BlackJack: Hard Error Detection with Redundant Threads on SMT

Ethan Schuchman and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{erys, vijay}@purdue.edu

Abstract

Testing is a difficult process that becomes more difficult with scaling. With smaller and faster devices, tolerance for errors shrinks and devices may act correctly under certain condition and not under others. As such, hard errors may exist but are only exercised by very specific machine state and signal pathways. Targeting these errors is difficult, and creating test cases that cover all machine states and pathways is not possible. In addition, new complications during burn-in may mean latent hard errors are not exposed in the fab and reach the customer before becoming active.

To address this problem, we propose an architecture we call BlackJack that allows hard errors to be detected using redundant threads running on a single SMT core. This technique provides a safety-net that catches hard errors that were either latent during test or just not covered by the test cases at all.

Like SRT, our technique works by executing redundant copies and verifying that their resulting machine states agree. Unlike SRT, BlackJack is able to achieve high hard error instruction coverage by executing redundant threads on different front and back-end resources in the pipeline. We show that for a 15% performance penalty over SRT, BlackJack achieves 97% hard error instruction coverage compared to SRT's 35%.

1 Introduction

Technology scaling has yielded smaller and faster transistors which have enabled higher performance. Unfortunately, scaled devices are more susceptible to hard errors (i.e., permanent errors) because each device has smaller margins for correct operation. For instance, a small charge trapped in the gate oxide can permanently damage the transistor. In addition, because margins are small, other (hard to quantify) variables can cause devices to operate correctly in some cases and incorrectly in others. These variables arise from normal operating conditions such as signal paths, machine state, or localized temperatures and supply-voltage droops. With correct operation being dependent on such variables, testing becomes difficult. For good coverage, it is not enough to test a transistor under one operating condition, but every transistor must be tested considering intractably many signal paths and machine states.

The increasingly-difficult problem of *latent* defects complicates the testing process even more. CPU lifetime can be described best by a bathtub-shaped curve, where some chips have short lifetimes, a few have intermediate lifetimes, and most have long (acceptable) lifetimes. The lifetime is determined by how long it takes for a latent defect to worsen and become active. CPU manufacturers weed out the short-lifetime chips from the production flow by a process called burn-in. In burn-in, chips are run at high voltage and temperature to cause large amounts of wear-and-tear over a short amount of time. Burn-in causes chips with short lifetimes to fail before leaving the fab, so that chips that reach the customer can be expected to have a long (acceptable) lifetime. Burn-in has long been relied upon but its continued feasibility and coverage are now coming into question. As devices get smaller, they incur more wear-and-tear in burn-in causing even the long-lifetime chips to fail. In addition, an effect called *thermal run-away* is becoming a problem. In thermal run-away, devices undergoing burn-in get hotter increasing leakage which in turn increases the temperature, creating a positive feedback loop. If thermal-run-away is not controlled, even the good chips are destroyed [13]. If controlled, the coverage of burnin comes into question [8]. Have all devices been exercised long enough at high-enough temperatures so that all latent faults are exposed?

These two worsening difficulties attack two basic assumptions of testing for hard errors. The first difficulty implies that test cases cannot be created for every possible defect. The second difficulty implies that even if one could, some of the latent defects would remain unexposed by burn-in. Even today, not only is 100% coverage not achieved but even quantifying what has been covered is only an approximation [7].

Despite these difficulties, testing will not disappear, but will only become more important. However, hard errors will get by and will be exposed in the field, despite most valiant attempts. This paper proposes a technique which is a safety net, *not a replacement,* for testing. Our technique allows defects that are missed in the testing process (either because the error was never tested, or the error was latent at testing time) to be detected in the field, preventing hard errors from corrupting data. It may seem that continual testing throughout the lifetime of the chip would achieve our target. However, injecting test inputs into the chip requires testers, which are expensive, specialized machines, and are not available in the field.

We make the key observation that instead of testing for an inordinately large number of potential defects we can instead test only for the defects that are exercised by a program when the program runs. One such way is to redundantly execute the program and compare program state. Because redundant execution is the standard approach to handling soft errors, our observation implies that soft-error schemes can be applied to hard errors to allow previously untested or latent defects to be detected at run time.

Despite this implication, soft-error schemes can not be applied *as is* to hard errors. Because soft-error techniques rely on the errors being transient, the techniques exploit temporal redundancy. For instance, a Simultaneously and Redundantly Threaded processor (SRT [10]) executes two copies of a program, called *leading* and *trailing* threads, on one SMT core assuming that a soft error would affect only one copy. Because hard errors are permanent, temporal redundancy *alone* will not suffice. Because both copies of an instruction run on the same core, they may encounter the same hard error which would then elude detection. To ensure proper detection, each instruction copy must use different hardware. That is, the redundant executions must be *spatially diverse*.

Ensuring spatial diversity is the key challenge in using SRT for

hard-error detection. We address this challenge in our microarchitecture called *BlackJack*. Spatial diversity does not occur naturally in SRT because the trailing thread is mostly-identical to the leading thread, resulting in the same resources being used by most leading-trailing instruction pairs. A naive approach would be to shuffle SRT's trailing instructions that are issued together in one cycle, so that each trailing instruction goes to a different execution way than the corresponding leading instruction. However, as we explain later, there is no convenient point in the pipeline to shuffle the trailing instructions. Shuffling before rename violates program correctness due to lack of dependence information; shuffling after rename does not cover the frontend and severely complicates issue which is already timing-critical.

To avoid these difficulties, we make the key observation that the leading thread determines dependencies well before the trailing thread executes. Accordingly, our novel idea is that we borrow dependence information from the leading thread which allows us to shuffle the instructions *before* they are fetched by the trailing thread. The dependence information ensures that the shuffling preserves program correctness. Specifically, we shuffle the leading instructions that were co-issued in the same cycle, called a *packet.* Our shuffling is *not* random and is specifically designed to ensure that the leading and trailing executions are spatially diverse. We call this scheme *safe-shuffle* which allows us to cover both the frontend and backend. Because we perform the shuffling at the leading thread commit which is off the critical path, we do not affect timing-critical components.

Finally, there may be a marketing concern as to what happens when a defect is detected. We have not changed the marketing model. The key point is that the defect exists with or without BlackJack, and both cases will result in the chip being returned to the manufacturer. Without BlackJack, the user will return the chip after the defect causes data corruption. BlackJack prevents this corruption.

