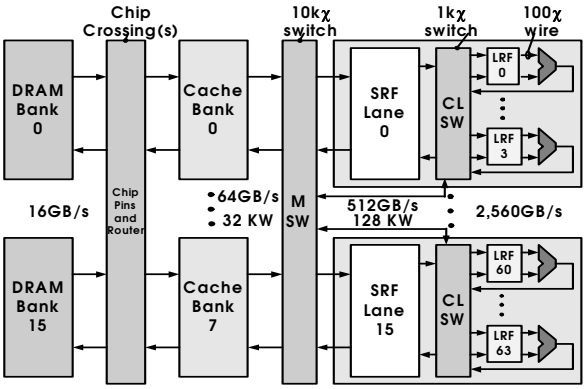

Figure 1: Architecture of Merrimac's stream core

As shown in Figure 1, Merrimac's stream architecture consists of an array of 16 clusters, each with a set of 4 64-bit multiply-add (MADD) FPUs, a set of *local register files* (LRFs) totaling 768 words per cluster, and a bank of the *stream register file* (SRF) of 8KWords to form a 1MB SRF for the entire chip. At the planned operating frequency of

1GHz each Merrimac processor has a peak performance of 128GFLOPS, where the functional units have a throughput of one multiply-add per cycle.

Each FPU in a cluster reads its operands out of an adjacent LRF over very short and dense wires. Therefore the LRF can provide operands at a very high bandwidth and low latency, sustaining 3 reads per cycle to each FPU. FPU results are distributed to the other LRFs in a cluster via the cluster switch over short wires, maintaining the high bandwidth required (1 operand per FPU every cycle). The combined LRFs of a single cluster, or possibly the entire chip, capture the *short term producer-consumer locality* of the application. This type of locality arises from the fact that the results of most operations are almost immediately consumed by subsequent operations, and can live entirely within the LRF. In order to provide instructions to the FPUs at a high rate, all clusters are run in *single instruction multiple data* (SIMD) fashion. Each cluster executes the same *very long instruction word* (VLIW) instruction, which supplies a unique instruction to each of the cluster's FPUs. Instructions are fetched from the on-chip micro-code store.

The second level of the register hierarchy is the stream register file (SRF), which is a software managed on-chip memory. The SRF provides higher capacity than the LRF, but at a reduced bandwidth of only 4 words per cycle for a cluster (compared to over 16 words per cycle on the LRFs). The SRF serves two purposes: capturing *long term producer-consumer locality* and serving as a data staging area for memory. Long term producer-consumer locality is similar to the short term locality but cannot be captured within the limited capacity LRF. The second, and perhaps more important, role of the SRF is to serve as a staging area for memory data transfers and allow the software to hide long memory latencies. An entire stream is transferred between the SRF and the memory with a single instruction. These stream memory operations generate a large number of memory references to fill the very deep pipeline between processor and memory, allowing memory bandwidth to be maintained in the presence of latency. FPUs are kept busy by overlapping the execution of arithmetic kernels with these stream memory operations. In addition, the SRF serves as a buffer between the unpredictable latencies of the memory system and interconnect, and the deterministic scheduling of the execution clusters. While the SRF is similar in size to a cache, SRF accesses are much less expensive than cache accesses because they are aligned and do not require a tag lookup. Each cluster accesses its own bank of the SRF over the short wires of the cluster switch. In contrast, accessing a cache requires a global communication over long wires that span the entire chip.

The final level of the register hierarchy is the inter-cluster switch which provides a mechanism for communication between the clusters, and which interfaces with the memory system which is described in the following subsection.

In order to take advantage of the stream architecture, the stream programming model is used to allow the user to express both the locality and parallelism available in the application. This is achieved by casting the computation as a collection of *streams* (i.e. sequences of identical data records) passing through a series of computational *kernels*. The semantics of applying a kernel to a stream are completely parallel, so that the computation of the kernel can be performed independently on all of its stream input elements and in any order. Thus *data level parallelism* is expressed and can be utilized by the compiler and hardware to drive the 64 FPUs on a single processor and the 1 million FPUs of an entire system. An additional level of *task parallelism* can be discerned from the pipelining of kernels. Streams and kernels also capture multiple levels and types of locality. Kernels encapsulate short term producer-consumer locality, or *kernel locality*, and allow efficient use of the LRFs. Streams capture long term producer-consumer locality in the transfer of data from one kernel to another through the SRF without requiring costly memory operations, as well as spatial locality by the nature of streams being a series of data records.

While the Merrimac processor has not been fabricated we have estimated an implementation of it based on a currently available 90nm fabrication process and a clock frequency of 1GHz (37 FO4 inverters in 90nm[35]). Each MADD unit measures 0.9mm $\times$ 0.6mm and the entire cluster measures 2.4mm $\times$ 1.6mm. Our area estimates for the baseline configuration with no fault tolerance is 144mm^2. The bulk of the chip is occupied by the 16 clusters while the second largest contributors to area are the high bandwidth external memory and network interfaces. Our current example design point for the scalar control calls for a MIPS64 20kc [22] scalar processor. The node memory system consists of a set of *address generators*, a line-interleaved eight-bank 64KWords (512KB) cache, and interfaces for 16 external Rambus DRDRAM chips. A network interface directs off-node memory references to the routers. We estimate that each Merrimac processor will cost about $200 to manufacture[1] and will dissipate a maximum of about 30W of power. Area and power estimates are for a standard cell process in 90nm technology and are derived from models based on a previous implementation of a stream processor [18, 17] and careful examination of modern commercial processor die photographs.

