Efficient Checker Processor Design

Saugata Chatterjee, Chris Weaver, and Todd Austin
Electrical Engineering and Computer Science Department
University of Michigan
{saugatac,chriswea,austin}@eecs.umich.edu

Abstract

The design and implementation of a modern microprocessor creates 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. In our previous work, we proposed a solution to these problems by adding a simple, easily verifiable checker processor at pipeline retirement. Performance analyses of our initial design were promising, overall slowdowns due to checker processor hazards were less than 3%. However, slowdowns for some outlier programs were larger.

In this paper, we examine closely the operation of the checker processor. We identify the specific reasons why the initial design works well for some programs, but slows others. Our analyses suggest a variety of improvements to the checker processor storage system. Through the addition of a 4k checker cache and eight entry store queue, our optimized design eliminates virtually all core processor slowdowns. Moreover, we develop insights into why the optimized checker processor performs well, insights that suggest it should perform well for any program.

1. Introduction

The enormous complexity of modern microprocessor design presents significant challenges in the verification of these systems. Architects and designers must ensure that designs operate correctly for all possible programs, and they must ensure that this correct functionality is maintained for all (including adverse) operating conditions. If they fail to meet these challenges, the repercussions can destroy profit margins, products, and even companies. In the worse case, failure to meet these challenges could even result in loss of life.

To avoid releasing faulty parts, designers spend considerable effort on functional and electrical verification. Unfortunately, the complexity of modern microprocessors makes this verification process both incomplete and imprecise. The test spaces of stateful design are simply too immense to fully test, necessitating the development of ad hoc test generation and coverage analysis tools to point to where to look for bugs, and when to stop looking. Moreover, the lack of any formality in system definition often leaves verification teams with a hazy definition of correctness. *Formal verification* [8] of a system works to increase test space coverage by proving a design is correct, either through model equivalence or assertion. The approach is significantly more efficient than simulation-based testing as a single proof can verify correctness over large portions of a design's state space. However, complex modern pipelines with imprecise state management, out-of-order execution, and aggressive speculation are too stateful or incomprehensible to permit complete formal verification.

To further complicate verification, new reliability challenges are materializing in deep submicron fabrication technologies (*i.e.* process technologies with minimum feature sizes below 0.25*u*m). Finer feature sizes are generally characterized by increased complexity, more exposure to noise-related faults, and interference from single event radiation (SER). It appears the current advances in verification (*e.g.*, formal verification, model-based test generation) are not keeping pace with these challenges. As a result, design teams are growing larger, development costs are increasing, and time-to-market lengthens. Without significant advances in the quality and speed of verification, the burden of verification will likely slow the rate at which designers can create higher-performance computing devices, a significant source of value in our industry.

1.1. Dynamic Verification

Recently we proposed the use of dynamic verification to reduce the burden of verification in complex microprocessor designs [1,2]. *Dynamic verification* is an online instruction checking technique that stems from the simple observation that *speculative execution is fault tolerant*. Consider for example, a branch predictor that contains a design error, *e.g.*, the predictor array is indexed with the most significant bits of the PC (instead of the least significant PC bits). The resulting design, even though the branch predictor contained a design error, would operate correctly. The only effect on the system would be significantly reduced branch predictor accuracy (many more branch mispredictions) and accordingly reduced system performance. From the point of view of a correctly designed branch predictor check mechanism, a bad prediction from a broken predictor is indistinguishable from a bad prediction from a correct predictor design. Moreover, predictors are not only tolerant of permanent errors (*e.g.*, design errors), but also transient errors (*e.g.*, noise-related faults or natural radiation particle strikes).

Given this observation, the burden of verification in a complex design can be decreased by simply increasing

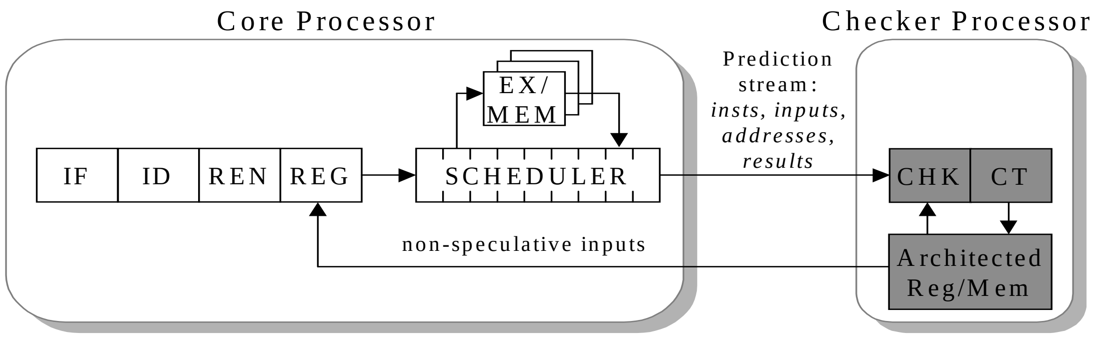

Figure 1. Dynamic Verification Architecture

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

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

The *checker processor* follows the core processor, verifying the activities of the core processor by re-executing all program computation in its wake. The checker processor is assumed to be correct since its simple design lends itself to easy verification (including formal verification). The high-quality stream of predictions from the core processor serves to simplify the design of the checker processor and speed its processing. Pre-execution of the program on the complex core processor eliminates all the processing hazards (*e.g.*, branch mispredictions, cache misses, and data dependencies) that slow simple processors and necessitate complex microarchitectures. In the event the core produces a bad prediction value (*e.g.*, due to a design errors), the checker processor will detect the bad value and flush all internal state from the core processor and restart it after the errant instruction. Once restarted, the core processor will resynchronize with the correct state of the machine as it reads register and memory values from non-speculative storage.

The resulting dynamic verification architecture should benefit from a reduced burden of verification, because only the checker needs to be built correctly. The checker processor will fix any errors in core processor computation, reducing the verification of the core to simply the process of locating and fixing *commonly occurring* design errors that could adversely impact system performance. Moreover, the simplicity of the checker processor design (which must be completely correct) lends itself to high-quality functional and electrical verification. In addition, dynamic verification may render other benefits and opportunities in the design of complex microprocessors. A number of promising directions that we are currently exploring (additional details are available in [1,2]) include: reduced time-to-market and design cost, SER and transient fault tolerance, aggressive core circuitry implementations, and reduced core processor complexity.

