Journal of Instruction-Level Parallelism 2 (2000) 1-6 Submitted 2/2000; published 5/2000

DIVA: A Dynamic Approach to Microprocessor Verification

Todd M. Austin TAUSTIN@EECS.UMICH.EDU
Advanced Computer Architecture Laboratory
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109 USA

Abstract

Building a high-performance microprocessor presents many reliability challenges. Designers must verify the correctness of large complex systems and construct implementations that work reliably in varied (and occasionally adverse) operating conditions. To further complicate this task, deep submicron fabrication technologies present new reliability challenges in the form of degraded signal quality and logic failures caused by natural radiation interference.

In this paper, we introduce *dynamic verification*, a novel microarchitectural technique that can significantly reduce the burden of correctness in microprocessor designs. The approach works by augmenting the commit phase of the processor pipeline with a functional checker unit. The functional checker verifies the correctness of the core processor's computation, permitting only correct results to commit. In the event of an incorrect result, the checker fixes the error and flushes any incorrect results from the core using the existing speculation recovery mechanism. Overall design costs can be dramatically reduced because designers need only verify the correctness of the checker unit. The core processor need not be fully correct, only sufficiently correct that its errors do not adversely affect performance.

We detail the DIVA checker architecture, a design optimized for simplicity and low cost. Using detailed timing simulation, we show that even resource-frugal DIVA checkers have little impact on core processor performance. To make the case for reduced verification costs, we argue that the DIVA checker should lend itself to functional and electrical verification better than a complex core processor. Finally, future applications of dynamic verification are suggested.

1. Introduction

Reliable operation is perhaps the single most important attribute of any computer system, followed closely by performance and cost. Users need to be able to trust that when the processor is put to a task the results it renders are correct. If this is not the case, there can be serious repercussions, ranging from disgruntled users to financial damage to loss of life. There have been a number of high-profile examples of faulty processor designs. Perhaps the most publicized case was the Intel Pentium FDIV bug in which an infrequently occurring error caused erroneous results in some floating point divides [1]. More recently, the MIPS R10000 microprocessor was recalled early in its introduction due to implementation problems [2]. These faulty parts resulted in bad press, lawsuits, and reduced customer confidence. In most cases, the manufacturers replaced the faulty parts with fixed ones, but only at great expense. For example, Intel made available replacement Pentium processors

to any customer that had a faulty part and requested a replacement, at an estimated cost of $475 million [3].

1.1 Designing Correct Processors

To avoid reliability hazards, chip designers spend considerable resources at design and fabrication time to verify the correct operation of parts. They do this by applying functional and electrical verification to their designs. [1]

Functional Verification Functional verification occurs at design time. The process strives to guarantee that a design is correct, *i.e.*, for any starting state and inputs, the design will transition to the correct next state. It is quite difficult to make this guarantee due to the immense size of the test space for modern microprocessors. For example, a microprocessor with 32 32-bit registers, 8k-byte instruction and data caches, and 300 pins would have a test space with at least 2^{132396} starting states and up to 2^{300} transition edges emanating from each state. Moreover, much of the behavior in this test space is not fully defined, leaving in question what constitutes a correct design.

Functional verification is often implemented with simulation-based testing. A model of the processor being designed executes a series of tests and compares the model's results to the expected results. Tests are constructed to provide good coverage of the processor test space. Unfortunately, design errors sometimes slip through this testing process due to the immense size of the test space. To minimize the probability of this happening, designers employ various techniques to improve the quality of verification including co-simulation [4], coverage analysis [4], random test generation [5], and model-driven test generation [6].

A recent development called *formal verification* [7] works to increase test space coverage by using formal methods to prove that a design is correct. Due to the large number of states that can be tested with a single proof, the approach can be much more efficient than simulation-based testing. In some cases it is even possible to completely verify a design. However, this level of success is usually reserved 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 [8, 9].

Electrical Verification Functional verification only verifies the correctness of a processor's function at the logic level, it cannot verify the correctness of the logic implementation in silicon. This task is performed during electrical verification. Electrical verification occurs at design time and fabrication time (to speed bin parts). Parts are stress-tested at extreme operating conditions, *e.g.*, low voltage, high temperature, high frequency, and slow process, until they fail to operate.[2] The allowed maximum (or minimum) for each of these operating conditions is then reduced by a safe operating margin (typically 10-20%) to ensure that the part provides robust operation at the most extreme operating conditions. If after this

1. Often the term "validation" is used to refer to the process of verifying the correctness of a design. In this paper we adopt the nomenclature used in the formal verification literature, *i.e.*, *verification* is the process of determining if a design is correct, and *validation* is the process of determining if a design meets customers' needs. In other words, verification answers the question, "Did we build the chip right?", and validation answers the question, "Did we build the right chip?".
2. Or fail to meet a critical design constraint such as power dissipation or mean time to failure (MTTF).

process the part fails to meet its operational goals (*e.g.*, frequency or voltage), directed testing is used to identify the critical paths that are preventing the design from reaching these targets [10].

Occasionally, implementation errors slip through the electrical verification process. For example, if an infrequently used critical path is not exercised during electrical verification, any implementation errors in this circuit will not be detected. Data-dependent implementation errors are perhaps the most difficult to find because they require very specific directed testing to locate. Examples of these types of errors include parasitic crosstalk on buses [11], Miller effects on transistors [12], charge sharing in dynamic logic [12], and supply voltage noise due to $\frac{dI}{dt}$ spikes [13].

1.2 Deep Submicron Reliability Challenges

To further heighten the importance of high-quality verification, new reliability challenges are materializing in deep submicron fabrication technologies (*i.e.*, process technologies with minimum feature sizes below $0.25\mu m$). Finer feature sizes result in an increased likelihood of noise-related faults, interference from natural radiation sources, and huge verification burdens brought on by increasingly complex designs. If designers cannot meet these new reliability challenges, they may not be able to enjoy the cost and speed advantages of these denser technologies.

Noise-Related Faults Noise related faults are the result of electrical disturbances in the logic values held in circuits and wires. As process feature sizes shrink, interconnect becomes increasingly susceptible to noise induced by other wires [11, 14]. This effect, often called *crosstalk*, is the result of increased capacitance and inductance due to densely packed wires [15]. At the same time, designs employ lower supply voltages to decrease power dissipation, resulting in even more susceptibility to noise as voltage margins are decreased.

Natural Radiation Interference There are a number of natural radiation sources that can affect the operation of electronic circuits. The two most prevalent radiation sources are gamma rays and alpha particles. Gamma rays arrive from space. While most are filtered out by the atmosphere, some occasionally reach the surface of the earth, especially at higher altitudes [16]. Alpha particles are created when atomic impurities (found in all materials) decay [17]. When these energetic particles strike a very small transistor, they can deposit or remove sufficient charge to temporarily turn the device on or off, possibly creating a logic error [18, 14]. Energetic particles strikes, sometimes called single-event radiation (SER), have been a problem for DRAM designs since the late 1970's when DRAM capacitors became sufficiently small to be affected by energetic particles [17].

It is difficult to shield against natural radiation sources. Gamma rays that reach the surface of the earth have sufficiently high momentum that they can only be stopped with thick, dense materials [16]. Alpha particles can be stopped with thin shields, but any effective shield would have to be free of atomic impurities, otherwise, the shield itself would be an additional source of natural radiation. Neither shielding approach is cost effective for most system designs. As a result, designers will likely be forced to adopt fault-tolerant design solutions to protect against SER-related upsets.

