A Fault Tolerant Approach to Microprocessor Design

Chris Weaver
Todd Austin
Advanced Computer Architecture Laboratory
University of Michigan
{chriswea, taustin}@eecs.umich.edu

Abstract

We propose a fault-tolerant approach to reliable microprocessor design. Our approach, based on the use of an on-line checker component in the processor pipeline, provides significant resistance to core processor design errors and operational faults such as supply voltage noise and energetic particle strikes. We show through cycle-accurate simulation and timing analysis of a physical checker design that our approach preserves system performance while keeping area overheads and power demands low. Furthermore, analyses suggest that the checker is a fairly simple state machine that can be formally verified, scaled in performance, and reused. Further simulation analyses show virtually no performance impacts when our simple checker design is coupled with a high-performance microprocessor model. Timing analyses indicate that a fully synthesized unpipelined 4-wide checker component in 0.25um technology is capable of checking Alpha instructions at 288 MHz. Physical analyses also confirm that costs are quite modest; our prototype checker requires less than 6% the area and 1.5% the power of an Alpha 21264 processor in the same technology. Additional improvements to the checker component are described which allow for improved detection of design, fabrication and operational faults.

1 Introduction

High-quality verification and testing is a vital step in the design of a successful microprocessor product. Designers must verify the correctness of large complex systems and ensure that manufactured parts work reliably in varied (and occasionally adverse) operating conditions. If successful, users will trust that when the processor is put to a task it will render correct results. If unsuccessful, the design can falter, often resulting in serious repercussions ranging from bad press, to financial damage, to loss of life. There have been a number of high-profile examples of faulty microprocessor designs. Perhaps the most publicized case was the Intel Pentium FDIV bug where an infrequently occurring functional design error caused erroneous results in some floating point divides [1]. Functional design errors are not the only source of problems; the MIPS R10000 microprocessor was recalled due to manufacturing problems that resulted in unreliable operation [2].

1.1 Reliability Challenges

The challenges that must be overcome to build a reliable microprocessor design are great. There are many sources of errors, each requiring careful attention during design, verification, and manufacturing. We broadly classify the faults that can reduce reliability into three categories: design faults, manufacturing faults, and operational faults.

1.1.1 Design faults

Design faults are the result of human error, either in the design or specification of a system component, that renders the part unable to correctly respond to certain inputs. The typical approach used to detect these bugs is simulation-based verification. A model of the processor being designed executes a series of tests and compares the model's results to expected results. Unfortunately, design errors sometimes slip through this testing process due to the immense size of the test space. To minimize the probability of undetected errors, designers employ various techniques to improve the quality of verification including co-simulation [4], coverage analysis, random test generation [5], and model-driven test generation [6].

Another popular technique, *formal verification*, uses equality checking to compare a design under test with the specification of the design. The advantage of this method is that it works at a higher level of abstraction, and thus can be used to check a design without exhaustive simulation. The drawback to this approach is that the design and the instruction set architecture it implements need to be formally specified before the process can be automated.

Complex modern designs have outpaced the capabilities of current verification techniques. For example, a microprocessor with 32 32-bit registers, 8k-byte instruction and data caches, and 300 pins cannot be fully examined with simulation-based testing. The design has a test space with at least 2^{132396} starting states and up to 2^{300} transition edges emanating from each state. While formal verification has improved detection of design faults, full formal verification is not possible for complex dynamically scheduled microprocessor designs [7,8,9]. To date, the approach has only been demonstrated for in-order issue pipelines or simple out-of-order pipelines with small window sizes. Complete formal verification of complex modern microprocessors with out-of-order issue, speculation, and large instruction windows is currently an intractable problem [10,11].

1.1.2 Manufacturing defects

Manufacturing defects arise from a range of processing problems that manifest during fabrication. For example, step coverage problems that occur during the metallization process may cause open circuits, or improper doping in the channel of CMOS transistors may cause a change in the threshold voltage and timing of a device. *Nonconcurrent testing* techniques, which place the part into a special testing mode, are the primary vehicle for diagnosing these type of errors. Testing of the system is accomplished by adding special test hardware.

Scan testing adds a MUX to the inputs of flip-flops that allow for reading and writing of latches during test mode. This method provides direct checking of flipflop operation and indi-

rect checking of combination logic connected to the scan latch. Using the scan chain, test vectors are loaded into the flipflops and then combinational logic is exercised to determine if the implementation is faulty. Built in self test (BIST) adds specialized test generation hardware to reduce the time it takes to load latches with test vectors. BIST test generators typically employ modified linear shift feedback register (LSFR) or ROMs to generate key test vectors that can quickly test for internal logic defects such as single-stuck line faults [12]. To obtain sufficient design coverage in current designs, BIST can take as much as 15% of the design area[14,15,16]. A more global approach is taken by *IDDQ testing*, which uses onboard current monitoring to detect if there are any short-circuits. During testing, power supply currents are monitored while the system is exercised; any abnormally high current spikes are indicative of short-circuit defects [13]. The advantage of Iddq is that it is straightforward to test a large area all at once, however, it requires careful regulation to get the correct current limits.

1.1.3 Operational faults

Operational faults are characterized as sensitivity of the chip to environmental conditions. It is useful to subdivide these type of errors into categories based on their frequency [17]: permanent, intermittent, and transient faults. *Permanent faults* occur consistently because the chip has experienced an internal failure. Electrometal migration [18] and hot electrons [19][20] are two examples of permanent faults that can render a design irrevocably damaged. We also classify latch-up, which is caused by a unity gain in the bipolar transistor structures present in a CMOS layout, as a permanent fault; however, this fault can be cleared by powering down the system.

Unlike permanent faults, *intermittent faults* do not appear continuously. They appear and disappear, but their manifestation is highly correlated with stressful operating conditions. Examples of this type of fault include power supply voltage noise [21] or timing faults due to inadequate cooling [19]. Data-dependent design errors also fall into this category. These implementation errors are perhaps the most difficult to find because they require specific directed testing to locate.

Transient faults appear sporadically but cannot be easily correlated to any specific operating condition. The primary source of these faults are single event radiation (SER) upsets. SER faults are the result of energized particle strikes on logic which can deposit or remove sufficient charge to temporarily turn the device on or off, possibly creating a logic error [24][25]. While shielding is possible, its physical construction and cost make it an unfeasible solution at this time[26].

Concurrent testing techniques are usually required to detect operational faults, since their appearance is not predictable. Three of the most popular methods are timers, coding techniques and multiple execution. Timers guarantee that a processor is making forward progress, by signaling an interrupt if the timer expires. Coding techniques, such as parity or ECC, use extra information to detect faults in data. While primarily used to protect storage, coding techniques also exist for logic [27]. Finally, data can be checked by using a k-ary system where extra hardware or redundant execution is used to provide a value for comparision[17,28,29,30].

