A Survey of Checker Architectures

Rajshekar Kalayappan
Computer Science and Engineering
Indian Institute of Technology, Delhi
and
Smruti R. Sarangi
Computer Science and Engineering
Indian Institute of Technology, Delhi

Reliability is quickly becoming a primary design constraint for high end processors because of the inherent limits of manufacturability, extreme miniaturization of transistors, and the growing complexity of large multicore chips. To achieve a high degree of fault tolerance, we need to detect faults quickly, and try to rectify them. In this paper, we focus on the former aspect. We present a survey of different kinds of fault detection mechanisms for processors at the circuit, architecture, and software level. We collectively refer to such mechanisms as *checker architectures*. First, we propose a novel two-level taxonomy for different classes of checkers based on their structure and functionality. Subsequently, for each class we present the ideas in some of the seminal papers that have defined the direction of the area along with important extensions published in later work.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General

General Terms: Checker Architectures

Additional Key Words and Phrases: Reliability, Checker Architecture, Fault Tolerance

1. INTRODUCTION

With continued device scaling, it is getting increasingly difficult to ensure reliability at a device, circuit, and architectural level. Traditional methods of design and testing are proving insufficient to ensure reliable error free operation for today's multi-billion transistor chips [Borkar 2004]. Consequently, the budgets for verification and testing are now roughly two thirds [Bacchini et al. 2004] of the cost of developing a high end processor. Even then, occasionally some defects slip into production silicon, and have resulted in catastrophic failures as documented by the popular press [Sarangi et al. 2007]. Other than such congenital faults [Sarangi 2007], processors are susceptible to a wide variety of transient and permanent faults during regular operation. Hence, along with extensive testing and careful design, it is necessary to reckon that defects will manifest in the field, and we need to

Address of the institute for both the authors
Indian Institute of Technology
Hauz Khas, New Delhi, 110016
India

make changes in the design of processors to dynamically detect and correct them. Researchers are converging to the view that we need to design *reliable processors* with increasingly unpredictable and unreliable components.

The need for reliable processors has been felt since the mid seventies when computers started to be used in major banks, financial institutions, and airlines. The designers of the Tandem computer [Horst and Chou 1985] were one of the early pioneers in this field. They used three processors in parallel, and decided the results based on voting. They considered a *fail-stop* failure model, which guarantees that a processor will completely stop functioning upon an error. However, since then, faults have become far more insidious in nature. In the mid nineties transient faults due to cosmic rays and alpha particle strikes came into prominence. As a result, IBM augmented its mainframe processors with a spare pipeline that executed a redundant thread of computation [Spainhower and Gregg 1999].

In the last decade, transistors have seen further miniaturization, and have entered the nanometer era. Along with a significantly increased chance of transient faults, transistors are now vulnerable to a host of wear out related faults that manifest over time. Secondly, designs have become extremely complicated, especially, after the advent of multicores. This leads to a host of design defects. Consequently, there is a vast body of literature on different kinds of checker architectures tailored to detect different kinds of faults. Some of them use variants of classical n-modular execution; however, most schemes use a host of novel techniques to quickly detect faults. There are some additional constraints such as power consumption and complexity, which make the space of solutions even more diverse.

1.1 Scope

The problem of designing reliable processors can be broken down into three parts – fault characterization (Section 2.1), efficient fault/error detection (Sections 3 to 7), and checkpointing/ recovery (Section 2.3). In this survey paper, we focus on online detection mechanisms for faults in hardware. In specific, we primarily look at approaches at the architecture level. Additionally, we look at some circuit and software level approaches that are fairly oblivious to low-level hardware details, and mostly concern themselves with architectural features.

1.2 Organization

We start by providing a novel taxonomy of fault detection mechanisms in Section 2.2. At the highest level, we classify the different detection mechanisms based on two major criteria. The first criterion is the type of the checker – specialized circuit, spare core, extra thread, and software module. The second criterion is the type of target processor – single core, or multicore. For a second level classification, we have three orthogonal sets of criteria namely the type of fault detected, degree of coverage, and the structure of the checker. We observe that when the checker architecture is limited to specialized circuits(see Section 3), there is a significant amount of diversity in solutions. The solutions for different types of faults vary significantly. However, most of the solutions at the level of a core, thread, or CMP, follow one of several major patterns as described in Sections 4 to 6. These patterns were proposed in a few seminal papers. Most schemes add extra constraints, change the fault model, or change the type of coverage. The approaches for find-

ing hardware faults in software (see Section 7) either rely on classical redundancy techniques, or check if a certain set of invariants hold.

We finally conclude in Section 8 by observing that we need an even more diverse set of solutions for checking architectures of the future. Over the next few years, we expect processors to have tens of cores, many specialized accelerators, 3d integration, and novel on-chip interconnects. They will have their own unique runtime verification challenges.

2. BACKGROUND

We describe different types of faults in Section 2.1, provide a taxonomy of different types of checking systems in Section 2.2, and briefly survey different methods of taking checkpoints and performing rollback in Section 2.3.

2.1 Faults in Hardware

2.1.1 *Overview.* Traditional fault tolerance literature [Koren and Krishna 2007; Mourad and Zorian 2000] classifies faults in two different ways. They can be classified by their duration – transient, intermittent, or permanent. Alternatively, they can be classified based on their nature – stuck-at, timing, and functional. Stuck-at faults can be modeled as a permanent open circuit or short circuit for some wire or transistor. When a circuit becomes slow, and signals do not reach their destination on time, the circuit suffers from a timing fault. Lastly, a logical flaw in the design of the circuit is referred to as a functional fault. Given the fact, that transient faults in circuits can typically be modeled as stuck-at faults, and in a processor's time frame, there is no significant difference between an intermittent and permanent fault, we can coalesce the different types of faults into four categories – transient, timing, hard, and design.

2.1.2 *Transient Faults.* Transient faults are caused by events that occur once and their effects persist for an extremely small duration (typically less than one cycle). Some of the most common causes include alpha particles, and neutrons generated by cosmic radiation, or spontaneous degeneration of unstable isotopes in the packaging material [Ziegler et al. 1996]. These events induce a current pulse, which is potent enough to flip the value stored in a latch. There are other relatively infrequent causes such as crosstalk and sudden fluctuations in supply voltage. Both logic as well as memory elements such as latches and unprotected SRAM arrays are susceptible to these faults. Larger yet slower transistors can partially mitigate this problem.

2.1.3 *Timing Faults.* Timing faults are typically caused by severe voltage or temperature fluctuations, or as a result of gradual wear out. Severe temperature and voltage fluctuations typically are intermittent in nature, and can possibly last till hundreds of milliseconds. During this period, transistors can slow down, and lead to the emergence of timing faults. Secondly, due to ageing processes [Tiwari and Torrellas 2008], interconnects and transistors can gradually wear out, and become slower. One of the prominent ageing processes is negative bias temperature instability(NBTI), which refers to the increase in threshold voltage of PMOS transistors due to negative gate voltages applied at a high temperature.

Fault	Rule for Detection	Rule for Correction
Transient	Any scheme that assesses the correctness of the program execution.	Spatial or temporal re-execution of the affected sequence of instructions.
Timing	1) A spare copy that is physically isolated from the master copy. Eg., separate core, or separate circuit with its own clock. or 2) A scheme that detects an invariant.	A spare that effectively runs at a safe frequency.
Hard	A scheme that uses spatial redundancy or invariants.	Contains one or multiple spare copies that are guaranteed to be correct.
Design	A method that checks some high level properties of the program execution that are independent of the architecture.	An entity that computes the correct result through an alternative method.

Table I. Rules for classifying faults

2.1.4 *Hard Faults.* Ageing processes can ultimately lead to a complete disintegration of the affected component (transistors or interconnects). Along with NBTI, typical examples of ageing processes that ultimately lead to hard faults are electromigration, stress migration, thermal cycling, and dielectric breakdown. *Electromigration* refers to the phenomena in which wires tend to gradually thin over time, because flowing electrons transfer a part of their momentum to the surrounding atoms. A similar migration of atoms known as *Stress Migration* happens due to thermal fluctuations. *Thermal Cycling* refers to metal fatigue around I/O contacts due to temperature cycles.

2.1.5 *Design Faults.* Design faults refer to bugs in RTL. Even after extensive verification, some design bugs slip into production silicon [Sarangi et al. 2007], and permanently impair some aspect of the processor's functionality. One such example is the infamous Pentium division bug [Blum and Wasserman 1996]. Because of the increasing complexity of modern architectures, it is becoming increasingly difficult to completely eliminate them.

2.1.6 *Detection and Correction of Faults.* In the subsequent sections, we label each major approach based on the types of faults it detects and corrects. We have observed that in some cases, the original paper was slightly vague about the fault coverage. Secondly, it is possible that through trivial extensions of the proposed architecture, it is possible to detect a wider variety of faults.

Table I provides an alternate perspective, viewed in terms of the faults, rather than the fault detection/correction schemes. The four types of faults are discussed in terms of the nature of the support that is required to detect and correct them. We use these rules to classify different architectures on the basis of their fault coverage. Please note that when we say that a certain architecture *detects* a certain fault, we mean that it either detects the fault or the error resulting from the fault. It is not necessary for it to distinguish different fault types all the time, as long as it can detect all the resultant errors.

2.2 Taxonomy

We propose a taxonomy of checkers in this section. Traditional fault tolerance literature [Koren and Krishna 2007] has looked at three major classes of solutions: spatial redundancy (run spare copies in parallel), temporal redundancy (run the same copy repeatedly), and information redundancy (error checking logic). We observe that these criteria prove to be extremely nebulous for classifying checker architectures. For example, [Rashid et al. 2005] proposes to parallelize the process of checking for different chunks of execution. These copies lag behind the master copy. We cannot neatly place this idea in any of the bins, because it contains aspects of both temporal as well as spatial redundancy.

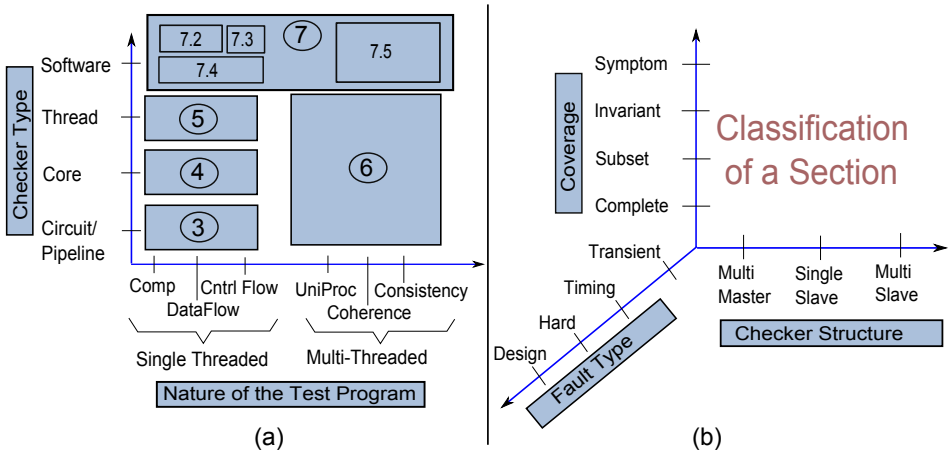

Fig. 1. *Taxonomy of checkers*

Consequently, we base our high level classification on a novel set of criteria shown in Figure 1(a). Before considering the taxonomy in detail, we need to reckon that it is often computationally intractable to check the correctness of hardware in totality. Obtaining an acceptable amount of coverage using pure hardware checking/verification methods can also be very time consuming. Consequently, the methods that we survey, primarily focus on the correctness of the execution of a given *test program*. If this test program does not execute correctly, then we can infer a hardware fault. Please note that the test program can either be a regular application, or it can be a targeted micro-benchmark tailored to detect a specific category of faults.

The x-axis represents the nature of the test program – single threaded, or multi-threaded. For a single threaded execution, we can check the computation (*Comp*), data flow (*DataFlow*) and control flow (*CntrlFlow*). For a multi-threaded execution, we can check for just uniprocessor semantics (*UniProc*), or mutiprocessor semantics – cache coherence, and memory consistency. In the y-axis, we list the different types of design alternatives for checkers. They can either be specialized circuits or intra-pipeline functional units, spare cores, spare threads, or software modules. We divide

the 2D design space into rectangles, and discuss each rectangle in a separate section in this paper..

We now define a set of second level criteria in Figure 1(b) to dissect each section. Before proceeding, let us define the terms – *master* and *slave*. A *master* is an entity (hardware or software), which represents the default implementation. It is afflicted by faults, and has a minimal amount of instrumentation in a checker based architecture. In comparison, a *slave* is a separate entity (hardware of software), which is significantly impaired in either ability or performance as compared to the master. A slave is not strictly required to guarantee correctness. Secondly, it might either execute subsets of the original program, or perform a totally separate computation to detect faults. A checker based architecture needs to detect faults by observing the outputs of the masters and the slaves.