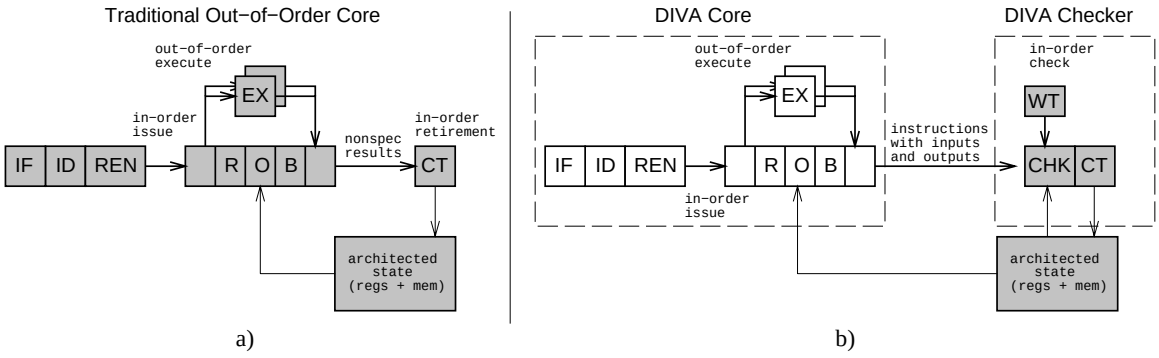


Figure 1: Dynamic Verification. Figure a) shows a traditional out-of-order processor core. Figure b) shows the core augmented with a checker stage (labeled CHK). The shaded components in each figure indicate the part of the processor that must be verified correct to ensure correct program execution.

Increased Complexity With denser feature sizes, there is an opportunity to create designs with many millions and soon billions of transistors. While many of these transistors will be invested in simple regular structures like cache and predictor arrays, others will find their way into complex components such as dynamic schedulers, new functional units, and other yet-to-be-invented gadgets. There is no shortage of testimonials from industry leaders warning that increasing complexity is perhaps the most pressing problem facing future microprocessor designs [19, 20, 21, 22]. Without improved verification techniques, future designs will likely be more costly, take longer to design, and include more undetected design errors.

1.3 Dynamic Verification

Today, most commercial microprocessor designs employ fault-avoidance techniques to ensure correct operation. Design faults are avoided by putting parts through extensive functional verification. To ensure designs are electrically robust, frequency and voltage margins are inserted into acceptable operating ranges. These techniques, however, are becoming less attractive for future designs due to increasing complexity, degraded signal integrity, and natural radiation interference.

Traditional fault-tolerance techniques could address some of the reliability challenges in future designs, but only at great cost. System-level techniques such as triple modular redundancy (TMR) [23] can detect and correct a single transient fault in the system at the expense of three times the hardware plus voting logic. Logic-level fault-tolerant solutions, such as self-checking circuits [24], often have lower cost (typically only twice as much hardware), but they are still costly and can slow critical circuits. Moreover, these approaches only address transient errors, *i.e.*, errors which manifest temporarily (such as energetic particle strikes). They cannot address design errors if the error occurs within each of the redundant components - the redundant units will simply agree to take the wrong action.

In this paper, we introduce *dynamic verification*, a novel microarchitecture-based technique that permits detection and recovery of all functional and electrical faults in the processor core, both permanent and transient. A *Dynamic Implementation Verification Architecture* (DIVA) extends the speculation mechanism of a modern microprocessor to detect errors in the computation of the processor core. As shown in Figure 1, a DIVA processor is created by splitting a traditional processor design into two parts: the deeply speculative *DIVA core* and the functionally and electrically robust *DIVA checker*. The *DIVA core* is composed of the entire microprocessor design except the retirement stage. The core fetches, decodes, and executes instructions, holding their speculative results in the re-order buffer (ROB). When instructions complete, their input operands and results are sent in program order to the *DIVA checker*. The DIVA checker contains a functional checker stage (CHK) that verifies the correctness of all core computation, permitting only correct results to pass through to the commit stage (CT) where they are written to architected storage. If any errors are detected in the core computation, the checker fixes the errant computation, flushes the processor pipeline, and then restarts the processor at the next instruction.

The DIVA checker detects and correct errors in the core processor by re-executing the non-speculative instruction stream (observable at core retirement). It does this using a simple (but complete) pipeline that leverages a stream of high-quality branch predictions, input value predictions, and cache prefetches from the core processor. Operating in the wake of the complex core processor eliminates the control and data hazards that would otherwise slow the simple checker pipeline.

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. To ensure that the processor core continues making forward progress in the event of an unrecoverable design fault, the checker is able to complete execution of the current instruction before restarting the core processor.

On the surface, it may seem superfluous to add hardware to perform a verification function that is today accomplished at design time. However, there are at least four powerful advantages of dynamic verification:

- The approach concentrates functional and electrical verification into the checker unit. As a result, the core processor has no burden of functional or electrical correctness, and no requirement of forward progress - it need only be correct often enough to meet performance goals. If the checker design is kept simple, the approach can reduce the cost and improve the overall quality of processor verification.

- Transistors outside of the checker unit can scale to smaller sizes without fear of natural radiation interference. If these transistors experience an energetic particle strike and produce incorrect results, the checker will detect and correct any errant computation.

- The fault-avoidance techniques used to produce electrically robust designs are very conservative. By leveraging a dynamic verification approach, voltage and timing mar-

gins in the core can be significantly tightened, resulting in faster and cooler implementations.

- As long as checker fault rates are kept in check, it becomes possible to simplify the processor by eliminating infrequently used functionality. For example, rarely used circuits can be eliminated to improve the speed or reduce the size of critical circuit paths.

In the remainder of this paper, we present and evaluate the DIVA checker architecture. In Section 2, we detail the architecture and operation of the DIVA checker unit. We also present arguments why the DIVA checker should be inexpensive to build and lend itself to functional and electrical verification, more so than the complex core processor it monitors. In Section 3, we present analyses of the runtime impacts of dynamic verification. Through detailed timing simulation, we examine the performance impacts of various DIVA checker architectures. We also study the effect of fault rates on core processor performance. Section 4 describes related work, and Section 5 suggests other applications of dynamic verification. Finally, Section 6 gives conclusions.

2. The DIVA Checker Architecture

The DIVA checker architecture presented in this section makes a number of important assumptions concerning the underlying microarchitecture. First, it is assumed that all architected registers and memory employ an appropriate coding technique (*e.g.*, ECC) to detect and correct any storage-related faults. As a result, any value the DIVA checker reads or writes to a register or memory will complete without error. Second, it is assumed that the record of instructions fetched by the DIVA core are correctly communicated to the DIVA checker. Once again, coding techniques can be used to detect and correct errors in this communication. (Note that it is not assumed that accesses to instruction or data storage occurred in the right order or to the correct address, the DIVA checker will verify these requirements.) Finally, it is assumed that the core and the checker share the same architected state (register and memory system). Later, we examine the implications this has on core processor performance.

2.1 The Invariants of Serial Program Semantics

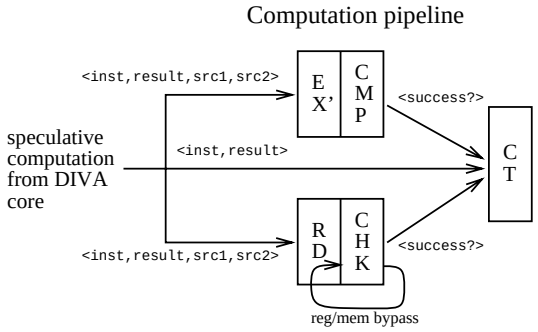
To ensure that the core processor is functioning correctly, the checker unit verifies four architectural invariants on the execution of each instruction. These architectural invariants are:

Correct Computation All operations produce a correct result given their inputs.

Correct Communication The last write of storage is visible by the next read of the same address.

Correct Control Processor control changes as per the semantics of branch instructions.