1.2. Contributions of this Paper

In this paper, we examine in detail the performance of our initial checker processor design. We found that for many programs, slowdowns from the checking process are minimal, but for others (especially floating point codes) slowdowns were non-trivial. We attribute the primary source of these slowdowns to *a)* core processor decoder stalls because of checker processor backpressure at retirement, *b)* storage hazards created as the core and checker processor compete for storage access ports, and *c)* cache misses experienced by the checker pipeline. Our analyses suggest the addition of a dedicated checker processor register file and store queue to relieve any retirement backpressure. Remaining storage hazards and checker processor cache misses are eliminated with the addition of a dedicated checker processor cache. Using the core processor as a (near) oracle prefetcher, a dual ported 4k checker processor cache has virtually no misses and provides sufficient bandwidth for checker processor accesses. The resulting design demonstrates that online instruction checking can be implemented with no slowdowns, and moreover, our results suggest that the checker should perform well for any program.

In Section 2 we give a brief description of the baseline checker processor design. Details pertaining to the pipeline design can be found in [1,2]. Section 3 presents a detailed performance analysis of our initial checker processor design, revealing the specific sources of core processor slowdown. In Section 4, we motivate design changes that eliminate these slowdowns; detailed performance analyses of the optimized design confirm that

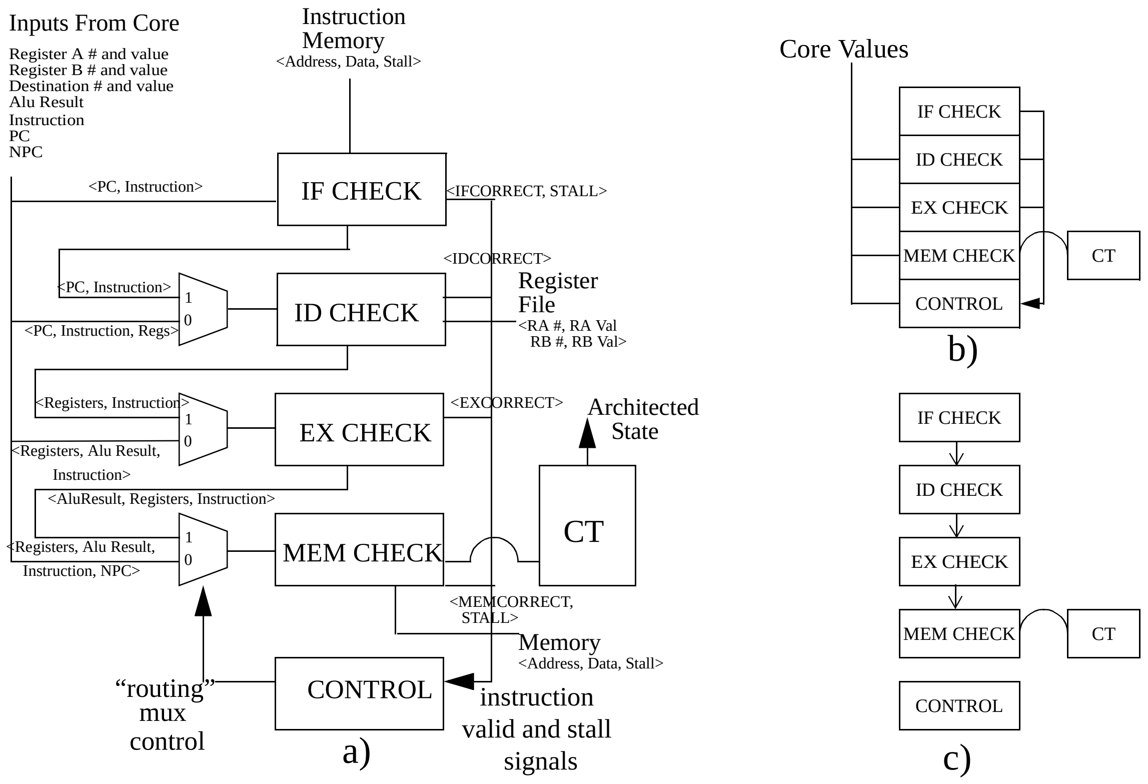

Figure 2. Checker processor pipeline structure for a) a single wide checker processor, b) a checker processor in CHECK mode, and c) a checker processor in RECOVER mode

our enhancements work well. Section 5 details additional related work (not covered in [1,2]). Finally Section 6 summarizes, draws conclusions and discusses future work.

2. Checker Processor Architecture

For dynamic verification to be viable, the checker processor must be simple and fast. It must be simple enough to reduce the overall design verification burden, and fast enough to not slow the core processor. A single-issue two-stage checker processor is illustrated in Figure 2a. The design shown is very general, it can detect and correct any error in core processor computation. Later, we will describe how this design can be simplified if portions of the core processor design are verified to be correct.

Figure 2b show the checker in its normal checking mode. When the core processor retires an instruction, the checker pipeline receives an instruction with core processor predictions. These predictions include the next PC, instruction, instruction inputs, and addresses referenced (for loads and stores). The checker processor ensures the correctness of each component of this transfer by using four parallel stages, each of which verifies a separate component of the prediction stream. Each parallel stage implements a substep of instruction execution and verifies the computed value is identical to that received from the core. If each prediction from the core processor is correct, the result of the current instruction (a register or memory value) as computed by the checker processor is allowed to retire to non-speculative storage in the *commit (CT)* stage of the checker processor.

In the event any prediction information is found to be incorrect, the bad prediction is fixed, the core processor is flushed and restarted, and the core and checker processor pipelines are restarted after the errant instruction. Core flush and restart use the existing branch speculation recovery mechanism contained in all modern high-performance pipelines. As shown in Figure 2b and 2c, the routing MUXes can be configured to form a parallel checker pipeline or a recovery pipeline, respectively.

In recovery mode the pipeline is reconfigured into a serial pipeline, very similar to the classic five-stage pipeline [7]. In this mode, stage computations are sent to the next logical stage in the checker processor pipeline, rather than used simply to verify core predictions. Unlike the classic five-stage pipeline, only one instruction is allowed to enter the recovery pipeline at a time. As such, the recovery pipeline configuration does not require bypass datapaths or complex scheduling logic to detect hazards. Processing performance for a single instruction in recovery mode will be quite poor, but as long as faults are infrequent there will be no perceivable impact on program performance [4]. Once the instruction has retired, the checker processor re-enters normal processing mode and restarts the core processor after the errant instruction. An important aspect of the checker design is that the check and recovery modes use the same checking modules, thereby reducing the area cost of the checker and its design complexity.