In summary, the main contributions of BlackJack are:

- We show that shuffling the trailing thread allows SRT, a soft-error technique, to detect hard errors as well.
- We propose *safe-shuffle;* a novel scheme which allows us to shuffle instructions before they enter the trailing thread while still ensuring correctness. Thus safe-shuffle allows coverage of hard-errors in both the pipeline frontend and backend.
- We show that for a 15% performance degradation over SRT, BlackJack is able to achieve 97% hard error instruction coverage while SRT achieves 34%.

The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 provides background on SRT. Section 4 describes BlackJack. Section 5 describes our experimental methodology, and Section 6 presents our results. We conclude in Section 7.

2 Related Work

There is a large body of past work [12] on error checking logic and error correcting codes (ECC). Although applicable to some modern microarchitectural structures (mostly array memory structures and some data path components) and implemented in some modern processors [2], these techniques cannot cover much of the faster and more complex control-dominated modern microarchitectures. For example, it is hard to build checker logic that checks the issue queue operation every cycle and correctly responds to wakeup and select actions without significantly degrading cycle time.

Because of such deficiencies in error checking logic for modern processors, there has been extensive work in architectural redundancy techniques for soft errors. SRT with recovery (SRTR) [17] extends SRT to detect and recover from soft errors, but still would not provide good hard-error instruction coverage because of lack of spatial diversity. Though primarily targeting soft errors, Redundant Multithreading (RMT) [9] proposes using redundant threads for hard errors. Using a clustered microarchitecture, RMT achieves spatial diversity by executing the redundant threads on different clusters. However, because the frontend of the pipeline is not clustered, the technique does not provide coverage for the frontend, ensuring spatial diversity only after rename. Moreover, RMT can provide backend coverage only because the issue queue is statically segmented among the clusters at design time, so that the leading and trailing instructions can be dispatched to different issue-queue segments. However, a segmented issue queue would incur substantial performance loss in an SMT compared to a conventional unified issue queue. A unified issue queue allows both threads to occupy as much of the issue queue as needed whereas as a segmented issue queue artificially limits each thread to its own segment. Such segmentation defeats SMT's purpose of improving throughput by flexibly sharing the pipeline resources (one of the most important of which is the issue queue) among the threads.

Chip-level redundant threading with recovery (CRTR) [6] proposes a CMP-based solution for soft-error detection and recovery. Despite being designed for soft-error recovery, CRTR would naturally provide good spatial diversity by running the redundant threads on two different cores. However,CRTR requires many values (*every* load and store value and address) to be sent at high rates between the cores. As such, satisfying such high bandwidth demand may not be realistic as it would require deeply-pipelining long, wide buses by introducing numerous latches and buffers which are power-hungry and increase chip area. Furthermore, CRTR forces running one copy each of two different programs on one core (limiting one core to run one copy of only one program would halve execution throughput and is not an option). The two programs may thrash in the core's i-cache, and may contend for pipeline resources increasing the complexity of the OS in ensuring that each program gets its due share of resources. The contented resources may not be visible to the OS, which would amount to not just worse complexity but more uncertainty in the OS. In contrast, SRT runs both copies on the same core, containing each program to one core and avoiding these OS complexity and uncertainty problems.

A recent paper on lifetime reliability [16] discusses the effect on reliability of detecting and disabling defective resources, but provides no such techniques to support such features.

DIVA[1], although intended to catch design bugs, not fabrication bugs, can catch some hard errors. DIVA uses an additional simple pipeline that checks committed instructions. A recent work, [3], uses multiple DIVA checkers to provide on-line diagnosis of failures. Online diagnosis may not be advantageous to the user beyond detection, if degraded operation is not acceptable and defective chips are still to be returned. As proposed in [3], the multiple DIVA checkers and additional area overhead may increase

the likelihood of the chip having a failure. Furthermore, [3] relies on randomization logic in the timing-critical select-map logic for spatial diversity. Finally, to keep DIVA checkers simple, [3] can not cover hard errors in large parts of the pipeline such as the register file.

A recent paper, [5], proposes defect-tolerance techniques for CMP interconnect switches. This work is orthogonal to ours because we target defects in the processor pipeline.

another recent work [18] proposes using standard SMT to run test case applications in the background and evaluates performance overhead. [18] does not rely on redundancy and therefore its coverage depends on the quality, number and frequency of test cases being ran.

Finally, Rescue [11] proposes architectures that allow detected hard errors to be isolated at test time and avoided at run time, but provides no support for hard errors missed at test time.

3 SRT Background

SRT [10] provides a hardware technique for detecting soft errors. SRT uses SMT hardware to allow two copies of a single program, called leading and trailing threads, to be executed concurrently on one SMT core. SRT detects soft errors whenever the corresponding stores in the leading and trailing threads disagree in address or data. Specifically, the leading store waits in the store buffer for the trailing store. Upon successful checking, SRT commits the store to the memory hierarchy. SRT commits register writes, however, in the respective threads without any checking. Because incorrect values propagate through computations and are eventually consumed by stores, checking only stores suffices for soft-error detection.

In SRT, the trailing thread executes behind the leading thread by some specified amount of *slack*. This slack provides two important performance advantages. First, the slack enables the leading branches to be resolved well ahead of the trailing thread, so that branch outcomes are passed to the trailing thread to be used as the trailing thread's prediction, allowing the trailing thread never to mispredict (assuming fault-free operation). Second, the slack enables leading load misses to be resolved well ahead of the trailing thread, so that leading load values can be passed to the trailing thread, allowing the trailing thread never to miss. Thus, only the leading thread accesses the cache. The second advantage also addresses a correctness issue: Duplicating cached loads is problematic because memory locations may be modified by an external agent (e.g., another processor) between the time the leading thread loads a value and the time the trailing thread tries to load the same value. Then, the two threads may diverge if the loads return different data, resulting in loss of redundancy.

Together these two advantages mean that the trailing thread executes far fewer instructions than the leading thread and executes the remaining (only non-speculative) instructions at high IPC. By using SMT, SRT allows much of the trailing thread's execution to be hidden during leading thread stalls. To implement the passing of branch outcomes and load values from the leading thread to the tailing thread, SRT uses a Branch Outcome Queue (BOQ) and a Load Value Queue (LVQ).