Forward Progress The processor is making progress toward completion of the next instruction to retire.



a)

stage / inst class	RD	CHK	CT
ALU/ AGEN/ BR	read src1 read src2	check src1 check src2 wait for CHKcomp	WB result or exception
LD	read addr read mem	check addr check mem	WB result or exception
ST	read addr read st data	check addr check st data	ST mem or exception

b)

Figure 2: A Dynamic Implementation Verification Architecture (DIVA). Figure a) illustrates the DIVA architecture and its interface to the core processor. Figure b) details the DIVA communication pipeline operation for each instruction class.

If each of these invariants hold for a particular instruction, it may safely retire its result. This simple approach to verification is possible because the underlying microarchitecture, although complex and a great challenge to verify directly, implements a relatively simple interface at the instruction set. In other words, the complexity in the core processor is primarily the result of architecturally invisible performance optimizations. For example, if renaming occurs, it does not affect whether or not writes to an address are visible to the next read. This property also serves to strengthen the checkers ability to detect design errors, since errors in the *specification* of the microarchitecture cannot affect the correctness of the checker.

2.2 Basic Operation

Figure 2a details the architecture of the DIVA checker. The DIVA core processor sends completed instructions in program order to the DIVA checker. With each instruction, it also sends the operands (inputs) and result (output) values as computed by the DIVA core processor. The checker verifies the instruction result using two parallel and independent verification pipelines. The *computation pipeline* verifies the integrity of all core processor computation and control. The *communication pipeline* ensures that register and memory communication between core processor instructions occurred without error. If both pipelines report correct operations (and no earlier instruction declared an error), the core computation succeeded with the correct inputs. Hence, the instruction result is correct and it can be safely retired to architected storage in the commit stage (CT) of the pipeline. In the event forward progress is lost in the core (*e.g.*, due to a deadlock or livelock), the watchdog timer will expire and restart the checker pipeline with the next instruction.

Computation Pipeline The computation pipeline verifies the integrity of all functional unit computation. In the EX' stage of the computation pipeline, the result of the instruction is re-computed. In the CMP stage, the re-computed result is compared to the functional unit result delivered by the core processor. If the two results differ, the DIVA checker raises an exception which will correct the errant core result with the value computed in the EX' stage. Control is also checked in this pipeline. Branches compute their actual target addresses, and compare this to the predicted address from the core (the result of the branch) - if they do not match, an exception is declared to fix program control.

Because instructions are delivered by the DIVA core to the computation pipeline *with pre-computed inputs and outputs*, there are no inter-instruction dependencies to slow pipeline progress. For example, if a long latency divide enters the checker followed by a dependent add, the dependent operation may start in same cycle (before completion of the divide) using the inputs values supplied by the core. As long as instructions are checked in program order, any incorrect input predictions will be detected and corrected. Core input value predictions create a tremendous amount of ILP, as a result, checker bypass datapaths are not required and pipeline control logic is trivial. The resulting checker pipeline is both simple and fast.

It may seem redundant to execute the instruction twice: once in the functional unit and again in the computation pipeline, however, there is good reason for this approach. First, the implementation of the computation pipeline can take advantage of a simpler algorithm to reduce functional unit verification costs. Second, it can be implemented with large transistors (that carry ample charge) and large timing and voltage margins, making it resistant to natural radiation interference and noise-related faults.

Communication Pipeline Figure 2b details operation of the communication pipeline for each instruction class. For the purpose of demonstration, it is assumed that the underlying architecture is a simple load/store instruction set (although this is not required). In addition, load and store operations are decomposed into two sub-operations: an address generation operation (AGEN) which adds a register and constant to produce an effective address, and a corresponding load (LD) or store (ST) primitive that accepts the effective address. This decomposition simplifies the mapping of load and store operations onto the communication pipeline.

The communication pipeline verifies that the processor core produced the correct register and memory input operands for each instruction. We observe that *at retirement, the correct inputs for an instruction reside in architected registers and memory.* By probing this state just before retirement, it is possible to check if the core processor produced the correct register and memory inputs. This simple check works independent of the underlying mechanism used to implement communication in the core processor pipeline, *e.g.*, register renaming, dependence speculation, or dynamic scheduling will not affect this invariant.

As shown in Figure 2b, the communication pipeline re-executes all communication in program order just prior to instruction retirement. In the RD stage of the communication pipeline, the register and memory operands of instructions are read from architected storage. In the CHK stage of the pipeline, these values are compared to the input values delivered by the core processor. If the operands delivered by the core processor match those read by the RD stage, the processor core successfully implemented instruction communication and processing may continue. Otherwise, the DIVA checker raises a register or memory

DIVA Exception	Exception Priority	Recovery Mechanism	How the Exception is Corrected by DIVA
Watchdog Timer Expiration	0 (highest)	1) reset DIVA pipes with next inst 2) restart DIVA pipes @ PC 3) flush core, restart @ NPC	Watchdog exception jumpstarts DIVA checker and core at next instruction
CHKcomm (register value)	1	1) reset DIVA pipes with correct register input 2) restart DIVA pipes @ PC 3) flush core, restart @ NPC	CHKcomm RD stage register value (always correct) is injected into DIVA verification
CHKcomm (memory value)	2	1) reset DIVA pipes with correct memory input 2) restart DIVA pipes @ PC 3) flush core, restart @ NPC	CHKcomm RD stage memory value (always correct) is injected into DIVA verification
CHKcomp	3 (lowest)	1) reset DIVA pipes with correct register result 2) restart DIVA pipes @ PC 3) flush core, restart @ NPC	CHKcomp EX' result (correct if no other exceptions) is injected into DIVA verification

Figure 3: Fault Handling in the DIVA Checker.

exception which will restore the correct input operand with the operand value read in the RD stage. A single bypass exists across the CHK stage to handle the case of an instruction checking an input written by the immediately previous instruction. Since this value is not visible until the CT stage of the pipeline (when the value is written to architected state), a single bypass is provided to eliminate any stalls.

It may seem superfluous to re-execute all storage operations in the communication pipeline, especially given the assumption that all storage is protected from faults using coding techniques such as ECC. However, coding techniques are insufficient to detect all communication errors. While coding can detect storage values that were damaged by transient errors, it cannot detect failed or misdirected communication in the processor core. For example, if the register renamer points an operand to an incorrect physical storage location, or if the store forward buffers miss a communication through aliased virtual memory, coding techniques will not detect these errors. These errors are, however, detected by the communication pipeline as it re-executes register and memory communication. Instruction fetch accesses, on the other hand, do not need to be re-executed because the order of accesses to this storage is not important (save self-modifying code writes).

2.3 Fault Handling

In the event a fault is detected in the DIVA core computation, the DIVA checker will raise an exception to correct the errant condition and restart the processor. Figure 3 shows the DIVA exceptions that can occur, their priority, and the specific method used to recover machine state.

Exceptions are handled in program order at the commit (CT) stage of the pipeline. If an instruction declares multiple exceptions in the same cycle, the exception with the highest priority is always handled first. When any exception is taken, the DIVA checker fixes the errant instruction with the correct value (returned by either checker pipelines), flushes the processor pipeline, and then restarts the DIVA checker and processor core.

Program	Input	# instr fwd (M)	# instr exec (M)	% ld exe	% st exe	Base CPI	RUU Occ	Mem Util	BP Acc
compress	ref.in	0	93	26.7	9.4	0.60	84.2	0.41	90.2
GCC	1stmt.i	100	100	24.6	11.5	0.64	25.5	0.32	85.4
go	2stone9.in	100	100	30.7	8.2	0.61	23.3	0.42	76.1
ijpeg	vigo.ppm	100	100	18.5	5.6	0.38	132.0	0.39	88.6
li	boyer.lsp	100	100	25.8	15.1	0.47	52.9	0.44	93.1
perl	scrabble.pl	100	100	22.7	12.2	0.48	55.0	0.39	93.7
hydro2D	hydro2D.in	100	100	20.7	8.7	0.46	106.4	0.37	96.3
tomcatv	tomcatv.in	100	100	20.4	8.7	0.42	50.2	0.40	95.6
turbo3D	turbo3D.in	100	100	23.6	16.2	0.38	149.7	0.42	94.9

Table 1: Program statistics for the baseline architecture.