Table 1. Benchmarks and baseline statistics

Benchmark	#instr. fwd. (M)	#instr. exec. (M)	%ld exec.	%st. exec.	Base IPC
Compress95	100	250	11.0%	1.0%	1.4303
Crafty00	100	250	32.4%	5.9%	1.4748
Gap00	100	250	25.5%	11.3%	2.2079
Gcc	100	250	25.7%	11.0%	1.2951
Go	100	250	29.3%	8.1%	1.3061
Ijpeg	100	250	18.4%	8.0%	2.6679
Li	100	250	24.3%	11.3%	1.7986
Perl	100	250	23.3%	11.2%	1.2566
Twolf00	100	250	26.3%	8.6%	2.2663
Applu00	100	250	25.9%	10.4%	1.8859
Hydro2d	100	67	23.1%	7.3%	2.2125
Lucas00	100	250	18.4%	0.5%	3.3628
Mesa00	100	250	27.0%	7.3%	2.0450
Tomcatv	0	79	20.1%	7.6%	1.7343
Turb3d	100	250	23.4%	14.9%	2.6035

Pipeline scheduling in the checker processor is trivial. If any checker pipeline is blocked for any reason, all checker processor modules are stalled. This simplifies control of the checker processor and eliminates the need for instruction buffering or complex non-blocking storage interfaces. Since there are no dependencies between instructions in normal processing, checker processor pipeline stalls will only occur during a cache miss or structural (resource) hazard.

The design description given assumes a single wide checker, but we believe that scaling a checker processor is a simple enough task. To make a deeper checker, the separate checker modules simply need to be pipelined deeper. This can be accomplished without adding bypass or complex control because of the minimal inter-pipestage dependencies. To make a wider checker, inter-instruction dependencies must also be verified, but can be done so in parallel with normal instruction checking. Checker scalability will be explored more fully in future work.

3. Performance of the Checker Processor

3.1. Experimental Framework

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [5], 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 [14] and six of the SPEC2000 benchmarks. There are nine integer programs and six FP ones. 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). The six Spec2000 benchmarks were compiled on an Alpha 21264 under OSF/1 V4.0 operating system using at least -O4 optimization. Table 1 shows the benchmarks, the number of instructions that were executed (fast forwarded) before actual simulation began, and the number of instructions simulated for each program (up to 250 million). Also shown are the percentage of dynamic instructions that were loads and stores and the baseline machine IPC.

3.2. Baseline Core Processor 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 16K direct mapped instruction cache and a 16k 4-way set-associative data cache. 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 256k 4-way set-associative cache with 32 byte blocks, with a 6 cycle cache hit latency. If there is a second-level cache miss it takes a total of 34 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. Checker Processor Baseline (Shared) Architecture

The checker processor 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 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, no structural hazards are introduced. The baseline communication pipeline takes two cycles unless there are structural hazards in accessing register file and

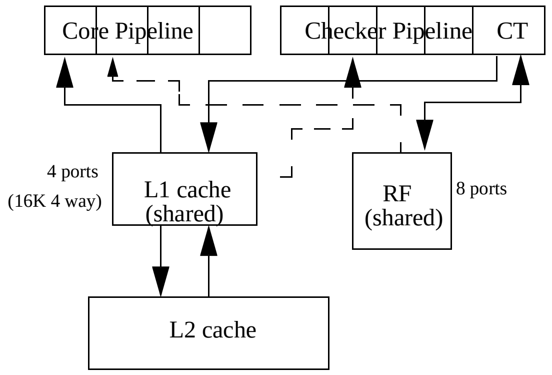

Figure 3. Baseline Checker Processor Architecture

cache ports. In the baseline checker architecture, the storage accesses compete with the core processor for eight architected register file ports and four cache ports, with priority given to the 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). Reorder buffer entries are not deallocated until instructions exit the commit stage of the pipeline, after the checker verifies the operation. The watchdog timer countdown is reset to 60 cycles (the round trip latency to memory) whenever an instruction commits.

We omitted the instruction fetch check stage of the checker processor from our experiments. We believe that it is sufficiently straightforward to protect against incorrectly fetched instructions, simply verify correctness of the core processor instruction cache design and protect all instruction storage in the core processor with ECC. As such, there is no need to determine if the instruction has been correctly fetched, the checker processor need only determine if the PC of the fetch was correct.

Our baseline checker processor architecture (Figure 3) is identical to that presented in our previous report [1]. The baseline storage system looks very similar to a traditional microprocessor system with the checker processor inserted just before commit. The core and checker processors share ports to a common architected register file and L1 data cache (possessing eight and two read ports, respectively). The checker has priority while accessing the storage elements, and the core can only access them in a cycle where ports are not fully consumed by checker accesses. When a core instruction is ready for retirement, it is pushed into the checker processor pipeline. During checking, the core processor must continue to hold speculative state resources (e.g., reorder buffer and load/store queue entries) for the instruction being checked. Once checking is complete, the commit stage retires nonspeculative values into the register file and the cache and the core processor may release speculative storage associated with the finished instruction.

In this design, there are three ways the checker processor can slow the progress of the core processor. First, any contention for the ports between the checker and core will lead to core processor stalls. Second, the checker processor pipeline delays the retirement of instructions, forcing the core processor to hold speculative state longer, thus creating backpressure at retirement. If speculative state resources fill, the core processor decoder will stall as it will not be able to allocate re-order buffer and load/store queue resources. Finally, checker processor cache misses stall the entire checker pipeline, which again can lead to increased pressure on core processor speculative state. The checker processor will only experience misses when data referenced by the core processor is replaced before the checker processor is able to re-execute the memory reference.

As can be seen from Table 2 the average slowdown from the fifteen benchmarks was 2.48%. Overall, the slowdown factors are kept mostly in check, however, there was a wide disparity in the performance for a few individual benchmarks. The most notable slowdowns came from the floating point programs *hydro2d* and *tomcatv*, which had slowdowns of roughly 10.5% and 13% respectively. *Hydro2d* also experiences the largest increase in the number of decoder stalls, a leading factor to its poor performance. *Hydro2d* is a very computationally intensive program, which is used to solve hydrodynamically Navier Stokes equations in astrophysics applications. The program is well tuned and it makes quite efficient use of machine resources, such that any backpressure will manifest in the form of slowdowns. Storage access stalls in the core were infrequent; there are two primary reasons for this. First, storage hazards are not created when the core processor accesses physical registers or load/store queue entries. Second, for the integer programs especially, many storage hazards occur during misspeculation and thus do not slow nonspeculative core progress.

There were a non-trivial number of L1 cache misses experienced by the checker processor, especially for the floating point codes. These programs tend to stream through memory, touching cache lines only a few times and then quickly replacing them with other memory contents. Figure 5 graphs the average delay (in cycles) between first execution of an instruction and its final re-execution on the checker pipeline. We term this delay the *slip* of the instruction. *Tomcatv*, which computes fluid dynamics and geometric translations, experienced the worst slowdown in conjunction with the largest increase in L2 accesses. For programs with large slip values, especially the floating point codes, delaying the second check reference to retirement can create a non-trivial number of L1 data cache misses.