Now, we define three kinds of configurations based on the number of masters and slaves – *MultiMaster* (multiple masters, no slaves), *SingleSlave* (single master, single slave), and *MultiSlave* (single master, multiple slaves). *MultiMaster* is similar to classical n-modular redundancy [Koren and Krishna 2007], where the existence of a fault is decided by comparing the outputs of the different redundant units. *SingleSlave* refers to a weak slave unit, which typically checks the execution of the master/underlying system, or does some other computation that is helpful in detecting a fault. The *MultiSlave* configuration parallelizes the work done by the slaves. We use this classification as the x-axis in Figure 1(b).

Let us define the term, *test set*, which refers to the set of all dynamic instructions in the execution of the test program that are relevant to the subsystem under test. Now, for the y-axis in Figure 1(b), we consider the degree of coverage – *complete*, *subset*, *invariant*, and *symptom*[1]. *Complete* coverage refers to the verification of every single instruction in the test set; whereas, *subset* coverage considers only a proper subset of all instructions. The *invariant* coverage metric seeks to verify a certain set of high level assertions. Lastly, the *symptom* coverage metric is conceptually the inverse of *invariant*; it checks for signs of something wrong. However, in most cases, it is not a strict boolean inverse of the invariant conditions. For example, let us consider a program that finds a root of a quadratic equation. *Complete* coverage is tantamount to checking all the instructions. We might just decide to check that the instructions to compute the discriminant are executed correctly (*subset* coverage), or we might substitute the value of the obtained root in the original equation, and check if it equates to zero (*invariant* coverage). Lastly, we can run a fault inference algorithm only upon the occurrence of an extraordinary event such as a segmentation fault (*symptom* coverage).

Finally, we use the type of fault – transient, timing, hard, and design – in the z-axis. For different types of checkers, we do not have solutions for all points in this 3D space. Consequently, we use different set of axes in different orders of priority for each section based on the diversity of prior work.

2.3 Checkpoints and Recovery

In this section, we discuss the process of recovery from an error. There are a few classical forward error recovery techniques, which use n-modular redundancy or

[1]We use *italics* for the coverage metrics to distinguish them from other uses of the same word in the rest of the text

error checking logic. In the former case, the correct output is decided by voting, and latter case we use error correcting codes to fix erroneous bits in the output. However, because of their limited applicability as well as implementation overhead, prior work has primarily looked at backward error recovery(BER), which involves the process of creating a checkpoint of a correct state, and a rollback upon the detection of an error. Please note that both these areas are extremely well studied; in this section, we are just outlining the major concepts.

For BER, the first design point that we need to consider is the extent of propagation of the error at the time it is detected. The erroneous output might have propagated to logic in the same functional unit(FU), other stages in the pipeline, or other elements in the memory system such as caches.

2.3.1 *Functional Units.* If the error is localized to the FU, then we first need to restore the state bits of the FU to default values. Subsequently, for error recovery we have two generic options. We can either reissue the instruction to the same FU, or use the result of a spare FU [Spainhower and Gregg 1999].

2.3.2 *Pipeline.* If the error has propagated to other pipeline stages, then we can treat this event as an exception. We need to flush the pipeline, and restore the register state. The correct register state can be obtained from either a register checkpoint taken previously or from a separate slave thread, which is guaranteed to be correct. As explained by Smith and Sohi [Smith and Sohi 1995], there are two methods of taking a register checkpoint. The first technique uses a history buffer to take a checkpoint of the values of all the architectural registers at periodic intervals. To rollback, we need to find the relevant check point in the history buffer. The second method, uses a separate architectural register file(ARF), which maintains the register state for only the committed instructions. To flush all the instructions in flight, we just need to copy the state from the ARF to the physical register file in the processor.

2.3.3 *Memory System.* When bugs propagate to the memory system, the process of recovery can be complicated. First, we need to use checkpoints for the pipeline as mentioned in Section 2.3.2. Subsequently, there are four broad approaches for taking checkpoints for memory [Prvulovic et al. 2002] – full separation, partial separation, renaming, and logging. "Full separation" proposes to keep all the speculative state in a dedicated store buffer. For any access, we need to query this buffer first, before accessing the caches. We can rollback to a checkpoint by flushing the store buffer. To create a new checkpoint, we need to drain the store buffer to the L1 cache, and reinitialize it. However, this approach is not very scalable. "Partial separation" uses the caches as the store buffer. Before creating a checkpoint, we need to ensure that all the speculative data in the cache is flushed to lower levels. Subsequently, we need to ensure that we are not evicting speculative data from the cache. If there is a conflict miss, we can use a victim cache to ameliorate the situation. After detecting a fault, we can go back to the last checkpoint by flushing the cache.

As compared to separation based approaches, we can use "renaming" at the page level. We can use a copy-on-write scheme for pages. After the creation of a checkpoint, whenever a word in a page is written, we need to create a new copy of the page. To restore a checkpoint, we need to restore the state of the page

table at the beginning of the checkpoint. The most scalable method especially for multiprocessors [Prvulovic et al. 2002; Sorin et al. 2002] is to use variants of "logging", which propose to record the old value of a memory word before it is overwritten for the first time. These logs can be maintained either in software or in hardware. Regular read accesses are oblivious of the logs. Since faults are extremely infrequent, we will need to restore the values in the logs very rarely. Consequently, we can afford to maintain very large logs.

For multiprocessors, there is an additional issue [Prvulovic et al. 2002] – checkpoint consistency. A checkpoint is consistent if it contains the effect of an event, only if, it contains the cause of the event. For example, it should never be the case that processor 1 has written X to location A, processor 2 has read X from there, and the read has been recorded but the write has not been recorded. We can have many such cases in systems with relaxed memory consistency models. Consequently, there is a need to take a co-ordinated checkpoint across different cores. We can either have global co-ordination or optimized algorithms to take co-ordinated local checkpoints [Prvulovic et al. 2002; Sorin et al. 2002; Ahmed et al. 1990].

3. CIRCUIT AND PIPELINE LEVEL TECHNIQUES

3.1 Summary

In this section, we look at approaches at the circuit and pipeline level. For a second level classification, we classify the approaches by the type of fault that they are primarily designed to detect. At this level, the approaches to detect different kinds of faults vary significantly. We shall observe in the succeeding sections that at a higher level, the nature of the fault is not very important, and it is hard to distinguish between them.

Table II shows a list of some of the seminal techniques. Most of them are designed to primarily detect one kind of fault. Some proposals such as Argus [Meixner et al. 2007] can detect both hard as well as transient faults. We have classified them based on their conceptual similarity with other schemes in the same category. We mention the nature of faults covered in the description of the technique as well as in Table II.

After classifying the techniques based on the type of fault they detect, we classify them based on coverage. As compared to higher level approaches, circuit level approaches are much more reliant on symptoms and invariants. Lastly, we observe that there is also a preponderance of *MultiSlave* techniques, because, a pipeline level checking algorithm typically uses a different type of slave for each important stage.

3.2 Transient Faults

3.2.1 *Complete Coverage.* [Ray et al. 2001; Parashar et al. 2004] advocate the use of classical n-modular redundancy for the execution stage of the pipeline (see Figure 3). There are replicated execution units, and each instruction is dispatched to a multitude of units. This replication is done at the rename stage. At the end of the execution, the hardware compares the outputs of the replicated units. If there is a mismatch that tends to recur, we can infer a hard fault.

Proposal	Sub-Section	Perf. Ovhd.	HW Ovhd.	Checker Type	Coverage	Faults
IRTR [Gomaa et. al. 2005]	Transient Faults (3.2)	≈ 0%	Minimal	*MultiMaster*	*Subset*	T
ReStore [Wang et. al., 2006]	Transient Faults (3.2)	≈ 0%	Minimal	*MultiSlave*	*Symptom*	T MH
RAZOR [Ernst et. al., 2003]	Timing Faults (3.3)	≈ 0%	Extra flip-flop per latch	*SingleSlave*	*Complete*	M
Wearmon [Zandian et. al., 2010]	Timing Faults (3.3)	≈ 0%	1 Delay monitor per stage	*SingleSlave*	*Invariant*	M
BulletProof [Shyam et. al., 2006]	Hard Faults (3.4)	4 – 18%	5.8%	*MultiSlave*	*Complete*	MH
Argus [Meixner et. al., 2007]	Hard Faults (3.4)	< 4%	< 17%	*MultiSlave*	*Complete*	TMH
Phoenix [Sarangi et. al., 2006]	Design Faults (3.5)	≈ 0%	0.05%	*MultiSlave*	*Complete*	D
Fault Types : (T → Transient, M → Timing, H → Hard, D → Design)						
→ detect and correct, → only detect						

Table II. Summary of circuit/pipeline level approaches

3.2.2 *Subset Coverage.* Given the overheads of *Complete* coverage, we observe the need to consider a subset of all the instructions. Gomaa et. al. [Gomaa and Vijaykumar 2005] propose to use the resources of the pipeline when it is idle to re-execute portions of the program. Such idle phases can be caused by L2 misses, or low-IPC phases. They present the design of a reuse buffer that contains a set of instructions along with their input and output values. A part of the pipeline such as the issue queue, and the functional units are augmented with extra hardware to verify the results. Qureishi et. al. [Qureshi et al. 2005] propose a similar scheme, and augment it by making the verification phase visible to higher levels.

Reese [Nickel and Somani 2001] proposes to use idle cycles to only verify ALU results. [Hu et al. 2005] extends this idea to consider the load imbalance between the integer ALU and the floating point ALU. The floating point ALU can be used to check the results of the integer ALU.

3.2.3 *Symptom Coverage.* The ReStore [Wang and Patel 2006] architecture uses symptoms to detect transient faults. Through a detailed set of simulations, the authors establish that about 80% of failure inducing transient faults can be classified as processor exceptions, access to illegal memory addresses, and control flow violations. These can be detected by a specialized symptom detector.

3.2.4 *Invariant Coverage.* Early variants of IBM's G5/S390 servers [Spainhower and Gregg 1999] had processors that performed inline checking by performing parity prediction or duplicating selected functional units. Parity prediction refers to a process of quickly predicting the parity of the output of an ALU without doing the actual computation. This can be used to find errors in the execution. [Ossi et al. 2009] uses Berger codes to verify the operation of an ALU. A Berger code

is a concise representation of a binary number. Moreover, it is possible to find the Berger code representation of the result of an ALU operation from the Berger codes of the operands. [Carretero et al. 2009] generalizes this approach to consider a variety of signatures. It uses these signatures to verify the control logic of the issue queue and the register files. Secondly, it also proposes to use signatures based on arithmetic codes, which can be used to verify the operation of ALUs.

3.3 Timing Faults

3.3.1 *RAZOR – Complete Coverage.* The RAZOR [Ernst et al. 2003] scheme was originally proposed to reduce power. However, it proved to be a seminal technique and has subsequently been used in [Avirneni et al. 2009] for fault tolerance. RAZOR proposes to augment a pipeline latch with an additional RAZOR flip-flop.

The intuition behind the RAZOR pipeline is as follows. Let us assume that the ideal clock cycle is one unit of time. It is possible that due to a timing fault, the delay of a pipeline stage increases to κ, where $\kappa > 1$ for some inputs. Now, if we resample the output of the stage at the instant, $\kappa - 1$, in the subsequent cycle, and compare it with the value that was originally stored in the pipeline latch, we can detect a timing fault.

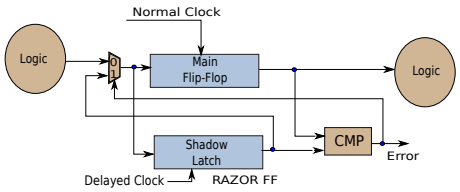

Fig. 2. *RAZOR*

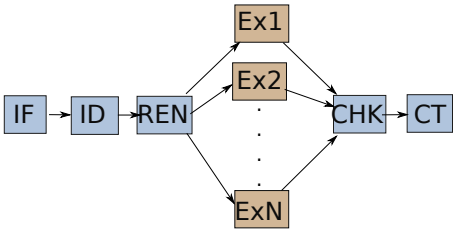

Fig. 3. *Dual use of superscalar datapath*

Figure 2 shows the design of a RAZOR flip-flop. The flip-flop has a normal latch and a shadow latch. The shadow latch uses a clock that is offset from the main clock by $\kappa - 1$ units of time. It is assumed that by the time the outputs are read into the shadow latch, they have stabilized to the correct values. Consequently, we compare the results in the shadow latch, and the main flip flop and infer faults accordingly. It is possible to proceed by using the value in the shadow latch in the subsequent stages. The erroneous computation can be discarded by inserting a pipeline bubble or by flushing the pipeline [Blaauw et al. 2008].

