Ultra Low-Cost Defect Protection for Microprocessor Pipelines

Smitha Shyam Kypros Constantinides Sujay Phadke
Valeria Bertacco Todd Austin
Advanced Computer Architecture Lab
University of Michigan, Ann Arbor, MI 48109
{smithash, kypros, sphadke, valeria, austin}@umich.edu

Abstract

The sustained push toward smaller and smaller technology sizes has reached a point where device reliability has moved to the forefront of concerns for next-generation designs. Silicon failure mechanisms, such as transistor wearout and manufacturing defects, are a growing challenge that threatens the yield and product lifetime of future systems. In this paper we introduce the *BulletProof* pipeline, the first ultra low-cost mechanism to protect a microprocessor pipeline and on-chip memory system from silicon defects. To achieve this goal we combine area-frugal on-line testing techniques and system-level checkpointing to provide the same guarantees of reliability found in traditional solutions, but at much lower cost. Our approach utilizes a microarchitectural checkpointing mechanism which creates coarse-grained epochs of execution, during which distributed on-line built in self-test (BIST) mechanisms validate the integrity of the underlying hardware. In case a failure is detected, we rely on the natural redundancy of instruction-level parallel processors to repair the system so that it can still operate in a degraded performance mode. Using detailed circuit-level and architectural simulation, we find that our approach provides very high coverage of silicon defects (89%) with little area cost (5.8%). In addition, when a defect occurs, the subsequent degraded mode of operation was found to have only moderate performance impacts, (from 4% to 18% slowdown).

Categories and Subject Descriptors B.8.1 [*Hardware*]: Performance and Reliability—Reliability, Testing, and Fault-Tolerance

General Terms Reliability, Design

Keywords Reliability, Defect-Protection, Low-Cost, Pipelines

1. Introduction

As silicon technologies move into the nanometer regime, there is growing concern for the reliability of transistor devices. Leading technology experts have begun to warn designers that device reliability will wane in the 45nm regime and beyond [7, 6]. In fact, device scaling aggravates a number of long standing silicon failure mechanisms, and it introduces a number of new non-trivial failure modes. Unless these reliability concerns are addressed, either through on-line detection and correction, or with the introduction of more robust devices, component yield and lifetime will soon be compromised. In this paper, we introduce a low-cost mechanism for tolerating a small number of silicon failures that occur in the field, *i.e.*, while the device is in operation.

1.1 The (Bumpy) Road Ahead

The following list highlights the types of silicon failures addressed by the reliable solution presented in this work. Each of these failure mechanisms have received significant attention in the process technology literature, and each has been identified as a growing concern for deep-submicron silicon.

Device Wear-out. Metal electro-migration and hot carrier degradation are traditional mechanisms that lead to eventual device failure [28]. While these mechanisms continue to be a problem for deep-submicron silicon, new concerns arise due to the extremely thin gate oxides utilized in current and future process technologies, which lead to gate oxide wear-out (or Time Dependent Dielectric Breakdown, TDDB). Over time, gate oxides can break and become conductive, essentially shorting the transistor and rendering it useless. Fast clocks, high temperatures, and voltage scaling limitations are well-established architectural trends that conspire to aggravate this failure mode [33].

Transistor Infant Mortality. Extreme device scaling also exacerbates early transistor failures, due to weak transistors that escape post-manufacturing testing. These weak transistors work initially, but they have dimensional and doping deficiencies that subject them to much higher stress than normal. Quickly (within days to months from component deployment) they break down and render the device unusable. Traditionally, early transistor failures have been addressed with aggressive burn-in testing, where, before being placed in the field, devices are subjected to high voltage and temperature testing, to accelerate the failure of weak transistors [4]. Those transistors that survive this grueling birth are likely to be robust devices, thereby ensuring a long product lifetime. In the deep-submicron regime, burn-in becomes less effective as devices are subject to thermal run-away effects, where increased temperature leads to increased leakage current, which in turn leads to yet higher temperatures and further increases in leakage current [23]. The end result is that aggressive burn-in can destroy even robust devices. Manufacturers are forced to either sacrifice yield with an aggressive burn-in or experience more frequent early transistor failures in the field.

Manufacturing Defects that Escape Testing. Optical proximity effects, airborne impurities, and processing material defects can all lead to the manufacturing of faulty transistors and interconnect [28]. Moreover, deep-submicron gate oxides have become so thin that manufacturing variation can lead to currents penetrating the gate, rendering it unusable [30]. In current 90nm devices these oxides are only about 20 atoms of thickness. In 45nm technology, this thickness is expected to reduce below 10 atoms. This prob-

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010. . . $5.00

lem is compounded by the immense complexity of current designs, which makes it more difficult to test for defects during manufacturing. Vendors are forced to either spend more time with parts on the tester, or risk having untested defects escape into the field.

1.2 Contributions of This Paper

While there is no consensus on the absolute rate of defects in future technologies, or as to when these problems will potentially derail the silicon manufacturing industry, there is, however, broad agreement that device reliability will begin to wane in the 45nm regime and beyond [13, 33, 17]. In this paper, we introduce the *Bullet-Proof* pipeline. It is the first ultra low-cost defect protection mechanism for microprocessor pipelines and on-chip cache memories. In this work, we target specifically low in-field defect rates. The usage mode we envision for our technology is that it will be installed into a microprocessor product. The technology will continuously monitor the system's health until the first defect is encountered. At that point, the system will stay operative but at a lower performance level. The user (and/or system controller) will be notified and will have to choose to either: i) live with the degraded mode performance, or ii) repair the system. And above all, our goal is to provide all of these capabilities for a minimal cost. Specifically, this research paper makes the following contributions to the area of reliable microarchitecture design:

- We present the first low-cost reliable system design approach which provides fine-grained detection, diagnosis, recovery, and repair of silicon defects that occur while the system is in operation in the field. While traditional approaches require at least 100% overhead due to duplication of critical resources, our on-line testing-based approach provides the same level of protection with an overhead of 5.8%.
- We provide a physical-level analysis of coverage and performance impact of our technique, in the context of a low-cost embedded VLIW processor design. We chose this target design because it is *i*) an important target due to its high reliability needs for safety-critical applications, and *ii*) a challenging environment to implement defect tolerance due to its high cost sensitivity. Moreover, it should be noted that very few reliability solutions in the computer architecture literature quantify the corresponding fault coverage. In contrast, we evaluate the coverage of our solution through a physical-level analysis (synthesized gate-level netlist) and find that it provides coverage against 89% of potential defect locations.

Our approach to defect detection is markedly different than previous solutions utilizing spatially or temporally redundant computation. We leverage instead a combination of *on-line distributed testing* with *microarchitectural checkpointing* to efficiently identify defects, and recover from their impact. The microarchitectural checkpointing mechanism provides a *computational epoch*, which is a period of computation over which the processor's hardware is checked. During a computational epoch, on-line distributed built-in self-testing (BIST) techniques exploit idle cycles to completely verify the functional integrity of the underlying hardware. When the on-line testing completes without finding faults, the underlying hardware is known to be free of silicon defects and the epoch's computation is allowed to safely retire to non-speculative state. By contrast, if the underlying hardware is found to be faulty, the results of the computational epoch are thrown away, and the system's state is restored to the last known-good machine state at the start of the epoch. Before continuing execution from this point, the defective component is disabled and the system continues in a performance degraded mode without the broken resource.