Deep submicron fabrication technologies (i.e., process technologies with minimum feature sizes below 0.25um) heighten the importance of operational fault detection. Finer feature sizes result in smaller devices with less charge, increasing their exposure to noise-related faults and SER. If designers cannot meet these new reliability challenges, they may not be able to enjoy the cost and speed advantages of these denser technologies.

1.2 Contributions of this Paper

In this paper, we detail a novel on-line testing approach, called *dynamic verification*, that addresses many of the reliability challenges facing future microprocessor designs. Our solution inserts an on-line checking mechanism into the retirement stage of a complex microprocessor. The checker monitors the results of all instructions committed by the processor. If no error is found in a computation, the checker allows the instruction result to be committed to architected register and memory state. If any results are found to be incorrect, the checker fixes the errant result and restarts the processor core with the correct result. We show in this paper that the checker processor is quite simple, lending itself to high-quality formal verification and electrically robust designs. In addition, we demonstrate through simulation and timing analysis of a physical design that checker costs are quite low. The addition of the checker to the pipeline causes virtually no slowdowns to the core processor, and area and power overheads for a complete checker design are quite modest.

The simple checker provides significant resistance to design and operational faults, and provides a convenient mechanism for efficient and inexpensive detection of manufacturing faults. Design verification is simplified because the checker concentrates verification onto itself. Specifically, if any design errors remain in the core processor, they will be corrected (albeit inefficiently) by the checker processor. Significant resistance to operational faults is also provided. We introduce in this paper a low-cost and high-coverage technique for detecting and correcting SER-related faults. Our approach uses the checker processor to detect energetic particle strikes in the core processor; for the checker processor we've developed a re-execute-on-error technique that allows the checker to check itself. Finally, we demonstrate how the checker can be used to implement a low-cost hierarchical approach to manufacture testing. Simple low cost BIST tests the checker module, then the checker can be used as the tester to test the core processor for manufacturing errors. The approach could significantly reduce the cost of late-stage testing of microprocessors, while at the same time reducing the time it takes to test parts.

In Section 2, we detail dynamic verification including the architecture of the checker processor and its operation. Section 3 gives results of a detailed analysis of our prototype checker processor design for the Alpha instruction set. These analyses include performance impacts on the core processor, and area and power overheads due to the addition of the checker processor. Section 4 introduces simple extensions to our prototype design that provide additional resistance to operational errors and improved manufacturing test capabilities. Section 5 details related work, and Section 6 gives conclusions and suggests future directions.

2 Dynamic Verification

2.1 System Architecture

Recently we proposed the use of dynamic verification to reduce the burden of verification in complex microprocessor designs [31][32][33]. *Dynamic verification* is an on-line instruction checking technique that stems from the simple observation that *speculative execution is fault tolerant*. Consider for example, a branch predictor that contains a design error, *e.g.*, the predictor array is indexed with the most significant bits of the PC (instead of the least significant PC bits). The resulting design, even though the branch predictor contained a design error, would operate correctly. The only effect on the system would be significantly reduced branch predictor

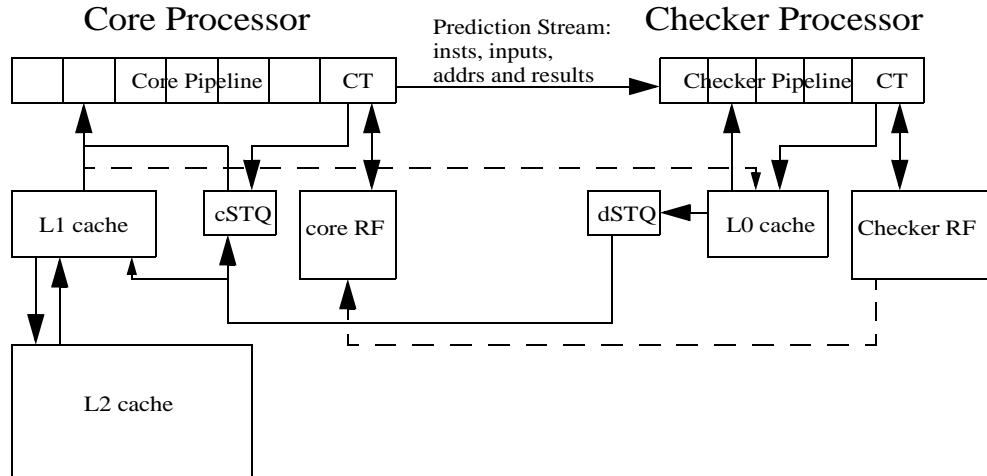Core Processor   Prediction Stream: Checker Processor

Figure 1: Dynamic Verification System Architecture.

accuracy (many more branch mispredictions) and accordingly reduced system performance. From the point of view of a correctly designed branch predictor check mechanism, a bad prediction from a broken predictor is indistinguishable from a bad prediction from a correct predictor design. Moreover, predictors are not only tolerant of permanent errors (*e.g.*, design errors), but also manufacturing defects and operational errors (*e.g.*, noise-related faults or natural radiation particle strikes).

Given this observation, the burden of verification in a complex design can be decreased by simply increasing the degree of speculation. Dynamic verification does this by pushing speculation into all aspects of core program execution, making the architecture fully speculative. In a fully speculative architecture, all processor communication, computation, control and forward progress is speculative. Accordingly, any permanent (*e.g.*, design error, defect, or failure) and transient (*e.g.*, noise-related) faults in this speculation do not impact correctness of the program. Figure 1 illustrates the approach.

To implement dynamic verification, a microprocessor is constructed using two heterogeneous internal processors that execute the same program. The *core processor* is responsible for pre-executing the program to create the *prediction stream*. The prediction stream consists of all executed instructions (delivered in program order) with their input values and any memory addresses referenced. In a baseline design, the core processor is identical in every way to the traditional complex microprocessor core up to (but not including) the retirement stage. In this baseline design, the complex core processor is "predicting" values because it may contain latent bugs that could render these values incorrect.

The *checker processor* follows the core processor, verifying the activities of the core processor by re-executing all program computation in its wake. The high-quality stream of predictions from the core processor serves to simplify the design of the checker processor and speed its processing. Pre-execution of the program on the complex core processor eliminates all the processing hazards (*e.g.*, branch mispredictions, cache misses, and data dependencies) that slow simple processors and necessitate complex microarchitectures. Thus is possible to build an inorder checker without speculation that can match the retirement bandwidth of the core. In the event the core produces a bad prediction value (*e.g.*, due to a design errors), the checker processor will fix the errant value and flush all internal state from the core processor, and restart it after the errant instruction. Once restarted, the core processor will resynchro-

nize with the correct state of the machine as it reads register and memory values from non-speculative storage.