4 Hard Error Detection with SMT

The key to detecting hard errors on SMT is ensuring *spatial diversity* — i.e., trailing instructions do not use the same pipeline resources used by the corresponding leading instructions. We note that in an out-of-order pipeline an instruction is processed in one *frontend way* crossing over only at the issue queue to a *backend way* which the instruction uses till write back. Figure 1 shows a diagram of the instruction flow. Once in a frontend way or backend way, an instruction uses resources dedicated to that way. It suffices to ensure that a trailing instruction uses a different frontend way and a different backend way than the leading instruction to ensure that the trailing instruction uses different pipeline resources than the leading instruction. Spatial diversity does not occur naturally in SRT because the trailing thread is an instruction stream mostly-identical to the leading thread, resulting in the same resources being used by most leading-trailing instruction pairs.

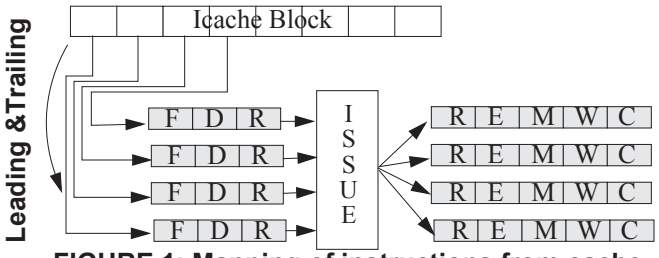

FIGURE 1: Mapping of instructions from cache frontend way and from issue to backend ways

In Section 4.1, we first discuss the problems with a straight-forward but naive approach to enforcing spatial diversity in the execution of the leading and trailing threads. We then continue on in Section 4.2. to describe our *safe-shuffle* which shuffles leading instructions, borrowing dependence information from the leading thread, to produce trailing instructions so that spatial diversity is enforced while maintaining program correctness in the trailing thread. In Section 4.3, we describe execution of the trailing thread. Finally, in Section 4.4, we discuss comparison of the leading and trailing state, and how we verify the dependence information that was passed from the leading thread to the trailing thread for safe-shuffle. This verification is needed so that dependence information corrupted by a hard error and propagated from the leading thread to the trailing thread, does not result in the error going undetected.

4.1 A Naive Approach

A naive approach to forcing the trailing instructions to use different pipeline resources from their leading counterparts, would be to perform some sort of shuffling on the trailing thread. However, there is no convenient point in the pipeline at which the trailing instructions can be shuffled. One option is to do the shuffling before rename, but then the trailing thread will not preserve dependencies and will diverge from the leading thread, resulting in loss of redundancy. A second option, is to shuffle after rename but before dispatch (i.e., insertion into the issue queue as done in [9]), but this introduces two problems: (1) Because both leading and trailing threads are fetched from the I-cache and the instruction location within a cache block does not change in leading and trailing threads, both leading and trailing instructions would exercise the same pipeline-frontend way resulting in zero coverage of the frontend. (2) The issue queue may undo the shuffling and map the trailing instructions to the same execution way as the corresponding leading instruction. A third option is shuffling after issue but this would require updating issue's data structures to reflect the

modified pipeline resource usage (some resources in use upon issue may not be in use upon shuffle, and vice versa). In addition, although there are enough resources to issue some instructions, spatial diversity may sometimes require that fewer instructions be issued and the excess be held back. The issue-queue updating and excess handling would severely complicate the pipeline. The final option is for shuffle to occur in the timing-critical issue. In this option, the select logic has to ensure that there is no excess due to shuffling, in addition to the usual constraints. And the mapping logic has to ensure that spatial diversity is maintained, in addition to the usual constraints. These additional requirements would severely complicate issue.

The following sections contain descriptions in detail of how instructions are fetched out-of-order, shuffled, and checked at commit. This detail should not be misinterpreted as complexity. Many common superscalar techniques (e.g., renaming) would seem complex if described at such fine detail. In addition, BlackJack has been carefully designed that all new hardware is off the critical path. Seemingly simpler options, which we describe above, that can not be moved off the critical path will have severe impact on performance.

4.2 Safe-Shuffle

Because we want to cover both the frontend and backend it is necessary that shuffling be done before the fetch of the trailing thread. Here we address the problem that instruction dependencies, needed to guarantee the correctness of such shuffling, have not yet been determined.

There seems to be a catch-22 that prevents us from covering the frontend: we cannot shuffle the instructions without first knowing the dependencies among the instructions to be shuffled, but we cannot know the dependencies without fetching the instructions in the original program order, yet we cannot fetch the instructions in program order if we want to cover the frontend. We make the key observation that because we are executing the same program redundantly in the leading and trailing threads and because there is a slack between the threads, the leading thread has already determined the dependencies before the trailing execution begins. Accordingly, in safe-shuffle we borrow the dependence information from the leading thread to allow shuffling before the trailing thread is fetched.

We implement safe-shuffle in a two-phase process: In the first phase, we collect information on the leading instructions' independence, rename maps (i.e., logical to physical register maps), and pipeline-resource-usage. In the second phase, we use the independence and resource-usage information to shuffle the leading instructions for producing the trailing thread that is spatially diverse from the leading thread. Shuffling produces the trailing thread in the leading thread's *issue order*. Though issue order preserves true dependencies, it removes false dependencies and overlaps multiple live ranges of the same logical register. Fortunately, the leading rename maps correctly identify the live ranges, allowing the trailing thread to maintain program correctness. We describe the first phase in Section 4.2.1, and the second in Section 4.2.2.

4.2.1 Collecting Independence Information

Our technique relies on the observation that instructions co-issued in the *same* cycle are independent and can be shuffled without causing correctness problems. As such we use the execution of the leading thread to record co-issued instructions, called a *packet*, and also the pipeline resources used by the instructions and their rename maps. We collect this information in a simple queue called the Dependence Trace Queue (DTQ). Each entry contains information for one issued leading instruction and the entries are allocated for all issued leading instructions *in issue order*. The order that entries are allocated *within* a packet can be arbitrary. The instructions collect information during execution and record information at commit.

Leading instructions in a packet are allocated consecutive entries, and the last instruction in the packet has a bit set to demarcate the end of the packet. Because the allocation of DTQ entries need be completed only before writeback, the allocation need not be done in the timing-sensitive select and map logic. When leading instructions are in the pipeline they record, and carry with them until commit, two IDs to identify the pipeline resources that the instructions used. One ID specifies the frontend-way, and the other specifies the backend-way. The instructions also carry the logical to physical register maps for their source and destination operands. Upon commit, the leading instruction records its undecoded instruction, its rename maps, and its frontend and backend IDs in its DTQ entry. Because the DTQ holds only the committed leading instructions (albeit in issue order) which are shuffled to produce the trailing thread, our trailing thread does not execute any misspeculated instructions, as is the case in SRT.