Relying on on-line testing, rather than traditional redundancy techniques, allows us to achieve dramatically lower overhead than previous proposed techniques. Redundant approaches such as triple-modular redundancy [31] and N-version hardware [31] utilize redundant hardware on a cycle-by-cycle basis to detect and correct errant computation resulting from silicon defects. For each of these previous techniques, redundant hardware is used to verify the integrity of computation on the baseline hardware component, resulting in cost overheads of 100% or more [11]. An on-line testing-based approach, in contrast, is much less expensive because the hardware necessary to verify integrity and provide checkpoint/recovery is quite modest. Our entire facility only adds 5.8% additional hardware to a 4-wide VLIW processor with 32-KBytes of instruction and data cache.

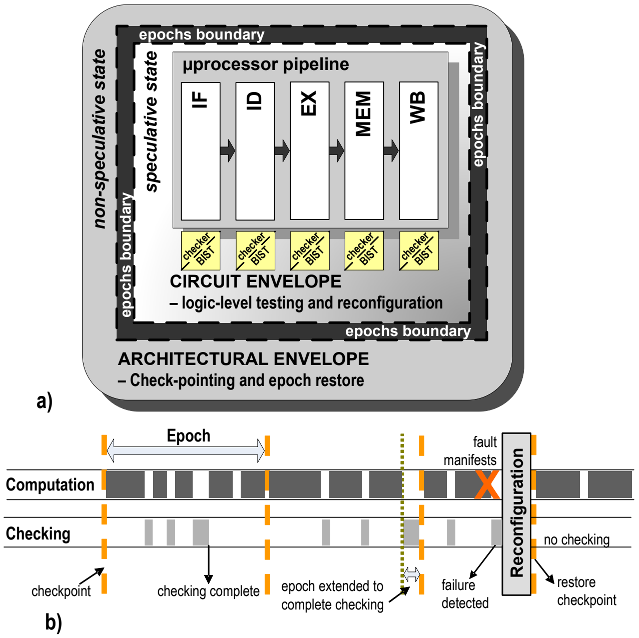

Figure 1. BulleProof pipeline architecture. Part a) shows how we equip a microprocessor pipeline for defect protection: Component-specific hardware testing blocks are associated with each design component to implement test generation and checking mechanisms. When a failure occurs, it is possible that results computed in the microprocessor core are incorrect. However, the speculative "epoch"-based execution guarantees that the computation can always be reversed to the last known-correct state. Part b) shows three possible epoch scenarios.

The remainder of this paper introduces our approach to defect tolerance and evaluates its impacts on design cost and performance. In Section 2 we describe in detail how on-line testing can be combined with checkpoint recovery to provide high-levels of defect tolerance at low cost. Section 3 presents a detailed simulation-based evaluation of the approach, using physical design analysis to gauge area costs and architectural simulation to judge performance impacts. Section 4 details previous work in the areas of defect tolerant microarchitecture design, on-line defect testing, and microarchitectural checkpoint recovery techniques. Finally, Section 5 gives conclusions and suggests future directions.

2. Testing and Recovery

Figure 1 illustrates the high-level system architecture of our defect tolerance approach, and it shows a timeline of execution that demonstrates its operation. At the base of our approach is a microarchitectural checkpoint and recovery mechanism that creates *computational epochs*. A computational epoch is a protected region of computation, typically at least 1000's of cycles in length, during which the occurrence of any erroneous computation, in this case due to the use of a defective device, can be undone by rolling the computation back to the beginning of the computational epoch. During the computational epoch, on-line distributed BIST-style tests are performed in the background, checking the integrity of all system components. Ideally, this checking will occur while functional units, decoders, and other microprocessor components are idle, as is often the case in a processor with parallel resources. By the end of the computational epoch, there are three possible scenarios that the control logic will handle. The first scenario (shown in the first epoch of Figure 1b) is when the checking completes before the end of the computational epoch. In this scenario, the hardware is known to be free of defects, thus, the results of the computational epoch are known to be free of defect-induced errors, and it can be safely retired to non-speculative storage.

In the second scenario (shown in the second epoch of Figure 1b), the computational epoch ends before the on-line testing infrastructure could complete the testing of all of the underlying hardware components. This scenario can occur because our microarchitectural checkpointing mechanism has only a finite amount of storage into which speculative state can be stored – once this space is exhausted, the computational epoch must end. I/O requests can also force early termination of a computational epoch. In this event, testing will be the only activity allowed on the processor, and it will run to completion while the processor pipeline is stalled. If at the end of testing the hardware is still deemed free of defects, the epoch's speculative state can safely retire to non-speculative storage.

Finally, the third scenario, depicted in the third epoch of Figure 1b, is when the on-line testing infrastructure encounters a defect in an underlying component, due to a transistor wearout, early transistor failure, or manifestation of an untested manufacturing defect. In this event, the execution from the start of the computational epoch to the point where the defect was detected cannot be trusted as correct, because this unchecked computation may have used the faulty component. Consequently, the results of the computed during the epoch are discarded, and the underlying hardware must be repaired. We do so by disabling the defective component. In a processor with instruction-level parallelism (ILP), there are typically multiple copies of virtually all components. Once a component is disabled the processor will continue to run in a performance-degraded mode. Additionally, a software interrupt is generated which notifies the system that the underlying hardware has been degraded, so the user can optionally replace the processor.

2.1 On-line Testing Techniques

The on-line testing infrastructure is responsible for fully verifying the integrity of the underlying hardware components. Our testing techniques are adopted from built-in self-test (BIST) [25], although we have tailored our designs to minimize the area of the testing hardware, and hence the area of our defect-protection infrastructure. For each of the pipeline components, we store a high quality input vector set in an on-chip ROM, which is fed into the modules during idle cycles. A checker is also associated with each component to detect any defect in the system. The primary techniques we utilize to verify the integrity of the underlying hardware are illustrated in Figure 2 and described below.

Decoder Checker: The decoders are validated by sending the same test vector to multiple decoders, and then comparing their outputs. The decoder test harness is illustrated in Figure 2a. In the event that the outputs do not match, one of the decoders has experienced a defect-related failure. In addition, it is important to determine which of the decoders has failed. Consequently, three decoders are sent the test vector, and a majority operator is used to identify which of the decoders has failed. Admittedly, it is rare that three decoders are not used within a single cycle, however, fully testing one of the decoders for stuck-at-0 and stuck-at-1 faults requires only 63 carefully selected vectors. In the case that the architecture has more than three decoders, each can be tested by including it in a battery of tests with any two other decoders.

Register File Checker: Register file integrity is checked using a four phase split-transaction test procedure, as illustrated in Figure 2b. The register file is unchanged from the original design, except that it has two address decoders (one for read and one for write), which allows testing of address decoder faults. In the first phase, a register file entry is read from the register file and stored in the *replacement register*. Testing of that register may now proceed whenever free read/write ports are available. If the register under test is read or written by the processor, the value is supplied by the replacement register. This same register is used to repair a broken entry, as described later. In the second phase, a random vector (generated with a linear feedback shift register, LFSR) is written into the register being tested, and in the third phase it is read back out and compared to the original vector. Finally, in the last phase the register entry (previously copied into the replacement register during the first phase) is written back into the appropriate register.