3.3.2 *Invariant Coverage.* Typically, the clock period of a processor is designed to accommodate for worst case propagation delays of all the pipeline stages. The difference between the propagation delay of a signal and the clock period is indicative of the safety margin. If it is too small or negative, then we can infer the occurrence of a timing fault. [Blome et al. 2007; Zandian et al. 2010] propose extra circuitry to measure this difference through periodic sampling, and by using different test vectors. These circuits are activated periodically or during periods of low activity.

3.4 Hard Faults

[Shyam et al. 2006] proposes *BulletProof*, a circuit-level solution that protects the pipeline and the on-chip memory. Each stage of the pipeline (decoder, register file, functional units, cache) has an associated checker, which get activated during idle slots. A decoder checker sends the same instruction to multiple unused decoders in the pipeline and verifies the results. A register file checker generates random inputs, and writes them to unused registers. It reads those registers at a later point of time, and compares the results. Each ALU has a 9-bit mini ALU that can verify some of the results generated by the main ALU. [Constantinides et al. 2007] proposes a new set of instructions that can manipulate a microprocessor's internal state. The proposal envisages firmware that can periodically suspend the execution of the microprocessor, capture its state, run a set of test vectors loaded by software, and then restore the state back. The coverage of both of these techniques is dependent on the type of test vectors, and checking algorithms.

Argus [Meixner et al. 2007] can detect both transient and hard faults. It has a dataflow, control-flow, and computation checker. Using static analysis, the compiler embeds a signature representing the control flow and data flow in every basic block. The signature represents the flow of values between different registers in a basic block, and the unique id of the basic block. To verify the data flow within a basic block, the hardware needs to compute the signature dynamically, and compare it with the embedded signature. For the case of control flow, the compiler embeds the signatures of the possible successors at the end of the basic block. The hardware verifies the signature of a newly entered basic block with this list. For verifying computation, Argus uses a low cost adder for adds, simpler redundant units for logical operations, and a modulo-arithmetic based approach for checking multiplication/division operations.

3.5 Design Faults

Sarangi et. al. [Sarangi et al. 2007] proposed the *Phoenix* system that can detect design faults. They analyze the errata sheets published by processor vendors, and characterize the design defects based on the micro-architecture level signals that activate them. For example, a bug in Pentium IV, can possibly manifest when in the same cycle, the L2 cache is being flushed, there is an external snoop request, and the processor is in a low power state. They subsequently characterize the defects from a certain processor family, and use this information to design appropriate hardware to collect all the signals for a future processor. There is an elaborate network of programmable logic arrays that can collect and route signals. Each fault is associated with a combination of signals.

After releasing a processor, vendors should continue to test their processor for design faults. As and when they detect faults, they need to issue a hardware patch that can train the reconfigurable on-chip hardware to monitor appropriate combinations of signals, and flag an error if the combination gets activated. Software or dedicated hardware can then initiate recovery, and take measures to circumvent the problem. [Constantinides et al. 2008] extends this work by considering more comprehensive and flexible monitoring of signals.

4. CORE BASED APPROACHES

4.1 Summary

In this section, we explore approaches that use single or multiple fully functional cores as the checker. We survey a set of major techniques for each of the major classes of checkers, and their implications in terms of power, performance and complexity. We highlight some of the seminal proposals in Table III.

Proposal	Sub-Section	Perf. Ovhd.	HW Ovhd.	Checker Type	Coverage	Faults
NSAA† [Bernick et. al. 2005]	4.2	Minimal	300%	*MultiMaster*	*Complete*	TMH
DIVA [Weaver et. al., 2001]	4.3	$< 3\%$	6%	*SingleSlave*	*Complete*	TMHD
SlipStream [Purser et. al. 2000]	4.4	12% speed-up	100%	*SingleSlave*	*Complete*	T MH
CGVP* [Rashid et. al., 2005]	4.5	$< 8\%$	200% + post-commit buffer	*MultiSlave*	*Complete*	T MH
Fault Types : (T → Transient, M → Timing, H → Hard, D → Design)						
▭ → detect and correct, ▭ → only detect						
CGVP* → Coarse Grain Verification Parallelism, NSAA† → Non-Stop Advanced Architecture						

Table III. Summary of core level approaches

We first look at the classical N-modular redundancy based *MultiMaster* configuration in Section 4.2. *MultiMaster* based approaches can detect all kinds of faults, and can provide complete coverage. The issue here is that we cannot run modern processors in lock-step due to non-determinism introduced by the memory system, and bus protocols [Sarangi et al. 2006]. Consequently, we need to devise protocols to run processors in a loosely synchronized fashion, and compare outputs infrequently.

We subsequently look at two broad families of *SingleSlave* schemes – DIVA (Section 4.3) and SlipStream (Section 4.4). DIVA uses a simple in-order checker processor, and SlipStream uses a redundant core that lags behind the master core. The slave is meant to be slower as compared to the master such that we can ensure that it is relatively immune to faults. However, for the slave to keep up with the master, it is necessary for it to get hints – branch outcomes and memory values – from the master. The *SingleSlave* approaches are clearly better than *MultiMaster* approaches in terms of power. We can also reduce power consumption by considering a *MultiSlave* approach(MSSP family described in Section 4.5), where the task of checking is distributed among several slaves. Each of them can be run at a slower frequency. Since power is a cubic function of frequency, we can achieve substantial power savings.

4.2 *MultiMaster* Approaches

Conceptually, we can either use Dual Modular Redundancy (DMR) to just detect errors, or Triple Modular Redundancy (TMR) to correct them using voting. How-

ever, due to non-deterministic events in state of the art processors, it is difficult to run processors in lockstep and compare results every cycle. Different processors might read asynchronous events at slightly different times. Signal propagation delays using source synchronous buses [Sarangi et al. 2006] are typically variable. Power and thermal management events typically introduce delays. Lastly, due to process variation [Borkar 2004], different processors might run at different speeds or might have I/O interfaces, which do not have the same latency. Consequently, we need to explore loosely coupled solutions, which can tolerate some amount of slack between the processors.

4.2.1 *Lockstep Processors.* Tandem Computers Inc. (now Hewlett Packard Nonstop Enterprise Division) [Horst and Chou 1985; Joel et al. 1986] were the first to design and commercially market highly reliable servers that used dual modular redundancy. The cores had a minimal amount of slack between them. The NonStop I was released in 1976. The system consisted of 2 to 16 processors connected by a dual-bus interconnect. Each processor worked in a *failstop* mode. A *failstop* processor stops as soon as it detects a fault. The Tandem processor had extensive support for built in self test modules and parity checkers.

The first two versions of the Tandem processor used messages in software to exchange state across redundantly executing processes. To facilitate the exchange of messages, the NonStop kernel [Bartlett 1981] was developed. These processors used redundant memory elements and hard disks. These designs used messages to keep the checker processor in sync with the master. Whenever the master failed, it was taken off the system, and the checker was connected in lieu of it.

By 1993, designers realized that internal BIST mechanisms are not sufficient to detect all types of faults. Pure DMR solutions were deemed necessary. Hence, later versions employed a pairing scheme for processors - two processors, working with the same clock, executed the same program in a *lock-stepped* fashion (see Figure 4(a)). If their results did not match, a fault was inferred, and both the processors were removed from the system.

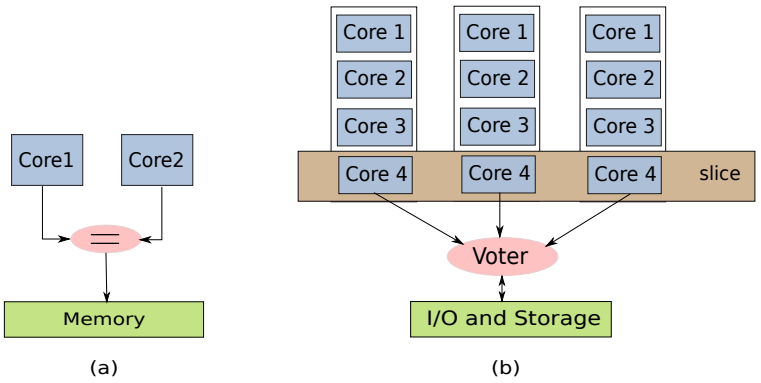

Fig. 4. NonStop architecture

4.2.2 *Loosely Coupled Processors.* This *lock-stepped* manner of execution is not desirable as it poses restrictions on possible power optimizations, advanced I/O protocols, and wide out-of-order machines. As an alternative, the *NonStop Advanced Architecture* or NSAA was proposed [Bernick et al. 2005]. NSAA follows a more loosely coupled approach. The comparison of results is done at a much larger granularity, thus, allowing for minor deviations in execution behavior, and in the frequency at which the processors run. The comparison is done only before every I/O operation.

The working of the NSAA architecture is explained in Figure 4(b). The architecture consists of groups of three 4-way Itanium SMP server processors. In a group, three corresponding cores, each in one server processor are referred to as a *slice*. All three cores in a slice execute the same program. They have separate memory address spaces. Please note that this architecture is only suitable for single threaded applications. Before every I/O event or interprocessor message, all three cores send a message to the voter. This message typically contains the content of the I/O event. The voter compares the results for three cores. If all of them agree, then we proceed normally. Otherwise, the recovery system marks the core that produced the erroneous output. It is possible that the faulty core had a soft error. In this case, we set the state of the faulty core to the state of a core that did not suffer from a fault by copying the register and memory contents. However, if a core repeatedly suffers from a fault or has a BIST failure, then it has most likely suffered from a hard error. In this case the entire SMP system is removed. The two cores in each slice can function in a standard DMR configuration, where we can detect errors, but we cannot correct them without using extensive checkpoint-rollback recovery. It is possible to later add another 4-way SMP system in place of the faulty one to complete each slice. This can be done though automatic reconfiguration or manually.

[Nomura et al. 2011] proposes *Sampling + DMR*, which employs DMR for a fraction of time. The remainder of the time has the system working in the uncovered mode. This allows massive savings in terms of power and performance (as now fewer comparisons are done). It must be noted that this is a solution that covers only permanent faults.

4.3 *SingleSlave* – DIVA Family

Austin proposed the *Dynamic Instruction Verification Architecture* (DIVA) [Austin 1999], which is one of the earliest works in this area, and serves as the basis for many subsequent proposals. This architecture uses a small slave coprocessor to verify the execution of the larger master core. Since the slave typically has lesser complexity, lower frequency, and uses fault tolerant transistors, it is significantly more immune to faults. The main research challenge is to ensure that the slave can keep up with the master in terms of performance.

4.3.1 *DIVA.* Figure 5 shows a high level view of the architecture.

We can view the DIVA pipeline as a deeper pipeline, with verification required before the commit stage. Elaborating further, the primary core performs all the functions of a standard processor (in-order or out-of-order) other than stores and commits. After the execute stage and memory access (for loads), the checker core verifies the execution of the master core in an in-order fashion. The verification

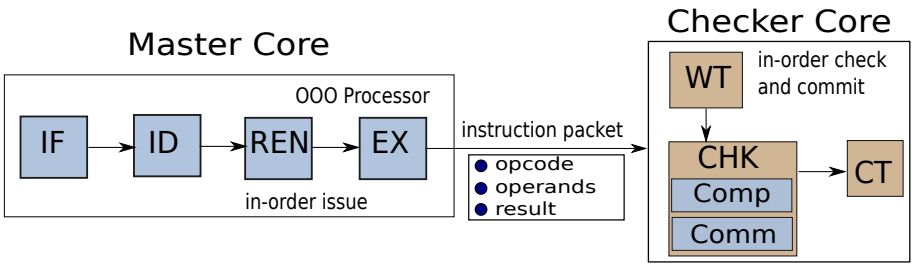

Fig. 5. *Basic Idea of DIVA*

consists of 2 parts : checking the communication(*Comm*), and checking the computation (*Comp*). Comm involves checking if the operands were read correctly. Meanwhile, performed in parallel, Comp involves re-performing the operation and checking if the results were computed correctly. If both the checks pass successfully, the checker commits the instruction and performs a store operation if required.

Additionally, the checker core is also interrupted by a watch-dog timer. This timer detects if the master is suffering from a deadlock. When a fault is detected (logical or deadlock), the checker flushes the pipeline of the master. It commits all the instructions till the erroneous instruction, and then starts the master from the subsequent instruction.

The checker is significantly simpler when compared to the master core because it processes instructions in-order. Thus, there is no need for complex structures like reorder buffers and instruction windows, which potentially increase the chances of a fault. The checker core can also match the master core's performance because it is expected to have a higher IPC, as it does not suffer from data or control hazards. It is provided with the values of all the operands by the master.

Unfortunately, this design has a few shortcomings. Both the checker and the master access the register file as well as the L1 cache. We will require extra ports. Secondly, we need more entries in the load store queue and reorder buffer, because it takes longer to commit an instruction. It is also possible that the master and checker might destructively interfere in the L1 cache.