It is crucial that the DIVA checker be able to correct whatever condition resulted in a DIVA exception. If the DIVA checker were not able to correct a particular exception condition, it would not be able to guarantee program forward progress in the presence of a permanent core fault (*e.g.*, a design error or stuck-at fault). As shown in Figure 3, all exception conditions are corrected. In fact, the DIVA checker is sufficiently robust that it can completely take over execution of the program in the event of a total core failure. However, its performance would be very poor, especially if it had to rely on the watchdog timer to expire before starting each instruction.

There is slightly different handling of the watchdog timer exception. When the watchdog timer expires, the DIVA checker fetches the next instruction to execute and injects it into the DIVA pipe with zero value inputs and outputs. The checker then restarts the processor. The checker pipelines will correct these operands and results as incorrect values are detected, eventually completing execution of the stalled instruction.

2.4 Working Examples

Figure 4 shows two examples of the DIVA checker in operation. In Figure 4a, an AGEN operation produces an incorrect result that it forwards to a LD operation. The computation pipeline detects the incorrect result in the CMP stage and then declares an exception which corrects the result of the AGEN operation, allowing it to retire three cycles later.

Figure 4b shows the operation of the DIVA checker in the event of a catastrophic core processor failure. In this example the core is not attempting to retire instructions, thus the DIVA checker must completely execute each instruction. The example starts out with a watchdog timer reset that forces insertion of the next instruction with zero value inputs and outputs. The instruction first detects in the communication pipeline that its inputs are incorrect which results in a register value exception that fixes the inputs. Next, the computation pipeline detects that the result is incorrect which declares an exception that fixes the result. Finally, the instruction completes without an exception and retires its results to the architected register file.

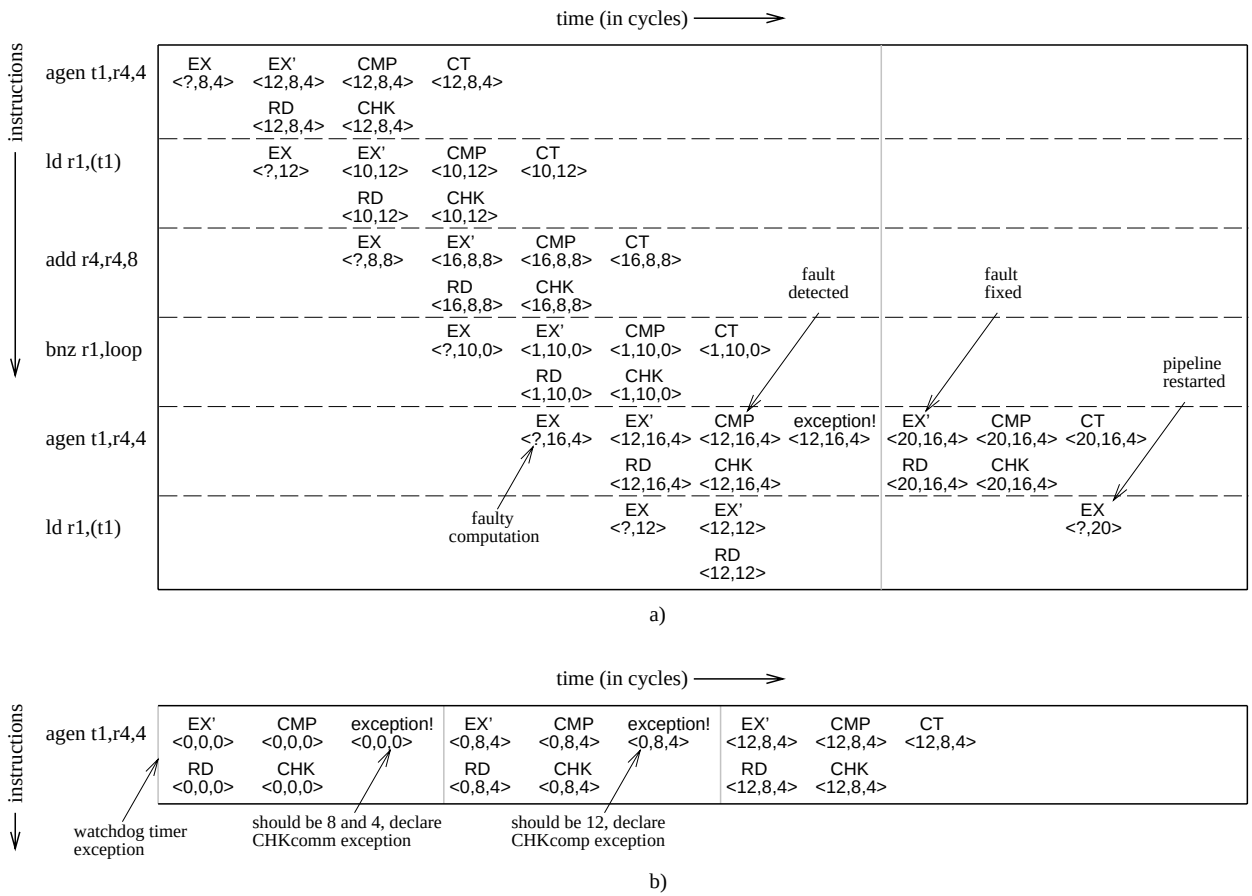


Figure 4: Example Operation of the DIVA checker. Two working pipeline examples are shown in Figures a) and b). In the pipeline diagrams, program execution runs from top to bottom, the instruction executed is shown to the left of the pipeline. Time runs from left to right; instructions list which pipeline stage they are in for each cycle they are active. Below the pipeline stage designators are listed the one output and two input values for each instruction. The vertical bars represent declarations of DIVA checker exceptions.

2.5 Verification of the DIVA Checker

Paramount to the success of dynamic verification is a functionally correct and electrically robust DIVA checker implementation. It had better work correctly all the time, otherwise, it may impair correct operation of the processor core. Moreover, the cost of DIVA checker verification should be lower than the cost of verifying a traditional processor design, otherwise, there is no overall gain to the employing dynamic verification. It is difficult to quantitatively assess the ease (or difficulty) in building a correct DIVA checker in a paper design such as this. To accurately assess these costs would require the construction of a real DIVA checker in VLSI. In lieu of this level of detail, we describe the attributes of the DIVA checker design that we feel will lend the approach to high-quality and low-cost functional and electrical verification.

Simple: The DIVA checker is inherently simpler than a traditional processor core. It contains only the mechanisms necessary to check the function of the program, and it lacks all of the mechanisms used to speed computation, *e.g.*, predictors, renamers, dynamic schedulers, etc. In addition, the pre-computed inputs and outputs from the core processor eliminate the inter-instruction dependencies and stall conditions that complicate traditional high-performance pipeline designs.

Latency-Insensitive: With sufficient buffering of speculative core results, the latency of the DIVA checker will not impact core processor performance. As a result, wide and deeply pipelined implementations are possible. These designs will permit checker implementations with large timing margins and large (and slow) transistors, affording the checker high resistance to transient faults and natural radiation interference. Since there are few dependencies between instructions, widening or lengthening the DIVA pipeline is quite straightforward.

Scalable: The DIVA checker design is more reusable than traditional processor cores, making it possible to leverage correctness established in previous designs. Since the checker sits at retirement, new designs need only scale with the retirement bandwidth of the new core it is checking. Retirement bandwidth scales very slowly from generation to generation, any additional bandwidth requirements can be accommodated by simply lengthening or widening the DIVA checker pipelines. Moreover, the design of the checker is independent of the core microarchitecture (as it checks architectural invariants), as a result, its design can be completely decoupled from the core design.