4. Increasing Checker Processor Efficiency

4.1. Eliminating Decoder Stalls and Storage Hazards

It is a simple process to eliminate decoder stalls and storage hazards, they are both structural hazards that can be eliminated by simply increasing the number of resources available to the checker processor. In this case, speculative state and storage ports. Figure 4 illustrates our approach to adding these extra resources; we call this design the *FastShared* model. We add a dedi-

Table 2. Analysis of Shared Model Checker Performance

	Performance	Stalls							
Benchmark	Slowdown (Shared Vs. Base)	Decoder		Storage Ports		Cache Misses		L2 Traffic Increase [and incur. %]	
		Base	Shared	Base	Shared	Base	%of Dl1 misses from Shared	Base	Increase in L2 Traffic for shared
compress95	0.01%	0.00%	0.00%	0	0.00%	0	0.00%	0 [0]%	-3789 [-0.08%]
crafty00	0.04%	0.26%	0.21%	0	2.17%	0	3.17%	0 [0]%	89472 [0.58%]
gap00	3.91%	19.23%	17.50%	0	2.27%	0	80.30%	0 [0]%	1118312 [19.52%]
gcc	2.00%	1.67%	2.72%	0	0.82%	0	28.20%	0 [0]%	1013148 [6.87%]
go	0.18%	0.28%	0.37%	0	0.83%	0	12.12%	0 [0]%	402933 [3.69%]
ijpeg	1.20%	11.31%	9.14%	0	1.93%	0	14.59%	0 [0]%	142868 [12.83%]
li	0.34%	0.52%	0.49%	0	0.57%	0	6.48%	0 [0]%	255959 [3.67%]
perl	0.95%	0.35%	0.74%	0	0.36%	0	43.76%	0 [0]%	738170 [4.06%]
twolf00	0.05%	4.64%	3.69%	0	1.49%	0	19.15%	0 [0]%	28457 [0.72%]
applu00	0.59%	42.32%	35.37%	0	2.21%	0	6.53%	0 [0]%	318688 [2.54%]
hydro2d	10.47%	34.47%	40.14%	0	1.45%	0	27.06%	0 [0]%	803670 [19.40%]
lucas00	2.59%	0.00%	0.00%	0	2.16%	0	99.96%	0 [0]%	328945 [50.02%]
mesa00	-0.09%	1.95%	2.02%	0	1.93%	0	92.43%	0 [0]%	9639 [0.22%]
tomcatv	12.99%	6.31%	2.02%	0	0.45%	0	52.07%	0 [0]%	1630372 [37.01%]
turb3d	1.93%	28.61%	9.32%	0	2.66%	0	43.23%	0 [0]%	1100773 [34.91%]
AVERAGE	2.48%	10.13%	8.25%	0	1.42%	0	33.40%	0 [0]%	531841 [13.06%]

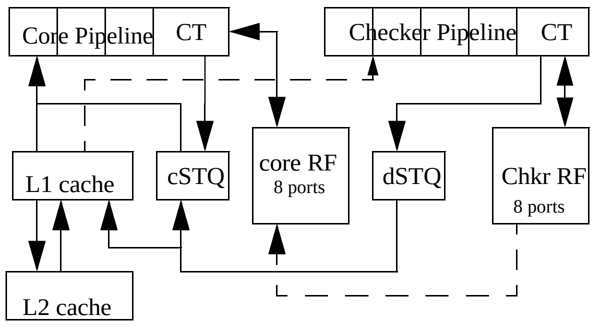

Figure 4. *FastShared* Storage Model
(16K 4way) 4 ports (2 dedicated checker Wr. Ports)

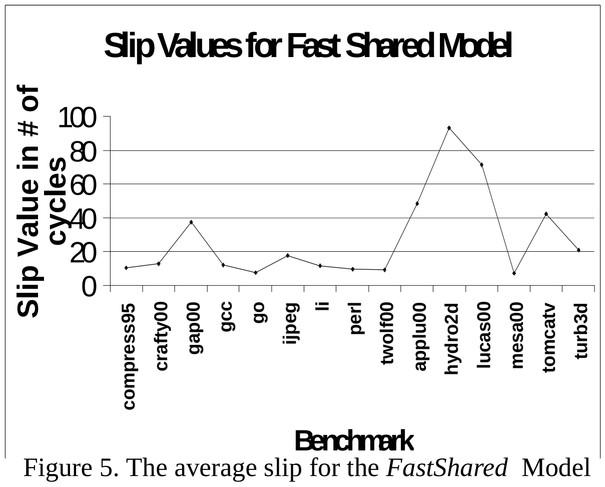

Figure 5. The average slip for the *FastShared* Model

cated checker processor register file, two store queues, and additional ports to the L1 cache (2 total). The split register file permits the core to retire register values (speculatively) into its private register file, thus allowing it to release its internal reorder buffer speculative state at the time instructions are transferred into the checker processor pipeline. The checker processor register file contains truly non-speculative state, if the checker processor detects an error in core processor computation, it must re-synchronize the core processor register file. This can be easily implemented by copying the checker processor register file contents to the core processor register file when faults are declared. The L1 data cache is still shared between the core and checker processors, however, extra ports eliminate storage related stalls.

The core store queue (cSTQ) permits the core processor to release load/store queue entries when instructions pass to the checker processor, thereby further relieving speculative state pressure. The cSTQ holds retired stores until they pass through the checker at which time the space can be reclaimed. Since the checker writes back store values to the shared cache, the cSTQ need not write back the same store values. To ensure a coherent view of speculative memory state, the core processor probes the cSTQ in parallel with the L1 data cache, if any cSTQ entries match the referenced address, the latest value is used in lieu of the L1 data cache value. Checker stores write to the checker STQ (dSTQ). The dSTQ writes to the L1 cache and releases cSTQ and dSTQ entries for the store. When the core executes a load, if it does not hit in the LSQ, the L1 cache and the cSTQ have to be searched in parallel with the latest STQ value overriding L1 cache values. If either the cSTQ or the dSTQ fills up, the corresponding stage stalls until the store at the head of the queue is committed and an entry becomes available. The (dSTQ) serves a similar purpose: to prevent checker pipeline stalls on store write misses, store value are placed into the dSTQ and later written to memory when L1 data cache write ports are available. The space cost for the store queues is minimal - two small queues with eight entries each virtually eliminated all backpressure in the core processor.