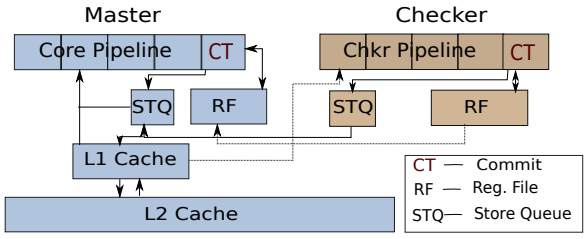

Fig. 6. *FastShared* Model

4.3.2 FastShared *Model.* The goal in this design [Chatterjee et al. 2000] is to mitigate the structural hazards due to the sharing of register files and the store

queue. The checker contains a copy of the register file and the store queue. The master core's register file is speculative. Upon the detection of a fault, the checker restores the contents of the master's register file with values from its own register file. The checker performs normal reads and writes to its register file. This optimization eliminates register file checkpoints, and also reduces the number of ports in both the register files.

There is a similar contention in the case of a shared store queue. Consequently, the checker maintains its own store queue. The core checks for a possibility of load-store forwarding by reading values from its own store queue. However, it does not remove any entries from its store queue. The checker removes entries from its queue, as well as the master core's queue, after a store commits.

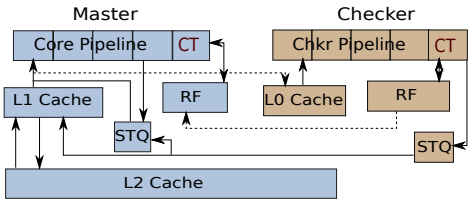

Fig. 7. *MiniDiva*

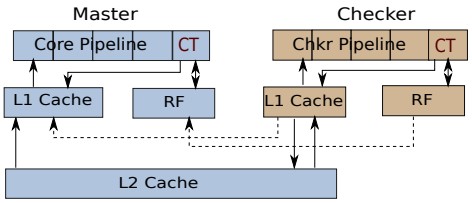

Fig. 8. *SplitDiva*

4.3.3 MiniDiva *and* SplitDiva. In the FastShared model, we still have the issue of increased contention at the level of the L1 cache. In the *MiniDiva* [Chatterjee et al. 2000] model (see Figure 7), we introduce an extra L0 cache for the checker. It is loaded with data that is touched by the master core. Along with reducing contention in the L1 cache, it also helps in prefetching values for the checker core. While writing data, the checker writes the data to its store queue and the dedicated L0 cache. The data is further propagated to the L1 cache, when it has free ports available.

We have, uptil now, assumed the L1 cache to be reliable. The *SplitDiva* [Chatterjee et al. 2000] design (see Figure 8) addresses this problem, and covers faults in the L1 cache. There are two L1 caches - one for the master core, and one for the checker. The master core's L1 is speculative. It is not allowed to writeback modified data to the shared L2 cache. Once a store is committed, the data is written to the checker's L1. Upon an eviction, the checker's L1 writes the data back to the L2 cache.

4.3.4 *Filtered Checkers.* [Yoo and Franklin 2008] proposes hierarchical verification, to minimize the performance penalty of the checker in a DIVA-like architecture by changing the coverage metric from *complete* to *subset*. The basic idea is as shown in Figure 9.

An instruction can be deemed critical or otherwise, by looking at various characteristics of the instruction such as the number of bits in the instruction that are useful, or the degree of usage of the result of an instruction. A filter checker sets the non-criticality of every instruction. The filtering of instructions can be proactive or reactive. Based on the non-criticality of an instruction, and the congestion at the checker, the checker decides whether or not to drop an instruction. This

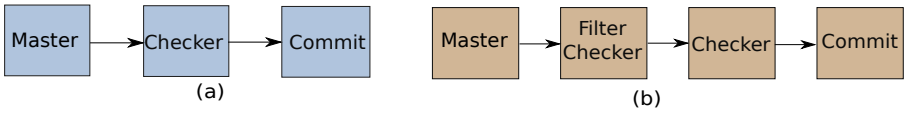

Fig. 9. *a) Traditional DIVA b) Hierarchical Verification*

scheme helps in reducing the power usage as well as the bandwidth between the core and the checker. Secondly, for very high IPC programs, the checker can become a bottleneck. By filtering out instructions that are not critical we can avoid this scenario.

4.4 *SingleSlave* – SlipStream Family

4.4.1 *SlipStream Processors.* Purser et. al., proposed the Slipstream processor [Sundaramoorthy et al. 2000; Purser et al. 2000] in 2000. This had a lot of conceptual similarities with DIVA (see Section 4.3). However, unlike DIVA, the slave core is equivalent to the master core in terms of size.

In this architecture, two versions of an application are executed on two separate cores. One version is called the *advanced stream* or the *A-stream*. The other is called the *redundant stream* or the *R-stream*. The R-stream lags behind the A-stream. The original paper primarily focused on increasing performance. They achieved this through reducing the number of dynamic instructions in the A-stream by eliminating redundant, and predictable computations. The R-stream could keep up with the A-stream in terms of performance because it received periodic dataflow and control-flow hints from the A-stream. The enhanced accuracy of different predictors ensured that the R-stream had an elevated IPC. However, on a side note the authors mention that this scheme can be used to increase reliability by treating the A-stream as the master and R-stream as the slave. The reason we mention this as a seminal technique is because the SlipStream approach forms the basis of a plethora of subsequent proposals. Most extensions to this idea leverage the hints from the A-stream to make the R-stream slower by either reducing its frequency, lowering its voltage, or by using bigger transistors. In either case, the R-stream is made more immune to faults. Secondly, since the core idea is to reduce the performance of the R-stream to make it more reliable, it satisfies our definition of a slave processor (see Section 2.2).

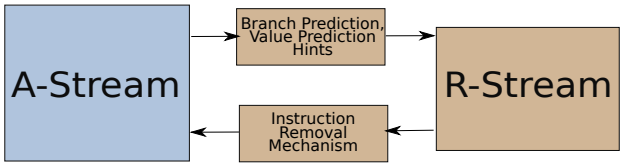

Fig. 10. *Slipstream processors*

Figure 10 shows a high level view of the architecture. The A-stream and R-stream processors are connected by a delay buffer, which is a queue containing dataflow and

control flow values. The A-stream writes entries into it, and the R-stream removes them. The R-stream treats these values as hints and trains its predictors (branch and load-latency). It thus has a higher IPC, and can match the A-stream in terms of performance. Subsequently, the R-stream compares the results of the A-stream with its own results. If a discrepancy is detected, then the R-stream initiates a process of recovery. It flushes its pipeline, and resets the A-stream. Subsequently, it restores its checkpointed register state and sends it to the A-stream also. For memory values, this paper assumes that the two processors have separate address spaces. A recovery controller tracks the store addresses that have been retired in the A-stream but not in the R-stream. Upon detection of a fault, the recovery controller restores the values of these addresses by reading them from the address space of the R-stream. The R-stream thus commits only correct values. We further observe that it has *complete* coverage, and it is mostly suitable for detecting and recovering from transient faults.

4.4.2 *Fingerprinting.* [Smolens et al. 2004] aims to reduce the inter-core communication bandwidth required to exchange information for verification. If the granularity of checking is small (say, every instruction executed is individually checked), then this involves frequent exchanges of messages between the cores while checking. If the granularity is increased, the amount of information to be exchanged also increases. Every change to the architectural state since the last verification round should be intimated to the other core. Fingerprinting suggests creating a cryptographic hash of all this information and exchanging simply the hashes, thus requiring much less bandwidth.

4.4.3 *RESEA and RECVF.* There have been other endeavors to reduce the usage of the Network-on-Chip. [Subramanyan 2010] proposes *Reduced Execution based on Simple Execution Assistance* (RESEA). Instead of forwarding the results of all computations to the checker, the master only sends the results of load and branch instructions. This is based on the fact that the *criticality* of these instructions is generally greater than the rest. Important data and control hazards are resolved, allowing the attainment of a higher IPC in the checker.

[Subramanyan et al. 2010] proposes *Reduced Execution based on Critical Value Forwarding* (RECVF). The critical path of execution is identified in the master, and the results of computations on this path (and not every instruction) are forwarded to the checker. The criticality of an instruction is determined heuristically. Two of the important heuristics determined are *freedN* and *fanoutN*. *freedN* marks an instruction as critical if on completion, it wakes up at least N instructions. *fanoutN* deems an instruction as critical if the value it produces is consumed by at least N live instructions.

4.4.4 *Dual Core Execution.* [Ma et al. 2007] follows in the same vein, proposing *Dual Core Execution* (DCE) to improve reliability and/or power efficiency. The idea again is to have the master core execute the program in a fast, highly accurate way with the checker executing redundantly to guarantee accuracy. Redundant execution implies a large power budget. Solutions are proposed to reduce the power consumed. One important source of power consumption is the large effective instruction window, which results as a result of redundant execution. The authors suggest dynamic adaptive sizing of the instruction window based on the behavior

of the program.

4.4.5 *Decoupled Execution.* [Garg and Huang 2008] proposes an architecture similar to Slipstream with a leading and a trailing core. The leading core runs a reduced version of the program called the *skeleton* program. The skeleton program is built by dynamic profiling and removal of code that is rarely exercised. Consequently, branches and the computation involved in computing the branch decision are removed. The leading core thus runs faster than the base version since it processes fewer instructions. Correctness is guaranteed by the trailing core that executes the entire program in a normal fashion. The trailing core is able to keep pace with the leader as it receives branch prediction hints from the latter, and the leader core helps prefetch memory locations. Memory operations also benefit from shorter latencies in the trailing core. Discrepancies detected between the leading and the trailing core trigger recovery mechanisms to restore a stable state.

It might not be always possible to dedicate a separate core for the process of checking. There are some novel proposals that suggest using dead cores in a chip, which have had faults detected by BISTs (Built-in Self Test). [Ansari et al. 2010] is one such proposal, which proposes using dead cores in a Slipstream fashion.

4.5 *MultiSlave* Approaches

In this section, we look at another seminal work, MSSP, and relevant extensions. The primary aim of MSSP was to improve performance. However, later works have used the same idea to improve reliability by dividing the work of checking among more than one slaves. As we shall see in Section 4.5.2, the main advantage of *MultiSlave* approaches is the reduction of power consumption.

4.5.1 *Master Slave Speculative Parallelization (MSSP).* Zilles and Sohi [Zilles and Sohi 2002] proposed the *Master/Slave Speculative Parallelization* (MSSP) framework in 2002. The authors propose to match the rate of the master and the slave by parallelizing the process of checking. They propose multiple checkers that each check a portion of the master's execution. This architecture was very influential in the field of speculative parallelization also, primarily because data flow violations between threads can be modeled as faults.

Here an approximate (referred to as *distilled* in the paper) version of the binary is run on a core designated as the master. A distilled binary lacks large chunks of code that are not likely to be executed in the common case. The process of creating it is an approximate compiler transformation devoid of any correctness guarantees.

Now, MSSP generates a checkpoint periodically (approximately after every hundred instructions). All instructions between two checkpoints constitute a *task*. A *task* is then assigned to a slave core. The slave core executes instructions from the original program, and not the approximate one.

Now if the master is fast enough, it may generate enough number of checkpoints to require the use of multiple slaves, executing different tasks, in parallel. When a slave completes its task, its state is compared with the corresponding checkpoint. If a discrepancy is found, which is a rare event, a recovery process is initiated. Subsequent, works have used this paper as a basis because it is possible to make minor changes to this scheme to ensure reliability. Instead of the distilled binary, we need to run the original binary on the master. Using this scheme, we can easily

detect transient faults because it is very unlikely that a master and a slave will be afflicted by a transient fault at exactly the same point of time.

4.5.2 *Coarse Grain Verification Parallelism.* [Rashid et al. 2005] extends MSSP for reliability. The main motivation is to reduce the total energy dissipated in the process of checking. In the *SingleSlave* scenario, the difference in frequencies of the master and the slave is limited to 20-30%. However, in a system with two slaves, we can run each slave at half the frequency. Since the supply voltage is roughly proportional to the frequency, the theoretical power consumption of the two slaves combined is 25% of that of one slave running at the nominal frequency.

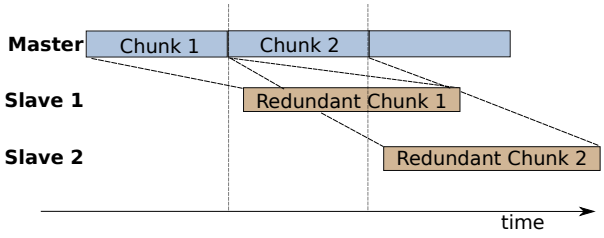

Fig. 11. *Coarse-grain verification parallelism*

The approach chosen by the authors involves having a large slack between the leading core and the checking cores. The unverified instructions are divided into *chunks*. Each chunk contains hundreds of instructions. Each checker is given the starting state of a chunk (register file contents and starting PC), and an ending state. The checker starts at the initial state, and keeps fetching instructions, till it reaches the final state, or times out. If it reaches the final state, and there is a discrepancy in the register contents or the values that need to be stored, then there is an error.