In addition to these attributes, we are currently investigating formal verification of the DIVA checker. The DIVA checker resembles a simple in-order processor with little microarchitectural state and few inter-instruction dependencies – properties that simplify formal verification [8, 9]. We believe the DIVA checker will also lend itself to formal verification, making it possible to formally verify large complex microarchitectures by only verifying the correctness of the DIVA checker.

3. Experimental Evaluation

In this section, we examine the impact of dynamic verification on processor core performance. Core slowdowns are measured, using detailed timing simulation for DIVA checkers with varied resource configurations, checker latency, and fault rates.

3.1 Methodology

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [25], 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. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results for nine of the SPEC95 benchmarks [26]. All programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and Fortran compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table 1 shows the data set we used in gathering results for each program, the number of instructions that were executed (fast forwarded) before actual simulation began, and the number of instructions simulated for each program (up to 100 million). Also shown are the percent of dynamic instructions that were loads and stores, the baseline machine CPI, the average number of entries in the instruction window (RUU), the fraction of time the memory ports were in use, and the branch predictor accuracy for each program.

3.2 Baseline Architecture

Our baseline simulation configuration models a future generation out-of-order processor microarchitecture. We've selected the parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch and issue up to 4 instructions per cycle. It has a 256 entry re-order buffer with a 64 entry load/store buffer. Loads can only execute when all prior store addresses are known. In addition, all stores are issued in program order with respect to prior stores. There is an 8 cycle minimum branch misprediction penalty. The processor has 4 integer ALU units, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The processor we simulated has a 32k 2-way set-associative instruction and data caches. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with 2 ports. The data cache access latency is one cycle (for a total load latency of two cycles). There is a unified second-level 512k 4-way set-associative cache with 32 byte blocks, with a 10 cycle cache hit latency. If there is a second-level cache miss it takes a total of 60 cycles to make the round trip access to main memory. We model the bus latency to main memory with a 10 cycle bus occupancy per request. There is a 32 entry 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

3.3 DIVA Checker Baseline Architecture

The DIVA checker in all experiments is a four instruction wide pipeline that instructions enter when they have completed and are the oldest instruction in the machine that has not yet entered the DIVA checker pipeline. Instructions are processed in-order, any instruction

that stalls causes later instructions to also stall. In the baseline configuration, the computation pipeline latency is one cycle longer than the functional unit it checks (for the result comparison). It is assumed that there is a computation pipeline for each of the functional units, as a result, there are no structural hazards introduced. The baseline communication pipeline takes two cycles (for RD and CHK) unless there are structural hazards in accessing register file and cache ports. In the baseline checker architecture, the RD stage competes with the core processor for four architected register file ports and two cache ports, with priority given to the DIVA checker accesses. The core processor only accesses the architected register file when an operand is not found in the physical register file (*i.e.*, it is not in flight). Re-order buffer entries are not deallocated until instructions exit the commit (CT) stage of the pipeline, after the DIVA checker verifies the operation. The watchdog timer countdown is reset to 60 cycles (the round trip latency to memory) whenever an instruction commits.

3.4 DIVA Checker Impact on Core Processor Performance

In Figure 5, we show the impact of the DIVA checker on core processor performance. All performance numbers are normalized to the CPI of an unchecked core processor. Results are shown with varied register and memory storage bandwidth. Experiment (*+0*) has no extra register file or cache ports (compared to the baseline unchecked microarchitecture). Without dedicated ports into the architected register file and data cache, the DIVA checker must compete for bandwidth with the core processor. This competition can create structural hazards which can slow core processing. Experiment are also shown with with 4 extra register file ports dedicated to the DIVA checker (*+R*), with one extra dedicated memory port (*+M*), and with 4 extra register file ports and one extra memory port (*+R+M*).

Even without extra storage bandwidth, *i.e.*, experiment (*+0*), the cost of employing the DIVA checker is quite low. Average program slowdown was only 3%. In general, there was a high correlation between pipeline utilization (*e.g.*, branch prediction accuracy, RUU occupancy, and memory port utilization) and slowdown. When the processor pipeline is efficiently utilized, any additional DIVA checker register file and cache accesses create structural hazards that slow core processing. *Turbo3D* had the largest slowdown of 14% without additional resources. This benchmark is highly efficient, it has high branch predictor accuracy, high RUU occupancy and high memory utilization. GCC and and GO, on the other hand, have poor branch prediction and thus poor pipeline utilization; additional DIVA checker resource usage has little impact on the core processor performance of these programs.

By increasing the bandwidth to the register file and caches, we can reduce the impact of structural hazards on core processor performance. Experiment (*+R*) adds four more read ports to the architected register file for use by the DIVA checker communication pipeline. These additional register ports eliminate most structural hazards into the architected register file. This change had little impact on overall performance (at most an improvement of 0.9% for *Hydro2D*). Since many of the register accesses are satisfied by the physical register file (which has its own access ports), there appears to be sufficient bandwidth left into the architected register file for DIVA checker accesses.

Experiment (*+M*) adds one more read port to the data cache for use by the DIVA checker communication pipeline. This additional cache port eliminates nearly all structural

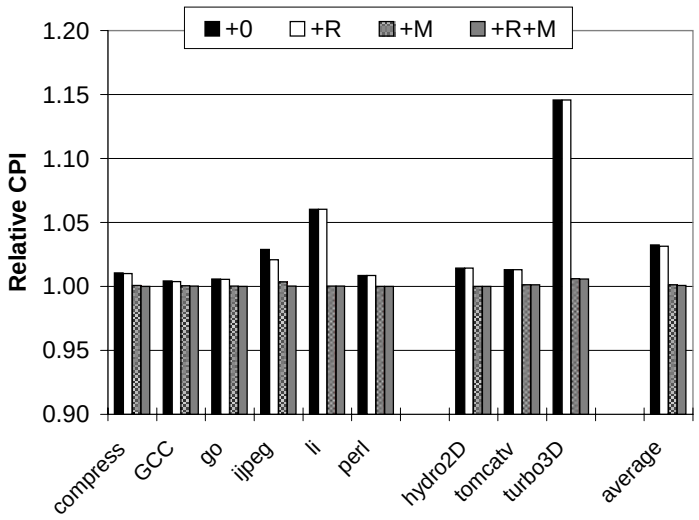


Figure 5: DIVA Checker Impact on Core Processor Performance for Varied Register File and Cache Bandwidth. All performance numbers are normalized to the CPI of an unchecked core processor. Results are shown with no extra register file or cache ports (*+0*), with 4 extra register file ports (*+R*), with one extra memory port (*+M*), and with 4 extra register file ports and one extra memory port (*+R+M*).

hazards into the data cache. Adding this port has a noticeable impact on core processor performance. Most core processor performance impacts are eliminated and overall slowdown drops to only 0.1%. Finally, in experiment (*+R+M*), four register ports and two memory ports are added for DIVA checker use. With an additional memory port, the extra register file ports provide little benefit, and overall slowdown drops to 0.03%.

In addition to structural hazards on architected storage resources, we observed that retirement delays could slow the core processor. Delays in the retirement of an instruction increase pressure on core processor speculative storage, *e.g.*, re-order buffer (ROB) and load/store queue (LSQ) entries. If these structures become full, they can stall the decode and issue of instructions in the core processor, resulting in reduced ILP and decreased program performance. During normal checker pipeline operation, checking only extends the latency of retirement by a few cycles. But as evidenced by the small slowdowns (especially when storage hazards were eliminated), these effects were minimal. We believe that while increased speculative storage pressure does stall core progress, it only stalls the issue of instructions that would likely not retire. In other words, the probability that instructions that would fill the speculative state resources would retire is very low due to the large degree of speculation required to reach these instructions. As a result, increased pressure on speculative state has little effect on overall performance.