2.2 Memory system

The memory system of Merrimac is designed to support efficient stream memory operations. As mentioned in Subsection 2.1 a single stream memory operation transfers an entire stream, which is typically many thousands of words long, between memory and the SRF. Merrimac supports both strided access patterns and gathers/scatters through the use of the stream address generators. Each processor chip has 2 address generators, which together produce up to 8 single-word addresses every cycle. The address generators take a base address in memory for the stream, and either a fixed stride and record-length or a pointer to a stream of indices in the SRF. The memory system provides high-bandwidth access to a single global address space for up to 16,384 nodes. Each Merrimac chip has a 128KWords cache with a bandwidth of 8 words per cycle (64GB/s), and directly interfaces with the node's external DRAM and network. The 2GB of external DRAM is composed of 16 Rambus DRDRAM chips providing a peak bandwidth of 38.4GB/s and roughly 16GB/s, or 2 words per cycle, of random access bandwidth. Remote addresses are translated in hardware to network requests, and single word accesses are made via the interconnection network. The flexibility of the addressing modes, and the single-word remote memory access capability simplifies the software and eliminates the costly pack/unpack routines common to many

[1] Not accounting for development costs.

parallel architecture. The Merrimac memory system also supports floating-point and integer streaming add-and-store operations across multiple nodes at full cache bandwidth. This *scatter-add* operation performs an atomic summation of data addressed to a particular location instead of simply replacing the current value as new values arrive. This operation is used in the molecular dynamics application described, and is also common in finite element codes.

3. Existing Fault Tolerance Techniques

In this section we will describe existing techniques relating to soft-error fault detection. The failure mechanisms here are energetic particle strikes that cause hole-electron pairs to be generated, effectively injecting a momentary (< 1ns) pulse of current into a circuit node. Our fault model of this is a transient 1-cycle flip of a single bit in the circuit, which is also applicable in the case of supply noise briefly affecting a circuit's voltage level. For more detail, and for a discussion of the general fault-tolerance problem we refer the interested reader to [37] and [36].

As mentioned in the introduction fault-detection techniques can broadly be classified into software and hardware. We will briefly describe the main methods in each category as well as their strengths, weaknesses, and suitability for Merrimac. Note that many of the recently reported hardware techniques have been developed in the context of multi-threaded and multi-core CPUs and are not well-suited for compute-intensive architectures as discussed in Subsection 3.3.

3.1 Software Techniques

Software techniques range from algorithm specific modifications for fault-tolerance, through techniques relying on program structure and sanity of results, to full instruction replication.

3.1.1 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 redundancy encoded data, and that can decode the results to check for errors that 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 [7, 1]. 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 [7]. ABFT enabled algorithms have been developed for various applications including linear algebra operations such as matrix multiply [14, 8] and QR decomposition [30] as well as the compiler synthesis approach mentioned above, FFT [15], and multi-grid methods [23]. 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.

ABFT methods only apply to the actual computation and not control code, or data movement operations. This however fits well with the overall design of the Merrimac fault tolerance schemes as will be explained in Section 4, as it allows for efficient software fault-detection in conjunction with hardware detection when software algorithms are not available.

3.1.2 Assertion and Sanity-Based Fault Detection

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 [3, 21, 32]. 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 on cluster machines and grids [43].

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 in Merrimac without employing the hardware methods.

3.1.3 Instruction Replication Techniques

When the programmer cannot provide specialized fault-tolerance through the algorithm or assertions, the only remaining software option is to replicate instructions and re-execute the computation. Several automated frameworks have been developed in this context ranging from intelligent full re-execution [42] to compiler insertion of replicated instructions and checks [27, 29, 31]. These recent techniques devote much of their attention to control flow checking which is not necessary in the context of compute-intensive architectures. In Section 4.2 we discuss how similar, simpler, methods can be used within the Merrimac fault-detection scheme.

3.2 Hardware Techniques

Many hardware techniques have been introduced over the years to combat the problems of reliable execution in general. Even when limiting the discussion to transient errors, the amount of prior work is very large. The high-level overview below is not intended to summarize this vast body of knowledge, and only describes some of the techniques that are relevant to Merrimac. A more detailed treatment of several recent techniques can be found in [24]. A hardware fault-tolerant system will usually use a combination of multiple techniques, as shown in [38, 6, 2].