To eliminate the possibility of storage structural hazards, the checker processor has its own register file and instruction and data caches. A small dedicated data cache for the checker processor, called the *L0 cache,* is loaded with whatever data is touched by the core processor; it taps off the output port of the L1 cache. This prefetching technique greatly reduces the number of misses experienced by the checker. However, if the checker processor misses in the L0 cache, it blocks the entire checker pipeline, and the miss is serviced by the core L2 cache. Cache misses are rare for the checker processor even for very small caches, because the high-quality address stream from the core processor allows it to manage these resources very efficiently. Store Queues are also added to both the core and checker (cSTQ and dSTQ in Figure 1) to increase performance.

The resulting dynamic verification architecture should benefit from a reduced burden of verification, as only the checker needs to be completely correct. Since the checker processor will fix any errors in the instructions that are to be committed, the verification of the core is reduced to simply the process of locating and fixing *commonly occurring* design errors that could affect system performance. Since the complex core constitutes a major testing problem, relaxing the burden of correctness of this part of the design can yield large verification time savings. To maintain a high quality checker processor, we leverage formal verification to ensure correct function and extensive checker processor Built-In-Self-Test (BIST) to guarantee a successful implementation.

2.2 Checker Processor Architecture

For dynamic verification to be viable, the checker processor must be simple and fast. It must be simple enough to reduce the overall design verification burden, and fast enough to not slow the core processor. A single-issue two-stage checker processor is illustrated in Figure 2a). The design presented assumes a single-wide checker, but scaling to wider or deeper designs is a fairly straightforward task (discussed later).

In the normal operation (as shown in Figure 2b), the core processor sends instructions (with predictions) at retirement to the checker pipeline. These predictions include the next PC, instruction, instruction inputs, and addresses referenced (for
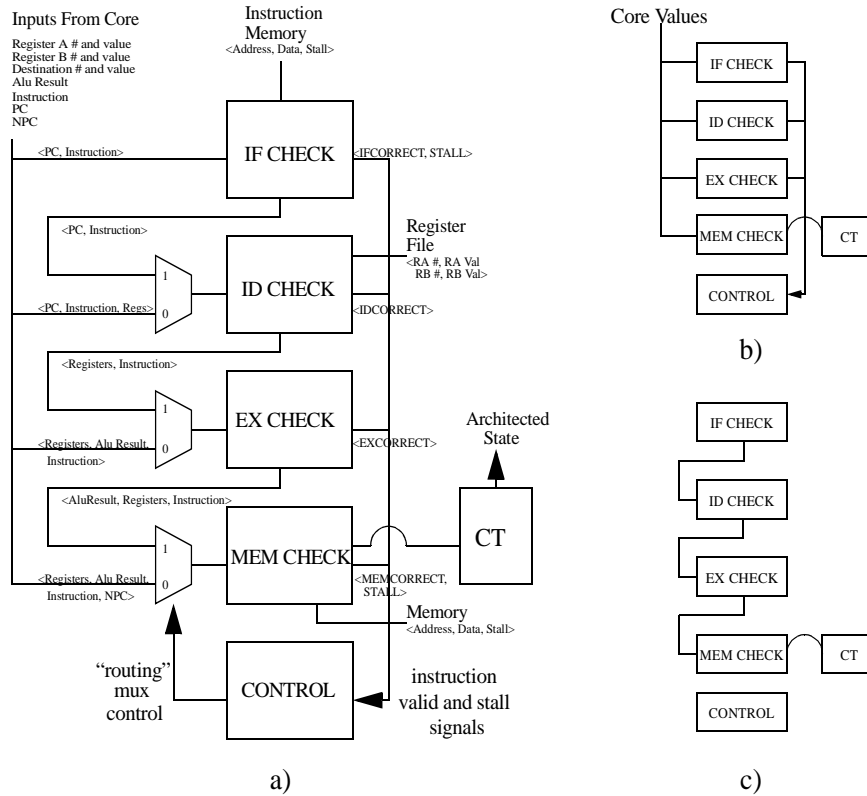
Figure 2: Checker Processor Pipeline Structure for a) a single wide checker processor, b) a checker processor in Check mode, and c) a checker processor in Execute mode

loads and stores). The checker processor ensures the correctness of each component of this transfer by using four parallel stages, each of which verifies a separate component of the prediction stream. The separation shown is done in a manner similar to the classic Hennessy and Patterson five stage pipeline[34]. The IFCHECK unit verifies the instruction fetch by accessing the instruction memory with the checker program counter. IDCHECK verifies that the instruction was decoded correctly by checking the input registers and control signals. EXCHECK re-executes the functional unit operation to verify core computation. Finally, the MEMCHECK verifies any loads by accessing the checker memory hierarchy.

If each prediction from the core processor is correct, the result of the current instruction (a register or memory value) as computed by the checker processor is allowed to retire to nonspeculative storage in the *commit (CT)* stage of the checker processor. In the event any prediction information is found to be incorrect, the bad prediction is fixed, the core processor is flushed, and the core and checker processor pipelines are restarted after the errant instruction. Core flush and restart use the existing branch speculation recovery mechanism contained in all modern high-performance pipelines.

As shown in Figure 2b) and 2c), the routing MUXes can be configured to form a parallel checker pipeline or a recovery pipeline, respectively. In recovery mode the pipeline is reconfigured into a serial pipeline, very similar to the classic H&P five-stage pipeline. In this mode, stage computations are sent to the next logical stage in the checker processor pipeline, rather than used to verify core predictions. Unlike the classic H&P five-stage pipeline, only one instruction is allowed to enter the recovery pipeline at a time. As such, the recovery pipeline configuration does not require bypass datapaths or

complex scheduling logic to detect hazards. Processing performance for a single instruction in recovery mode will be quite poor, but as long as faults are infrequent there will be no perceivable impact on program performance [31]. Once the instruction has retired, the checker processor re-enters normal processing mode and restarts the core processor after the errant instruction.

Figure 2 also illustrates that the checking and recovery modes use the same hardware modules, thereby reducing the area cost of the checker. Each stage only requires intermediate pipeline inputs - whether these are from the core processor prediction stream or the previous stage of the checker processor pipeline (in recovery mode) is irrelevant to the operation of the stage. This attribute serves to make the control and implementation of individual stages quite simple. In recovery mode, the check logic is superfluous as the inputs will always be correct, however, no reconfiguration is required to the check logic as it will never declare a fault during recovery.