Data cache misses in the checker pipeline can greatly extend the latency of instruction retirement as the checker pipeline completely blocks on data cache misses. These misses will quickly stop the progress of the core processor pipeline, however, we observed virtually no data cache misses in the checker pipeline. Since the checker follows in the wake of the

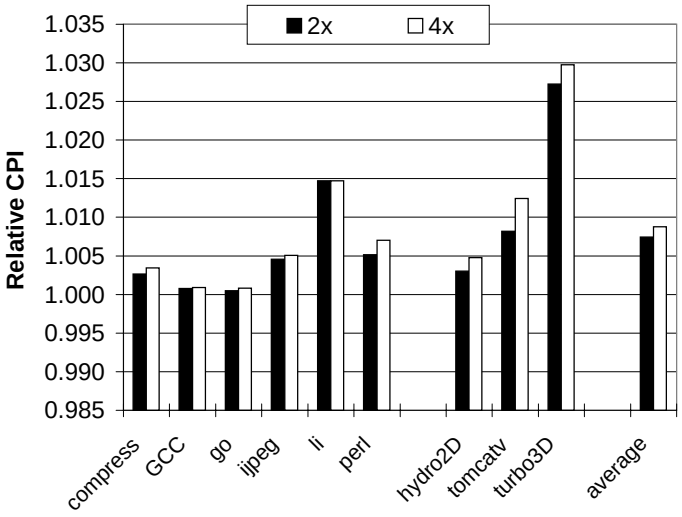


Figure 6: DIVA Checker Impact on Core Processor Performance with Varied DIVA Checker Latency. All performance numbers are normalized to the CPI of the baseline DIVA checker configuration (*+0*). Results are shown for a checker with two times the latency (*2x*), and a checker with four times the latency (*4x*).

core processor, the core processor serves a prefetch mechanism for the checker pipeline, eliminating nearly all data cache misses. We are currently developing refined DIVA checker designs that do not delay the release of speculative storage in the core processor pipeline, virtually eliminating core processor stalls due to instruction checking.

3.5 Effects of Increased DIVA Checker Latency

In an effort to build an electrically robust implementation of the DIVA checker, it may be necessary to construct it with large timing margins and large transistors (to resist noise and tolerate natural radiation interference, respectively). The most straightforward approach to achieve this is to deeply pipeline the DIVA checker. Deeply pipelining the DIVA checker, however, will increase its latency which can delay retirement of instructions. These delays may cause congestion in the instruction window, effectively reducing the instruction window size and amount of ILP that can be exploited. Figure 6 shows the impact of DIVA checker latency of core processor performance. All performance numbers are normalized to the CPI of a DIVA checker configuration with no extra register file or cache port resources, *i.e.*, experiment (*+0*) from Figure 5.[3] Results are shown for a checker with two times as many stages (*2x*), and a checker with four times as many stages (*4x*)

Overall, the impact of checker latency on core performance is quite low. At two times the latency slowdowns increase by 0.7%, and with four times the checker latency, slowdowns only

3. We compare to this configuration because it was the worst performing DIVA checker configuration, thus it will have the most instruction window congestion to start with and be most sensitive to DIVA checker latency.

increase by 0.8%. Even the programs with high branch predictor accuracy. *e.g.*, *Hydro2D*, cannot get sufficiently far ahead of the execution core to keep the instruction window full, as a result, adding extra checker latency has little impact on core performance.

3.6 Effects of DIVA Checker Exceptions

In the event that the DIVA checker detects a failed computation or communication, it will declare a DIVA exception and reset the checker and processor core. The performance penalty for exception handling is quite large, at least 8 cycles for the experiments simulated, more if the faulty instruction takes a while to reach retirement. To gauge the performance impact of DIVA exceptions, we ran experiments with random exceptions injected at random times but with a fixed exception interval (in core processor cycles). The results are shown in Figure 7. All performance numbers are normalized to the CPI of the baseline DIVA checker configuration (*+0*). Results are shown on a log scale for exception rates of one per one million core processor cycles (*1M*), one per 1000 processor cycles (*1k*), and one every processor cycle (*1*). The experiment labeled (*Core Lock*) examined the performance impacts of a catastrophic core failure in which the core is no longer attempting to commit instructions, consequently, instructions are not fetched until the watchdog timer counter expires (once every 60 cycles).

At an exception rate of one exception every one million cycles (an average of one exception every 2 msec on a 500 MHz processor), there was virtually no impact on core processor performance. At intervals of 1000 cycles (an average of one exception every 2 usec on a 500 MHz processor), impacts were higher but still small, with an overall slowdown of only 2.6%. With an exception every cycle (as would be the case for a catastrophic core processor failure), performance impacts rise dramatically. Overall, performance is $\frac{1}{14}$th normal speed. In this experiment, the DIVA checker is completely executing the program. When the core processor is locked up, performance is quite low, on average $\frac{1}{120}$th the performance of the unchecked core. In Section 5, we suggest a simple change to detect this case and provide more graceful degradation in processor performance.

It should be possible to keep fault rates well below the point where they have any perceivable effect on performance. The rate of faults caused by design errors should be quite low as the core will undergo some level of verification to excise frequently occurring design errors. The remaining design errors will be those that occur infrequently and thus are difficult to find and fix. These infrequent errors should not create performance concerns. Transient faults such as SER have been shown to be quite infrequent as well. For example, SER fault rates in adverse conditions (high altitudes) were measured at rates of one every few hours [18], nothing close to the rates that would be required to affect core processor performance.

3.7 Discussion

Core processor slowdowns (due to structural hazards, checker latency, and exception handling) are not the only costs associated with the DIVA checker, other costs include silicon area, power consumption, and more. While it is difficult to gauge these costs without building an actual DIVA checker implementation, we feel overall DIVA checker costs should be low for at least two reasons.

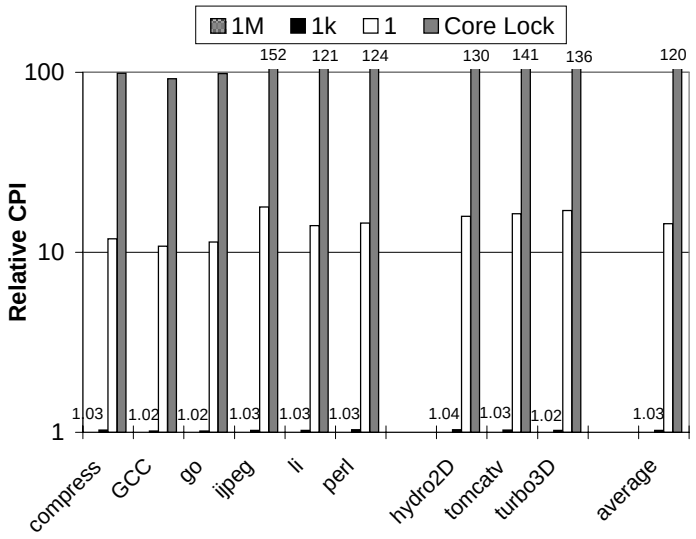


Figure 7: Performance Impact of Varied Exception Rates. All performance numbers are normalized to the CPI of the baseline DIVA checker configuration (*+0*). Results are shown on a log scale for exception rates of one per one million core processor cycles (*1M*), 1000 processor cycles (*1k*), and every processor cycle (*1*). The experiment labeled (*Core Lock*) examines the performance of a catastrophic core failure in which the core is no longer attempting to commit instructions. Bar values are shown for the (*Core Lock*) and (*1k*) experiments.

First, the DIVA checker only replicates the part of core pertaining to actual function. All core processing structures related to program performance (*e.g.*, predictors and schedulers) need not be represented in the DIVA checker. In addition, the DIVA checker design is a very simple pipeline. It uses in-order instruction processing and has few inter-instruction dependencies. These advantages make the DIVA checker significantly simpler than the core processor pipeline, and they should serve to reduce its size and cost.