While the leading thread's issue order helps us implement safe-shuffle, we also need the leading thread's program order to keep trailing loads and stores in program order in the load/store queue of the trailing thread's context, and to commit the trailing thread. Because the trailing thread is fetched from the DTQ, which is in issue order and not program order, we also need to record the leading thread's program order. One method for recording this ordering would be to actually allocate trailing thread active list and load/store queue entries in the trailing thread context when each leading instruction commits, which is well before the corresponding trailing instruction is fetched. Such early allocation would mean that idle instructions in the slack would be consuming trailing thread resources and consequently, slack would be limited by the size of the trailing context load/store queue (which is the smaller of active list and load/store queue). Instead, we allocate virtual active list and load/store queue entries at leading thread commit to record the ordering without requiring that instructions in the slack are actually assigned to any trailing thread resources. We store these allocations in the DTQ as well.

Thus, we have borrowed the leading thread's issue order, rename maps, and program order. Later we will describe how we verify the correctness of this borrowed data to ensure that a single error that corrupts both threads identically is still caught.

4.2.2 Using Independence to Shuffle

While the leading thread places its packets in the DTQ, shuffle waits for the packets to reach commit and then shuffles the instructions *within* a packet, one packet at a time, producing shuffled packets that will be fetched by the trailing thread.

To enforce spatial diversity, Shuffle must satisfy the following two constraints. First, when the packet is fetched by the trailing thread, each instruction in the packet must map to a different frontend than was used by the corresponding leading instruction. Sec-

ond, if *all* the instructions in the packet are co-issued together in the same cycle in the trailing thread and *no other* (leading or trailing) instruction is co-issued (Section 4.3.2 describes why these conditions may not be met and what happens then), each trailing instruction must be issued to a different backend than was used by the corresponding leading instruction. Therefore, shuffle cannot be random, must be aware of the policies used by both fetch and instruction map, but can work with any policy as long as the policy is deterministic.

We assume the following policies, which afford the most straightforward implementation and are consequently the most-commonly used: direct mapping policy for fetch where the first instruction in fetch order goes to first frontend way, the second instruction to the second frontend way and so on, and oldest-first mapping policy for instruction select and map, where the oldest instruction goes to the first free backend way that matches the instructions' type (e.g., ALU, memory, or branch type), the second oldest instruction to the second matching backend way and so on.

With the above policies, if shuffle produces a (shuffled) packet whose instructions are fetched and co-issued meeting the second constraint's conditions, then the packet's first instruction is guaranteed to use the first frontend way and the first matching backend way, the second instruction is guaranteed to use the second frontend way and the first of the remaining matching backend ways and so on. Consequently, to meet the above two constraints for spatial diversity, Shuffle need only determine an ordering so that each trailing instruction in a packet uses a different frontend way and different backend way than the ways used by the corresponding leading instruction.

To prevent issue and fetch from undoing our shuffle when there are fewer instructions in a packet than the issue width, we allow shuffle to insert NOPs. We require that the NOPs remain in the pipeline through writeback (i.e., each NOP occupies a frontend and backend way and an issue-queue slot).

We use the following simple, greedy algorithm which works well in most cases. The algorithm shuffles an input packet into an output packet. Each instruction in the input packet (in any arbitrary order) grabs the first free output-packet slot that is spatially diverse from the corresponding leading instruction (i.e., does not map to either the frontend or backend way used by the corresponding leading instruction). At a given output-packet slot, the instruction's new frontend way is easy to determine whereas the backend way is a little more involved: The instruction's new frontend way is the output-packet slot number, and the new backend way is the number of instructions in the packet that have already been allocated to lower slot numbers and use the same type of backend way. The given output-packet slot is acceptable if the new ways are spatially diverse from the leading instructions. If allocation of an instruction passes over an empty output-packet slot that the instruction cannot use because the slot maps to the corresponding leading instruction's frontend or backend way, the instruction puts an NOP in the slot and marks the slot with the instruction's type.

The process continues with each instruction in the input packet attempting to grab a free output-packet slot. If an instruction finds an acceptable slot containing a NOP marked with a matching instruction type, the instruction can claim the slot and replace the NOP. This replacement is what allows two instructions of the same type to swap backend ways as depicted in Figure 2. NOPs created by one type cannot be replaced by an instruction of another type

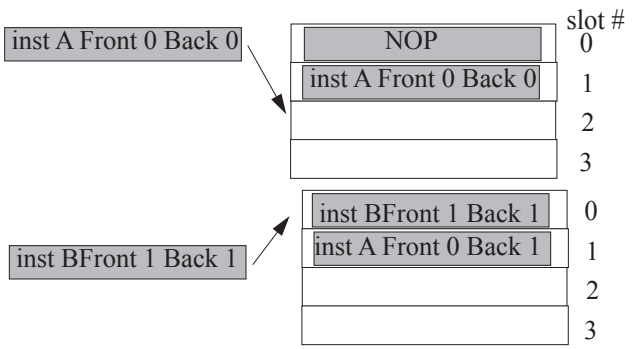

FIGURE 2: Safe-shuffle swapping resource allocations of two like instructions.

because doing so may require correcting (decrementing, to be precise) the backend mappings of some previously-allocated instructions (those whose backend way is larger than that of the NOP being replaced). Such correction would not fit the greedy nature of the algorithm and would complicate the algorithm. There will never be fewer output-packet slots than instructions if the issue width matches the frontend width. However, there may be slots with NOPs that cannot be replaced by later instructions either because the slot is not spatially diverse from the corresponding leading instruction or because the NOPs are allocated by a different instruction type. If an instruction cannot find a slot, the output packet is ended and the remaining instructions of the input packet start a new output packet (the input packet gets broken into two or more output packets). Breaking an input packet reduces parallelism and its impact on performance is discussed in Section 6.

Because of SRT's long slack, there are many cycles between commit of the leading thread and fetch by the trailing thread. Consequently, there is ample time to perform the shuffling, which, if needed, may be pipelined over multiple cycles.

The output packets are placed in the trailing thread's fetch queue. Because the input packets come from the DTQ and shuffling shuffles only the instructions *within* an input packet, the fetch queue inherits DTQ's leading-thread issue order *across* packets.