Table 3 details the performance of the *FastShared* model. It is quite effective in reducing core processor decoder stalls, in all cases (with the exception of *Applu00*) there are less stalls than in the baseline

Table 3. Analysis of FastShared Model Checker Performance

	Performance		Stalls							
Benchmark			Decoder		Storage Ports		Cache Misses		L2 Traffic Increase [and incr. %]	
	Slowdown (FastShared Vs Base)	Speedup (FastShared Vs Shared)	Shared	FastShared	Shared	FastShared	%of Dl1 misses from Shared	%of Dl1 misses from Fast Shared	Increase in L2 Traffic for shared	Increase in L2 Traffic for Fast-shared
compress95	0.00%	0.01%	0.00%	0.00%	0.00%	0	0.00%	0.00%	-3789 [-0.08%]	2 [0.00%]
crafty00	0.02%	0.02%	0.21%	0.20%	2.17%	0	3.17%	3.17%	89472 [0.58%]	1042317 [6.78%]
gap00	0.01%	0.03%	17.50%	7.68%	2.27%	0	80.30%	80.30%	1118312[19.52%]	1179802 [20.6%]
gcc	1.25%	0.77%	2.72%	1.19%	0.82%	0	28.20%	29.00%	1013148 [6.87%]	2562353[17.37%]
go	0.15%	0.03%	0.37%	0.08%	0.83%	0	12.12%	12.12%	402933 [3.69%]	1727555[15.80%]
ijpeg	0.28%	0.93%	9.14%	8.73%	1.93%	0	14.59%	14.57%	142868 [12.83%]	154662 [13.88%]
li	0.05%	0.30%	0.49%	0.31%	0.57%	0	6.48%	6.49%	255959 [3.67%]	726937 [10.41%]
perl	0.95%	0.01%	0.74%	0.31%	0.36%	0	43.76%	43.76%	738170 [4.06%]	2822083[15.53%]
twolf00	0.07%	-0.02%	3.69%	3.59%	1.49%	0	19.15%	19.15%	28457 [0.72%]	54349 [1.37%]
applu00	-0.13%	0.73%	35.37%	39.46%	2.21%	0	6.53%	6.53%	318688 [2.54%]	2354622[18.74%]
hydro2d	10.49%	-0.02%	40.14%	13.77%	1.45%	0	27.06%	27.06%	803670 [19.40%]	806233 [19.46%]
lucas00	2.59%	0.00%	0.00%	0.00%	2.16%	0	99.96%	99.96%	328945 [50.02%]	328945 [50.02%]
mesa00	-0.09%	0.00%	2.02%	2.01%	1.94%	0	92.43%	92.43%	9639 [0.22%]	9933 [0.22%]
tomcatv	12.99%	0.00%	2.02%	0.94%	0.45%	0	52.07%	52.07%	1630372[37.01%]	1419675[32.23%]
turb3d	1.74%	0.19%	9.32%	5.14%	2.66%	0	43.23%	43.56%	1100773[34.91%]	1754036[55.63%]
AVERAGE	2.03%	0.19%	8.25%	5.56%	1.42%	0	33.40%	35.34%	531841 [13.06%]	1126307[18.54%]

checker processor configuration. The approach even eliminates stalls that exist in the baseline model due to store write misses, resulting in less decoder stalls for most programs than the baseline experiments without a checker processor. Thus for some programs, *Applu00* and *Mesa00*, there are actually performance gains over the baseline. Storage ports stalls are completely negated by the addition of the additional register file and additional ports to the cache.

Overall, slowdowns improved for some programs, *e.g.*, *GCC* and *Ijpeg*, but worsened for other programs such as *Gap00*. The gain in performance would have been more notable, if this model did not aggravate L1 cache misses. This problem, discussed briefly before in conjunction with *Tomcatv's* slowdown, is the interaction of two working sets on the same cache. If the slip is too large there will be replacements in the cache before a load/store reaches the checker. Thus, the line must be retrieved again from the L2. This causes a large increase in the L2 traffic, which is illustrated in the L2 traffic increase column of the table. The *FastShared* model aggravates this condition by adding a store queue that can increase the lifetime of an instruction in the core before it is checked. Average slip measurements are shown in Figure 5. There is a large correlation between the high slip and low performance in the shared models. The size of the data set and number of memory access are also contributing factors. For example, *Lucas* which has the second highest slip value does not experience a large slowdown because it has comparatively very few data cache misses in the baseline. In other words, there is less conflict for the cache space since each working set is smaller.

4.2. Improving Checker Processor Cache Performance

We can address checker processor L1 cache misses by providing a mechanism by which both core and checker processor data working sets can easily co-exist. The simple approach to this is to split the data cache, giving separate caches to both the checker and the core processors. Figure 6 illustrates the approach, we call this new dynamic verification architecture the *MiniDiva* model.

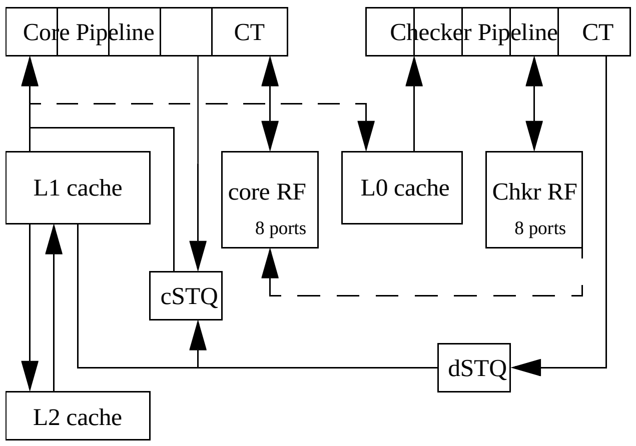

Figure 6. *MiniDiva* Storage Model
Core Cache 16K 4 way 2 rd 2 wr ports
Checker Cache 4K 8 way 4 ports

The *MiniDiva* model extends the *FastShared* model by removing the shared L1 constraint. It provides a small dedicated cache for the checker processor, called the *L0 cache*. The L0 cache is loaded with whatever data is touched by the core processor;*MiniDiva* it taps off the output port of the L1 cache. If the checker processor misses in the L0 cache, it blocks the entire checker pipeline, and the miss is serviced by the core L2 cache. The STQ mechanism and the separate RF are the same as described earlier. When stores commit, they write their