Pipeline scheduling is trivial, if any checker pipeline is blocked for any reason, all checker processor pipelines are stalled. This simplifies control of the checker processor and eliminates the need for instruction buffering or complex nonblocking storage interfaces. Since there are no dependencies between instructions in normal processing, checker processor pipeline stalls will only occur during a cache miss or structural (resource) hazard. This lack of dependencies makes it possible to construct the checker control as a three stage (Idle, Check and Execute) Moore state machine. The pipeline sits in the Idle state until the arrival of a retired core processor instruction. The pipeline then enters normal Check mode until all instructions are exhausted or an instruction declares a fault. If a fault is declared the pipeline enters the Execute state, reconfiguring

the pipeline to single serial instruction processing. Once the faulty instruction has completed, the pipeline returns to Idle or Check mode, depending on the availability of instructions from the core.

Certain faults, especially those affecting core processor control circuitry, can lock up the core processor or put it into a deadlock or livelock state where no instructions attempt to retire. For example, if an energetic particle strike changes an input tag in a reservation station to the result tag of the same instruction, the processor core scheduler will deadlock. To detect these faults, a *watchdog timer (WT)* is added. After each instruction commits, the watchdog timer is reset to the maximum latency for any single instruction to complete. If the timer expires, the processor core is no longer making forward progress and the core is restarted. At that time, pipeline control transitions to recovery mode where the checker processor is able to complete execution of the current instruction before restarting the core processor after the stalled instruction.

3 Performance Implications

In this section we detail the performance of a complex out of order processor that is enhanced with a checker. Cycle accurate simulation of a number of benchmarks is used to demonstrate that the performance impacts are minimal.

3.1 Experimental Framework

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [35], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 4-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. A 4-wide 3-stage checker with 4k data cache and 0.5K instruction cache was used for the analysis. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. To perform our simulation experiments, we collected results for nine of the SPEC95 benchmarks [36] and six of the SPEC2000 benchmarks. Of these program nine are primarily integer codes and six use floating point extensively.

Physical designs were constructed in synthesizable Verilog code. The code was compiled and simulated using Cadence's VERILOG-XL 2.1.3. The waveform data generated from the simulation was viewed using DAI Signalscan Version 6.4s4. Once the design had been debugged it was synthesized using Synopsys Version 2000.05, using the 0.25um Leda library. Synopsys designs were derived using a wire load from this library as well. The design was synthesized with timing as its primary optimization objective, and area as a secondary concern. After synthesis was complete, Synopsys timing, area and power reports were generated to assess the design.

3.2 Performance Impacts

With dynamic verification, there are two ways the checker processor can slow the progress of the core processor. First, checker processor stalls delay the retirement of instructions in the core processor, forcing the core processor to hold speculative state longer, possibly creating back pressure at core processor decode. If speculative state resources in the core processor fill, the decoder will have to stall as it will not be able to allocate re-order buffer and load/store queue resources to new instructions. The checker processor only stalls when an access to its data cache misses or when there are insufficient

Integer Benchmark	Slowdown (Checked Vs Base)	Floating Point Benchmark	Slowdown (Checked Vs Base)
compress95	0%	applu00	-1.2779%
crafty00	0.013561%	hydro2d	-0.46554
gap00	-0.73826%	lucas00	0%
gcc	0.015443%	mesa00	-0.09291%
go	0%	tomcatv	0.19605%
ijpeg	-0.25113%	turb3d	-3.16497%
li	0.0556%		
perl	0%		
twolf00	0.05295		
average	-.095%	average	-.800%
OVERALL AVERAGE		-0.3771%	

Table 1: Performance Results of a Processor Checker.

functional unit resources. Second, fault recovery will delay core progress as the core processor must be flushed and restarted after the errant instruction.

The performance impacts of dynamic verification are summarized in Table 1. These results reflect performance impacts without any core faults incurred during the simulation. The processor model with dynamic verification exhibits modest performance gains over the baseline model without a checker processor. This is due to the additional checker store queue (cSTQ), which eliminates store writeback misses at commit that can slow the baseline architecture (without a checker processor). This effect is demonstrated in the average decoder stalls percentage which went from 10.128% in the baseline to only 7.744% in the dynamically verified model.

An analysis of fault rates in a previous work indicated that recovery from faults had no perceivable impact on performance provided the faults were less than one every 1000 cycles [31]. Considering that faults are the result of design errors or infrequent transient faults, we don't expect fault rates to approach this threshold.

To summarize the performance results from [31,32,33], the checker processor keeps up with the core processor because the core processor clears the many hazards that would otherwise slow this very simple design. Branch hazards are cleared by the high quality next PC predictions from the core processor, generated using pre-execution of the program. Data hazards from long latency instruction are eliminated by the input value predictions supplied by the core processor. With these inputs values, instruction need not wait for earlier dependent operations to finish. As such, dependent long latency operation may execute in parallel in the checker pipeline. Finally, data hazards from cache misses are virtually non-existent in the core processor since the core processor runs ahead of the checker initiating cache misses (and tolerating their latency) in advance of checker execution. The checker cache need only hold the working set of the pipeline, i.e., data elements from the time they are first touched in the core processor pipeline until the time they are checked by the checker processor.