3.2.1 Code Based Fault-Tolerance

Arguably the most cost-effective and widely employed reliability technique is the protection of memory-arrays and buses using bit-redundancy. Typically, a set of code bits is added to a group of bits or bus-lines and uses a Hamming code to add redundancy. Examples include the common SEC-DED (single error correction – double error detection) code which requires 8 code bits for each 64 bit word, and simple 1-bit parity for small arrays and buses.

Merrimac's stream unit uses an SEC-DED code to pro-

tect the SRF, cache, micro-code store and global chip buses (Subsection 4.2.1).

3.2.2 Fault Tolerant Circuits

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) [24]. 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 [13, 20]. As we do not have a design of Merrimac at this point, we do not employ circuit level techniques. Such techniques may be applicable to Merrimac's external interfaces as mentioned in Subsection 4.2.3.

3.2.3 Replication of Hardware Units

Instead of replication at a circuit-level, entire structures can be replicated in the architecture or micro-architecture. This option is very useful when dealing with structures that consume modest die area, or are not visible to software. Good examples of employing these techniques are presented in [38, 6]. In Merrimac hardware replication is enabled through reconfigurability and a software-hardware hybrid approach (see Subsection 4.2.2).

3.3 Fault Tolerance for Superscalar Architectures

The Merrimac processor includes a scalar control-processor core, which can use some of the recently developed fault-tolerance techniques aimed at superscalar processors. We briefly reference a few such options and explain why they are not an appropriate choice for the compute-intensive stream unit.

[5] 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 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.

A different set of techniques relies on the fact that control-intensive processor execution resources are often idle, and utilize them for time-redundant execution that can be initiated by the micro-architecture [33]. In compute-intensive processors, resources are rarely idle and these schemes are not directly applicable.

Another fault-tolerance option is to concurrently run two (or more) copies of the program/thread on a chip multiprocessor. Again, hardware can be introduced to reduce software overhead for initiation and comparison [11]. Efficient comparison is an important issue [39] and its optimizations may apply to the scalar portion of the Merrimac processor.

4. Merrimac Fault Tolerance

In this section we detail the Merrimac fault-tolerance and detection mechanisms and perform a general analysis related to fault susceptibility, hardware costs, and impact on performance. A more detailed treatment of these analyses for specific case-studies appears in Section 5.

An overarching goal of the Merrimac architecture is to achieve high utilization of scarce resources such as off-chip bandwidth and on-chip memory capacity as emphasized in Section 2. These priorities pervade the fault-tolerance mechanisms of Merrimac as well. Hardware fault-tolerance mechanisms are favored where they can be supported without incurring high area overheads and eliminate potential performance or bandwidth bottlenecks. Where hardware mechanisms are costly, the flexibility to trade off performance and reliability is provided, particularly in cases where algorithmic/sanity techniques may be possible with low overheads.

We will first examine the fault susceptibility of the baseline architecture, and proceed by describing and discussing various options for fault-detection and recovery in Merrimac based on a dynamic redundancy scheme. ECC is used to enable large arrays and buses to mask a single fault (Subsection 4.2.1), but if two or more faults occur in a single word, or when an error is caught during execution in the compute clusters (Subsection 4.2.2), external interfaces (Subsection 4.2.3), or the scalar core (Subsection 4.2.4), the system will raise a fault-exception, roll back to a previously verified checkpoint, and restart execution (Subsection 4.2.5).

4.1 Fault Susceptibility

A full analysis of the architecture's susceptibility to faults is beyond the scope of this paper, as it requires a complete design as well as deep knowledge of the circuit style employed and foundry process parameters. Moreover, we hope that the research presented here is general and will be applicable to architectures and processors other than Merrimac as well, and therefore present and analyze multiple options and not just a single "optimal" point.

In order to understand the importance of good fault detection and the affect on performance, we will give an order-of-magnitude type analysis based on the observations of [36], which deal with trends of soft error rates (SER) for storage arrays and logic circuits, and [26] that discusses a methodology to refine a fault rate estimate based on the architectural vulnerability of the structure under consideration.

Table 1 lists the major components of the baseline Merrimac architecture (buses and scalar unit already include ECC based fault-detection) along with our area estimates, expressed in percentage of die area, the dominant circuit type, and an estimate of the *architectural vulnerability factor* (AVF). The AVF represents the likelihood that a fault in the structure will affect the outcome of a computation. For example, a fault in an invalid cache-line will not affect the result, so a cache will have an AVF < 1. The area estimates are based on extrapolating data gained from our experience of designing and fabricating the Imagine stream processor [17], as well as careful examination of commercial processor die-photos. Due to brevity considerations, we will not provide the full details of these and the AVF calculations.

Based on these numbers and the figures of [36], we report the expected soft-error rate (SER) in FIT/chip for both a current 90nm and a future 50nm technology in Table 2. We

also present the results in terms of contribution to total SER so that we can more clearly understand the effects of adding fault-tolerance techniques later in this section. Note that as we are assuming an existing CPU design as a scalar unit, we also assume SER rates typical of ECC protected arrays within the CPU (based on [36] and [37]). Even with architecture vulnerability taken into account we can expect a SER of about 500 FIT for a Merrimac chip in current technology, which amounts to a processor soft error every few days on a large supercomputer configuration with 16K nodes. Clearly fault tolerance is necessary and the Merrimac scheme will be described below.