This process effectively tests both the register storage as well as the address decoders in the register file. The register storage is tested by writing and reading a value from the register. The address decoders are tested by virtue of the fact that the value written and read is fairly unique (*i.e.*, it is randomly generated), thus if either the read or write address decoder incurs a defect, some other (likely another register value) value will incorrectly appear during the read phase of the register file testing. Because the value stored in the register entry under test is available at all times from the replacement register, the testing process can be implemented as a series of split transactions. Consequently, different phases can be executed in non-subsequent cycles, whenever a free port is available on the register file. This facet of the approach greatly contains the performance impact, as shown in Section 3. The register file testing procedure is repeated until all of the registers have been validated. For our test processor with 32 registers, we can fully test the register file with 128 cycles, spread out over an entire computational epoch in cycles when the register file is not in use.

ALU and Multiplier Checker: The ALU is checked using a 9-bit mini-ALU, as shown in Figure 2c. During each cycle a test vector from the BIST unit is given to the ALU and compared with the output of the mini-ALU. It takes four cycles for the mini-ALU to test the full output of the main ALU. We use a 9-bit ALU to validate the carry out of each 8th bit in the 32-bit output. The same type of ALU checker is also used to verify the output of the address generation logic. Using the mini-ALU checker, it is possible to fully verify that the ALU circuitry is free of stuck-at-0 and stuck-at-1 faults with only 20 carefully selected test vectors.[1] A similar approach is used to validate the multiplier, which employs arithmetic residue checks [3]. Given an n-bit operand x, the residue x_r with respect to r is the result of the operation $x\%r$. When applied to multiplication, residue codes adhere to the following property: $(x_r * y_r) = (x * y)_r$. When the value of $r = 2^a - 1$ for some a, the residue operations are much simpler to implement in hardware [3]. The resulting multiplication checker requires only a shifter and simple custom logic.

Residue codes can detect most of the faults in a multiplier except those that manifest as multiples of the residue, a small class of faults where a single fault at an internal node could manifest as a

[1] It should be noted that our testing approach is in contrast to traditional BIST-style testing techniques that store both the input and output vectors, with the output vectors being compared to the output of the ALU. We found that by computing the output vector on a smaller adder, we could produce a tester that was significantly smaller.

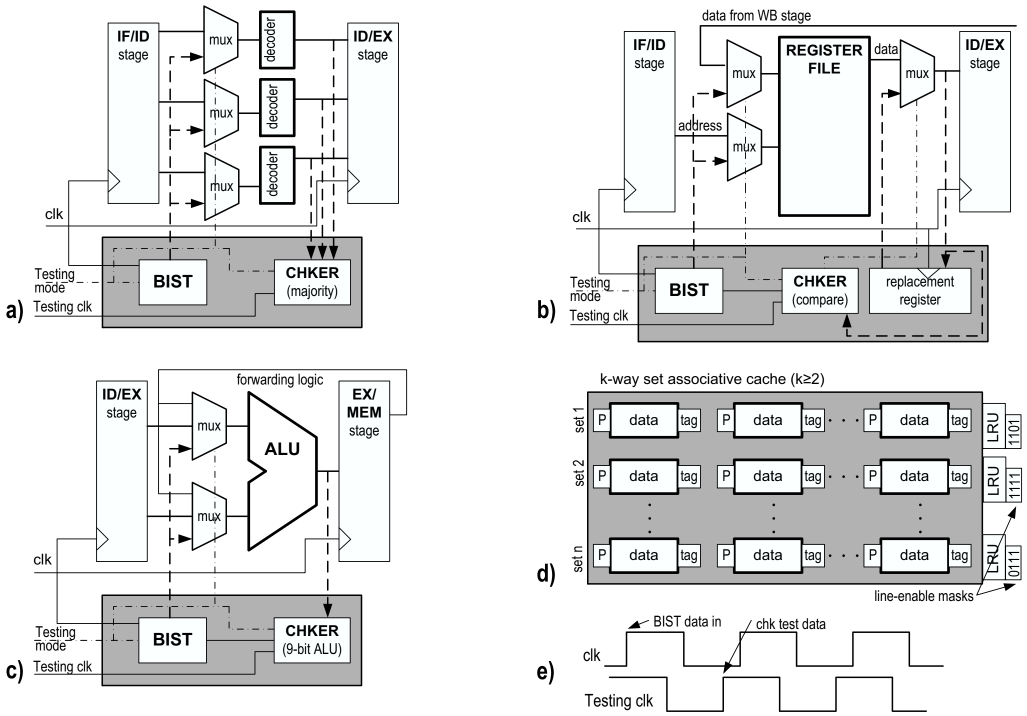

Figure 2. Component-specific on-line testing techniques. The decoders use a majority vote, as shown in part a). In part b) the register file tests one register at a time, by swapping to a replacement register. Part c) shows functional units exploiting arithmetic checkers. In part d) caches are equipped with a parity bit, a "volatile" bit to indicate the speculative state of the data stored in a line and bits to track a faulty cache line. Part e) shows the early clock edge for checker logic.

multiple of the correct value on the output. The errors missed by the residue checker are caught by a few additional carefully selected test vectors, against which the exact output is matched. Using this approach, the multiplier can be fully tested for stuck-at-0 and stuck-at-1 faults with a total of only 55 test vectors.

Cache Line Checker: Cache line integrity is maintained, as illustrated in Figure 2d, through the use of cache line parity. A single parity bit is associated with each line, holding the parity of the entire cache line plus the tag, valid bit, and LRU state for the line. When cache lines are written to the cache, the parity for the line is generated and stored. Subsequently, when the cache line is read, the parity is recomputed to verify the contents. In the event that the parity is correct, notwithstanding a multi-bit failure, which is beyond the scope of our single bit failure model, the cache line is known to be correct. In the event that a cache line parity check fails, a defect has been detected within the storage of the cache, consequently, the line must be disabled from further use and execution is rolled back to the last checkpointed epoch. Cache lines are disabled by setting a two bit field in the LRU state table, which indicates which line in the current set has been disabled. The disable bits in the LRU table are periodically reset to avoid soft errors in caches being interpreted as hard errors and rendering the cache lines unusable for the rest of the design's lifetime. Furthermore, at the end of each computational epoch, dirty cache lines are checked and written back to the next level of the memory hierarchy to guarantee recoverability in the presence of cache silicon defects. This approach is area-efficient, but it can only support a single failed line per set of a cache. Additional failed lines could be supported within a single set if more disable bits were to be included in the LRU logic.

The Test Clock: An important consideration in the testing of hardware components is the timing of the test vector samples. Since many transistor wearout-related failures manifest as progressively slower devices [13], the failure of the device may occur in a way where timing is no longer met for the component's critical path. Figure 2e shows how we address the issue by utilizing a slightly shorter clock cycle for sampling test vector outputs. The clock frequency safety margins in current microprocessors (e.g. to mitigate process variation) permits the use of this slightly shorter cycle testing clock with a negligible amount of false positives. This ensures that if a device is failing by showing slower response, it can be detected long before it affects any processor computation, which operates on the main clock cycle, longer than the testing cycle.