Indeed, it is this lack of pipeline hazards that also makes the checker processor design quite scalable. A deeper (or super-pipelined) checker processor will process instructions faster because it can employ a higher clock rate. The lack of dependences makes increasing the depth of the checker processor pipe stages fairly straightforward. A wider (or superscalar) checker processor design will process instruction faster by allowing more instructions to be processed in a single cycle. For the most part, widening the checker processor stages requires only replicating the resources such that more instruc-
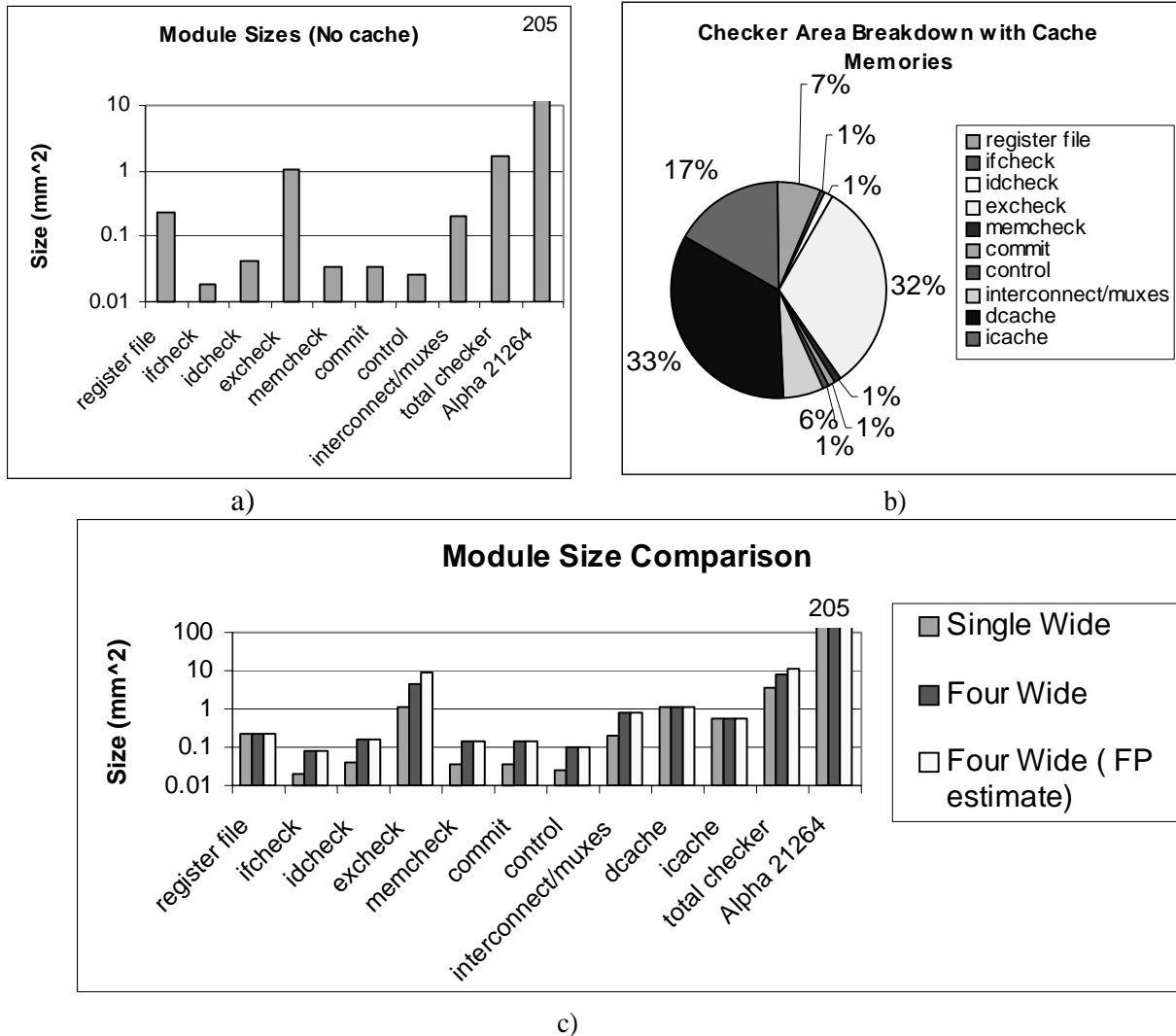
a)

b)

c)

Figure 4 Physical Characteristics: a) The area of the various checker modules. b) A breakdown of the area used in a single wide checker. c) A comparison of the sizes for a single wide checker, quad wide checker, quad wide check with floating point estimate and an Alpha 21264.

tions can be processed in a single cycle. Input value checks (for registers or memory) are somewhat more complicated, however, as multiple instructions may attempt to read from memory or registers in a single cycle while at the same time creating values that may be visible to later instruction in the same cycle. Fortunately, this is no more difficult than the dependence checks that occur during instruction issue, renaming, and commit. We are currently investigating the full impacts of scaling on the checker processor and will report on these findings in the future.

4 Physical Design

When considering any type of testing technique it is important to quantify its costs in addition to its benefits. This section will analyze the performance, area and power costs of a prototype checker processor.

To assess the performance impacts of a checker processor, a Verilog model for the integer subset of the Alpha instruction set was constructed. The design was tested with small hand coded assembly programs to verify correct checker operation. Floating point instructions were not implemented due to the time constraints. We do, however, estimate floating point overheads using measurements from an unrelated physical design in the same technology. The Synopsys toolset was used in conjunction with a 0.25um library to produce an unpipelined fully synthesized design that ran at 288MHz. Our high level simulations show that pipelining of the checker does not have a severe impact on performance. We are considering this approach as well as semi-custom designs as means to matching the speed of current microprocessors.

The Synopsys toolset will generate area estimates for a synthesized design given the cells used and the estimated interconnect. Boundary optimization, which exploits logical redundancy between modules, was turned off so that an assessment of each module's contribution to the overall area could be determined. A 1.65mm^2 single-wide checker size was produced by the toolset, which is fairly small in comparison to the 205mm^2 size of the Alpha 21264 microprocessor. A single-wide checker only amounts to 0.8% of the area of the Alpha

chip. However, as mentioned before only a subset of the instruction set was implemented. The checker area breakdown chart shows that the *excheck* module, which houses the functional units, contributes the most to the overall size. The floating point modules should be slightly larger then their integer counterparts due to the need to track the exponents. The memory elements in the checker design were estimated using the Cacti 2 cache compiler[43]. The estimates produced by Cacti were slightly pessimistic, but within a factor of 2 of known layouts. Cacti projected that the I-cache and D-cache would be 0.57mm^2 and 1.1408mm^2 respectively. These values are for the 512 byte I-cache and 4K byte D-cache that are described in previous section. Accounting for caches, total checker area rises to roughly 10mm^2, still much smaller than a complete microprocessor in the same technology.

4.1 Power Analysis

As transistor counts grow, power is a growing concern since adequate distribution and cooling can create design problems. Thus, checking mechanisms must budget their power usage to meet overall system goals. Synopsys tools are also able to model power usage at a high level. It estimates switching frequencies for logic to determine the power consumed. A 50% switching estimate is the default. Generally, this produces a slight overestimate of the power, but it is useful for a coarse grain analysis. Synopsys estimates the power of our Verilog design at 941mW, small in comparison to the 75 watts that are consumed by a typical core processor design like the Compaq Alpha 21264[42]. While this is a pessimistic estimate, we have isolated a few mechanisms to improve the power usage. To speed up the checker processor we are sending our operands to all the functional units at the same time, then selecting the result from the one we want. By increasing the depth of the checker pipeline, we can predecode the functional unit used and only change the inputs to that module. This will decrease the switching in the checker and thereby decrease the power required to process an instruction.

5 Design Improvements for Additional Fault Coverage

The design presented thus far is targeted at design faults. While the checker inherently aids in the detection of operational and manufacturing faults due to its checking capability and simplification of critical circuitry, it has not been the primary focus thus far. We will now explore how dynamic verification can further aid in the detection of operational and manufacturing faults.