Component	% of Die Area	Circuit Type	AVF
CPU	7%	mixed	0.1–0.2
FPUs	22%	logic	0.1–0.5
LRF	9%	SRAM	0.7
SRF	11%	SRAM	0.9
cache	9%	SRAM	0.8
AG	2%	logic	0.9
micro-controller	1%	SRAM	0.7
mem. interface	15%	mixed	0.9
net. interface	19%	mixed	0.7
global buses	5%	bus	0.6

Table 1: Area and AVF of major Merrimac components (based on total baseline area of 144mm^2 in 90nm ASIC technology)

Component	SER 90nm FIT	SER 50nm FIT
CPU	5 (1%)	10 (1%)
FPUs	5 (1%)	50 (7%)
LRF	40 (7%)	45 (6%)
SRF	270 (48%)	280 (38%)
cache	190 (34%)	200 (27%)
AG	1 (0%)	10 (1%)
micro-controller	40 (7%)	45 (6%)
mem. interface	5 (1%)	50 (7%)
net. interface	4 (1%)	40 (5%)
Total	560	730

Table 2: Expected SER contribution in current and future technologies with no fault-tolerance

4.2 Fault-Tolerance Scheme

Fault detection in the Merrimac architecture is tailored to the specific characteristics of each component. We will detail the schemes below and also provide an estimate of the hardware costs and impact on performance.

4.2.1 On-Chip Arrays and Buses

As can be seen in Table 2 large SRAM arrays are the main contributor to SER. Following standard practices we use SEC-DED ECC codes based fault-tolerance on all large arrays and buses. This includes the SRF, micro-code store, cache, and global chip buses such as the inter-cluster and memory switches. ECC techniques reduce the expected SER of these structure by a factor of 100 to over 1000 and require an area increase of roughly 10%. Analysis of the exact numbers is beyond the scope of this paper, and we will approximate the reduction in SER factor as 1000.

Although this area cost is not negligible, we are not aware of any more cost-effective mechanism for protecting these large arrays, and therefore require that ECC be used at all times. After adding ECC to the SRF and cache for example, their SER contribution drops from 48% to 0.45% and 34% to 0.31% respectively in 90nm technology, and the drop is even more dramatic in future processes.

There is little to no performance impact as error-detection can proceed concurrently with data movement.

4.2.2 Stream Unit Fault Detection

After protecting large arrays with ECC, the compute clusters are responsible for over 70% of the SER. In addition, we estimate that the stream execution unit, excluding the ECC protected SRF, occupies over 20% of the Merrimac processor die area. Therefore, replicating any significant portion of these resources leads to high area overheads, reducing the amount of compute clusters that can be accommodated on a fixed sized die. Such an overhead is particularly wasteful in cases where algorithmic fault detection techniques can be applied. Therefore, we explore a range of software-only and software/hardware hybrid techniques that can be selectively applied only in cases where algorithmic approaches are not applicable, and high reliability is desired. These techniques are discussed below.

We would like to emphasize that even if the LRF contribution were reduced using ECC or parity (a technique which may reduce the operating frequency and adds a significant amount of encoding/decoding logic), cluster logic will still account for almost 25% of the SER today and over 31% in 50nm technology. Moreover, the overall SER of a future chip without protecting logic will still be 160 FIT, equivalent to about two contiguous weeks of correct execution on a fully configured Merrimac supercomputer, just from soft faults in the processor chips. Therefore, we should look at techniques which are able to detect faults in both registers and logic.

We propose multiple techniques below, all of which effectively check for errors by performing the computation twice and comparing results at a granularity of either SRF or memory writes. Again, a full analysis of the SER reduction is beyond our means at this time, but based on a simple model given in [37] we estimate that this dual-modular redundancy approach with fine-grained checking should reduce SER by at least a factor of 1000, providing more than enough protection for this hardware unit, and bringing the chip SER to 15 FIT in 90nm and 100 FIT in 50nm. Unlike ECC on the memory arrays and buses which is able to correct single bit errors, faults detected in cluster logic are handled via the rollback technique described in Subsection 4.2.5.

Full Program Re-execution (FPR)

A straight-forward software-only technique is to execute entire applications twice and compare the final results. The main appeal of this technique is that it requires no changes to hardware or application binaries. However, effective system throughput for applications that require checking is reduced by more than half due to time spent in re-execution and result comparison.

Instruction Replication (IR)

An alternative software-only technique is to insert instructions at compile time to perform each computation twice and compare results. Unlike full program re-execution, this technique requires different versions of binaries for full throughput execution and fault-tolerant execution modes. However,

it has several advantages over full program re-execution. First, since scalar code and memory accesses are already protected through other mechanisms, redundant instructions need only be inserted in kernel code. Second, memory accesses are performed only once since only the computation is replicated. Third, on multiple-issue architectures, unused issue slots can be used to perform redundant computation and checking, reducing execution time relative to full program re-execution. However, the applications targeted by Merrimac have abundant parallelism, resulting in very high issue slot utilization [9, 16]. Therefore, on Merrimac, this technique results in essentially doubling kernel execution time. In addition, replicated computation within the instruction stream increases register pressure and instruction storage capacity requirements potentially leading to hardware overheads as well.