2.2 Micro-architectural Checkpointing

We rely on a microarchitectural rollback mechanism to restore correct program state in the event of a defect detection. The mechanism we employ is similar to the one described in [22]. During the execution of a computational epoch, the processor makes register and memory updates which would need to be discarded if a fault is detected. To prevent any memory updates with corrupted data, such updates are buffered in speculative state within the processor, until when the hardware is checked and certified to be functionally correct. It is worth noting that the same level of fault coverage is not feasible by simply stopping the computation and running the built-in tests on a regular basis, without any checkpointing, and reconfiguring the pipeline if a fault is found. In fact, with this approach it would not be possible to ensure that a detected fault had not corrupted earlier computation. In contrast, with the microarchitectural checkpointing facility, we can always roll back the state of the machine to the point when we last completed an on-line testing pass successfully (a point in the computation known to be correct). In addition, once the hardware is repaired, we can safely restart the program from this checkpoint.

As shown in Figure 3, register state is preserved by backing up the register file into a dedicated single-port SRAM at the begin-

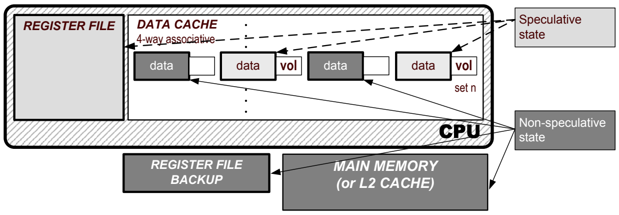

Figure 3. Checkpointing System. The storage in dark color indicates all the state that is non-speculative at any execution point. Light colored storage is speculative during each epoch, and it is only synchronized at epoch's boundaries.

ning of each computational epoch. The register backup can be done lazily by tagging the registers and copying them only before they get overwritten, so that there is no associated performance penalty. To support long epochs, memory updates are buffered within the local cache hierarchy. To implement in-cache speculative state, each cache line is augmented with a *volatile bit*. At the beginning of an epoch, all volatile bits are reset. When a value is stored to the cache, the volatile bit of the target cache line is set to indicate that the contents are speculative in the current epoch. The end of an epoch is determined by the ability of the local cache hierarchy to buffer the memory updates issued during the epoch. Therefore, it is designated by a cache miss on a cache set in which all of the cache lines are already marked as volatile. In this event, all speculative state resources have been exhausted and the processor must stall until the testing sweep is complete. Once the underlying hardware is determined to be defect-free, an epoch may end. At this point, all volatile bits from the cache lines are cleared, moving all formerly speculative state to non-speculative. To minimize performance costs of starting epochs (*i.e.*, copying the register file and clearing volatile bits), we extend each epoch as long as possible, until when speculative state resources are exhausted or a high-priority I/O request is generated, as discussed in Section 2.6. To provide even longer epochs, we introduce a small fully associative victim cache for volatile cache lines, so that the end of an epoch is now designated by a cache miss on a cache set with all lines marked as volatile, and while the victim cache is full of volatile lines. In this work we assume a uni-processor environment; hence, delaying the commit of stores to non-speculative storage has no effects on the system's performance. Similar microarchitectural checkpointing techniques that take into account the performance penalty of delayed stores in shared-memory multi-processor environments are described in [18].

2.3 Two-Phase Commit

Unfortunately, if only one checkpoint of the microprocessor's architectural state is preserved, there is a chance that errant computation from a new defect manifestation could be missed. The potential problem is illustrated in Figure 4: If a hardware check completes before a fault manifests, it becomes possible for an errant computation to be generated *later* in the same computational epoch. In this event, corrupted state updates would be committed to non-speculative state at the end of the epoch. The manifested fault will eventually be detected in the next epoch, but not before erroneous computation had a chance to be committed to non-speculative storage. We can solve this conundrum by adopting a two-phase commit procedure, which maintains two checkpoints of the processor's state. To implement this two-phase commit, we use an additional bit for each L1 data cache line. We also use an extra backup register file so that the processor's architectural state can be stored alternatively to one or the other of the two backup register files. This enables us to keep backups of the microprocessor's state for the last two epochs. Lines in the L1 data cache will be marked (using the two volatile bits) as being either non-speculative, in the previous epoch, or in the current epoch. At the end of each epoch, the volatile bits of the previous epoch are cleared, and the tags of the current epoch are updated to indicate that they refer to the previous epoch. During the new epoch, any access to the previous epoch's state must be first copied into the current epoch before being written, so that we do not corrupt the previous epoch's state. A similar technique for providing a sliding rollback window is described in [34].

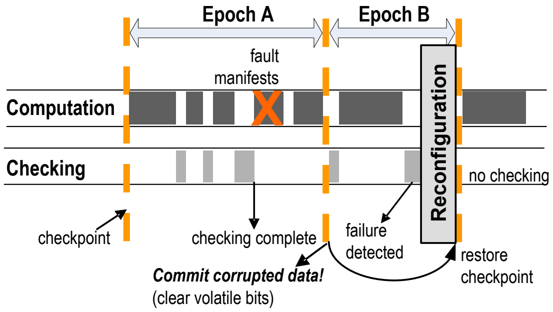

Figure 4. Incorrect recovery scenario. During the execution of epoch A, a fault manifests after the testing sweep is complete. The fault causes memory updates with corrupted data, which are committed at the end of the epoch. In epoch B, the fault is detected and recovery occurs. However, this happens too late to revert the corrupted memory updates.

2.4 Fault Recovery

In presence of a fault, recovery to a correct microprocessor architectural state is accomplished by flushing the pipeline and copying the architectural registers from the backup register file. The memory system is protected against possible corrupted updates issued after the fault manifestation by invalidating all the cache lines marked as volatile in the local cache hierarchy. Therefore, the presence of the fault is transparent to the application's correct execution. To provide forward progress the defective module must be disabled via hardware reconfiguration.

2.5 Repairing the Pipeline

In the event of a fault manifestation, the microarchitectural checkpointing mechanism will restore correct program state. However, before execution can safely continue, the underlying hardware must be repaired. We rely on the natural redundancy of ILP processors to reduce the cost of repair. Faulty components are removed from future operations, and the pipeline can keep running in a performance-degraded mode. To implement pipeline repair, the following facilities are included in the design:

1) Faulty functional units, such as ALUs, multipliers and decoders are disabled from further use. Consequently, further execution must limit the extent of parallelism allowed.

2) Faulty register file entries are repaired using the replacement register, as shown in Figure 2b. The replacement register overrides a single entry of the register file, thus, any value read or written to the defective register is now serviced by the replacement register.

3) Faulty cache lines are excluded using a two-bit register in the LRU logic. Upon detecting a faulty line, the LRU state register is updated to indicate that the defective line is no longer eligible as a candidate line during replacement.

Given enough silicon defects, it may be no longer possible to tolerate an additional defect in a particular subcomponent. The degree to which defects can be tolerated is dictated by the number of redundant components available. In general, with *N* components, it is possible to tolerate *N-1* defects. Once the *N-1*-th component fails, the hardware should generate a signal to the operating system to indicate that the system is no longer protected against defects. Finally, it should be noted that if the failure is the result of a transistor slowdown, e.g., due to gate oxide wearout or to a negative-bias temperature instability (NBTI), it may be possible to recover the failing component by slowing down the system clock or increasing the component's voltage. Specifics on how to accomplish this is beyond the scope of this paper; we will be exploring this possibility in future research.