4.3 Trailing Thread Execution

4.3.1 Frontend

The trailing thread fetches the shuffled packets from its fetch queue according to SRT's slack (Section 3) as depicted in Figure 3. When given fetch bandwidth, the trailing thread fetches one packet each cycle. Because of the direct mapping of instruction fetch to frontend ways, each instruction maps to the frontend way as intended by safe-shuffle and continues on to be decoded on different hardware than was used to decode the corresponding leading instruction.

It is important to note that the trailing-thread fetch is limited to fetch only one packet per cycle even if the packet is smaller than the fetch width of the pipeline. If multiple packets were fetched in one cycle, then the mapping of the packets' instructions to the frontend ways may be different than that intended by safe-shuffle, resulting in loss of spatial diversity. While this restriction ensures frontend spatial diversity, it potentially lowers the trailing thread's fetch bandwidth, degrading performance.

The trailing thread is fetched in leading thread's issue order, which while preserving true dependencies removes false depen-

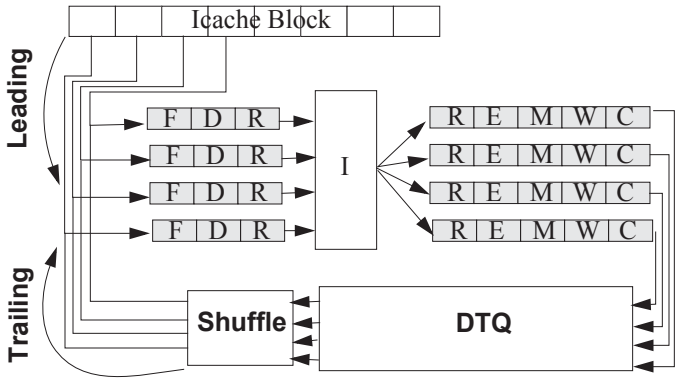

FIGURE 3: BlackJack Execution Flow.

dencies and overlaps multiple live ranges of the same logical register. Because of this overlap, the trailing thread renamer cannot correctly connect consumers to their producers by looking only at the *logical* registers of the instructions. Fortunately, leading thread rename maps can make this connection. Accordingly, the trailing thread renamer uses the leading thread's physical registers (sources and destination) as input, instead of the usual logical registers. That is, the trailing thread renamer renames the renamed leading instructions! Though this double renaming allows us to cover the frontend while preserving program correctness, the downside is that our rename tables have more rows because there are more physical registers than logical registers.

While there is also an issue with determining in rename which physical register each instruction should free, the actual freeing is done at commit, and we address the issue there.

Because the trailing thread fetches from its fetch queue without any branch prediction, the fact that branches appear in issue order (i.e., out of program order) in the trailing thread does not matter. Branches simply flow through the pipeline and execute. We use their execution to verify the trailing-thread program order which is borrowed from the leading thread.

Once decoded we use the virtual active list and load/store queue specifiers in the DTQ (Section 4.2.1) to allocate corresponding physical entries for each fetched instruction. We translate these virtual specifiers to physical load/store queue and active list entries by keeping the virtual to physical mapping for the head of the physical structures as a reference. Any virtual index which is *j* greater than the head's virtual index is allocated a physical index *j* away from the head. If *j* is larger than the size of the structure, then the frontend is stalled till there is space in the structure. Combined with out-of-order fetch, this allocation means that instructions that are fetched earlier than their commit order will allocate physical entries leaving the appropriate number of empty slots ahead of them.

4.3.2 Dispatch/Issue

After rename, register tags correctly encode instruction dependencies and instructions move on to be dispatched into the issue queue. Ideally, the constraints assumed by shuffle will be maintained at issue. Instruction packets will issue complete and alone, and each trailing thread instruction will issue to a different backend than it used in the leading thread.

Because we leave the issue queue and issue policy unmodified, the issue queue may undo our shuffling by breaking up the shuffled packets in the trailing stream and introducing unrelated leading and trailing instructions into the packets. Doing so would violate the conditions of safe-shuffle's second constraint, specified in Section 4.2.2, that the trailing packet should co-issue as a whole and no other unrelated instructions should co-issue with the packet, and may result in loss of spatial diversity and reduced coverage. The undoing occurs due to two types of *interference,* leading-trailing interference and trailing-trailing interference. However, because trailing thread is fetched in leading thread's issue order, both types of interference are rare. We explain the details next.

Leading-trailing interference occurs when leading instructions co-issue in the same cycle with trailing instructions. Leading instructions issuing with trailing instructions causes trailing packets to break apart, some parts of the packets issuing with leading instructions, and some issuing alone (or with other trailing-thread instructions). Fortunately, in SRT and BlackJack, dependencies naturally cause issue from each thread to be bursty, either only leading instructions or only trailing instructions are issued in most cycles though both threads are present in the issue queue. We quantify this bursty behavior in Section 6. While this bursty issue behavior makes leading-trailing interference rare, another reason makes the interference even rarer. The trailing thread is a high-IPC thread fetched in dependence order with no branch mispredictions or cache misses (see Section 3), while the leading-thread is a lower-IPC thread fetched in program order and the issue policy is oldest-first. Consequently, the trailing instructions issue out of the issue queue almost as soon as they are inserted, while the leading instructions take multiple cycles to issue. This difference in occupancy means that a trailing instruction has a slim chance of becoming older than any leading instruction in the issue queue. As such, the trailing instructions almost always have lower priority than any leading instructions that are ready. Therefore, trailing instructions cannot disturb the leading instructions ready to issue.

The lack of leading-trailing interference results in two distinct benefits: (1) Leading instructions rarely interfere with trailing packets. (2) Conversely, trailing instructions rarely interfere with leading instructions. If this converse were not true, the leading thread's backend resource usage information collected by safe-shuffle could be due to the leading thread either issuing in isolation or co-issuing with some trailing instructions. While the first case causes no problems for safe-shuffle, the second case implies that the leading packets are narrower than the issue width. Narrow leading packets force safe-shuffle either to put many NOPS in the shuffled packets to ensure spatial diversity or to use a shuffling algorithm more complicated than our simple, greedy one to try to combine multiple leading packets into one trailing packet.

Similar to leading-trailing interference, trailing-trailing interference is also rare. Trailing-trailing interference occurs when one packet co-issues with instructions from another packet. It may seem that it would be difficult to prevent trailing-trailing interference especially if the issue queue is to remain unmodified. However, the fact that the trailing stream is ordered in dependence order reduces trailing-trailing interference. As mentioned before, this order implies that trailing instructions enter and leave the issue queue without much delay. This quick departure combined with the fact that the trailing thread fetches only one packet per cycle (Section 4.3.1) implies that most often only one trailing packet resides in the issue queue at any given time, giving little opportunity for the issue queue to introduce trailing-trailing inter-

ference.