Figure 11 shows this scheme. The master divides its execution into chunks, and distributes the chunks across checkers. Each checker verifies the chunk assigned to it. If there are n parallel checkers, then each checker can theoretically run at $(1/n)^{th}$ the frequency. The main problem is to provide each chunk the correct view of the memory system. For this purpose the master maintains a list of uncommitted stores in a structure called the *Post-Commit Buffer*(PCB). Each chunk first checks the PCB for values before accessing the caches. [Zhao 2008] proposes optimizations to the PCB structure.

5. THREAD LEVEL APPROACHES

5.1 Summary

This section focuses on achieving reliability through multi-threading. We shall observe that achieving reliability through multiple threads is conceptually different from the approaches using multiple cores. It is much easier to communicate intermediate results across threads than cores. Secondly, since multiple threads are on the same physical core, there are no architectural differences that can be exploited. We did not have this constraint for core based schemes. The checker could have been very different from the master, like the DIVA scheme.

Proposal	Sub-Section	Perf. Ovhd.	HW Ovhd.	Checker Type	Coverage	Faults
IBM G5 [Spainhower et. al., 2005]	5.2	≈ 0%	Parallel Pipeline	*MultiMaster*	*Complete*	T MH
AR-SMT [Rotenberg, 1999]	5.2	10 − 30%	Delay Buffer	*MultiMaster*	*Complete*	T
SRT [Reinhardt et. al., 2000]	5.2	≈ 8%	Extra buffer/ queue	*MultiMaster*	*Complete*	T
SRTR [Vijaykumar et. al., 2002]	5.2	< 40%	Extra queue	*MultiMaster*	*Complete*	T
CRTR [Gomaa et. al., 2003]	5.2	< 30%	Extra Core	*MultiMaster*	*Complete*	T MH
DBCE [Vijaykumar et. al., 2002]	5.3	< 35%	Extra queue	*SingleSlave*	*Subset*	T
Fault Types : (T → Transient, M → Timing, H → Hard, D → Design)						
▭ → detect and correct, ▢ → only detect						

Table IV. Summary of thread level approaches

We observe that in the design space of multiple threads, the *MultiMaster* configuration (see Section 5.2) is relatively simpler to design as compared to the core based approaches. This is because the sources of non-determinism that afflict multiple cores such as clock skew, and variable delays in buses, are not relevant for multiple threads. Most of the time, threads share bus controllers, and their clocks are with each other. The *SingleSlave* approaches based on Slipstream are harder to design, because it is typically not possible to run different threads at different frequencies. However, there are some *SingleSlave* approaches that use the slave to check a subset of instructions (see Section 5.3).

5.2 *MultiMaster* Schemes – Complete Coverage

5.2.1 *IBM G5.* The earliest commercial system to incorporate thread level redundancy was the IBM G5 [Spainhower and Gregg 1999]. It had two pipelines running at the same time. All stores, and register writeback values were compared every cycle. If there was any discrepancy, both the pipelines were flushed. Effectively, the G5 architecture added one extra pipeline stage, whose job was to check the results of both the pipelines. We will outline several schemes that have tried to improve upon this basic idea.

5.2.2 *AR-SMT.* A seminal work in this area is called Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [Rotenberg 1999] proposed by Eric Rotenberg in 1999. He proposes to use two threads called the A-thread and R-thread (similar to SlipStream (see Section 4.4)). Both of them run the same copy of an application. However, there is a lag of tens of cycles between them. This is deliberately introduced to localize the effect of intermittent faults to one thread. The A thread runs ahead of the R-thread. It writes all its results to a delay buffer. The R-thread compares its results to values in the delay buffer. It commits an in-

struction only if the results match. If the R-thread, detects a discrepancy, both the threads roll back to the last checkpointed state. This state is the last committed state of the R-thread.

The two threads independently read and write from memory. They have separate address spaces. Consequently, there is no explicit sharing of values, even though there might be constructive or destructive interference in the caches for read only data, which might possibly be in shared pages. The paper further proposes using branch prediction and value prediction hints to speed up both the threads in a SlipStream fashion. The authors showed that it is possible to achieve complete fault coverage by incurring an overhead of 10% to 30%.

5.2.3 *SRT.* [Reinhardt and Mukherjee 2000] extended the AR-SMT idea and made it more generic. An important contribution of this paper is the introduction of the term *sphere of replication* shown in Figure 12(a) and 12(b). A *sphere of replication* is a reliable subcomponent of a system, which ensures reliability through computation redundancy. To support it, we need a well defined interface with the rest of the system. Additionally, we need to replicate the inputs to feed all the redundant units, and we need to compare the outputs for all of them. Figure 12(a) shows an example, in which the entire execution is replicated. In Figure 12(b), we just replicate the CPUs (not the memory or disk). IBM G5 follows this model.

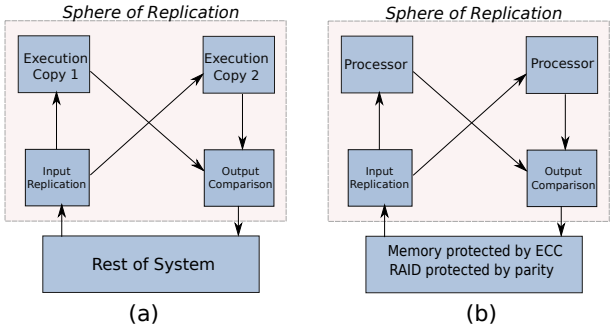

Fig. 12. *Sphere of Replication*

The authors observe that first, in an out-of-order system, the order of execution of instructions of the leading and the trailing threads cannot be guaranteed to be the same. Thus, a *lock-stepped* or per-cycle manner of comparing outcomes may not be advisable. Secondly, the overhead of essentially executing every instruction twice must be minimized as much as possible, while still providing fault tolerance. Another issue that requires handling is that the execution of synchronization constructs must be exactly similar in both the threads. Like Slipstream and DIVA, SRT passes branch outcomes between threads to speed the trailing checker thread. Furthermore, the leading thread effectively prefetches memory values for the trailing thread. This reduces its latency, and it can possibly run slower if required.

Two possible solutions for dealing with the synchronizing constructs are proposed: *Active Load Address Buffer* (ALAB) and *Load Value Queue* (LVQ).

The ALAB scheme forces loads to happen in pairs. After performing a load, the leading thread adds an entry in the ALAB if there is no previous entry for that address. Otherwise, it increments the count for that entry. When the trailing thread performs the same load, it decrements the count. If the count reaches zero, then the entry can be recycled. During the time the count is non-zero, the cache line cannot be replaced, or be written to. This ensures that the trailing thread gets the same value for the load. This scheme can suffer from deadlocks. The paper proposes appropriate solutions.

The LVQ is a much simpler solution. The leading thread inserts the results of load instructions in the LVQ upon committing the instruction. The trailing thread does not access the cache. Instead, it reads the results of load instructions in program order from the LVQ.

In this scheme and SRT, the trailing thread is assumed to hold the correct state. If there is a discrepancy between results when an instruction in the trailing thread commits, then it is necessary to flush the pipeline of the trailing thread, and restore the state of the leading thread.

5.2.4 *SRTR.* [Vijaykumar et al. 2002] proposes *Simultaneously and Redundantly Threaded Processors with Recovery* (SRTR) as an extension to SRT [Reinhardt and Mukherjee 2000]. The first problem with SRT recognized by the authors is that the leading thread is allowed to commit an instruction before verification. This alters the state of the system regardless of whether the instruction executed incorrectly or not. To avoid this, SRTR advocates checking the instructions before committing.

Verification of the outputs involves comparing the values of registers. These accesses increase pressure on the register file, which may degrade performance and increase power consumption. As a solution to this, the authors propose maintaining all unverified results of the leading thread in a *Register Value Queue* or RVQ. The trailing thread compares its results with the values stored in the RVQ.

[Gomaa et al. 2003] proposes *Chip-level Redundantly Threaded Processors with Recovery*, which is an extension of SRTR [Vijaykumar et al. 2002]. However, in this case each thread is located on a separate SMT core.

5.3 *SingleSlave – Subset* Coverage

5.3.1 *Dependence Based Checking Elision.* The authors of SRTR [Vijaykumar et al. 2002] also propose another scheme called *Dependency Based Checking Elision*(DBCE). They observe that it is possible to further reduce the pressure on shared data structures, and also speed up the checker. The basic insight here is to check the result of the last instruction in a long chain of dependent instructions. Unless there is logical masking, a fault anywhere in the long chain of computation should show up in the result of the last instruction with very high probability. If we can isolate some values, which have long dependence chains or graphs (backward slices), then we can check the execution of large parts of programs by verifying just this small subset of values. The authors propose to build such DBCE chains by propagating tokens across data dependent instructions.

5.3.2 *Other Approaches.* [Gomaa and Vijaykumar 2005] proposes Opportunistic Fault Tolerance, where redundant execution takes place only when the efficiency of single-thread execution is low.

[Kumar and Aggarwal 2008] also aims at improving efficiency through *Speculative Instruction Validation* (SpecIV). The basis for this work is that most instructions have predictable outputs. For example, almost every instance of a particular load instruction fetches from the same address. Similarly, reasoning may be extended to the addresses of store instructions, the outcomes of branch instructions and the results of computations. The idea is to leverage this predictability, and re-execute only those instructions that fail to behave as predicted. Thus, the number of instructions re-executed is reduced, thereby helping improve efficiency, while having only a minimal impact on the protection offered.

6. MULTITHREADED PROGRAMS

6.1 Summary

In Sections, 4, and 5, we tried to infer hardware faults by analyzing the execution of single threaded programs. In this section, we try to do the same by analyzing multi-threaded programs. We can divide the correctness of a multithreaded program into two parts – uniprocessor semantics and multiprocessor semantics. While considering uniprocessor semantics (Section 6.2), we assume that values produced by other processors are being correctly delivered at the right time. We just verify the execution of each individual thread of computation separately. While considering multiprocessor semantics, we are interested in the integrity of values transferred across processors, and the associated state machines. We separately look at detecting faults in the cache coherence protocols in Section 6.3 and in the implementation of the memory consistency model in Section 6.4.

Proposal	Sub-Section	Perf. Ovhd.	HW Ovhd.	Checker Type	Coverage	Faults
Reunion [Smolens et. al., 2006]	Uniprocessor Semantics(6.2.1)	$5-6\%$	100% + sign generator	*MultiMaster*	*Complete*	T MH
HDTLR [Rashid et. al., 2008]	Uniprocessor Semantics(6.2.2)	$\approx 40\%$	100% + PCB	*SingleSlave*	*Complete*	T MH
Repas [Sanchez et. al., 2009b]	Uniprocessor Semantics(6.2.2)	$< 21\%$	SVQ	*SingleSlave*	*Complete*	T MH
Token Coh. [Meixner et. al., 2007]	Coherence (6.3)	$< 7\%$ BW overhead	Central Verifier	*SingleSlave*	*Invariant*	T MHD
DVSC [Meixner et. al., 2005]	Consistency (6.4)	$< 20\%$	DIVA checker (ld/st reordering) + invariant checker	*SingleSlave*	Varied Coverage	T MHD
Fault Types : (T → Transient, M → Timing, H → Hard, D → Design)						
T → detect and correct, MHD → only detect						

Table V. Summary of approaches

Verifying uniprocessor semantics is similar to verifying executions at the core level. We thus discuss both *MultiMaster* and *SingleSlave* configurations. However, the main difference here is that synchronization operations force the master and

slave to converge thus reducing the lag to zero. The performance penalty is non-trivial, and consequently, this is a major research challenge. For verifying coherence and consistency mechanisms, we look at the *SingleSlave* model, and verify both the entire protocol as well as particular invariants. Table V summarizes the set of schemes.

6.2 Uniprocessor Semantics

We look at *MultiMaster* configurations in Section 6.2.1 and the *SingleSlave* configuration in Section 6.2.2.

6.2.1 MultiMaster - *Reunion.* [Smolens et al. 2006] proposes the *Reunion* scheme, which forms a logical pair of cores, where both of them execute the same thread. They compare their results with each other to detect the occurrence of faults. Each time a core finishes executing a pre-determined number of instructions, it creates a hash(*fingerprint*) of all the results and passes it to the partner core. Exchange of *fingerprints* consumes bandwidth, and so fixing the granularity of checking is an important decision.

The two cores of a pair are not completely symmetric - one is called the *vocal* core, and the other the *mute* core. The stores of the vocal core are allowed to propagate to the rest of the memory system, while those of the mute core are not. Secondly, unlike the vocal core, the mute core does not participate in coherence protocol actions. Now, we have observed in Section 4.2 that it is very difficult to make two cores operate in lock step. In modern processors there are a lot of non-deterministic delays due to process variation, and complex out-of-order execution.