5.1 Operational Errors

Several mechanisms reduce the probability of operational errors affecting functionality, especially those errors that can arise from SER particle strikes. If the checker is properly designed, the system is guaranteed to function correctly given any error in the core. This guarantee can be safely made because the checker will detect and correct any operational faults in the core. Ensuring correct operation is more challenging when we consider the possibility of operational errors in the checker circuits. Ideally, the checker would be designed such that its lifetime and radiation tolerance is higher then the

core. However this may be too costly or impossible due to environmental conditions.

	error occurred	error did not occur
error detected	**Functioning ok** checker detects error that occurred	**Case A** checker will re-check and realize instruction is correct
error not detected	**Case B** Small probability of occurring see equation in text	**Functioning ok** core is working correctly and checker does not find any errors

Table 2: Operational Faults in Checker Circuitry.

Table 2 enumerates the possible fault scenarios that may occur during the checking process. Case A represents the false positive situation where the checker detected a fault that did not really occur. To solve Case A, we have added additional control logic that causes the checker to re-check core values when an error is detected, before recovery mode is entered. This reduces the likelihood of this case occurring, at the expense of slightly slower fault recovery performance.

Case B can occur in one of two ways; the first is when an operational error causes an equivalent error to occur in both the core and the checker. This is the product of two small probabilities. Given a good design, the likelihood of this event is probabilistically small. However, replication of the functional units may be applied to further reduce this probability. The other possibility is that either the comparison logic or the control logic suffer an error. We can employ TMR on the control logic to help reduce the probability that this error occurs. In a system with a checker, the probability of a failure, shown in the following equation, is always the product of at least two unlikely events. For systems that require ultra-high reliability the checker can be replicated until the probability is acceptable for the application. This is a low cost redundant execution approach as only the small checker needs to be replicated.

$$P_{\text{failure}} = P_{\text{design error core}} * P_{\text{masking strike in checker}} + P_{\text{design error core}} * P_{\text{strike checker control}}$$

Figure 5 illustrates how TMR can be applied to the control logic of the checker to provide better reliability in the event of an operational error in the checker control. Again, a design analysis was done by synthesizing a Verilog model. The areas estimates previously given show that the simple control logic of the checker only contributes a small portion to the overall checker size. The addition of two more control logic units and voting logic only consumes an extra 0.12 mm^2. TMR still has a single point of failure within the voter logic, but the chance of a strike is reduced due to the area difference as compared to the control unit logic. Additionally, the voter logic can be equipped with transistors that are sized large enough to have a high tolerance of environmental conditions.

5.2 Manufacturing Errors

The checker processor can improve the success rate of manufacturing test, because the checker design lends itself to these tests, and only the checker must be completely free of defects for correct program operation. The structure of the checker, where values are latched in and compared to hardware, enhances the checkers suitability for nonconcurrent testing techniques. A simple scan chain can be synthesized in latches
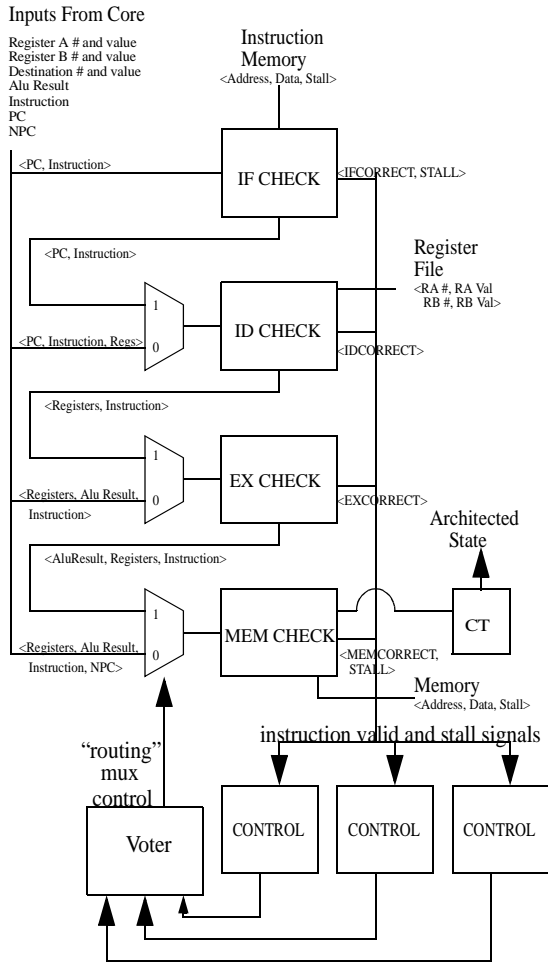
Figure 5: Checker Processor Pipeline Structure with TMR on the Control Logic.

Figure 6: Checker Processor Pipeline Structure with TMR on Control Logic and BIST.

Module Name	# of faults	test coverage	test patterns	pattern size	memory
IFCHECK	2108	100%	204	97	2.4KB
IDCHECK	6063	100%	366	215	9.6KB
EXCHECK	82810	99.96%	764	327	30.5KB
MEMCHECK	5024	100%	290	194	6.9KB
COMMIT	388	100%	26	140	455B
CONTROL	2038	100%	80	97	790B
Total					51.8KB

Table 3: BIST Analysis.

that supply the data to the checker. This type of testing can achieve enhanced fault coverage over the core, since the checker circuitry is significantly simpler. BIST is another option that holds great potential for this type of circuit.