Nevertheless, the issue queue may occasionally have more than one co-resident trailing packet due either to long-latency trailing instructions backing up in the issue queue or to leading-trailing interference. In such cases, the issue queue may wake up later packets in an order different than the trailing dispatch order (which is the same as the leading issue order). This different order occurs because the leading issue order is based on the latencies seen by the leading thread which are different from those seen by the trailing thread. The leading thread sees cache miss latencies which are hidden completely from the trailing thread due to the slack (see Section 3). Trailing loads access only the LVQ and not the cache hierarchy, and as such, may complete earlier than the dispatch order expects, creating opportunity for instructions in later packets to be woken up earlier and to be co-issued with earlier packets.

Both types of interference being rare implies that most often the trailing packets are co-issued as a whole and not co-issued with unrelated leading or trailing instructions. Consequently, the conditions of safe-shuffle's second constraint, specified in Section 4.2.2, are met, and spatial diversity is maintained in the common case.

This way of maintaining spatial diversity without modifying the issue queue does come at the price of some performance loss. Leading-trailing interference is reduced by trailing thread's issue-order fetch which does not negatively impact performance. However, preventing trailing-trailing interference relies on fetching one packet per cycle which limits trailing thread's fetch bandwidth and negatively impacts performance, as discussed in Section 4.3.1.

4.4 Commit and Correctness Check

After passing through the remainder of the backend pipeline, trailing thread instructions complete and wait for commit in program order. Because register dependencies are preserved, commit is in program order, and both threads maintain the ordering in the load/store queue, the result of each trailing instruction and its leading counterpart will be in agreement when there are no errors.

BlackJack checks for agreement by comparing the trailing stores against corresponding leading stores waiting in the store buffer in the same way as SRT (Section 3). However, safe-shuffle borrows dependence information from the leading thread to produce the trailing thread. Therefore, we must perform additional checks so that corruption of this information due to a hard error does not cause identical mistakes in the two threads, allowing the error to go undetected. We note that this additional check is in the same vein as SRT's branch outcomes. SRT passes leading branch outcomes to the trailing thread which uses the outcomes as prediction and not as result. Trailing branch execution validates the prediction which forms a separate check for the correctness of the outcomes.

The borrowed information includes dependence information in the form of leading issue order and leading rename maps, and leading program order.

To check the dependence information, we observe that we need to ensure that the trailing thread maintains the dependencies in the original program. To implement this check, we use a second rename table at trailing commit, in a slightly different fashion than normal which is described below.

As trailing instructions commit in program order, they use their logical source registers to look up their physical source registers in the second table. While normally a new physical register is allocated for the destination operand, the trailing instructions already have their physical destination register which they use to update the table as the new mapping for the logical destination register. We compare the looked-up physical source registers against the physical source registers that were provided by the first trailing rename and used by the instructions in trailing execution. A disagreement signifies either a leading-thread error that propagated to the trailing thread through safe-shuffle, or a trailing-thread error (including this dependence check). Because the second rename table is used only by the trailing thread and never by the leading thread, we maintain spatial diversity.

Because the first trailing rename is done out of program order, we do not free physical registers as determined by the first renamer (Section 4.3.1). Instead, we free the physical register that the second renamer reports as the previous mapping of the destination register because the second rename is in program order. Using the second renamer ensures that freeing reflects program order, not dependence order.

Finally, the trailing thread does not fetch its own instructions but obtains the instructions committed by the leading thread. Therefore, program-order errors in the leading thread could cause incorrect instructions to be fetched, instructions to be dropped or instructions to be added, and the trailing thread will duplicate these errors. To check for this kind of error we require an additional simple check at commit that checks that the program counters of the committed instructions are correct. If a committed instruction is a taken branch, the program counter of the next committed instruction should be the branch target; otherwise the program counter of the next instruction should be the program counter of the previous instruction plus the size of an instruction.

4.5 Coverage

While BlackJack ensures spatial diversity in the combinational logic present in the frontend and backend, spatial diversity in memory structures, specifically rename tables, load/store queue, active list, and the issue queue, need some explanation.

Because each SMT context has its own rename tables, load/store queue and active list, spatial diversity in these structures is inherent. An error in the leading thread's program order (e.g., omitted instructions) propagating to the trailing thread despite this spatial diversity is caught by our program-order checks as described in Section 4.4. Spatial diversity from having per-context structures ensures that any remaining errors are caught by disagreeing results of the two threads.

The issue queue may seem to be more of a problem because it is shared by the two threads and can not be spatially diverse. Because BlackJack ensures spatial diversity in the backend spatial diversity ends up being maintained in most of the issue queue hardware. Each backend way has a broadcast line to broadcast to its dependents. As such, if the backend way is assured to be spatially diverse so are the broadcast wires. Furthermore, each broadcast wire connects to a comparator in each entry. Therefore, by using a different broadcast wire, a trailing instruction is woken up by a different comparator than the corresponding leading instruction. The remaining concern is the spatial diversity of the issue queue entries themselves. We point out here that the function of the issue queue is simply to obey dependencies and find a valid issue order. As such, any issue queue failure that causes an invalid

issue order (even if it affects the order of both threads in the same way) is detected by the dependence check as described in Section 4.4. Consequently, we do not need to ensure spatial diversity in the issue queue entries.

There is a vulnerability in the issue queue's payload RAM which holds instruction payload while the instructions are in the issue queue. It is possible that an entry in the payload RAM corrupts bits in some deterministic way, and this entry is used by both copies of an instruction leading to identical incorrect results in the threads. There are many ways to address this vulnerability. We could additionally consider payload entry allocation policy in safe-shuffle, or insert NOPs at trailing dispatch if an allocation attempts to violate spatial diversity. But because the payload RAM is just a small RAM, having separate payload RAMs for the two threads is probably the simplest solution. Two payload RAMs obviously provide sufficient spatial diversity.

Although safe-shuffle ensures spatial diversity in the backend, including the register file ports, we do not explicitly cover the possibility that both copies of an instruction may be allocated the same physical register. However, because the two threads maintain their own register state and compete for free registers, allocation of identical registers is unlikely and not directly tied to program patterns and architectural policies and causes only a negligible loss in coverage.