Consequently, there will be a small amount of asynchrony between the two cores. This can lead to the *input incoherence problem*, i.e., two corresponding loads to the same address return different values. This is because of intervening stores to the same address. This scenario is indistinguishable from the case of a soft error in the case of Reunion. A somewhat related problem is that the two cores might observe different results for synchronization operations.

To check for discrepancies between the cores, we add a stage at the end of the pipeline to compare fingerprints across the two cores. If they don't match, then there might have been a soft error or an input incoherence event. In either case, both the cores flush their pipelines, and restore themselves to a safe state. It is assumed that in the absence of soft errors, the vocal core maintains a correct state. Furthermore, the vocal core transfers its register state to the mute core. To avoid the input incoherence problem, the mute core copies the value of the mismatched load instruction (if any) from the vocal core. Subsequently, both the cores resume execution. The authors observe that a slowdown due to synchronizing/serializing instructions is inevitable in master-checker architectures.

6.2.2 SingleSlave *Configurations.* We can naively use a solution based on DIVA or SlipStream (see Sections 4.3,4.4). To avoid costly multiprocessor memory checkpoints, we need to ensure that the effects of a fault are localized to the pipeline or possibly to higher level private caches. Consequently, we need to make a write and synchronization operation globally visible only when the entire execution till that point is guaranteed to be correct. This can be a prohibitive constraint for synchronization operations, if we do not allow master or slave threads to speculate

across them, especially in a proposal akin to DIVA or SlipStream. If we disallow speculation, then all the masters and slaves need to converge at the beginning of every synchronization operation. Researchers have measured a slowdown of upto 34% [Sanchez et al. 2009b] in this case. We thus need to mitigate this bottleneck in high performance implementations.

Dynamic Core Coupling
In DCC [LaFrieda et al. 2007] each thread has a redundant copy running on another core similar to *Reunion*. Here, there is some slack between the two cores(SlipStream pattern). With increasing slack, the probability of different forms of input incoherence events increases.

DCC attempts to solve this problem on a per-address basis. When the leading thread executes a load, it opens a *read window* for the address, and when it executes a store, it opens a *write window*. When both leading and trailing threads commit the load (store), the *read* (*write*) *window* is closed. Two read windows on the same address may overlap, but a read and a write window, or two write windows may not overlap. Enforcing this constraint ensures that shared memory operations in the leading and trailing thread behave in the same way.

HDTLR
Highly Decoupled Thread Level Redundancy or HDTLR [Rashid and Huang 2008] allows for large slacks. The cores that run the leading threads(masters) form a logical *computation wavefront* while the cores running the trailing threads(slaves) form the *verification wavefront*. Coherence activities in one wavefront do not affect the other.

The problem arising out of the large slack is that the sequence of memory operations need not be the same in the leading and trailing threads, due to complicated interactions between the threads of a wavefront. Thus, at the time of verification, states may not match, even when no error has occurred. Especially, race conditions pose a problem. We start out by dividing the entire execution into intervals called epochs. Both the wavefronts need to compare their state at the end of epochs.

To handle the issue of race conditions in programs, we partition epochs into *sub-epochs*. The partitioning is such that there are no two stores (even across threads) to the same address in the same sub-epoch. This is achieved by some amount of communication among the leaders, with each informing the others of its stores. If a leader has issued a store to an address, and receives a message from another core informing of a store to the same address, the leader places all further instructions in the next sub-epoch. The verification wavefront ensures that all instructions belonging to a sub-epoch are completed before moving onto the next one, thereby maintaining the ordering of shared memory operations as seen in the leading wavefront. All operations including memory races are replayed in exactly the same order.

REPAS
[Sanchez et al. 2009b; 2009a] proposes *Reliable Execution for Parallel Applications in Tiled CMPs* or REPAS. It extends earlier proposals CRTR [Gomaa et al. 2003] and SRT [Reinhardt and Mukherjee 2000] to provide a solution for checking multi-

threaded applications. The idea is to have two copies of each thread, both executing on the same SMT core. This removes the requirement for non-trivial inter-core bandwidths (DCC [LaFrieda et al. 2007]) and large central data-structures (HDTLR [Rashid and Huang 2008]). Stores made by the leading thread, go as far as the L1 cache, and on verification by the trailing thread, go beyond. Such a scheme requires the two threads to have a reasonably small slack between them. An additional structure called the *Store Value Queue* is used to support multiple unverified writes to the same cache block.

6.3 Multiprocessor Semantics - Cache Coherence

Most of the error detection schemes for cache coherence utilize the *SingleSlave* configuration consisting of a small dedicated unit that checks for errors in the execution of the protocol. Upon the detection of an error, the hardware needs to discard the victim memory operation or in some cases rollback to a checkpoint. We present three schemes, where each one of them considers a separate coverage model.

6.3.1 Complete *Coverage.* [J. F. Cantin and Smith. 2001] extends the DIVA scheme [Austin 1999] for cache coherent SMPs. Each node is paired with a checker processor, which checks its computation. The core sends coherence events and results to a private cache coherence checker immediately after a memory operation completes. The checker verifies the actions of the coherence protocol. The second phase of checking is global in nature. The checkers have a dedicated network for passing messages. They broadcast the states of lines, and then check for illegal global states.

6.3.2 Invariants . Token Coherence [Marty et al. 2005] is a token-based protocol for verifying cache coherence. Each block has N tokens, one of them being the owner token. For a memory element to be able to read a block it must hold at least one token. To write to a block, it must hold all the tokens. If a memory element wishes to read (or write) a block, and doesn't have sufficient tokens, then it broadcasts a token request. Other cores respond to this request by sending their tokens.

6.3.3 Symptoms . [Fernandez-Pascual et al. 2007] builds on Token Coherence. It handles faults in the on-chip interconnect. The detection of faults is solely based on time-outs. Every control message, which requires a reply according to the protocol is associated with a time-out period. Based on the kind of control message, expiry of the time-out results in resending of the message (fixed number of re-trials) or proceeding with the recovery mechanism. The recovery mechanism is to simply invalidate all current tokens, and generate new ones.

6.4 Multiprocessor Semantics – Memory Consistency

Similar to cache coherence, most of the schemes follow the *SingleSlave* DIVA pattern by proposing a small non-intrusive hardware unit to verify memory consistency. It takes inputs from the memory system. The schemes use invariants that are provably equivalent to the memory consistency model being verified.

[Meixner and Sorin 2005] proposes two schemes – DVSC-Direct and DVSC-Indirect. The DVSC schemes try to dynamically verify sequential consistency(SC) [Adve and Gharachorloo 1996]. The DVSC-Direct scheme tries to record the de-

pendence edges across multiple threads by timestamping all the memory accesses using a logical time base. If event, A, precedes event, B, then a logical time base guarantees that $time(A) < time(B)$. The dependence edges across memory operations in different threads are recorded through special units that track the loads and stores issued by each processor. Conceptually, there is a violation of SC, if there is a cycle in the dependence graph. Due to space constraints, this approach is impractical.

However, the authors use insights suggested by [Plakal et al. 1998] to make the design practical by actually verifying a set of sub-invariants. These sub-invariants are equivalent to SC. They, thus, propose the DVSC-Indirect protocol, which reduces the number of messages, and the amount of logging by an order of magnitude. [Chen et al. 2008] extends this scheme by using scalar timestamps. Lastly, DVMC [Meixner and Sorin 2009] extends DVSC to verify relaxed memory models. It checks the three sub-invariants – uniprocessor semantics, allowable memory reordering, cache coherence – that are provably equivalent to the memory model that it intends to verify. An extension of DVMC [Romanescu et al. 2010] recognizes physical address memory consistency (PAMC) and virtual address memory consistency (VAMC) as two separate problems. Virtual memory can create issues because of synonyms, and dynamic changes in page mapping.

7. SOFTWARE APPROACHES

7.1 Summary

This section discusses software-based approaches for providing fault tolerance in processors. In Section 7.2, we survey papers that attempt to verify only the computation and data flow. In Section 7.3, we look at approaches that try to exclusively verify the control flow. Section 7.4 discusses solutions that are capable of handling errors in both data flow as well as in the control flow. Lastly, approaches specially catering to the multi-processor domain are discussed in Section 7.5.

It must be noted that some of these solutions are purely software-based (for example, *EDDI* (Section 7.2)), while others require a certain amount of hardware support (for example, *SWAT* (Section 7.4.2)). A summary of the seminal papers in this area is provided in Table VI.

7.2 Computation and Dataflow Errors

In this section, we focus on techniques that try to infer hardware faults by analyzing errors in the computation and data flow of test programs. The most popular set of approaches use a *MultiMaster* configuration(Section 7.2.1). They run multiple copies of the same instruction and compare the results. Section 7.2.2 describes approaches that use the *SingleSlave* configuration. The slave instructions mostly check invariants.

7.2.1 MultiMaster *Schemes.*

Complete Coverage

[Rebaudengo et al. 1999] describes a general scheme that consists of duplicating all the computation and data. At compile time, each instruction is duplicated with a different set of registers. For instance, an operation of the form $a = b + c$; is

Proposal	Sub-Section	Perf. Ovhd.	HW Ovhd.	Checker Type	Coverage	Faults
EDDI [Oh et. al., 2002]	Comp. and Data Flow(7.2)	upto 100%	nil	*MultiMaster*	*Subset*	T
DDFV [Meixner et. al., 2007]	Data Flow (7.2)	≈ 1.8%	Signature HW + Modifications to ROB,RF	*SingleSlave*	*Invariant*	T MH
CCA [Kanawati et. al., 1996]	Control Flow (7.3)	20 − 50%	nil	*SingleSlave*	*Invariant*	T MH
SWIFT [Reis et. al., 2005]	All errors (7.4.2)	≈ 41%	ECC in memory	Multiple Checkers	Varied Coverage	T H
SWAT [Li et. al., 2008]	All errors (7.4.2)	≈ 0%	Symptom Detector HW	*SingleSlave*	*Invariant*	T MH
TSOtool [Hangal et. al., 2004]	Total Store Order (7.5)	Not Applicable	Not Applicable	*SingleSlave*	*Complete*	TMHD
mSWAT [Hari et. al., 2005]	All errors (7.5)	≈ 0%	Record/Replay Support	*MultiMaster*	*Invariant*	T MH
Fault Types : (T → Transient, M → Timing, H → Hard, D → Design)						
→ detect and correct, → only detect						

Table VI. Summary of approaches

transformed to $a_1 = b_1 + c_1; a_2 = b_2 + c_2;$. These two statements are followed by an assertion statement that checks for the equality of a_1 and a_2. Thus, any errors in the computation and the reading(writing) of operands(results) are detected with a high probability, since the recurrence of the same fault (only transient faults are covered by this technique) during the redundant computation is highly unlikely. [Chang et al. 2006] proposes SWIFT-R that uses TMR to both detect and recover from faults.

Subset Coverage

Error Detection by Duplicated Instructions (EDDI)[Oh et al. 2002b] extends this idea by changing the coverage model to *subset*. It checks the values that are written to memory or determine a branch direction. [Pattabiraman et al. 2007] also proposes to reduce the overhead through *Critical Value Recomputation*(CVR), a static technique, wherein during compile-time, the data-flow graph is analyzed, and results that have a high *fanout* (number of consuming instructions) are deemed critical values. The computation of these values is then covered by duplication. [Lyle et al. 2009] extends CVR, proposing the implementation of the checks in an on-chip progammable array to accelerate execution.

7.2.2 *Singleslave Schemes.* [Meixner and Sorin 2007] proposes *Dynamic Data Flow Verification* (DDFV), which starts out by creating signatures for each basic block at compile time. A signature is a hash of the *histories* of each register, where, *history*, refers to the ordered list of registers, whose values were used in setting the

value of the concerned register. This signature is embedded in the program source. During runtime, special DDFV hardware computes the signature for each executed block, and compares them with the statically determined values.

7.3 Control Flow Errors

A more challenging problem is the detection of errors that alter the control flow. This can be caused by errors in the backward slice of a branch instruction or by faults in the fetch logic. The general pattern of approaches to verify the control-flow of a program is to compare the observed basic block sequence with a set of patterns computed through static analysis. For example, let us assume that basic block B can be preceded by only basic blocks A, and C. If B is preceded by basic block, D, then we can infer an error. All the schemes that we have surveyed primarily use the *SingleSlave* pattern, and are mostly tailored to detect transient faults. They avoid redoing any computation to verify the control flow; instead, they rely on verifying invariants.

[Schuette and Shen 1994] proposes to assign identifiers to basic blocks at compile-time. There is a global register, KEY, which contains the identifier of the current basic block being executed. The first instruction of every basic block updates the basic block id in the KEY register. The compiler further embeds checking statements within a basic block to test the value of the KEY register. If the value is incorrect, then we can infer a control flow error. This means that the processor has not executed the first instruction of a basic block.