Second, the computation pipeline can be implemented with simple, area-efficient algorithms. For example, if the core uses a carry-select carry-lookahead adder for fast addition, the DIVA checker can instead use a pipelined ripple carry adder to reduce implementation costs. The pipeline will be slower, but as shown earlier, core performance is mostly insensitive to DIVA checker latency, making this a good tradeoff.

Ultimately, a more quantitative assessment of DIVA checker area and power costs will have to be made, first through circuit level simulation and later through an actual silicon implementation. We are currently working toward these goals.

4. Related Work

The idea of dynamic verification at retirement was inspired by Breach's Multiscalar processor simulator [27]. During the programming and verification of this very complex simulator, a functional verification stage was added at retirement. The approach was sufficiently robust that it could mask functional bugs that were introduced during the simulator's development

permitting its use for performance analysis before it was fully debugged. The DIVA checker works in precisely the same manner, but for a hardware system.

Rotenberg's AR-SMT processor [28] employs a time-redundant execution technique that permits an SMT processor to tolerate some transient errors. We borrow on this work; the DIVA checker leverages the idea of using an unreliable processor's results to speed the execution of instruction checking. We improve upon Rotenberg's work in a number of ways. Rotenberg's approach checks all aspects of execution, include program function and the mechanisms used to optimize program performance. The DIVA checker, on the other hand, only checks program function, permitting a simpler checker implementation. While Rotenberg's approach only detects some transient errors, a functionally correct and electrically robust DIVA checker can recover all permanent and transient core processor faults.

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 [29] 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 [30]. 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 DIVA 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 2 times as much hardware in the core processor. Additionally, the DIVA 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 [31] 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.

5. Other DIVA Applications

The fault-tolerance create by dynamic verification could allow designers to revisit many of the fundamental assumptions of computer design - assumptions grounded in a fault-avoidance design methodology. We believe that a transition to a fault-tolerant design style would break many of these venerable assumptions - creating significant opportunity at all levels of the design. In this section, we suggest a number of possible optimizations.

5.1 Beta-Release Hardware

Today, when parts are fully verified and released to the field, the process of verification (for the most part) is over. If customers find bugs, it may be necessary to implement expensive recalls of costly components. With dynamic verification, it becomes possible to safely release "beta" versions of the hardware to customers in the field (once the checker is fully verified). This early release of hardware will enable widespread in-field electrical verification, concurrent with initial deployment of the part. If any core processor errors occur during this testing phase, the checker will ensure that they only manifest as performance divots. As problems are identified and fixed, steppings of the hardware can be made without necessitating a replacement of earlier hardware – the new release of the part will simply be slightly faster because it will experience fewer core processor design errors. To facilitate this process, effective system monitoring mechanisms could be developed that better identify the source of core design errors, and communicate this information back to the manufacturer.

5.2 Scalable Low-Cost SER Protection

Single event radiation (SER) poses a significant threat to the reliability of deep submicron logic implementations. Dynamic verification provides natural protection from these problems, since SER-induced faults in the core processor will manifest as computation errors that are fixed by the checker. To prevent SER from affecting the checker, it can be made with larger transistor with ample charge to tolerate strikes, thus providing 100% coverage for SER by simply addressing the problem in the checker. As fabrication technology continue to scale to smaller feature sizes, it eventually becomes more area efficient to implement two copies of the checker logic. Since SER upsets are temporally and spatially sparse, it will be highly improbably that both checkers could be affected by SER in the same or subsequent cycles. Accordingly, if the checkers disagree on the correctness of an instruction one of the checkers has experienced an SER upset. In this event, the machine can be restarted at the same instruction and the checkers should once again agree as to the correctness of the instruction result. This approach is a completely scalable low-cost and low-impact (to the overall design) solution, and it provides complete coverage of all SER-related faults.

5.3 Self-Tuned Microprocessor Systems

The techniques used to provide reliability in VLSI logic implementations are very conservative. Systems are designed with frequency and voltage margins that ensure reliable operation in even the most adverse environments (*e.g.*, high temperature, high clock skew, slow transistors). This is one of the reasons why hobbyists can overclock [32] production parts for more performance. It should be possible to use the DIVA checker to reclaim much of the power and performance consumed by operating margins.

In a *self-tuned system* [33], clock frequency and voltage levels are tuned to the system operating environment, *e.g.*, temperature. The approach minimizes timing and voltage margins which can improve performance and reduce power consumption. Using the DIVA checker, a self-tuned system could be constructed by introducing a voltage and frequency control system into the processor, as shown in Figure 8. The control system decreases volt-

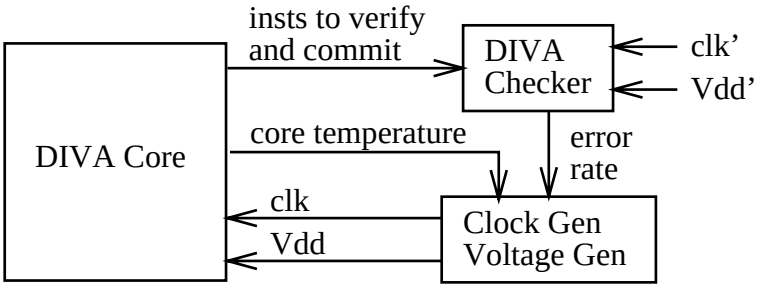


Figure 8: A Self-Tuned Microprocessor System.

age and/or increases frequency while monitoring system temperature and error rates until the desired system performance-power characteristics are attained. If the control system over steps the bounds of correct operation in the core, the DIVA checker will correct the error, reset the core processor, and notify the control system. To ensure correct operation of the DIVA checker, it is sourced by a fixed voltage and frequency that ensures reliable operation under all operating conditions (or conversely, it is design to operate reliably under the varied frequencies and voltages applied to the core processor).

5.4 Complexity-Effective Microarchitecture Designs

In the work of Palacharla *et. al.* [34], it was shown how reducing the complexity of a design could improve its performance by shortening critical circuit paths. We could leverage the checker to further reduce complexity in the core processor by eliminating any complexity that did not directly lead to performance improvements. Since the checker covers the complete semantics of the ISA, we can delete any core functionality to optimize performance, area, or design convenience. As long as that functionality is infrequently exercised (in which case the checker will detect an error and reset the pipelines), it will not have a significant impact on performance. This approach provides designers the option to *eliminate the uncommon case*. Consider how this might be used to streamline the design of of the load/store queue (LSQ):

- eliminate infrequently used support for partial forwards through memory
- eliminate rarely used support for store forwarding through virtual address synonyms
- eliminate the address check on the infrequently changing upper bits of virtual addresses

Taking this approach to its extreme, it becomes possible to design two machines: the simple checker pipeline which covers the entire functionality of the machine, and a de-featured core that implements a performance-oriented subset of the ISA. For an iA32 machine, the checker would implement all instructions, and the core would implement only the instructions and their semantics that were expected to achieve good performance, *e.g.*, register-register instructions, aligned memory accesses, and no partial register or memory definitions.

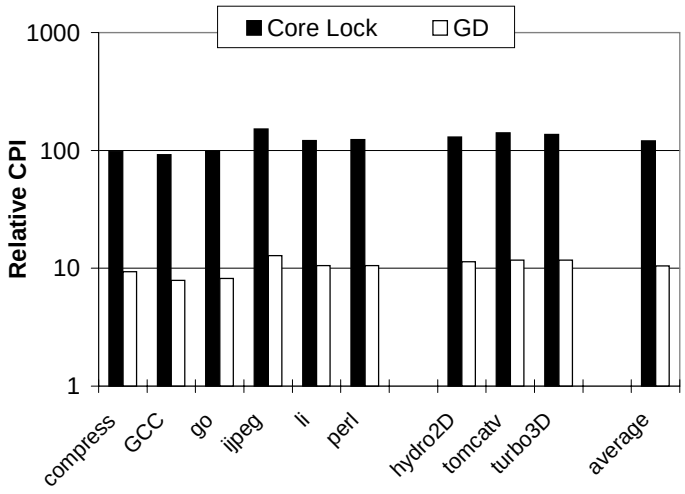


Figure 9: Graceful Degradation in the Presence of a Catastrophic Core Processor Failure. All performance numbers are normalized to the CPI of an unchecked core processor. Results are shown on a log scale for the baseline DIVA checker with a locked core (*Core Lock*), and for a DIVA checker with graceful degradation (*GD*).