As shown in Figure 6, built-in-self-test (BIST) hardware can be added to test for checker manufacturing errors. The BIST hardware uses a Test Generator to artificially stimulate the checker logic and it passes values into the checker stage latches that hold the instruction for the checker to validate. The most effective test generator will be a ROM of opcodes, inputs and expected results. Using this data and the error signals already present in the system, an efficient nonconcurrent test can be performed. Obviously, the ROM should have both good and bad sets of data such that the check modules are fully tested. The number of tests required to test for all faults is a function of the checker logic size and structure. Memories make up another huge portion of the design; fortunately, significant research has been done in how to effectively test memory. Marching ones and zeros is just one simple way to test a memory.
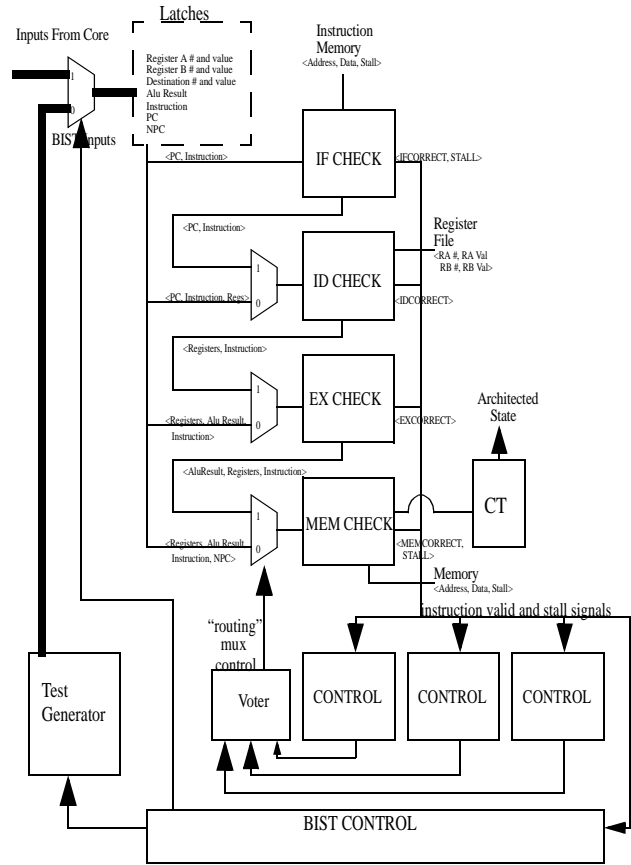
To assess the potential of applying BIST to the checker, we ran our synthesized design through Tetramax. Tetramax is an ATPG tool that is part of the Synopsys suite. Verilog models of the gates in the synthesized design are read into the Tetramax, allowing it to activate and propagate faults. The Verilog design is then entered into the tool and analyzed. Statistics and patterns are reported from the analysis. In addition, the ATPG tool will try to compress the vectors so that redundant patterns are deleted. The ATPG statistics are summarized in Table 3. The total ROM usage of 51.8KB was plugged into the Cacti 2 tool, which estimated the area at 0.057mm^2 (assuming a density of 10 ROM cells per RAM cell).

Although difficult to quantify, it is likely that the manufacturing process could benefit from the checker processor as well. In a fab where part production is limited by the bandwidth and latency of testers, the checker processor could improve testing performance. Once the checker is fully tested using internal BIST mechanisms, the checker itself can test the remaining core processor circuitry. No expensive external tester is required, only power and a simple interface with ROM to hold core testing routines and a simple I/O interface to determine if the checker passed all core processor tests.

6 Related Work

Rotenberg's AR-SMT[29] and more recently Slipstream[30] processors use multiple threads to eliminate hazards and verify correct operation. In Slipstream an advanced stream (A-stream) is used to aid a redundant stream (R-stream) by providing future knowledge to the trailing stream. As a result, performance can be increased and the redundancy can be used to detect transient faults. However, this technique does not provide total coverage or resistance to design errors. Other techniques for obtaining reliability include the redundant hardware approach, as in the IBM S/390 G5 microprocessor [38] and the redundant thread (or SRT) approach. In the SRT approach proposed in [10], redundant threads run concurrently and compare results for fault detection. Performance in SRT processors is improved with one thread prefetching cache misses and computing branch outcomes for other threads, similar to the core prefetching that the checker processor leverages.

A number of fault-tolerant processor designs have been proposed and implemented, in general, they employ redundancy to detect and/or correct transient errors. IBM's G4 processor[39] is a highly reliable processor design similar to the design in this paper in that it checks all instructions results before committing them to architected state. Checking is accomplished by fully replicating the processor core. An R-Unit is added to compare all instruction results, permitting only identical results to commit. If a failure in the processor is detected, it is addressed at the system level through on-line reconfiguration. The ERC32 is a reliable SPARC compatible processor built for space applications [40]. This design augments the microarchitecture with parity on all register and memory cells, some self-checking control logic, and control flow checking. In the event a fault is detected, a software interrupt is generated and the host program initiates recovery.

Unlike these designs, the checker can keep costs lower by only checking the function of the program computation. The G4 and ERC32 designs check both the function of the program and the mechanisms used to optimize program performance. This results in more expensive checkers, typically two times as much hardware in the core processor. Additionally, the checker can detect design errors. Simple redundant designs cannot detect design errors if the error is found in each of the redundant components.

Tamir and Tremblay [41] proposed the use of micro rollback to recover microarchitecture state in the event of a detected fault. The approach uses FIFO queues to checkpoint a few cycles of microarchitectural state. In this work, we use a similar parallel checking approach, but employ a global checking strategy to reduce checker cost. In addition, we use the existing control speculation mechanism to restore correct program state.

Blum and Wasserman discussed checkers with respect to the Pentium Division Bug [1]. In addition to describing complete checkers, they discussed how a partial checker could be utilized to achieve checking accuracy that was almost perfect.

They also postulate that if error rates are kept small enough, correcting procedures can be very time consuming without impacting the performance of the system.

7 Conclusions and Future Directions

Many reliability challenges confront modern microprocessor designs. Functional design errors and electrical faults can impair the function of a part, rendering it useless. While functional and electrical verification can find most of the design errors, there are many examples of non-trivial bugs that find their way into the field. Additional faults due to manufacturing defects and operation faults such as energetic particle strikes must also be overcome. Concerns for reliability grow in deep submicron fabrication technologies due to increased design complexity, additional noise-related failure mechanisms, and increased exposure to natural radiation sources.

To counter these reliability challenges, we proposed the use of dynamic verification, a technique that adds a checker processor to the retirement phase of a processor pipeline. If an incorrect instruction is delivered by the core processor the checker processor will fix the errant computation and restart the core processor using the processor's speculation recovery mechanism. Dynamic verification focuses the verification effort into the checker processor, whose simple and flexible design lends itself to high-quality functional verification and a robust implementation.

We presented detailed analyses of a prototype checker processor design. Cycle-accurate architectural simulation of our design monitoring an Alpha 21264-like core processor resulted in virtually no performance penalties due to on-line instruction checking. The simple checker can easily keep up with the complex core processor because it uses pre-computation in the core processor to clear the branch, data, and communication hazards that could otherwise slow the simple checker pipeline.

We presented analyses of our prototype physical checker design. Timing analyses indicate a fully synthesized and unpipelined 4-wide checker processor design in 0.25um technology is capable of running at 288 MHz. We are currently hand optimizing this design through better layout and stage pipelining. We fully expect our follow on efforts will demonstrate that the checker design is also quite scalable. In addition, area and power analyses of our physical design were presented. Overall, the checker processor requires less than 6% the area and 1.5% the power of an Alpha 21264, confirming that our approach is low cost. Finally, we presented novel extensions to our baseline design that improve coverage for operational faults and manufacturing fault detection.

We feel that these results strengthen the case that dynamic verification holds significant promise as a means to address the cost and quality of verification for future microprocessors. Currently, we continue to refine our physical checker processor design, optimizing its layout and investigating technique to scale its performance through pipelining. In parallel with this effort, we are also examining how we might further leverage the fault tolerance of the core processor to improve core processor performance and reduce its cost. One such approach is to leverage the fault tolerance of the core to implement self-tuning core circuitry. By employing an adaptive clocking mechanism, it becomes possible to overclock core circuitry, reclaiming design and environmental margins that nearly always exist. We will report on this and other optimizations in future reports.