Table 4. Analysis of MiniDiva Model Checker Performance

	Performance		Stalls							
Benchmark	Slowdown (MiniDiva Vs Base)	Speedup (Mini Diva Vs FastShared)	Decoder		Storage Ports		Cache Misses		L2 Traffic Increase [and incr.%]	
			FastShared	Mini Diva	FastShared	Mini Diva	%of Dl1 misses from FastShared	%of Dl1 misses from Mini Diva	Increase in L2 Traffic for Fast shared	Increase in L2 Traffic for Mini Diva
compress95	0.00%	0.00%	0.00%	0.00%	0	0	0.00%	0	2 [0.00%]	9593 [0.20%]
crafty00	0.01%	0.01%	0.20%	0.21%	0	0	3.17%	0	1042317 [6.78%]	819 [0.01%]
gap00	-0.74%	4.81%	7.68%	12.26%	0	0	80.30%	0	1179802 [20.6%]	6293 [0.11%]
gcc	0.02%	1.25%	1.19%	1.65%	0	0	29.00%	0	2562353[17.37%]	151763 [1.03%]
go	0.00%	0.15%	0.08%	0.22%	0	0	12.13%	0	1727555[15.80%]	39180 [0.36%]
ijpeg	-0.25%	0.53%	8.73%	10.35%	0	0	14.57%	0	154662 [13.88%]	57265 [5.14%]
li	0.06%	-0.01%	0.31%	0.38%	0	0	6.49%	0	726937 [10.41%]	127888 [1.83%]
perl	0.00%	0.96%	0.31%	0.35%	0	0	43.76%	0	2822083[15.53%]	65049 [0.36%]
twolf00	0.05%	0.02%	3.59%	3.59%	0	0	19.15%	0	54349 [1.37%]	-191 [-0.01%]
applu00	-1.28%	1.15%	39.46%	40.85%	0	0	6.53%	0	2354622[18.74%]	3980 [0.03%]
hydro2d	-0.47%	12.23%	13.77%	31.75	0	0	27.06%	0	806233 [19.46%]	75965 [1.83%]
lucas00	0.00%	2.66%	0.00%	0.00%	0	0	99.96%	0	328945 [50.02%]	0 [0.00%]
mesa00	-0.09%	0.00%	2.01%	2.01%	0	0	92.43%	0	9933 [0.22%]	482 [0.01%]
tomcatv	0.20%	14.71%	0.94%	5.87%	0	0	52.07%	0	1419675[32.23%]	327437 [7.43%]
turb3d	-3.17%	5.00%	5.14%	6.67%	0	0	43.56%	0	1754036[55.63%]	271265 [8.60%]
AVERAGE	-0.38%	2.90%	5.56%	7.74%	0	0	35.34%	0	1126307[18.54%]	75786 [1.80%]

result to the L0 cache and the dSTQ. When a free store port is available on the L1 cache, the store in the dSTQ is retired to the L1 cache. If the dSTQ fills the checker processor pipeline stalls until an entry can be written back.

The performance of the *MiniDiva* architecture is summarized in Table 4. Splitting the caches yields almost a 3% improvement over the *FastShared* model. In fact, the *MiniDiva* model even exhibits modest performance gains over the baseline model without a checker processor. We believe that this is due to the cSTQ, which eliminates store writeback misses at commit that can slow the baseline architecture (without a checker processor). This effect is demonstrated in the average decoder stalls percentage which go from 10.13% in the baseline to only 7.74% in the *MiniDiva* model. Second, our data shows that some backpressure may actually cause positive interference in the executing loads and stores. The LSQ and STQ are considered to have infinite bandwidth, thus if a memory access can grab the value it needs from these queues it does not have to reserve a cache port.

The *MiniDiva* model is quite efficient, it eliminates virtually all the checker processor stalls that can slow the core processor, as well as a few core processor stalls. In addition, cache performance improves because the checker processor cache now contains the working set of the checker processor, *i.e.*, loads and stores in the window of core processor execution. With the working set in the L0 cache, core processor activity can no longer displace checker processor working set. A 4k checker cache experiences virtually no misses for any programs. We also looked at slowdowns for a 2k cache, overall slowdown was only 0.77%, but the worst case slowdown rose 4.40% for *Ijpeg.*

While on the surface it may seem adding a second cache is an expensive proposition, there are two mitigating factors. First, the L0 cache eliminates the need for extra ports on the core processor L1 cache, which increases its size and slows its accesses. Second, the cache need only needs to hold the data from when it is first touched in the core until the time it is verified by the checker processor (*i.e.*, slip latency). As such, the *MiniDiva* L0 cache be made very small.

4.3. Eliminating Common Mode L1 failures

In the *MiniDiva* model, the checker processor has its own dedicated L0 data cache. The checker processor register file and L0 and L1 caches hold the architected nonspeculative state of the machine, whereas the core register file holds speculative states. If an error is detected by the checker processor, it rewrites the core register file with values from its own register file. Since the checker processor relies on correct information in the core processor L1 cache, any design errors in this component will manifest as a common failure that could impair correct program execution. For many designs this may not be a significant concern, however, we are currently exploring core processor design strategies, such as self-tuning circuits, that benefit greatly if the core processor L1 cache state is speculative as well.

Figure 7 illustrates an approach to eliminate common mode failures from the core processor L1 cache, we call this design the *SplitDiva* design. There are no STQs present in this design. The core commit stage speculatively commits results to the (speculative) L1 core cache and the core register file. As in the *MiniDiva* model, commits in the core processor are not stalled by the checker processor pipeline. When a fault is detected by the checker processor, it must re-synchronize the core processor L1 with non-speculative storage (*i.e.*, L2 cache state). This process can be easily accomplished by invalidating all core processor L1 cache state. Given that faults are infrequent, the performance implications of this simple (but expensive) approach should be minimal.

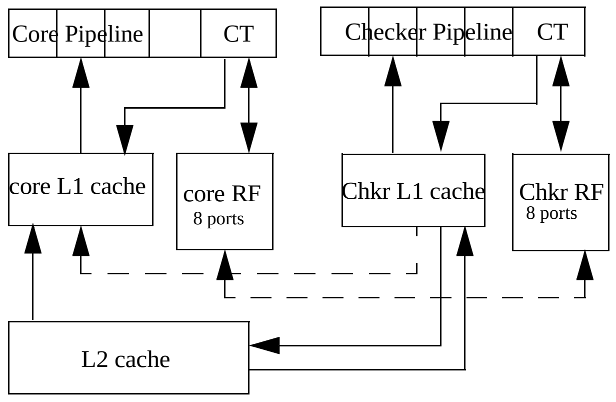

Figure 7. Split Storage Model
Core Cache- (16k 4 way) 2 ports
Checker Cache- (4K 8 way) 4ports

To enable a high hit rate for the checker processor L1 cache, whenever the cache is accessed in the core for loads and stores, the address stream of the reference is sent as a prefetch to the checker processor cache. The checker cache services misses directly from the L2. Thus, there is additional traffic between the checker processor cache and the L2, unlike in the *MiniDiva* model. To minimize L2 traffic increases, lines written from the L2 to the core processor L1 are also immediately placed in the checker processor L1 cache, using a common line bus. In addition, speculative state in the core processor L1 need not be written back to the L2.