[Kanawati et al. 1996] proposes *Control-Flow Checking using Assertions*(CCA), which adds more information to every basic block. During compilation, each basic block is assigned a block identifier BID, and a control flow identifier $CFID$. All blocks sharing the same parent block (or predecessor block in the control flow) have the same $CFID$. The BIDs help detect entry into the middle of a basic block similar to[Schuette and Shen 1994], while the $CFID$s serve to verify that upon exit from the current basic block, a legally succeeding block (in terms of the control flow) is executed. A drawback of this approach is the extra storage to maintain the BID and $CFID$, and the time needed to compare them.

[Alkhalifa et al. 1999] proposes *Enhanced Control-Flow Checking using Assertions*(ECCA). By using prime numbers for the identifiers and number theoretic techniques, it is possible to compress the BID and $CFID$ to a single number. Secondly, ECCA proposes to create large hyperblocks from more than one basic blocks. By performing checking at the hyperblock level, we can further reduce the time and space overhead.

[Oh et al. 2002a; Goloubeva et al. 2003] propose another signature-based technique termed *Control-Flow Checking through Software Signatures*(CFSS). The authors propose to perform the invariant check in only a subset of blocks, as opposed to every single block.

[Venkatasubramanian et al. 2003] proposes *Assertions for Control-Flow Checking*(ACFC). The novelty in this technique is that instead of associating an identifier with each basic block (and then updating a common register during runtime), ACFC maintains *Execution Status* words, with each block being represented by a single bit in these words. The flow of control modifies these *Execution Status* words based on which block has just been entered, and the invariant check is based on

their current value.

[Borin et al. 2006; Vemu and Abraham 2006] propose the *Edge Control-Flow Technique* where we ascertain the signature of the next block at the end of every basic block. After entering the target block, the compiler inserts code to ascertain that the correct basic block is being executed. This is done through comparing the id of the predicted basic block with the id of the basic block currently being executed. [Borin et al. 2006] further proposes to use the same method at the level of larger hyperblocks.

7.4 All Types of Errors

In this section we look at software schemes that detect hardware faults by considering programs holistically.

7.4.1 *Multimaster Techniques.* [Banerjee et al. 1990] proposes application-specific techniques to detect errors in matrix multiplication, Gaussian elimination, and Fourier transforms. Let us consider a matrix multiplication problem – $C = A * B$. Here a controller core gives each master core, i, the matrix B and a portion of the matrix A, A_i. Each core i computes $C_i = A_i * B$ and sends it to the controller. The controller then aggregates these components (C_i) to derive C. Now, to enable the detection of faults, the controller core makes a logical pair of masters called *mates*. Each mate computes the sub-matrix, C_i. The two mates exchange these matrices (whole or signature) among each other. If there is a discrepancy, then at least one of the mates is faulty. The controller can then assign the task to another set of masters.

Classical techniques described in [Koren and Krishna 2007] propose running n copies of the same program, and decide the final or intermediate outputs through voting. There is another famous paradigm known as N-version software [Avizienis 1985], which considers n independently designed versions of the same program. Such approaches can ensure a better detection of design faults because it is possible that one version might be exercising a certain functional unit such that a bug is exposed. Foutris et. al. [Foutris et al. 2011] consider a variant of n-version programming by generating code for the same benchmark in multiple ways. For example, it is possible that one version can use an instruction like *swap* and the other version can replace it with a sequence of moves. These approaches increase the diversity of the programs and help us detect a wide variety of faults, inclusive of design and intermittent faults, because different sequences of instructions exercise different functional units.

7.4.2 *Singleslave Techniques.* We categorize the different techniques for the *SingleSlave* configuration based on coverage.

Subset Coverage

[Constantinides et al. 2007] aims at a flexible reliability solution through a software-based BIST (Built in self test) technique. Periodically, the firmware suspends execution and runs tests on the processor. These tests can be explicitly invoked by the user or the compiler as the instruction set is augmented with additional monitoring capabilities. [Pellegrini and Bertacco 2010] also proposes to periodically run tests, but only on units that are being exercised by the user's application, in a bid to

reduce overhead. Another solution, *Relax* [de Kruijf et al. 2010], uses hardware error detection-recovery techniques like *Argus* [Meixner et al. 2007]. These are used only on those instructions for which the programmer or compiler have explicitly requested for additional protection.

Fault Detection Based on Invariants

[Ersoz et al. 1985] proposes software implementations of a watchdog processor with which other applications can register invariant assertions. The *WatchDog Task* periodically verifies all registered invariants, enabling the detection of errors. [Leeke et al. 2011] tries to automatically deduce the invariants through a combination of rigorous static analysis and profiling.

[Reis et al. 2005] proposes SWIFT, a software-based transient error detection mechanism, inclusive of both control and data flow errors. It is blend of duplication for dataflow errors, ECC-based memory protection, and invariant based control flow error detection. Additionally, it proposes some optimizations to reduce overhead, like performing checks only on those values that are to be written to memory.

Fault Detection Based on Symptoms

Some recent works propose techniques that are implemented partly in hardware and partly in software. [Li et al. 2008] introduces *Software Anomaly Treatment* (SWAT). The findings illustrate that a vast majority of the hardware faults that propagate into software can be detected with simple hardware support. The principle is to detect suspicious activities (*symptoms*) such as *(i)* fatal hardware traps, *(ii)* abnormal application exit, *(iii)* application or OS hangs, and *(iv)* abnormally high OS activity. The occurrence of one or more of these activities implies the occurrence of an error with good probability. In [Sahoo et al. 2008], the authors aim to improve the coverage of SWAT with iSWAT. They suggest the usage of training inputs to determine likely ranges of selected program variables. At runtime, we can verify these invariants.

7.5 Detecting Faults in Multiprocessors

We present two schemes here – mSWAT and TSOtool. mSWAT can detect transient, timing, and hard faults; TSOtool, can also detect design faults.

7.5.1 *mSWAT.* mSWAT [Hari et al. 2009] is an extension to SWAT (Section 7.4.2) for multi-core systems. mSWAT aims to identify the faulty core in a set of n cores, and it further assumes that only one core has suffered a fault. Whenever the mSWAT hardware observes a symptom, it rolls back the entire system to a correct checkpoint, and restarts execution. If the symptom does not recur, then we can infer a transient fault; otherwise, we can infer a permanent fault or software bug. In the latter case, mSWAT restarts the application from the checkpoint, and collects a detailed trace of all the events. The trace is a log of all loads performed by each core. In the first round, each core sends its checkpoint and trace to another core. The other cores executes the received trace, and compares the outputs. A lack of divergence implies a software bug. If there two divergences, then there must be a core in common between the two pairs since we assume only one faulty core.

This core can be declared faulty. If there is just one divergence, then we have two suspect cores (core A that generated the trace, and core B that ran it). In the next round, we run the trace that had a divergence on a fault free core (known from the first round). If there is a divergence, then core A is at fault, else, core B is at fault.

7.5.2 *TSOtool.* The TSOtool project [Hangal et al. 2004] aims to find errors in the implementation of a processor's memory model. In specific, it checks for the TSO (total store order), which allows the processor to relax the write to read ordering. It first generates a set of multi-threaded programs that randomly read and write from a small set of locations. Each write operation stores a unique value to memory. Now, by analyzing the values read by load instructions, it is possible to create a dependence graph between all the dynamic instructions. The graph will have some edges induced by the program order and some edges from producing stores to consuming loads. The last part of the algorithm checks for cycles in this graph. If there are cycles, then we can be sure that there is a violation of the underlying memory model, TSO. The algorithm for finding cycles has performance issues since the size of the graph is typically very large. Chen et. al. [Chen et al. 2009] propose algorithms to speed up this process for memory models that implement atomic writes. The complexity of their algorithm is linear in the number of operations.

8. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we presented a comprehensive survey of most of the state of the art techniques for detecting faults in processors. In Section 2.2, we proposed a taxonomy of different checker architectures. We observed that there is a significant variability in the nature of techniques for solutions at the circuit/pipeline and the software level. However, solutions at the thread, core, and multiprocessor level follow a set of major patterns, and have a roughly similar structure.

Most of the techniques that we presented were proposed in the last decade. The next decade is extremely exciting for computer architecture research since a host of new technologies [Torrellas 2009] such as optical interconnects, non-volatile RAMs, FinFets , 3D stacking, and near threshold operation are expected to be introduced. These new technologies have very different fault mechanisms, and consequently detecting them with a minimal amount of hardware is a challenge.

In specific, futuristic chips are expected to have extensive on-chip networks possibly containing many different types of interconnects. We would need a dedicated checker substrate for the network to verify different safety and liveness properties. Novel interconnect technologies such as optical interconnects are sensitive to process variation, and thus their health needs to be monitored. Non-volatile memory extends the lifetime of a fault. Faults can persist even after a system restart, and can propagate to lower levels and I/O devices more easily. Consequently, early fault detection, and efficient fault containment are important challenges. Novel process technologies using FinFets, high k transistors, and 3D stacking allow for greater on-chip transistor densities. We would need to detect faults at multiple layers in the 3D stack simultaneously. For example, if we have processor and memory on a single die, it is possible that high temperature in the processor layer can cause faults in the DRAM memory layer. Any fault detection mechanism for such processors will probably need to take such interactions into account. We thus foresee

an exciting decade ahead for research in fault detection architectures.

REFERENCES

Adve, S. V. and Gharachorloo, K. 1996. Shared memory consistency models: A tutorial. *IEEE Computer 29,* 12, 66–76.

Ahmed, R., Frazier, R., and Marinos, P. 1990. Cache-aided rollback error recovery (carer) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium.* 82 –88.

Alkhalifa, Z., Nair, V., Krishnamurthy, N., and Abraham, J. 1999. Design and evaluation of system-level checks for on-line control flow error detection. *Parallel and Distributed Systems, IEEE Transactions on 10,* 6 (jun), 627 –641.

Ansari, A., Feng, S., Gupta, S., and Mahlke, S. 2010. Necromancer: enhancing system throughput by animating dead cores. In *Proceedings of the 37th annual international symposium on Computer architecture.* ISCA '10. ACM, New York, NY, USA, 473–484.

Austin, T. 1999. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on.* 196 –207.

Avirneni, N. D. P., Subramanian, V., , and Somani, A. K. 2009. Soft error mitigation schemes for high performance and aggressive designs. In *SELSE.*

Avizienis, A. 1985. The n-version approach to fault-tolerant software. *Software Engineering, IEEE Transactions on SE-11,* 12 (dec.), 1491 – 1501.

Bacchini, F., Damiano, R. F., Bentley, B., Baty, K., Normoyle, K., Ishii, M., and Yogev, E. 2004. Verification: what works and what doesn't. In *DAC.* 274.

Banerjee, P., Rahmeh, J., Stunkel, C., Nair, V., Roy, K., Balasubramanian, V., and Abraham, J. 1990. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on 39,* 9 (sep), 1132 –1145.

Bartlett, J. F. 1981. A nonstop kernel. In *Proceedings of the eighth ACM symposium on Operating systems principles.* SOSP '81. ACM, New York, NY, USA, 22–29.

Bernick, D., Bruckert, B., Vigna, P. D., Garcia, D., Jardine, R., Klecka, J., and Smullen, J. 2005. Nonstop® advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks.* DSN '05. IEEE Computer Society, Washington, DC, USA, 12–21.

Blaauw, D., Kalaiselvan, S., Lai, K., Ma, W.-H., Pant, S., Tokunaga, C., Das, S., and Bull, D. 2008. Razor ii: In situ error detection and correction for pvt and ser tolerance. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International.* 400 –622.

Blome, J., Feng, S., Gupta, S., and Mahlke, S. 2007. Self-calibrating online wearout detection. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO 40. IEEE Computer Society, Washington, DC, USA, 109–122.

Blum, M. and Wasserman, H. 1996. Reflections on the pentium division bug. *IEEE Trans. Comput. 45,* 385–393.

Borin, E., Wang, C., Wu, Y., and Araujo, G. 2006. Software-based transparent and comprehensive control-flow error detection. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on.* 13 pp.

Borkar, S. 2004. Microarchitecture and design challenges for gigascale integration. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* MICRO 37. IEEE Computer Society, Washington, DC, USA, 3–3.

Carretero, J., Chaparro, P., Vera, X., Abella, J., and González, A. 2009. End-to-end register data-flow continuous self-test. *SIGARCH Comput. Archit. News 37,* 3 (June), 105–115.

Chang, J., Reis, G., and August, D. 2006. Automatic instruction-level software-only recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on.* 83 –92.

Chatterjee, S., Weaver, C., and Austin, T. 2000. Efficient checker processor design. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. MICRO 33. ACM, New York, NY, USA, 87–97.