8 References

[1]M. Blum and H. Wasserman,"Reflections on the Pentium Division Bug", Intel Corporation, Oct. 1997.

[2]M. Kane, "SGI stock falls following downgrade,recall announcement." PC Week, Sept. 1996

[3]M. Williams, "Faulty Transmeta Crusoe Chips Force NEC to Recall 300 Laptops." The Wall Street Journal, Noc 30,2000

[4] P. Bose, T. Conte, and T. Austin, "Challenges in processor modeling and validation," IEEE Micro, pp. 2-7, June 1999.

[5] A. Aharon,"Test program generation for functional verication of PowerPC processors in IBM," in Proceedings of the 32nd ACM/IEEE Design Automation Conference, pp. 279-285, June 1995.

[6] R. Grinwald, "User defined coverage, a tool supported methodology for design verication," in Proceedings of the 35nd ACM/IEEE Design Automation Conference, pp. 1-6, June 1998.

[7]Anonymous "Scalable Hybrid Verification of Complex Microprocessors," submitted for publication

[8]M. K. Srivas and S. P. Miller,"Formal Verification of an Avionics Microprocessor," SRI International Computer Science Laboratory Technical Report CSL-95-04, June 1995.

[9]M. C. McFarland, "Formal Verification of Sequential Hardware: A Tutorial," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol 12, NO. 5. May 1993.

[10]J. Burch and D. Dill, "Automatic verication of pipelined microprocessors control," Computer Aided Verication, pp. 68-80, 1994. 24

[11] J. Sawada, "A table based approach for pipelined microprocessor verification," in Proc. of the 9th International Conference on Computer Aided Verication, June 1997.

[12] H. Al-Asaad and J.P. Hayes, "Design verification via simulation and automatic test pattern generation." In Proceedings of the International Conference on Computer-Aided Design. IEEE Computer Society Press, Los Alamitos, CA, 174-180 1995.

[13]E. Bohl, Th. Lindenkreuz, R. Stephan, "The Fail-Stop Controller AE11," International Test Conference, pp 567-577, 1997.

[14] B. T. Murray and J.P. Hayes, "Testing ICs: Getting to the Core of the Problem," Computer, Vol. 29, No.11, pp.32-45, Nov. 1996.

[15] M. Nicolaidis, "Theory of Transparent BIST for RAMs," IEEE Trans. Computers, Vol. 45, No. 10, pp. 1141-1156, Oct. 1996.

[16]M. Nicolaidis, "Efficient UBIST Implementation for Microprocessor Sequencing Parts," J. Electronic Testing: Theory and Applications, Vol. 6, No. 3, pp. 295-312, June 1995.

[17]H. Al-Asaad, B.T. Murray and J.P. Hayes, "Online BIST for Embedded Systems," IEEE Design and Test of Computers, pp.17-24, October-December 1998.

[18]D. G. Pierce and P. G., "Electromigration: A review," Microelectronic Reliability, Vol. 37. No. 7, pp. 1053-1072, 1997.

[19]R. S. Muller and T.L. Kamins, "Device Electronics for Integrated Circuits Second Edition," John Wiley&Sons, New York, NY 1986.

[20]S. Wolf, "Silicon Processing for the VLSI Era Volume 3- Submicron MOSFET" Lattice Press, Sunset Beach, CA 1995.

[21] K. Seshan, T. Maloney, and K. Wu, "The quality and reliability of Intel's quarter micron process." Intel Technology Journal, Sept. 1998.

[22] M. Bohr, "Interconnect scaling: the real limiter to high-performance ULSI," in Proceedings of the International Electron Devices Meeting, pp. 241-244, Dec. 1995.

[23] N. Weste and K. Eshragian, Principles of Cmos VLSI Design: A Systems Perspective. Addison-Wesley Publishing Co., 1982.

[24] J. Z. et al, "IBM experiments in soft fails in computer electronics," IBM Journal of Research and Development, vol. 40, pp. 3-18, Jan. 1996.

[25] P. Rubinfeld, "Managing problems at high speed," IEEE Computer, pp. 47-48, Jan. 1998.

[26] J. Ziegler, "Terrestrial cosmic rays," IBM Journal of Research and Development, vol. 40, pp. 19-39, Jan. 1996.

[27] D. P. Siewiorek, R. S. Swarz, "Reliable Computer Systems, Design and Evaluation Second Edition," Digital Press, Burlington MA 1992.

[28]S.K. Reinhardt and S. S. Mukherjee "Transient Fault Detection via Simultaneous Multithreading," Proceedings 27th Annual Int'l Symp. on Computer Architecture (ISCA), June 2000.

[29]E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," Proceedings of the 29th Fault-Tolerant Computing Symposium, June 1999.

[30]Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," 33rd International Symposium on Microarchitecture, December 2000.

[31]T. Austin. "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design". In *Micro-32*, Nov 99.

[32]T. Austin. "DIVA: A Dynamic Approach to Microprocessor Verification." *The Journal of Instruction-Level Parallelism Volume 2,* 2000.

[33]S. Chatterjee, C. Weaver and T. Austin. "Efficient Checker Processor Design." In Micro-33, Dec 2000.

[34]J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc. 1996.

[35]SPEC newsletter. Fairfax, Virginia, Sept. 1995.

[36]D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[37]H. Al-Assad, J.P. Hayes and B.T. Murray "Scalable test generators for high-speed datapath circuits," Journal of Electronic Testing:Theory and Applications, Vol. 12, Nos. 1/2, February/April 1998

[38]T.J Slegel et al. "IBM's S/390 G5 Microprocessor Design." *IEEE Micro*, pp 12-23, March/April 1999.

[39]L. Spainhower and T. Gregg, "G4: A fault-tolerant CMOS mainframe," in Proceedings of the 28th Fault-Tolerant Computing Symposium, June 1998.

[40]J. Gaisler, "Evaluation of a 32-bit microprocessor with built-in concurrent error detection," in Proceedings of the 27th Fault-Tolerant Computing Symposium, June 1997.

[41]Y. Tamir and M. Tremblay, "High-performance fault tolerant VLSI systems using micro rollback," IEEE Transactions on Computers, vol. 39, no. 4, pp. 548-554, 1990.

[42] Cahners Microprocessor Report, vol 15, pp. 50, August 2000.

[43]Glenn Reinman, Norm Jouppi. "An Integrated Cache Timing and Power Model." http://research.compaq.com/wrl/people/jouppi/cacti2.pdf