2.6 Handling Input/Output Requests

Instructions that perform input and output requests require special handling in our defect tolerant microprocessor design. Since I/O operations are typically non-speculative, they can only be executed at the end of a computational epoch. To accommodate them efficiently, we introduce three flavors of I/O requests into our design: high-priority, low-priority, and speculative (the type of I/O request is associated with the memory address, and it is specified in the corresponding page table entry).

High priority I/O requests are deemed extremely time sensitive, thus, they force the end of a computational epoch, which may force the processor to stall to complete the testing sweep. After this, the I/O request executes safely, and another epoch can start immediately after it.

Low priority I/O requests are less time sensitive. Hence, they are held in a small queue where they age until the end of the current epoch, at which point they are all serviced. To prevent I/O starvation in programs with long computational epochs, low-priority I/O requests are only allowed to age for a small fixed period of time (about one μsec in our design). In addition, the computational epoch must end when any attempt is made to insert a low-priority request into a full I/O queue.

Speculative I/O requests are I/O requests that are either insufficiently important to care about the impacts of unlikely defects (*e.g.*, writes to video RAM, which could be easily fixed in the next frame update), or they are idempotent (*e.g.*, the reading of a data packet from a network interface buffer). Such requests are allowed to execute speculatively before the end of a computational epoch. If a defect is encountered during the epoch in which they execute, they will just be re-executed in the following epoch, once the defective component has been disabled.

2.7 Assumptions and Limitations

While our approach to providing defect protection for a microprocessor pipeline and on-chip memory system provides low cost with very limited performance impact, it does have a a number of error model assumptions and usage limitations which we detail below in this subsection.

First, we assume a fairly treacherous error model for this work. Specifically, we assume that devices can suffer from catastrophic failures at any time, which can be successfully detected with our online functional tests. In addition, transistors can suffer gradual slowdown, for example from gate oxide wearout or negative-bias temperature instability (NBTI), in which case transistors gradually slow down until they do not meet frequency requirements. In this case, the aggressive online testing clock will detect this condition before it affects computation.

The primary limitation of this approach in its current form is that it cannot be used to detect and correct transient faults. This is because the design assumes that if a computation is corrupted, the defect that led to that corruption is observable forever. We made a design choice to implement defect tolerance without transient error tolerance, as there are already emerging low-cost solutions in this latter domain. Examples are techniques using time-borrowing (e.g., Razor [2]) or time-redundancy (e.g., microarchitecture-based introspection [27]). Consequently, any cost-competitive technique to correct permanent silicon defects must be either *i*) a single low-cost technology that corrects both defects and soft errors, or *ii*) a technology complementary to existing soft error tolerance techniques with comparably low cost. Our approach targets this second group. In addition, to our knowledge, no solution is yet available in the former one.

Finally, the use of our approach places a few restrictions on the pipeline and on-chip cache organizations. In particular, the approach of disabling defective functional units requires multiple units of each class, otherwise, a single defect in a critical non-replicated unit could render the processor broken. Given the abundance of resources in most modern ILP processors, this limitation is not a significant drawback for most designs. Additionally, the cache organization must be set-associative to accommodate both speculative and non-speculative state.

3. Experimental Evaluation

In this section we present a detailed physical design of a 4-wide VLIW processor including instruction and data cache and enhanced with our technology. We analyze the performance of the design, using both circuit timing simulation and architectural simulation and gauge the impacts of defect protection, both during normal operation and after a component has been disabled. Finally, we examine the cost of the defect protection technology by measuring area overhead of the testing logic (e.g., vector generation and checkers). We also evaluate the coverage of our approach, *i.e.*, what fraction of defects randomly placed are protected, by carefully measuring the portion of silicon area protected.

3.1 Experimental Framework

Below we detail the infrastructure used for our studies.

Circuit-Level Evaluation: The 4-wide VLIW prototype was specified in Verilog, and synthesized for minimum delay using Synopsys Design Compiler. This produces a structural Verilog netlist of the processor mapped to Artisan standard cell logic using the TSMC 0.18um fabrication technology. The design was then placed and routed using Cadence Sedsm, which in turn yields a physical design with wire capacitances and individual component areas. Finally, we back annotated the design to obtain a more accurate delay profile, and simulated it with Synposys' PrimeTime to verify its timing and functional correctness.

We verified, for each component and test vector set, that all stuck-at-0 and stuck-at-1 faults are detected. In general, test vector sets were identified using carefully hand-selected vectors, or by randomly cycling through random vector sets until a small group of effective vectors was located. Test vector coverage is verified by inserting a hard fault at each net of the design and then determining if a change in the output is observable for the current input test vector set. For a test vector set to provide full coverage, there must be at least one vector that identifies a hard fault in all nets of the design. Once the test vector set is identified, it is encoded into a on-chip ROM storage unit, created using Synopsys design tools.

Architectural Evaluation was done using the Trimaran tool set, a re-targetable compiler framework for VLIW/EPIC processors [35], and the Dinero IV cache simulator [16]. The simulator was configured to model the VLIW baseline configuration and memory hierarchy as detailed in the following section. We chose to evaluate our designs using benchmarks from SPECint2000, MediaBench [21] and MiBench [14] benchmark suites. These benchmarks cover a wide range of potential applications, including desktop applications, server workloads, and embedded codes.

Coverage Analysis: We implement coverage analysis by injecting faults into a logic timing level simulation of the detailed VLIW processor physical design. Since characterization of silicon defects

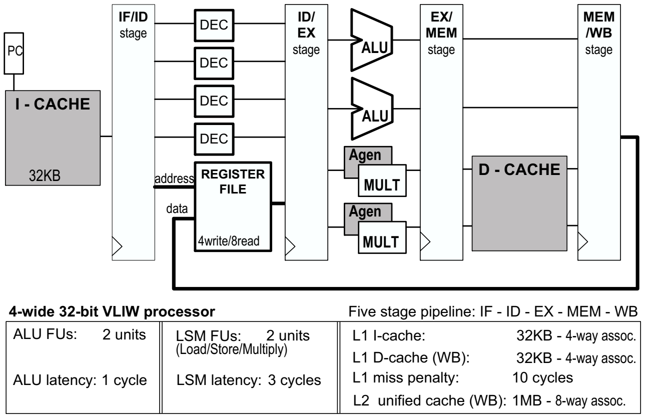

Figure 5. Baseline processor: 4-wide 32-bit VLIW.

in nanometer-sized technologies is still an open research problem [12], we opted for a stuck-at-0 and stuck-at-1 fault model. Defects are injected into a placed-and-routed implementation of the design. Faults are assigned to gates and wires so that the probability of a device X becoming defective p_{defect} is equal to: $p_{defect} \: \alpha \: A_x * \lambda_x$ where A_x is the area of the device and λ_x is the average estimated activity of the device. As such, large devices with high activity rates are most apt to failure, while small components or components with little activity are at lower risk.