An interesting feature of this design is that ECC is no longer needed to protect the core processor L1 cache. Since this state is completely speculative, transient errors due to, for instance, energetic particle strikes will not impair the correct operation of the program. Of course the checker processor L1 cache will require ECC to protect the non-speculative program state. Depending on the size of the checker processor L1 cache and the ECC coding technique, the saving in core processor L1 ECC could go quite far to make up for the area costs of the checker processor and its L1 cache.

Table 5 details the performance of the *SplitDiva* model. Again, the slip between the core and the checker permits memory addresses to be sent as a prefetch stream. Thus stalls in the checker due to cache misses are kept to a minimum. However, unlike the small working set requirements of the *MiniDiva* model, the *SplitDiva* checker processor L1 cache must hold the entire program working set (or as much of it as it can). Fortunately, the core processor address stream presents a very high-quality view of future references well in advance of their use, thus with sufficient L2 cache bandwidth, most checker processor misses can be averted.

Comparing the slowdown of the *SplitDiva* model to the baseline, we notice that this model,with a 4K data cache, has an average slowdown of 1% for all the benchmarks. This is superior to all the storage models discussed so far except the *MiniDiva* model. The *SplitDiva* model, however, offers us greater reliability over all the other storage models, permitting the core L1 cache to be stripped of ECC. This added reliability comes at a cost, though, as can be seen from Table 5.

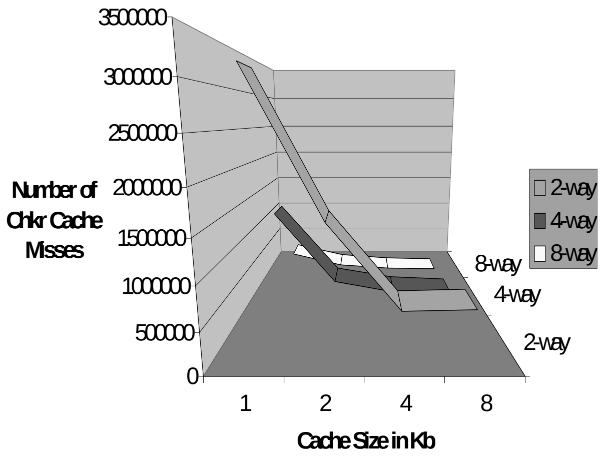

Figure 8. Cache Misses With Respect to Diva cache size and associativity

The *SplitDiva* model has an average slowdown of 1.46% compared to the *MiniDiva* model. The leading factor for this is the increase in L2 traffic for the split model, which is around 75 times that of the *MiniDiva* model.

Figure 8 illustrates the sensitivity of the *SplitDiva* model to checker processor L1 cache geometry. Results are shown for increasing cache sizes and associativity. Clearly, checker processor cache performance is quite good at 4k or higher sizes, and with more associativity (i.e., 4 or 8 way) for even 2k and 1k configurations. However, there is a clear trade-off between cache size and a L2 traffic increases. For the 4-way cache configurations, for example, L2 traffic increases by 86%, 147%, and 245% for decreasing cache sizes of 4k, 2k, and 1k respectively.

4.4. Sensitivity to Other Key Design Parameters

Function Units. To test the *MiniDiva* checker's tolerance for functional units the number of integer ALUs and floating point ALUs was varied. Not surprising, it was found that as the number of ALUs were decreased so did performance. We saw an average slowdown across the fifteen benchmarks of roughly 12% when the number of ALUs were halved from 4 to 2. A subset of the benchmarks was then simulated with only one floating point ALU and one integer ALU in the checker. This subset of three floating point *i.e.*, *Applu00*, *Lucas00* and *Mesa00*, and three integer programs, *i.e.*, *Crafty00*, *Gap00* and *Twolf00*, averaged a 20% slowdown. Clearly, reducing the number of functional units below that of the core execution resources can have significant negative impacts on core processor performance. The number of functional units in the checker must be equal to the core to maintain a good balance with the core commit speed, thereby preventing any slowdowns for efficient programs.

Memory Ports. The versatility of the *MiniDiva* checker was further tested through sensitivity analysis on the

Table 5. Analysis of SplitDiva Model Checker Performance

	Performance		Stalls							
Benchmark			Decoder		Storage Ports		Cache Misses		L2 Traffic Increase [and incr.%]	
	Slowdown (Split Vs Base)	Speedup (Split Vs MiniDiva)	MiniDiva	Split	MiniDiva	Split	%of Dl1 misses from MiniDiva	%of Dl1 misses from Split	Increase in L2 Traffic for Fast MiniDiva	Increase in L2 Traffic for Split
compress95	0.00%	0.00%	0.00%	0.00%	0	0	0	0	9593 [0.20%]	5251070[107.5%]
crafty00	0.06%	-0.05%	0.21%	0.31%	0	0	0	0	819 [0.01%]	12461427[81.0%]
gap00	0.14%	-0.87%	12.26%	20.02%	0	0	0	0	6293 [0.11%]	2816646[49.17%]
gcc	0.09%	-0.07%	1.65%	1.71%	0	0	0	0	151763 [1.03%]	8861076[60.07%]
go	-0.01%	0.01%	0.22%	0.27%	0	0	0	0	39180 [0.36%]	13466178[123%]
ijpeg	0.34%	-0.59%	10.35%	11.69%	0	0	0	0	57265 [5.14%]	1601495[143.7%]
li	-0.01%	0.06%	0.38%	0.55%	0	0	0	0	127888 [1.83%]	2976652[42.62%]
perl	-0.01%	0.01%	0.35%	0.39%	0	0	0	0	65049 [0.36%]	11272662[62.0%]
twolf00	0.01%	0.04%	3.59%	4.76%	0	0	0	0	-191 [-0.01%]	396975 [10.02%]
applu00	0.89%	-2.13%	40.85%	42.97%	0	0	0	0	3980 [0.03%]	19384421[154%]
hydro2d	7.43%	-7.35%	31.75	38.24%	0	0	0	0	75965 [1.83%]	2143317[51.74%]
lucas00	6.86%	-6.42%	0.00%	0.00%	0	0	0	0	0 [0.00%]	1973517[300.1%]
mesa00	0.00%	-0.09%	2.01%	1.96%	0	0	0	0	482 [0.01%]	43803 [0.98%]
tomcatv	0.57%	-0.38%	5.87%	6.94%	0	0	0	0	327437 [7.43%]	627439 [14.24%]
turb3d	1.06%	-4.08%	6.67%	29.91%	0	0	0	0	271265 [8.60%]	2997647[95.07%]
AVERAGE	1.16%	-1.46%	7.74%	10.65%	0	0	0	0	75786 [1.80%]	5751622[86.38%]

number of memory ports on the checker processor cache. When the number of memory ports was lowered from 4 to 2 a slowdown of 1.7% was experienced in the benchmarks. A 8.1% slowdown was the result of lowering the memory ports to 1. As is the case for functional unit resources, they should match that of the core to reduce the possibility of core processor slowdown. Slowdowns, however, were mitigated by the dSTQ, is it was able to reduce the impacts of not immediately having more ports available.