5.5 Highly Available Microprocessor Designs

As shown earlier, if the core processor completely fails and no longer attempts to retire instructions, processor performance will be severely impacted. Without any instruction retiring, the DIVA checker will have to wait for the watchdog timer to timeout before an instruction can be retired and the next instruction fetched. As a result, one instruction will complete for each interval of the watchdog timer, one per 60 clock cycles in the experiments presented.

It would be relatively straightforward to detect the case where the core processor was no longer contributing to the program execution. For example, a counter could record the number of watchdog timer exceptions with no intervening instruction retirements. If a suitable threshold was reached, it could be assumed that the core processor has failed and the DIVA checker could take over execution of the program. A simple change to the DIVA design would permit it to fetch and execute instruction much quicker, simply by permitting it to execute the next instruction without having to wait for the watchdog timer to expire.

As shown in experiment (*GD*) in Figure 9, this graceful degradation approach improves processor performance considerably in the event of a catastrophic core processor failure. Program performance degrades to only $\frac{1}{10}$th the performance of the original working core processor. This is a marked improvement over the (*Core Lock*) experiment, in which the core processor locks and instructions proceed at watchdog timer exceptions, an average of $\frac{1}{120}$th the speed of the working core processor. The graceful degradation mode also outperforms a core processor that is still retiring instructions but always incorrect results, *i.e.*, experiment (*1*) from Figure 7, with an overall performance of $\frac{1}{14}$th the speed of the working core. In

this case, the DIVA checker benefits from not having to wait for the core processor pipeline to first execute the instruction incorrectly.

In addition, the availability of a system using the DIVA checker could be further improved by replicating the DIVA checker to detect errors within its own circuitry, or by applying triple modular redundancy (TMR) [23] to detect and correct permanent faults in the DIVA checker.

5.6 Dynamic Simulator Verification

Not only is dynamic verification an effective technique to lower the cost of hardware verification, it is also an effective technique to lower the cost of simulator software verification. A detailed execution-driven simulator that incorporates dynamic verification (*i.e.*, a functional checker at retirement) will permit architects to quickly explore design tradeoffs without impairing the correctness of the microarchitectural simulator. The simulator can report a coverage metric indicating what fraction of the time the simulator retired correct results, giving the architect a clear indication of the accuracy of their modifications.

6. Conclusions

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. Concerns for reliability grow in deep submicron fabrication technologies due to increased noise-related failure mechanisms, natural radiation interference, and more challenging verification due to increased design complexity.

To counter these reliability challenges, we introduced dynamic verification, a technique that adds a functional checker to the retirement phase of a processor pipeline. The functional checker ensures that all core processor computation is correct, and if not, the checker fixes the errant computation, and restarts the core processor using the processor's speculation recovery mechanism. Dynamic verification focuses the verification effort into the checker unit, whose simple and flexible design lends itself to functional and electrical verification. Using this approach, the core processor carries no burden of correctness or any requirement for forward progress. The DIVA checker architecture was presented as a checker design optimized for simplicity and low cost.

We showed through detailed timing simulation that the simple checker is also very fast. The design achieves high throughput because the core processor eliminates most of the control and data hazards that might otherwise slow its progress. Control hazards are completely resolved in the fault-tolerant core processor; the checker need only verify that the program counter is updated as per branch results. Data hazards, caused by long latency operations and long latency communications, are virtually eliminated. Long latency operations execute independent of each other using the high-quality input value predictions delivered by the core processor. And long latency communication delays (the result of data cache misses) are virtually non-existent as the core processor serves as a prefetcher for the checker, warming up its caches in advance of instruction checks.

Overall, we have found that even resource-frugal checker designs have little impact on core processor performance. For SPEC95, overall simulated slowdown for a modern pipeline was less than 3%, with most of the slowdown attributed to floating point codes that made very efficient use of all pipeline resources. With the addition of a single memory port for use by the DIVA checker, slowdowns were negligible. Also, increased DIVA checker latency had only a small impact on core processor performance. The DIVA checker can tolerate very high fault rates without significantly impacting processor core performance. At rates of one fault per 1000 processor cycles, the core processor only slowed by 2.6%; at intervals of 1M cycles, performance impacts were negligible.

Finally, other applications of dynamic verification were proposed. A self-tuned clock and voltage scheme was proposed in which dynamic verification is used to reclaim frequency and voltage margins. A highly available checker design was also proposed. The design provides more graceful degradation in system performance in the event of a total core processor failure. Applications to improve the quality and complexity of future designs were also suggested.

We feel that dynamic verification holds significant promise as a means to address the cost and quality of verification for future microprocessors, while at the same time creating opportunities for faster, cooler, and simpler designs. The next step in this work is to better quantify the benefits and costs of dynamic verification, refine our initial design to further decrease core processor performance impacts, and further develop applications of the fault tolerant processor core.

References

[1] "Statistical analysis of floating point flaw in the Pentium processor." Intel Corporation, Nov. 1994.

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

[3] A. Wolfe, "For Intel, it's a case of FPU all over again." EE Times, May 1997.

[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 verification 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 verification," in *Proceedings of the 35nd ACM/IEEE Design Automation Conference*, pp. 1–6, June 1998.

[7] W. Hunt, "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, pp. 429–460, Dec. 1989.

[8] J. Burch and D. Dill, "Automatic verification of pipelined microprocessors control," *Computer Aided Verification*, pp. 68–80, 1994.

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

[10] J. Bockhaus, R. Bhatia, M. Ramsey, J. Butler, and D. Ljung, "Electrical verification of the HP PA-8000 processor." Hewlett-Packard Journal, Aug. 1997.

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

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

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

[14] P. Rubinfeld, "Managing problems at high speed," *IEEE Computer*, pp. 47–48, Jan. 1998.

[15] R. Anglada and A. Rubio, "An approach to crosstalk effect analyses and avoidance techniques in digital CMOS VLSI circuits," *International Journal of Electronics*, vol. 6, no. 5, pp. 9–17, 1988.

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

[17] T. May and M. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electronic Devices*, vol. 26, no. 2, 1979.

[18] 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.

[19] P. Bose and T. Conte, "Performance analysis and its impact on design," *IEEE Computer*, vol. 31, pp. 41–49, May 1998.

[20] M. Tremblay, "Increasing work, pushing the clock," *IEEE Computer*, pp. 47–48, Jan. 1998.

[21] G. Grohoski, "Reining in complexity," *IEEE Computer*, pp. 47–48, Jan. 1998.

[22] B. Colwell, "Maintaining a leading position," *IEEE Computer*, pp. 47–48, Jan. 1998.

[23] D. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

[24] W. J, *Self-Checking Circuits and Applications*. New York: North–Holland, 1978.

[25] 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.

[26] "SPEC newsletter," Fairfax, Virginia, Sept. 1995.

[27] S. Breach, *Design and Evaluation of a Multiscalar Processor*. PhD thesis, University of Wisconsin, Madison, 1999.

[28] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the 29th Fault-Tolerant Computing Symposium*, June 1999.

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

[30] 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.

[31] 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.

[32] B. Machrone, "You too can be an overclocker," *PC Magazine*, 1999.

[33] T. Kehl, "Hardware self-tuning and circuit performance monitoring," in *Proceedings of International Conference on Computer Design*, 1993.

[34] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.