Kernel Re-execution (KR)

A third software-only technique is to re-execute each kernel twice in succession using the same SRF input streams, and compare the result streams. This enables much of the benefit of instruction replication for applications with high issue slot utilization. However, register pressure in LRFs and instruction storage requirement increases are avoided since comparisons are performed on streams in the SRF. Since the kernel is executed twice, the start-up and shut-down overheads must be performed two times as well. In addition, instructions must be inserted to first read the first kernel's result from the SRF and to compare with the newly calculated results. As a mismatch on the comparison signifies a fault there is no need to write the results a second time. This overhead can be eliminated through modest hardware support. During the second execution of a kernel the hardware will perform the comparison as it writes the second result, using the already existing stream-buffers and newly added comparators.

Mirrored Clusters (MC)

A hardware-software hybrid technique is to perform the same computation on a pair of compute clusters simultaneously and compare the stream writes. This mimics the behavior of replicated compute clusters, but allows the software to determine whether to duplicate execution for high reliability, or to use each cluster to perform an independent computation for high performance. Since the SRF is protected by ECC, it is not necessary to replicate SRF contents between mirrored clusters. Hence the entire SRF capacity is available even when clusters perform mirrored computation. Further, SRF reads are broadcast to both clusters in a mirrored pair, avoiding redundant SRF reads. On SRF writes, the results from mirrored cluster pairs are compared to detect transient errors. Figure 2 shows a simplified diagram of the read and write paths between a mirrored pair of clusters and the corresponding SRF banks. Therefore, only very modest additional hardware is required in the form of the communication paths for SRF reads between the two clusters and the result comparators.

Table 3 compares the performance and bandwidth overheads of the four options considered above for detecting transient faults in the compute clusters. For applications with no known efficient ABFT techniques, Merrimac implements the MC approach due to its minimal performance and bandwidth overheads relative to the other options considered. Where efficient ABFT techniques do exist, no redundant execution is performed in hardware, enabling high system throughput.

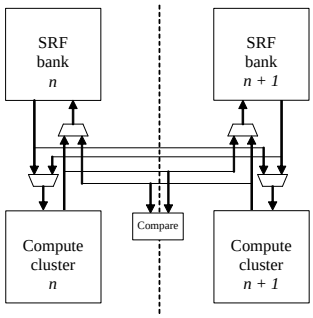

Figure 2: Mirrored cluster hardware arrangement

	FPR	IR	KR	MC
Requires recompilation		x		x
≈2x application execution time	x			
≈2x kernel execution time		x	x	x
≈2x memory system accesses	x			
≈2x SRF accesses	x		x	
Increased register pressure		x		

Table 3: Stream execution unit fault detection performance and bandwidth overheads

4.2.3 *External Interfaces*

The memory and network interfaces, including the address generators, also rely on hardware replication and circuit-based detection techniques. Note that applying software-based re-execution techniques at these units would double the off-chip bandwidth requirements. While the address generators are simple structures occupying only 2% of the chip area and can easily be replicated, the memory and network interfaces are significantly larger and more complex. However, given the design goal of avoiding redundant accesses at the performance-critical off-chip interfaces, we intend to fully replicate the logic portions of these units. Based on our current area estimates, this would increase the chip area by almost 20%. We are not aware of any research conducted on efficient fault-detection within these structures.

4.2.4 *Scalar Unit*

In the scalar processor, which is essentially a general-purpose CPU, fault-detection relies on one of the superscalar specific techniques mentioned in Section 3. The current specification is to use full hardware replication and checking of results between two independent cores, as described in [39], which should reduce the SER contribution of the scalar core by a factor of over 1000 at a cost of full replication. However, as the choice of a specific scalar core and design have not been finalized, the details of the scalar fault-detection scheme may change as well. It is also interesting to note that the scalar unit occupies a small portion of the total chip area, and therefor does not greatly contribute to the overall SER. It may be possible to use less expensive software-based techniques with no hardware replication to protect it, provided the slowdown in the control processor

does not dramatically reduce the performance of the stream unit under its control.

4.2.5 Recovery from Faults

The above techniques are aimed at detecting the occurrence of transient faults. Once a fault is detected, the system must recover to a pre-fault state. The ECC protecting the memories are capable of correcting single bit errors, and hence do not require system-level recovery. However, on all other detectable faults, recovery is based on checkpoints. Periodically, the system state is backed up on non-volatile storage. Once a fault is detected, the system state is restored from the last correct checkpoint. A simple calculation, that is not presented in this paper, can be performed to determine the optimal checkpoint interval and expected performance degradation due to re-execution. Our initial estimates point to an interval of several hours with less than 5% slow-down. Note that this a good solution for the Merrimac supercomputer as it is intended to run a single job over a long period of time.