Baseline Architecture: Our baseline architecture, which we enhanced with our low-cost defect protection technology, is a 4-wide VLIW architecture, with a 32-KByte instruction and data caches. We chose this architecture for our evaluation because it represents a mainstream embedded target, often used in applications where cost and reliability are paramount concerns. An overview of the architecture and details of its components are shown in in Figure 5. The baseline pipeline is a 4-wide VLIW processor with 32-bit fixed-point datapaths. The instruction set of the processor is loosely based on Alpha instruction set. Each VLIW instruction bundle is 128-bit long, consisting of 4 independent 32-bit instructions. The processor pipeline has five stages. The instruction fetch (IF) stage is responsible for fetching the 128-bit VLIW instruction from the 32-KByte instruction cache. The instruction decode (ID) stage decodes 4 independent instructions per cycle and fetches register operands from a register file with 8 read ports and 4 write ports. The execute (EX) stage performs arithmetic operations, multiplications, and address generation. The memory (MEM) stage accesses the 32-KByte data cache and main memory. Finally, the writeback (WB) stage retires instruction results to the register file.

3.2 Testing Performance and Design Coverage

In this section we examine the cost of on-line testing. In particular, we examine the bandwidth requirements for on-line testing each component (*i.e.*, number of vectors required to fully test a component), the area costs of our test harnesses, and we compute the overall design defect coverage.

On-line Testing Bandwidth Requirements: The bandwidth requirements of testing are the number of vectors needed to fully test components for stuck-at-0 and stuck-at-1 faults. Table 1 lists the number of vectors to fully test each component, showing that few vectors are required to test each unit. Considering that the length of a computational epoch will typically be 1000's of cycles, it is quite promising that testing can be completed using only occasional idle cycles. The caches are not listed in Table 1 because the use of parity bits allow for the continuous detection of defects.

Test Harness Area Overheads and Design Coverage: The addition of test vector ROMs, where test vectors are stored, plus the checkers and checkpointing infrastructure bears a cost on the overall size of the design. Table 2 lists the total area of the defect tolerant component (Total area), the defect protection infrastructure area (Checker area), and the area that is covered by the test harness (Protected area). The coverage of the component is also shown as a percentage, this is the total fraction of the final design in which a defect that occurs will be detected and repaired. This metric can also be thought of as the probability that a defect in the component would be detected, given a random occurrence of a defect.

Component	Test vectors (or cycles)
ALU	20
MUL	55
Decoder	63
RegFile	128

Table 1. Test vectors. Number of test vectors to achieve 100% coverage for stuck-at-0 and stuck-at-1 faults.

Design block	Total area (um^2)	Checker area (um^2)	% of tot. area	Protected area (um^2)	% of tot. area
IF	131323	4523	3.4	118190	90.0
ID	278396	22776	8.2	237726	85.4
RF	2698213	133213	4.9	2501787	92.7
EX	2140100	375580	17.5	1740486	81.3
WB	394673	4763	1.2	250165	63.4
Overall Core	5642705	540855	9.6	4848354	85.9
I-cache 32KB	2037062	13012	0.6	1881416	92.4
D-cache 32KB	2047472	13012	0.6	1891826	92.4
Overall System	9727239	566879	**5.83**	8621596	**88.6**

Table 2. Area costs of individual design components and of the overall system. The table reports the total area of each design block, the area dedicated to checkers, and the portion of the overall area that is protected as a result of the specific solution.

As shown in Table 2, area overheads for defect protection are quite modest, with most overheads less than 10%. The overheads within the caches are even lower, less than 1% for the prototype. Consequently, the overall overhead for defect protection is quite low. Adding support for defect protection increased the total area of the design by only 5.83%. The defect coverage is also quite good, with most components in the 80 and 90 percentiles. The overall coverage of the design, *i.e.*, the total area of the final defect tolerant design in which a defect could be detected and corrected, is 88.6%. In other words, 9 out of 10 randomly placed defects would be detected and corrected by our prototype design.

Analysis of coverage results indicates a number of opportunities to improve coverage in future work, without significantly increasing protection overheads. Examination of the design indicates that currently 89% of the area is protected from defects. Consequently, devising protection schemes for the remaining fraction of the design, even if very expensive, would not incur a significant area cost. The unprotected area of the design mainly consists of resources that do not exhibit inherent redundancy in the design. Such resources include interconnection and control logic. Currently, we are developing techniques to provide defect tolerance for these resources, such as system-level protection solutions, that periodically check the resources. Based on our preliminary studies, we estimate that such techniques will not contribute significantly to the total area cost of the protection mechanism, while pushing the total protection coverage closer to 100%.

3.3 Run-time Performance

In this section we examine the impact of our defect protection mechanism on the performance of programs running on the defect tolerant prototype design. The primary source of potential slowdown occurs when a computational epoch is too small (or the test-

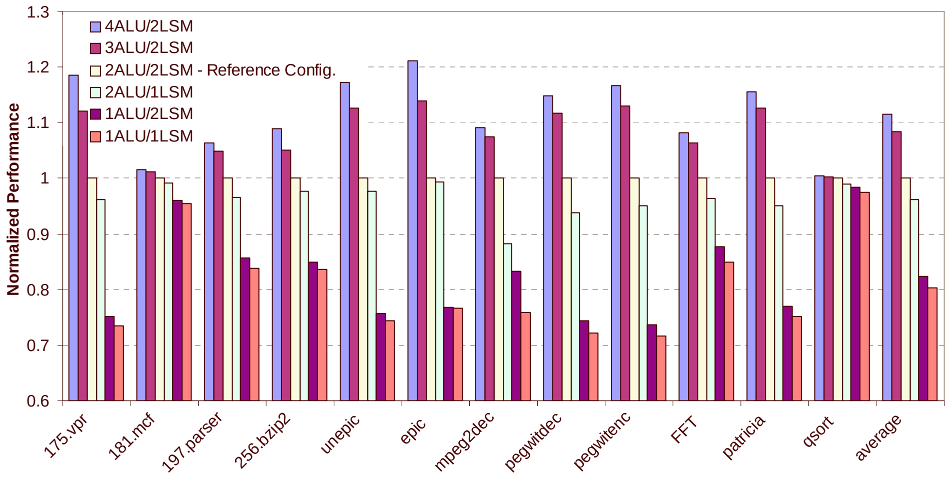

Figure 6. Performance degradation. The graph shows the performance of a variety of prototype processor pipelines that have been impaired through reconfiguration. A configuration with *n-ALU/m-LSM* indicates that the prototype processor pipeline has n ALUs and m address generation/multiplier units.

Benchmark	Avg. epoch size (cycles)	Data L1 miss rate	Avg. ALU util. (%)	Avg. LSM util. (%)	Avg. Dec. util. (%)	Avg. reg. rw/cycle
175.vpr	50499	3.10	69.71	18.41	59.00	4.72
181.mcf	120936	3.54	36.89	10.70	67.00	5.36
197.parser	106380	2.10	54.22	19.71	52.25	4.18
256.bzip2	162508	8.88	55.91	33.93	73.50	5.88
unepic	33604	17.16	68.70	14.29	55.50	4.44
epic	196211	6.60	72.80	8.28	29.25	2.34
mpeg2dec	1135142	0.59	55.81	54.55	46.25	3.70
pegwitdec	169617	10.42	62.15	45.06	62.50	5.00
pegwitenc	304310	12.81	69.09	42.19	63.75	5.10
FFT	23145	1.49	56.88	43.95	33.50	2.68
patricia	139952	1.19	55.20	37.69	57.75	4.62
qsort	1184756	2.55	20.08	18.74	32.25	2.58
Average	**302254**	**5.87**	**56.45**	**28.96**	**52.71**	**4.22**

Table 3. Epoch statistics for the baseline configuration. Listed is the average epoch size in cycles along with L1 data cache miss rates and statistics regarding the utilization of ALUs, L1 data cache memory ports (LSM), decoders, and register file ports.

ing requirements too great) to allow testing to complete within the time speculative state resources are exhausted.