Finally, there may also be a concern that while BlackJack covers single hard errors much as SRT covers single soft errors, multiple errors are more likely in the case of hard errors than soft errors. BlackJack can be effective for multiple uncorrelated errors. It is true that certain structures may be more prone than others to multiple correlated errors, and in a highly-defective chip many like structures (e.g., all RAM structures or all CAM structures) may be defective. However, we do not target this class of defective chips. Such chips will be eliminated early in the testing process. Our technique is a run-time technique that catches hard errors that were small enough to be missed during test or only became active in the field. We target chips that seem error-free but silently corrupt data.

4.6 Complexity

Blackjack exploits pre-existing ordering information from the leading thread issue order to permit shuffling and reordered fetch. Because this ordering information is available from the leading thread, it can be used far in advance of trailing thread fetch (within the leading/trailing slack) thus clearly placing shuffle off the critical path with little chance of complicating current structures or degrading cycle time.

5 Methodology

We modify SimpleScalar 3.2b[4] to simulate SRT and BlackJack. We use the parameters listed in Table 1. Note that we use two integer multipliers and two integer dividers in both SRT and BlackJack, because without two of each type of resource, spatial diversity is not possible.

We evaluate the hard error instruction coverage of SRT and BlackJack. SRT is not a hard-error technique but provides some hard error coverage due to accidental spatial diversity. We use SRT as a reference point.

We measure hard error instruction coverage as the fraction of instruction pairs that execute on spatially diverse hardware multiplied by the core area used by the pair. Because instruction pairs can be spatially diverse for part of the execution, and use identical resources in others, we allow for partial coverage of single instructions. We make the simplifying assumption that equal chip areas have equal probability of hard error. We use the HotSpot [15] area model to estimate the core area that remains vulnerable to hard defects under redundant threading. We divide the area into three classes: issue, frontend and backend. We give SRT the benefit of assuming full coverage of hard errors in the issue queue although SRT can cover hard errors in the issue queue only by chance. BlackJack, on the other hand provides coverage as described in Section 4.4. Of the remainder of the core, 34% is accessed by instructions in the frontend pipeline stages. The remaining 66% is accessed in the backend.

Table 1: Processor Parameters

Out-of-order issue	4 instructions/cycle
Active list	512 entries (64-entry LSQ)
Issue queue	32-entries
Caches	64KB 4-way 2-cycle L1s (2 ports); 2M 8-way unified L2
Memory	350 cycles
Int ALUs	4 int ALUs, 2 int multipliers,
FP ALUs	2 FP ALUs, 2 FP multipliers
Store Buffer	64 entries
LVQ	128 entries
BOQ	96 entries
Slack	256 instructions
DTQ	1024 instructions

We run 16 SPEC2000 benchmarks, fast-forward to the early-simpoint specified by [14], and then run 100 million instructions.

6 Results

In this section we present our results for coverage and performance for SRT and BlackJack. First, we discuss the hard error coverage provided by BlackJack and then move on to discuss the performance impact.

6.1 Instruction Coverage

Figure 4a and b show hard error coverage achieved by SRT and BlackJack. SRT is shown by white bars, and BlackJack by black bars. Figure 4a shows hard error instruction coverage of the entire processor, including frontend and backend. As described in Section 4.1, execution in which frontend ways is determined solely on the instruction's cache block location and thus SRT has zero frontend spatial diversity. Because BlackJack deterministically places instructions so they map to spatially diverse frontends in the leading thread, BlackJack achieves 100% spatial diversity in the frontend. Figure 4b shows instruction coverage only of the backends which is dependent on timing and resource availability.

From Figure 4a, we see that SRT achieves limited spatial diversity and hence provides modest coverage of hard errors. On average SRT provides 34% coverage of hard errors. SRT's worst coverage of 25% occurs in sixtrack and its best coverage in vortex is at 41%. BlackJack on the other hand covers 97% of hard errors on average, with its lowest of 94% occurring in bzip and its high-

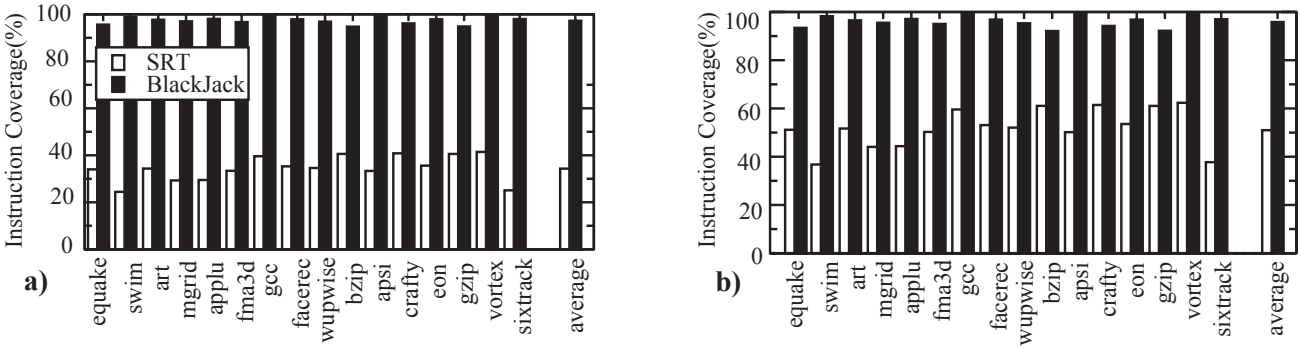

FIGURE 4: Instruction coverage of SRTand BlackJack in a) entire pipeline and b) backend only.

est occurring in vortex at 99%.

To help understand the program behaviors that reduce BlackJack's coverage we breakdown the coverage-reducing interference into leading-trailing and trailing-trailing (described in Section 4.3) and present them in Figure 5. Training-trailing is represented by white bars, and leading-trailing by black bars. The y-axis is the percentage of issue cycles where the specified type of interference causes instructions to violate spatial diversity. On average across all benchmarks, 0.5% of issue cycles lose coverage due to trailing-trailing interference, and 2.3% lose coverage due to leading-trailing interference.