4.2.6 Summary of Fault-Detection Scheme

Table 4 summarizes the estimated expected SER reduction, as well as area and performance overheads of the techniques described above. Each line in the table represents adding a technique to the design and the cumulative increase in area compared to the baseline described in Subsection 4.1.

Adding ECC to the large arrays (including the LRF) is the most cost-effective way to increase fault-tolerance. With an area overhead of 4% the SER is reduced by a factor of almost 30 down to 21 FIT in 90n, technology. However, this method will not be sufficient in future technologies, where we still expect an SER of 161 FIT arising from logic. The focus of this paper is on fault-tolerance for the compute clusters, and we see that with practically no area over-head we are able to improve SER by close to an additional 40% over ECC alone. The final two methods at our disposal are costly hardware replication of the external interfaces and scalar unit. While protecting the interfaces is necessary in 50nm, we can see that the scalar unit contributes only a small amount to the total SER as it consumes a small fraction of the total chip area.

Technique	Cumulative Area Overhead	Performance Impact	SER 90nm	SER 50nm
Baseline	0%	N/A	560	730
ECC on Arrays	4%	minimal	21	161
IR/KR/MC	4%	$\sim 2\times$ kernel exec. time	15	100
Interface Redundancy	23%	none	6	11
Scalar Redundancy	30%	none	0.6	0.7

Table 4: Summary of Fault-Detection Schemes

5. Case Studies

To demonstrate the trade-offs involved we now evaluate the techniques discussed in the previous section on two case study applications. The first example is of dense matrix-matrix multiplication that can employ a very efficient algorithmic based fault tolerance technique [14], and the second is a molecular dynamics application [40, 10] that has more dynamic behavior and for which no ABFT technique has been developed.

5.1 Methodology

We measured the performance of each application with all applicable reliability schemes using the Merrimac cycle-accurate stream simulator. The simulator reflects the Merrimac stream processor described in Section 2, and can be run at full performance with 16 clusters or in mirrored-cluster mode with only 8 active clusters accessing the entire SRF. As mentioned before, our experiments only measure the effect of adding reliability on performance and do not involve fault injection.

Full program re-execution runtime is measured by running the application twice and accounting for the comparison time of the final results including the memory access times. To evaluate instruction-replication we manually introduced the redundant computation instructions into the kernels' source code and disabled the compiler's dead code elimination optimization. To support kernel re-execution we modified the Merrimac stream compiler to run the kernel twice back-to-back (the comparison of results written back to the SRF is performed by hardware comparators in this scheme), and shift any dependencies to the later kernel. ABFT is applied to matrix multiplication only, as described below.

5.2 Matrix Multiplication

Our matrix multiplication example consists of multiplying two dense matrices A and B of sizes (304×512) and (512×384) to form the result matrix C of size (304×384). The matrices sizes were chosen to be large enough to exceed the size of the SRF, thus being representative of much larger matrices while not requiring unduly large simulation times on the cycle accurate simulator. The computation was hierarchically blocked to take advantage of locality at both the SRF and LRF levels, as well as utilize Merrimac's inter-cluster communication. The full details of this implementation are not described in this paper. The resulting code is memory bandwidth limited on Merrimac, yet still achieves over 80% of Merrimac's 128GFLOPS only accounting for actual matrix computation instructions.

Our ABFT implementation of matrix multiplication follows the ideas discussed in [14] and uses a simple column checksum. As we process a block of the A matrix in the SRF we compute the sum of each of its columns. This checksum row is then multiplied with the block of the B matrix and stored back to the SRF. Once an entire block of the result matrix C is computed in the SRF we run a kernel that computes the column sums of the C matrix block and compares them against the multiplied checksum row. Due to the linearity of matrix multiplication, any difference in the results indicates an erroneous computation and a fault exception can be raised.

Table 5 summarizes the results for matrix multiplication. The runtime is given relative to the baseline implementation that does not use any of the stream unit fault-detection mechanisms. The register-pressure column lists the maximum number of registers in a single LRF required to run each variant, and the instruction-pressure column shows the number of instructions that need to be present in the microcode store.

As expected, FPR, the simplest fault-detection method,

Scheme	Normalized Runtime	Register Pressure	Instruction Pressure
Baseline	1.00	17	117
FPR	2.09	17	117
IR	2.51	52	258
KR	1.77	17	117
MC	1.76	17	117
ABFT	1.03	22	255

Table 5: Evaluation of matrix-multiplication under the five reliability schemes

has a large impact on runtime requiring a full re-execution with an additional 10% increase due to the final comparison.

The poor performance of the IR software scheme, which has a slow-down of about 20% compared even to FPR, is the result of the large number of comparisons that were added to the compute kernel and the fact that the baseline kernel fully utilized all the available multiply-add units. Moreover, IR requires over two times as many instructions in the kernel and increases the register requirements in a single LRF by a similar factor.