Performance Impact of Defect Testing: Table 3 lists statistics about computational epochs for a variety of programs while running on the baseline VLIW processor with a 32 KByte 4-way set-associate data cache and an eight entry fully associative volatile victim cache. Listed is the average epoch size in cycles along with the L1 data cache miss rate. Also shown are statistics regarding the utilization of ALUs, L1 data cache memory ports (LSM), decoders, and register file ports. It is clear from this table that the performance overhead of defect testing is quite low. For the program with the shortest average epoch length (FFT), the number of test cycles is at most 0.5% of the total number of cycles within the epoch. For this program, even if we could not complete the testing during idle cycles, the performance impact would be negligible. We do not graph the performance impacts directly because there simply were none! All programs were able to complete testing within each epoch without delaying the start of the next.

It should be noted that there is a useful correlation between epoch length and average component utilization. For many of the programs with short epoch lengths (*e.g.*, FFT and unepic), there are correspondingly low functional unit utilizations. This is to be expected because a program with a short epoch length would have a large amount of cache turnover, which in turn would lead to many pipeline stalls and low functional unit utilization – and plenty of time for defect testing. While programs with long epochs tend to have higher component utilization, they do provide more time for the test harness to complete its task. In addition, we examined the effect of different cache geometries on average epoch size, and we found that there is little performance impact for defect testing for a wide range of cache geometries.

Performance Impact of Degraded Mode Execution: Once a defect has been located, the processor must be reconfigured by disabling the defective component. This reconfiguration will not allow as much parallelism as previously afforded in the unbroken pipeline, resulting in a performance degradation. Figure 6 graphs the performance of a variety of prototype processor pipelines that have been impaired through reconfiguration. In the experiments, *n-ALU/m-LSM* indicates that the experiment was run with n ALUs and m address generation units/multipliers. The number of resources is varied from one to four. As shown in Figure 6, losing an ALU in a 2ALU/2LSM machine configuration renders an average of 18% performance degradation. The average performance degradation is limited to only 4% when losing an address generation/multiplier unit in the same machine configuration. Machine configurations with more resources can exhibit even lower performance degradation after being impaired through resource reconfiguration. For example, machine configurations with four and three ALUs loosing one ALU results in an average performance degradation of 3% and 8% respectively.

4. Related Work

To date, only a few efforts have explored techniques to provide defect tolerance for microprocessor pipelines. In [29], the authors propose to use hardware redundancy and reconfiguration to improve yield and increase defect tolerance of future systems. The paper suggests that hardware redundancy use should not be limited only to memories but that inherent redundancy should be exploited in both uni-processors and multi-processors. The authors identify three primary types of redundancy that can be used in a system: component level redundancy (replicated functional units etc.), array redundancy (spare rows and columns in bit arrays), and dynamic queue redundancy (spare queue entries).

In [11], we explored the design space of defect-tolerant chip multiprocessor switch designs and the resulting tradeoff between defect tolerance and area overhead. We compared traditional defect tolerant mechanisms such as triple modular redundancy and error corrections codes with domain-specific techniques that include end-to-end error detection, resource sparing, and iterative diagnosis and reconfiguration and we concluded that such techniques are more effective and that designs are attainable to tolerate a larger number of defects with less overhead than traditional techniques. Further, we studied the tradeoff between the area overhead and the reliability provided for different granularities of applying the protection mechanisms and we proposed a technique for decomposing the design into modest-sized clusters for providing higher reliability with less area overhead.

In the NanoBox project [19], error detection and correction is distributed at the lower levels of logic blocks, and a self-correcting logic block consisting of a lookup table is proposed with appropriate error detection and correction entries. In order to provide adequate fault tolerance triple modular redundancy and information coding implementations of the NanoBox are used. However, these approaches have unrealistically high area (49x-181x) and delay (7x-8x) overheads over conventional implementations.

Both the Teramac [1] and the Phoenix [15] projects, proposed the creation of an external circuitry fabricated from reliable devices for periodically surveying the design and reconfiguring faulty parts. However, periodic design checking has a high performance overhead, and it is not scalable as the design size grows.

DIVA, an on-line checker component inserted into the retirement stage of a microprocessor pipeline, fully validates all computation, communication, and control exercised in a complex microprocessor core [36]. The approach unifies all forms of permanent and transient faults, making it capable of correcting design faults, transient errors (like those caused by natural radiation sources), and even untestable silicon defects. As such, the design is capable of tolerating silicon defects in the core processor. However, the DIVA approach corrects each operation after it occurs, but no mechanism is available for diagnosing problems and repairing the underlying computation fabric.

In [9], the authors present (SRAS), a Self-Repairing Array Structures hardware mechanism, for on-line repairing defected microprocessor array structures such as a reorder buffer and a branch history table. The proposed mechanism detects faults by employing dedicated "check rows". To achieve that, every time an entry is written to the array structure, the same data is also written into a check row. Subsequently, both locations are read and their values are compared which effectively detects any defected rows in the structure. In the proposed mechanism, when a faulty row is detected, it gets remapped by using a level of indirection. For circular access structures like the reorder buffer, this is achieved by mapping out the faulty rows, while for tabular structures like the branch history table, accesses to faulty rows are redirected to spare rows.

In [10], a fault-tolerant microprocessor design is presented, which exploits the existing redundancy in current microprocessors for providing higher reliability through dynamic hardware reconfiguration. The proposed microprocessor design uses DIVA checkers [36] for system-level error detection. The design also uses a mechanism for diagnosing hard faults, through tracking the instructions core structure occupancy from decode until commit. After diagnosing a hard fault, the microprocessor deconfigures the faulty part and continues operation at a gracefully degraded level of performance. However, since the proposed technique was not evaluated at the physical level (gate-level netlist) there are no clear estimates of its area cost and fault coverage.

A number of recent research efforts have utilized microarchitectural checkpointing techniques as a mechanism to recover from transient faults. The basic approach is to provide continuous computational checking, through redundant execution on idle resources [27] or dual-modular redundancy (DMR) [32]. Once a computation error is detected, the system is rolled back to the last state checkpoint, after which the system will retry the computation. Because of the sparseness of SER-related transient faults, it is likely that the computation will complete successfully on the second attempt. This earlier work does not discount the novelty of our paper, as our paper corrects for hard silicon defects. Interestingly, these techniques would likely be quite cost effective when employed in tandem as both techniques rely on microarchitectural checkpointing to perform their correction.

In the context of online testing of processors, various concurrent error detection schemes (CED) schemes have been proposed [24]. Most schemes incorporate a checker that compares the expected behavior with that of the unit under test. Duplication is the most common way used to detect defects, although some work has suggested that a diverse implementation of logic is more robust than simple duplication [24] as it offers protection against some common mode failures. Another solution proposed in the direction of online testing are Berger codes [5] which can detect all unidirectional errors, and Bose-Lin codes [8] which can detect t unidirectional errors (known as t-EC). These codes are suitable for the reliability of systems that have large occurrence of one type of binary numbers. However, it is non-trivial using these codes for online testing of datapaths as they impose constraints on the way the logic block is designed such that only unidirectional faults occur. Additionally, parity prediction schemes have long been used for ensuring fault-secure datapaths [26].