Chen, K., Malik, S., and Patra, P. 2008. Runtime validation of memory ordering using constraint graph checking. In *HPCA*. 415–426.

Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., and Pan, H. 2009. Fast complete memory consistency verification. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. 381 –392.

Constantinides, K., Mutlu, O., and Austin, T. 2008. Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. 282 –293.

Constantinides, K., Mutlu, O., Austin, T., and Bertacco, V. 2007. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. 97–108.

de Kruijf, M., Nomura, S., and Sankaralingam, K. 2010. Relax: an architectural framework for software recovery of hardware faults. *SIGARCH Comput. Archit. News 38,* 3 (June), 497–508.

Ernst, D., Kim, N. S., Das, S., Pant, S., Rao, R., Pham, T., Ziesler, C., Blaauw, D., Austin, T., Flautner, K., and Mudge, T. 2003. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. 7 – 18.

Ersoz, A., Andrews, D. M., and J., M. E. 1985. The watchdog task:concurrent error detection using assertions. Tech. rep., Center for Reliable Computing, Stanford Univ., CA, CRC-TR 85-8.

Fernandez-Pascual, R., Garcia, J., Acacio, M., and Duato, J. 2007. A low overhead fault tolerant coherence protocol for cmp architectures. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. 157 –168.

Foutris, N., Gizopoulos, D., Psarakis, M., Vera, X., and González, A. 2011. Accelerating microprocessor silicon validation by exposing isa diversity. In *MICRO*. 386–397.

Garg, A. and Huang, M. C. 2008. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. IEEE Computer Society, Washington, DC, USA, 306–317.

Goloubeva, O., Rebaudengo, M., Sonza Reorda, M., and Violante, M. 2003. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*. 581 – 588.

Gomaa, M., Scarbrough, C., Vijaykumar, T. N., and Pomeranz, I. 2003. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*. ISCA '03. ACM, New York, NY, USA, 98–109.

Gomaa, M. and Vijaykumar, T. 2005. Opportunistic transient-fault detection. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*. 172 – 183.

Hangal, S., Vahia, D., Manovit, C., and Lu, J.-Y. J. 2004. Tsotool: A program for verifying memory systems using the memory consistency model. In *Proceedings of the 31st annual international symposium on Computer architecture*. ISCA '04. 114–.

Hari, S., Li, M.-L., Ramachandran, P., Choi, B., and Adve, S. 2009. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. 122 –132.

Horst, R. and Chou, T. 1985. The hardware architecture and linear expansion of tandem nonstop systems. Tech. rep., Tandem Computers.

Hu, J. S., Link, G. M., John, J. K., Wang, S., and Ziavras, S. G. 2005. Resource-driven optimizations for transient-fault detecting superscalar microarchitectures. In *Proceedings of the 10th Asia-Pacific conference on Advances in Computer Systems Architecture*. ACSAC'05.

J. F. Cantin, M. H. L. and Smith., J. E. 2001. Dynamic verification of cache coherence protocols. In *Workshop on Memory Performance Issues*. WMPI '01.

Joel, B., Gray, J., and Horst, B. 1986. Fault tolerance in tandem computer systems. Tech. rep., Tandem Computers.

Kanawati, G., Nair, V., Krishnamurthy, N., and Abraham, J. 1996. Evaluation of integrated system-level checks for on-line error detection. In *Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International*. 292 –301.

Koren, I. and Krishna, C. 2007. *Fault Tolerant Systems*. Morgan Kaufmann.

Kumar, S. and Aggarwal, A. 2008. Speculative instruction validation for performance-reliability trade-off. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 405 –414.

LaFrieda, C., Ipek, E., Martinez, J., and Manohar, R. 2007. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. 317 –326.

Leeke, M., Arif, S., Jhumka, A., and Anand, S. 2011. A methodology for the generation of efficient error detection mechanisms. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. 25 –36.

Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S., and Zhou, Y. 2008. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ASPLOS XIII. ACM, New York, NY, USA, 265–276.

Lyle, G., Chen, S., Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. 2009. An end-to-end approach for the automatic derivation of application-aware error detectors. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. 584 –589.

Ma, Y., Gao, H., Dimitrov, M., and Zhou, H. 2007. Optimizing dual-core execution for power efficiency and transient-fault recovery. *IEEE Transactions on Parallel and Distributed Systems 18*, 1080–1093.

Marty, M. R., Bingham, J. D., Hill, M. D., Hu, A. J., Martin, M. M. K., and Wood, D. A. 2005. Improving multiple-cmp systems using token coherence. *High-Performance Computer Architecture, International Symposium on 0*, 328–339.

Meixner, A., Bauer, M. E., and Sorin, D. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. IEEE Computer Society, Washington, DC, USA, 210–222.

Meixner, A. and Sorin, D. 2007. Error detection using dynamic dataflow verification. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. 104 –118.

Meixner, A. and Sorin, D. J. 2005. Dynamic verification of sequential consistency. In *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05. IEEE Computer Society, Washington, DC, USA, 482–493.

Meixner, A. and Sorin, D. J. 2009. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Transactions on Dependable and Secure Computing 6*, 18–31.

Mourad, S. and Zorian, Y. 2000. *Principles of Testing Electronic Systems*. Wiley-Interscience.

Nickel, J. B. and Somani, A. K. 2001. Reese: A method of soft error detection in microprocessors. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*. DSN '01.

Nomura, S., Sinclair, M. D., Ho, C.-H., Govindaraju, V., de Kruijf, M., and Sankaralingam, K. 2011. Sampling + dmr: practical and low-overhead permanent fault detection. *SIGARCH Comput. Archit. News 39,* 3 (June), 201–212.

Oh, N., Shirvani, P., and McCluskey, E. 2002a. Control-flow checking by software signatures. *Reliability, IEEE Transactions on 51,* 1 (mar), 111 –122.

Oh, N., Shirvani, P., and McCluskey, E. 2002b. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on 51,* 1 (mar), 63 –75.

Ossi, E. J., Limbrick, D. B., Robinson, W. H., and Bhuva, B. L. 2009. Soft-error mitigation at the architecture-level using berger codes and instruction repetition. In *SELSE*. SELSE.

Parashar, A., Gurumurthi, S., and Sivasubramaniam, A. 2004. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. *SIGARCH Comput. Archit. News 32,* 2.

Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. 2007. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*. 211 –216.

Pellegrini, A. and Bertacco, V. 2010. Application-aware diagnosis of runtime hardware faults. In *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '10. IEEE Press, Piscataway, NJ, USA, 487–492.

Plakal, M., Sorin, D. J., Condon, A. E., and Hill, M. D. 1998. Lamport clocks: verifying a directory cache-coherence protocol. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '98. ACM, New York, NY, USA, 67–76.

Prvulovic, M., Zhang, Z., and Torrellas, J. 2002. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. 111 –122.

Purser, Z., Sundaramoorthy, K., and Rotendberg, E. 2000. A study of slipstream processors. *Microarchitecture, IEEE/ACM International Symposium on 0*, 269.

Qureshi, M. K., Mutlu, O., and Patt, Y. N. 2005. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *DSN*. 434–443.

Rashid, M. and Huang, M. 2008. Supporting highly-decoupled thread-level redundancy for parallel programs. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 393 –404.

Rashid, M. W., Tan, E. J., Huang, M. C., and Albonesi, D. H. 2005. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. *Parallel Architectures and Compilation Techniques, International Conference on 0*, 315–328.

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

Rebaudengo, M., Sonza Reorda, M., Torchiano, M., and Violante, M. 1999. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*. 210 –218.

Reinhardt, S. K. and Mukherjee, S. S. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*. ISCA '00. ACM, New York, NY, USA, 25–36.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., and August, D. I. 2005. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*. CGO '05. IEEE Computer Society, Washington, DC, USA, 243–254.

Romanescu, B. F., Lebeck, A. R., and Sorin, D. J. 2010. Specifying and dynamically verifying address translation-aware memory consistency. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 323–334.

Rotenberg, E. 1999. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. *Fault-Tolerant Computing, International Symposium on 0*, 84.

Sahoo, S., Li, M.-L., Ramachandran, P., Adve, S., Adve, V., and Zhou, Y. 2008. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. 70 –79.

Sanchez, D., Aragon, J., and Garcia, J. 2009a. Extending srt for parallel applications in tiled-cmp architectures. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 1 –8.

Sanchez, D., Aragon, J., and Garcia, J. 2009b. Repas: Reliable execution for parallel applications in tiled-cmps. In *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Lecture Notes in Computer Science, vol. 5704. Springer Berlin / Heidelberg, 321–333. 10.1007/978-3-642-03869-3_32.

Sarangi, S. R. 2007. Techniques to mitigate the effects of congenital faults in processors. Ph.D. thesis, Champaign, IL, USA. AAI3270016.

Sarangi, S. R., Greskamp, B., and Torrellas, J. 2006. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *Proceedings of the International Conference on Dependable Systems and Networks*. 301–312.

Sarangi, S. R., Narayanasamy, S., Carneal, B., Tiwari, A., Calder, B., and Torrellas, J. 2007. Patching processor design errors with programmable hardware. *IEEE Micro 27,* 1, 12–25.

Schuette, M. and Shen, J. 1994. Exploiting instruction-level parallelism for integrated control-flow monitoring. *Computers, IEEE Transactions on 43,* 2 (feb), 129 –140.

Shyam, S., Constantinides, K., Phadke, S., Bertacco, V., and Austin, T. 2006. Ultra low-cost defect protection for microprocessor pipelines. In *In Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*. 73–82.

Smith, J. and Sohi, G. 1995. The microarchitecture of superscalar processors. *Proceedings of the IEEE 83,* 12 (dec), 1609 –1624.

Smolens, J. C., Gold, B. T., Falsafi, B., and Hoe, J. C. 2006. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. IEEE Computer Society, Washington, DC, USA, 223–234.

Smolens, J. C., Gold, B. T., Kim, J., Falsafi, B., Hoe, J. C., and Nowatzyk, A. G. 2004. Fingerprinting: bounding soft-error detection latency and bandwidth. *SIGARCH Comput. Archit. News 32*, 224–234.

Sorin, D. J., Martin, M. M. K., Hill, M. D., and Wood, D. A. 2002. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th annual international symposium on Computer architecture*. ISCA '02. IEEE Computer Society, Washington, DC, USA, 123–134.

Spainhower, L. and Gregg, T. A. 1999. Ibm s/390 parallel enterprise server g5 fault tolerance: a historical perspective. *IBM J. Res. Dev. 43*, 863–873.

Subramanyan, P. 2010. Efficient fault tolerance in chip multiprocessors using critical value forwarding. M.S. thesis, Supercomputer Education and Research Center, Indian Institute of Science, Bangalore.

Subramanyan, P., Singh, V., Saluja, K., and Larsson, E. 2010. Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. 121 –130.

Sundaramoorthy, K., Purser, Z., and Rotenberg, E. 2000. Slipstream processors: improving both performance and fault tolerance. *SIGPLAN Not. 35*, 257–268.

Tiwari, A. and Torrellas, J. 2008. Facelift: Hiding and slowing down aging in multicores. In *MICRO*. 129–140.

Torrellas, J. 2009. Architectures for extreme-scale computing. *IEEE Computer 42,* 11, 28–35.

Vemu, R. and Abraham, J. 2006. Ceda: control-flow error detection through assertions. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*. 6 pp.

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

Vijaykumar, T., Pomeranz, I., and Cheng, K. 2002. Transient-fault recovery using simultaneous multithreading. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. 87 –98.

Wang, N. and Patel, S. 2006. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on 3,* 3 (july-sept.), 188 –201.

Yoo, J. and Franklin, M. 2008. Hierarchical verification for increasing performance in reliable processors. *Journal of Electronic Testing 24*, 117–128. 10.1007/s10836-007-5037-z.

Zandian, B., Dweik, W., Kang, S. H., Punihaole, T., and Annavaram, M. 2010. Wearmon: Reliability monitoring using adaptive critical path testing. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. 151 –160.

Zhao, H. 2008. Memory buffer element optimization for decoupled thread level redundancy. M.S. thesis, Department of Electrical and Computer Engineering The College School of Engineering and Applied Science University of Rochester, Rochester, New York.

Ziegler, J. F., Curtis, H. W., Muhlfeld, H. P., Montrose, C. J., Chin, B., Nicewicz, M., Russell, C. A., Wang, W. Y., Freeman, L. B., Hosier, P., LaFave, L. E., Walsh, J. L., Orro, J. M., Unger, G. J., Ross, J. M., O'Gorman, T. J., Messina, B., Sullivan, T. D., Sykes, A. J., Yourke, H., Enger, T. A., Tolat, V., Scott, T. S., Taber, A. H., Sussman, R. J., Klein, W. A., and Wahaus, C. W. 1996. Ibm experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development 40,* 1 (jan.), 3–18.

Zilles, C. and Sohi, G. 2002. Master/slave speculative parallelization. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on.* 85 – 96.