Both the KR and MC approaches essentially double the computation only without increasing register or instruction pressure and rely on comparators in hardware. As the baseline implementation is memory bandwidth limited on Merrimac, the additional computations fill the idle execution slots resulting in a speedup of 18% over naive FPR. The small difference of 1% in runtime in favor of MC over KR is due to the necessity of doubling the small kernel overhead in the KR scheme.

Finally ABFT is clearly superior for matrix multiplication, and requires only a small overhead of 3% to achieve fault detection. While the increase in code size is significant (a factor of almost 2.2), the total size is still small and easily fits within Merrimac's instruction store.

5.3 Molecular Dynamics

Unlike matrix multiplication, force calculation in molecular dynamics displays dynamic behavior and non-linear computations and we are not aware of any ABFT technique that is applicable to it. We use the Merrimac implementation of the GROMACS ([40]) water-water force calculation detailed in [10], as an example of such an application.

Scheme	Normalized Runtime	Register Pressure	Instruction Pressure
Baseline	1.00	27	170
FPR	2.01	27	170
IR	1.85	47	328
KR	1.90	27	170
MC	1.88	27	170

Table 6: Evaluation of molecular dynamics application under the four applicable reliability schemes

Table 6 lists our results for the molecular-dynamic application, and follows the format of Table 5 described in the previous subsection.

The force calculation requires significant computation and a large input set, yet only produces a small number of values as a result (the total force on each atom in the system). Therefore, FPR has little overhead for comparison and requires 2.01× the runtime of the baseline case. Another differentiation point from matrix-multiply is that the molecular dynamics application is not as memory limited[2]. This makes FPR a more attractive alternative, where the best performing reliability technique is only 8% faster.

While IR did not perform well in the case of matrix multiplication, it is the most efficient technique applicable to molecular dynamics. The reason is that the structure of the computation is such that there is significant slack in kernel schedule and not all instruction slots are filled during kernel execution (as in matrix multiplication). As a result some of the additional instructions required to duplicate the computation and perform the comparison can be scheduled in the empty slots leading to a kernel that requires less than twice the execution time of the baseline kernel. This is not the case for the KR and MC schemes, which fully double the execution time the main kernel loop, trailing the performance of IR by 3% and 2% respectively. However, achieving this speedup requires increasing the number of registers in each LRF and roughly doubling the kernels instruction count.

The performance difference between the KR and MC techniques is again quite low, at 1% due to the small overhead of priming the kernel software pipelined main loop.

5.4 Discussion

The two case studies presented above span diverse characteristics in the trade-off space. Matrix multiplication shows the power of applying an ABFT technique, but also represents applications with a relatively high overhead for final-result comparison, memory bandwidth limited applications, very tight kernel schedules, and low kernel startup overheads. Molecular dynamics on the other hand, requires very few comparisons of its final result, is not memory limited, and has kernels with significant scheduling slack. We can draw a few interesting observations from these characteristics.

The most obvious observation is that ABFT techniques should be applied when possible. While not all cases will be as efficient as dense matrix multiplication, we expect that ABFT will almost always offer better performance than other schemes, possibly requiring modifications to the input set of the application. One exception to this general rule is applications that are extremely memory bandwidth limited, for which even doubling the computation will not change the total runtime.

Another interesting point is that kernels that do not have a very tight schedule offer the opportunity to hide much of the cost of instruction replication. This matches with many observations of using this technique with conventional CPUs, which usually display significant idle execution slots.

In both applications discussed above, the kernel startup overheads are low, and therefore KR is almost as efficient as MC. The small amount of additional hardware required to enable MC execution (a local switch between the SRF lanes of each cluster pair) may be more attractive for other applications. Our future work will include modeling this behavior and testing a larger number of applications to explore this part of the trade-off space.

The final point we want to bring out regards the cost of protecting the external interfaces against undetected faults. As discussed in Subsection 4.2.3, we are not aware of an area-efficient hardware technique to perform fault-detection on

[2]We chose to use the "fixed" variant described in [10].

the memory and network interfaces. However, with the exception of FPR the fault-detection schemes discussed above rely on accurate data transfers. This is true for many ABFT techniques as well, including the one we employed for matrix multiplication. While it is not clear that the interfaces have to be protected in all cases (our analysis in Subsection 4.1 shows this to be the case in current technology), when fault-detection is added it might be more effective to resort to FPR instead. For example, growing the die-area by 30% to only gain 8% in runtime as is the case for molecular dynamics, may not be a good trade-off.

6. Conclusions

In this paper we describe the methods that allow the Merrimac streaming supercomputer to achieve reliable operation in the presence of soft errors. With modern VLSI logic processes we show that fault detection covering all logic modules is required for reliable operation. Our fault tolerance strategy is aimed at making the most efficient use of scarce and expensive off chip bandwidth and on-chip memory. In comparison it is relatively inexpensive to replicate on-chip logic. Our strategy uses dynamic redundancy with strong fault-detection followed by re-execution from checkpointed state. For fault-detection we made sure that hardware techniques were used when they conserve the expensive off-chip bandwidth and on-chip storage resources, and a hybrid hardware-software approach for dealing with computational redundancy.