5. Conclusions

In this paper, we presented the *BulletProof* pipeline, which is the first ultra low-cost defect protection mechanism for microprocessor pipelines and on-chip cache memories. The approach we take is markedly different from previous techniques that utilize replicated hardware to validate all computation. We instead use the more area efficient approach of combining on-line testing with microarchitectural checkpointing. A microarchitectural checkpointing mechanism creates speculative computational epochs during which distributed domain-specific on-line test techniques are used to verify the integrity of the underlying hardware components. If at the end of an epoch the hardware is determined to be correct, the speculative computation of the epoch is allowed to commit. Otherwise, the program state is rolled back to the beginning of the epoch, and the defective component is disabled thereby allowing the processor to continue correct execution in a degraded-performance mode.

We presented a detailed design of a prototype 4-wide VLIW processor, and demonstrated that area overheads due to our defect protection are quite small, with only a 5.8% increase in total area. Moreover, the coverage of our approach was also quite good, with 89% of the total area of the prototype design effectively protected against silicon defects. Additional simulations demonstrated that sufficient idle pipeline time exists for all programs to allow online defect testing to proceed concurrently with program execution, with no perceivable impact on program performance. Additionally, we examine the performance of prototype processors running with disabled components in a degraded mode. When a 4-wide VLIW has been impaired through resource reconfiguration after losing one resource, the performance degradation ranges from 4% to 18%.

We feel that this paper makes a strong case for the use of on-line testing and microarchitectural checkpointing to implement future defect tolerant designs. The approach is both efficient, with high coverage and low performance impacts, and also inexpensive, with small area overheads.

Looking forward we are working hard to continue to reduce the area overheads of our testing infrastructure while at the same time improving coverage. We are examining techniques to further compress test vectors using carefully designed finite-state automatons. In addition, we are developing techniques to provide low-cost defect tolerance for control logic, which constitutes the bulk of the portion of our prototype design that was not protected. We are also examining the applicability of our approach to desktop and server class microprocessors. Our early indications are that this technique will perform equally well for desktop and server microprocessors, as the checkpointing mechanism (and overheads) is unchanged, and the online testing techniques are agnostic to the underlying components such that infrastructure overheads stay low-cost for a wide variety of hardware structures. Finally, there is significant opportunity to lower realized costs by utilizing the on-line testing support and microarchitectural checkpointing for other value-added capabilities. For example, the on-line testing infrastructure could be used to tune frequency and voltage to eliminate ambient temperature and voltage margins [2]. Similarly, the microarchitectural checkpoint mechanism could also be used to provide support for transient fault tolerance or speculative shared memory access [20].

Acknowledgments

This work is supported by grants from NSF and the Gigascale Systems Research Center.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Int'l Conf. on Supercomputing (ICS)*, pages 1–6, June 1990.

[2] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, 2004.

[3] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Trans. on Computers*, C-20(II):1322–1331, 1971.

[4] T. S. Barnett and A. D. Singh. Relating yield models to burn-in fall-out in time. In *Proc. of Int'l Test Conference (ITC)*, pages 77–84, 2003.

[5] J. M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[6] K. Bernstein. Nano-meter scale CMOS devices (tutorial presentation). In *5th Int'l Symposium on Quality of Electronic Design*, 2004.

[7] S. Borkar. VLSI design challenges for gigascale integration (keynote address). In *18th Int'l Conference on VLSI Design*, 2005.

[8] B. Bose and D. J. Lin. Systematic unidirectional error-detecting codes. *IEEE Trans. on Computers*, 34(11):1026–1032, 1985.

[9] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, June 2004.

[10] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, Nov. 2005.

[11] K. Constantinides, J. Blome, S. Plaza, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. BulletProof: A defect-tolerant CMP switch architecture. In *Proc. of the Int'l Symposium on High-Performance Computer Architecture*, Feb. 2006.

[12] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman. Evaluation of test metrics: stuck-at, bridge coverage estimate and gate exhaustive. In *VLSI Test Symposium*, pages 66–71, 2006.

[13] P. Gupta and A. B. Kahng. Manufacturing-aware physical design. In *Proc. of Int'l Conference on Computer-Aided Design (ICCAD)*, pages 681–685, 2003.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characteristics*, pages 3–14, 2001.

[15] J. R. Heath, P. Kuekes, G. Snider, and S. Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280(5370):1716–1721, 1998.

[16] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. on Computers*, 38(12):1612–1630, 1989.

[17] A. M. Ionescu, M. J. Declercq, S. Mahapatra, K. Banerjee, and J. Gautier. Few electron devices: towards hybrid CMOS-SET integrated circuits. In *Proc. of the Design Automation Conference*, pages 88–93, 2002.

[18] B. Janssens and W. K. Fuchs. The performance of cache-based error recovery in multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 5(10):1033–1043, 1994.

[19] A. J. KleinOsowski and D. J. Lilja. The NanoBox project: Exploring fabrics of self-correcting logic blocks for high defect rate molecular device technologies. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 19–24, 2004.

[20] M. Kirman, N. Kirman, and J. Martinez. Cherry-MP: Correctly integrating checkpointed early resource recycling in chip multiprocessors. *Intl. Symposium on Microarchitecture (MICRO)*, Dec. 2005.

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Int'l Symposium on Computer Architecture*, pages 330–335, 1997.

[22] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, pages 3–14, 2002.

[23] M. Meterelliyoz, H. Mahmoodi, and K. Roy. A leakage control system for thermal stability during burn-in test. In *Proc. of Int'l Test Conference (ITC)*, Nov. 2005.

[24] S. Mitra and E. J.McCluskey. Which concurrent detection scheme to choose? In *Proc. of Int'l Test Conference (ITC)*, 2000.

[25] B. T. Murray and J. P. Hayes. Testing ICs: Getting to the core of the problem. *IEEE Computer*, 29(11):32–38, 1996.

[26] M. Nicolaidis, R. de Oliveira Duarte, S. Manich, and J. Figueras. Fault-secure parity prediction arithmetic operators. *IEEE Design & Test of Computers*, 14(2):60–71, 1997.

[27] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *Proc. of Int'l Conference on Dependable Systems and Networks (DSN)*, 2005.

[28] J. M. Rabaey. *Digital integrated circuits: a design perspective*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[29] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of Int'l Conference on Computer Design (ICCD)*, 2003.

[30] M. Shulz. The end of the road for silicon. *Nature Magazine*, June 1999.

[31] D. P. Siewiorek and R. S. Swarz. Reliable computer systems: Design and evaluation, 3rd edition. *AK Peters, Ltd*, 1998.

[32] J. Smolens, B. Gold, K. J, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: Bounding the soft-error detection latency and bandwidth. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[33] J. H. Stathis. Reliability limits for the gate insulator in CMOS technology. *IBM Journal of Research and Development*, 46(2/3):265–286, 2002.

[34] R. Teodorescu, J. Nakano, and J. Torrellas. SWICH: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro*, 2006.

[35] Trimaran. An infrastructure for research in ILP. www.trimaran.org

[36] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Proc. of Int'l Conference on Dependable Systems and Networks (DSN)*, pages 411–420, 2001.