Pipedepth. We experimented with increasing the pipeline depth and functional unit depth for the checker pipeline. Similar to earlier presented experiments [1,2], the results were quite encouraging. Increasing the checker pipeline length four-fold (from 2 stages to 8) increased average slowdown for the *MiniDiva* model to only 2.4%, demonstrating that overall the checker processor pipeline is quite tolerant of processing latency. The integer codes were extremely latency tolerant, average slowdown for these programs was only 0.04%. These results suggest that superpipelining the checker processor to scale checker processor bandwidth will likely produce very good results. We are currently exploring this design in detail in conjunction with the implementation of a physical checker processor design.

I-Cache Check. Fetching is a simple operation which can be easily made reliable, using encoding techniques like ECC. Nonetheless, we added a small I-cache (512 entries) to the checker to test performance impacts. The slowdown experienced was less than 1%. High spatial locality in instruction memory and core-driven prefetching eliminated virtually all misses.

Fault Rate. Analysis in our previous work indicated that recovery from faults did not impact performance provided the faults were infrequent enough (less than one every 1000 instructions).

5. Related Work

The related work mentioned here is in addition to those found in [1,2].

Blum and Wasserman discussed checkers with respect to the Pentium Division Bug [4]. In addition to talking about complete checkers, they discussed how a partial checker could be utilized to achieve checking accuracy that was almost perfect. They also postulate that if error rates are kept small enough, correcting procedures can be very time consuming without impacting the performance of the system.

A similar idea of having a simple inorder pipeline that runs in parallel with the main engine is presented by Nakra et al in [9]. In that paper, instructions that already have their operations decoded execute on the simple pipeline upon detection of misprediction in the main engine.

Rotenberg's AR-SMT[11] and more recently Slipstream[13] processors use multiple threads to eliminate hazards and verify correct operation. In Slipstream an advanced stream (A-stream) is used to aid a redundant stream (R-stream) by providing future knowledge to the trailing stream. As a result performance can be increased and the redundancy can be used to detect errors. However this technique does not provide total coverage or resistance to design errors.

Other techniques for obtaining reliability include the redundant hardware approach, as in the IBM S/390 G5 microprocessor [12] and the redundant thread (or SRT) approach. In the SRT approach proposed in [10], redundant threads run concurrently and compare results for fault detection. Performance in SRT processors is improved with one thread prefetching cache misses and branch outcomes for other threads, similar to the prefetching ideas in the *SplitDiva* model.

In a previous study Dundas and Mudge[6] showed that a performance gain was possible, by ignoring cache misses and pre-executing instructions while the processor would normally be stalled. When a cache miss occurred the register file and instruction address would be backed up and the processor would execute in runahead mode. By doing this they were able to generate a fairly accurate prefetch stream. This would in turn warm up the cache and reduce future stalls. A checker processor can use similar techniques, because there is advanced knowledge of the addresses that will be touched by instructions when they enter the checker. A similar mechanism for obtaining prefetches was proposed in an earlier paper by Chen and Baer [3].

6. Conclusions and Future Work

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 previously proposed the use of dynamic verification, a technique that adds a checker processor to the retirement phase of a processor pipeline. If an incorrect instruction is delivered by the core processor, for instance due to a design error or transient fault, the checker processor will fix the errant computation and restart the core processor using the processor's speculation recovery mechanism. Dynamic verification focuses the verification effort into the checker processor, whose simple and flexible design lends itself to high-quality functional and electrical verification.

We presented detailed analyses of a baseline checker processor design identical to the one presented in our initial proposal [1]. We find three factors lead to occasional poor program performance: 1) checker processing latency delays core processor retirement leading to decoder stalls, 2) shared storage resources create competition for storage ports which can force the core processor to delay issuing instructions, and 3) checker processor cache misses occur that also create backpressure on core retirement leading to additional decoder stalls.

To eliminate decoder stalls, we provide a dedicated register file for the checker processor, thereby permitting the core processor to immediately release speculative state resources when instructions enter the checker processor. In addition, a store queue is added to the core processor design , that permits the load/store queue to also release speculative storage when instructions enter the checker pipeline. Checker processor cache misses are eliminated by giving the checker processor its own dedicated data cache. Our approach employs the core processor reference stream as a high-quality prefetch oracle, driving the placement of data into the checker processor cache in advance of any references. We examine two designs, one that draws data from the core processor L1 cache and a more flexible design that draws data directly from the L2 cache. Both designs performed quite well, with 4k 8-way set-associative caches experiencing virtually no misses, however, L2 cache traffic was understandably higher for the latter *SplitDiva* design. Our refined design now exhibits negligible slowdowns for all programs examined, while at the same time keeping checker pipeline and storage costs quite low.

We feel that these results strengthen the case that dynamic verification holds significant promise as a means to address the cost and quality of verification for future microprocessors, while at the same time creating opportunities for faster, cooler, and simpler designs. Currently, we are in the process of better quantifying area, power, and performance aspects of our optimized checker processor design through the development of a physical checker processor design. In conjunction with this effort, we are continuing to refine the checker processor design. Currently, we are exploring strategies to better manage the checker processor cache using advanced prefetching techniques. We are also examining the cost and utility of partial checking mechanism, where reduced functionality checkers are employed. We will report on these results in a future paper.

7. References

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

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

[3] J.-L. Baer and T.F Chen, "An effective on-chip preloading scheme to reduce data access penalty", In *Proc. of Supercomputing* , pages 176-186, 1991.

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

[5] 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.

[6] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss", *Proc. 1997 Acm Int. Conf. on Supercomputing*, July 1997.

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

[8] W. Hunt, "Microprocessor design verification", *Journal of Automated Reasoning,* 5(4):429-460, Dec. 1989.

[9]T. Nakra, R. Gupta, and M.L. Soffa, "Value Prediction in VLIW Machines", *In ACM/IEEE 26th International Symposium on Computer Architecture,* May 1999.

[10]S. K.Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading", *In 27th Annual International Symposium on Computer Architecture (ISCA),* June 2000.

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

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

[13] K. Sundaramoorthy, Z. Purser, and E.Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance", *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

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