In our analysis we showed that the bulk of the reduction in soft error rate (SER) can be gained by employing ECC methods on large arrays, and that in 90nm technology a low SER of 21 FIT can be achieved. However this standard techniques will not be enough in future semiconductor technologies, where we expect the SER to be over 161 FIT. Then, we will need to both protect the execution clusters and the external interfaces. Merrimac and other recently introduced compute-intensive architectures dedicate a large portion of the chip to functional units, and rely on simple control, no speculation, and optimized static compilation to gain high cost/performance and power/performance efficiencies. As a result, many of the techniques developed for superscalar processors do not carry over. Moreover, as logic consumes significant area, replication and circuit fault-detection methods are very costly.

Our hybrid software-hardware approach allows the programmer or programming system to use the full computational capabilities of the Merrimac chip, when effective and efficient software fault-tolerance techniques are available. Yet, when no software-technique is known, the processor can be configured to mimic hardware functional-unit duplication while utilizing the full available bandwidth and storage. For example algorithmic based fault tolerance was used for the matrix-multiply case study, and instruction replication in the hybrid schemes was employed in the molecular dynamics example.

A final interesting observation about the results, is that the scalar unit's contribution to the total SER is very small yet protecting it in hardware is relatively expensive. In this paper we proposed several cost-effective mechanisms for detecting faults in the clusters, but have simply used expensive replication for the interfaces and scalar unit. We believe further study is warranted into reducing this overhead and this study is left for future work. In the case of the scalar unit is is possible to employ software-only detection mechanisms, yet careful analysis of the effects a slowdown in the control processor will have on overall performance is necessary. As for the external interfaces, a more careful analysis of the micro-architecture is required to identify cost-effective solutions.

7. REFERENCES

[1] 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.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 microprocessor. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 702–705, 2003.

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

[4] ATI. Radeon X800 product site. http://www.ati.com/products/radeonx800.

[5] 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*, pages 196–207, 1999.

[6] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. The STAR (Self Testing And Repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. *IEEE Trans. Comput.*, C-20(11), November 1971.

[7] V. Balasubramanian and P. Banerjee. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. *IEEE Trans. Comput.*, 39(4):436–446, 1990.

[8] 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 Trans. Comput.*, 39(9):1132–1145, 1990.

[9] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[10] M. Erez, J. Ahn, A. Garg, W. J. Dally, and E. Darve. Analysis and performance results of a molecular modeling application on Merrimac. In *SC'04*, Pittsburgh, Pennsylvaniva, November 2004.

[11] 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*, pages 98–109, 2003.

[12] T. R. Halfhill. Floating point buoys ClearSpeed. *Microprocessor Report*, November 17, 2003.

[13] 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*, pages 617–620, September 2003.

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

[15] J.-Y. Jou and J. A. Abraham. Fault-tolerant FFT networks. *IEEE Trans. Comput.*, 37(5):548–561, 1988.

[16] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, August 2003.

[17] B. Khailany. *The VLSI Implementation and Evaluation of*

Area- and Energy-Efficient Streaming Media Processors. PhD thesis, Stanford University, June 2003.

[18] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owen, and B. Towles. Exploring the VLSI scalability of stream processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*, pages 153–164, Anaheim, California, USA, February 2003.

[19] K. Krewell. Cell moves into the limelight. *Microprocessor Report*, February 14, 2005.

[20] 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.

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

[22] MIPS Technologies. *MIPS64 20Kc Core*. http://www.mips.com/ProductCatalog/P_MIPS6420KcCore.

[23] A. Mishra and P. Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput.*, 52(9):1089–1099, 2003.

[24] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, 2005.

[25] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *Eleventh International Symposium on High-Performance Computer Architecture (HPCA-11)*, San Francisco, California, February 2005.

[26] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 29, 2003.

[27] 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.

[28] NVIDIA. NVIDIA GeFORCE FX. http://www.nvidia.com/docs/lo/2430/SUPP/PO_GFFX_Consumer_030503.pdf.

[29] N. Oh, S. Mitra, and E. J. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51(2):180–199, 2002.

[30] A. L. N. Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Trans. Comput.*, 39(10):1304–1308, 1990.

[31] 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)*, pages 243–254, 2005.

[32] 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*, page 394, 1996.

[33] 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*, page 84, 1999.

[34] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA'03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433, 2003.

[35] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, 2001 Edition.

[36] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[37] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[38] 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*, 19(2):12–23, 1999.

[39] 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*, pages 224–234, 2004.

[40] D. van der Spoel, A. R. van Buuren, E. Apol, P. J. Meulenhoff, D. P. Tieleman, A. L. T. M. Sijbers, B. Hess, K. A. Feenstra, E. Lindahl, R. van Drunen, and H. J. C. Berendsen. *Gromacs User Manual version 3.1*. Nijenborgh 4, 9747 AG Groningen, The Netherlands. Internet: http://www.gromacs.org, 2001.

[41] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 61, 2004.

[42] 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*, pages 458–469, 1976.

[43] 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*, pages 130–137, 2005.