From Figure 5 we can see that one of the lowest covered benchmarks, equake (95.6%), suffers from both trailing-trailing interference (1.5%) and leading-trailing interference (2.5%). Its trailing-trailing result is notable because it is three times greater than the average across all benchmarks. This elevated trailing-trailing interference is a consequence of equake's low IPC; equake is the slowest benchmark. Trailing-trailing interference is inherent to the low IPC because with slow benchmarks, fetching of the trailing thread outpaces issue and allows trailing instructions to build up in the issue queue. With larger trailing thread issue queue occupancy there is a greater chance for trailing instruction to issue out of their fetch order and interfere with and lose diversity. In fast-paced benchmarks, issue more closely matches fetch and there is little opportunity for trailing interference. The difficulty of interference (both leading-trailing and trailing-trailing) is made worse because equake is an FP application. Because our machine has only 2 FP ALUs, and 2 FP multipliers, once a leading instruction has issued, unless the trailing thread goes to the other equivalent unit there will be loss of coverage. As such, benchmarks that heavily use resources for which only a few copies exist are inherently more susceptible to interference. Benchmarks that use resources for which multiple copies exist are less sensitive. In benchmarks heavily dependant on basic integer operations (such as Vortex) trailing instructions must avoid only a single backend way and the remaining three ways are spatially diverse. In such cases, interference still has a good chance of sending the instruction to a favorable (although unintended) backend way.

To help explain the high leading-trailing interference in the higher-IPC benchmarks, gzip, crafty, and bzip, (which are to the right in Figure 5) we additionally provide Figure 6. Figure 6 plots the percentage of issue cycles, in which only one context is issued in. Recall from Section 4.3.2, that the bursty nature of instruction issue prevents leading-trailing interference. Figure 6 quantifies this burstiness. While the average across all benchmarks is 70% gzip, crafty, and bzip range from 54% to 63%. In fact, gzip is the lowest of all benchmarks at 54%. The fact that issue is more likely to issue from both contexts in the same cycle naturally implies there will be more interference and greater loss of coverage. Figure 5 reinforces this fact, showing that both gzip and bzip have the highest leading-trailing interference at 7.0% and 5.6% respectively.

6.2 Performance

Figure 7 plots the performance of SRT and BlackJack with no shuffle (BlackJack-NS), and BlackJack. As we explain later, BlackJack-NS helps understand the components of BlackJack's performance. All are normalized to non-fault-tolerant single thread performance. Benchmarks are plotted from left to right in the order of increasing IPC. White bars represent SRT, gray bars BlackJack-NS and black bars BlackJack. In general SRT and therefore also BlackJack show larger performance degradation with higher-IPC benchmarks, because there are less idle cycles to hide the execution of the redundant thread. On average across all benchmarks, compared to non-fault-tolerant single-thread, SRT has a slowdown of 21%. and BlackJack has a slowdown of 33%.

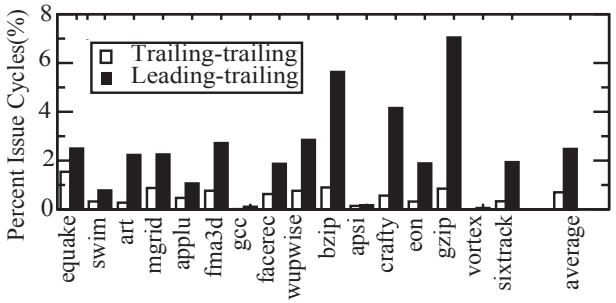

FIGURE 5: Percent of issue cycles with trailing-trailing and leading-trailing interference.

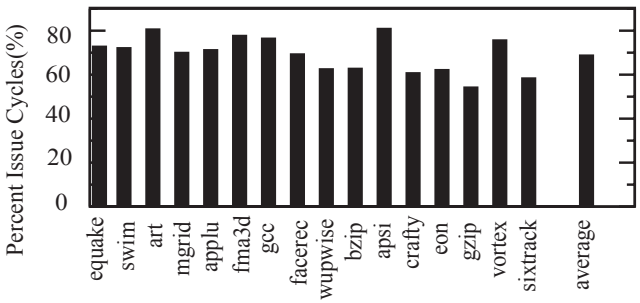

FIGURE 6: Percentage of issue cycles when all instructions issued are from the same context.

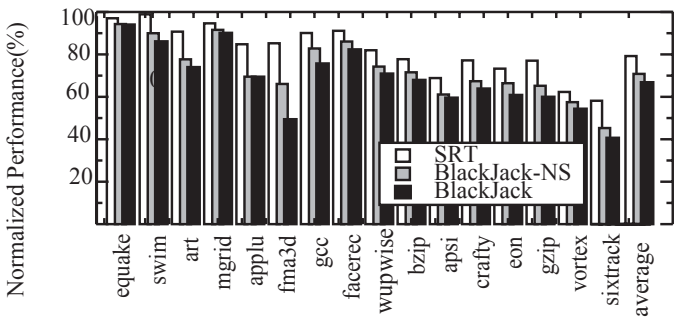

FIGURE 7: Performance of SRT, BlackJack with no shuffle (BlackJack-NS) and Blackjack.

These slowdowns represent a 15% slowdown for BlackJack beyond SRT.

The difference between BlackJack-NS and BlackJack represents the performance degradation due to the cases where safe-shuffle's greedy algorithm sometimes splits packets for high coverage. BlackJack-NS never splits packets (and never shuffles) and so it is able to issue a greater number of instructions per cycle (but has low coverage). Averaged across all benchmarks, adding shuffle in BlackJack results in a 5% slowdown over BlackJack-NS. If BlackJack were to have an ideal shuffle algorithm that provided good coverage without ever splitting packets, BlackJack would incur a 10% slowdown over SRT. Better shuffle algorithms may be able to approach this 10% slowdown.

This remaining difference between BlackJack-NS and SRT is due to BlackJack's policy of fetching only a single packet per cycle. This policy prevents co-issue of multiple trailing packets. This prevention is a simple method for reducing trailing-trailing interference (as seen in Figure 5) but comes at the cost of performance. Trailing-trailing interference is often good for performance, allowing two or more small packets that took multiple cycles to issue in the leading thread to be combined into one large packet issuing in a single cycle. Combining two such packets while maintaining spatial diversity requires that the packets are known to be independent. For simplicity, BlackJack's shuffle algorithm and fetch policy assumes all leading instruction not co-issued in the same cycle may have dependencies. It is important to note that all information about the dependencies between packets is available for shuffle to borrow from the leading thread. Hence it is possible for a more complex shuffle algorithms to use this additional information to close the gap between BlackJack and SRT.

7 Conclusions

This paper presents BlackJack; a microarchitecture that addresses the increasing difficulty of test. With smaller and faster devices, tolerance for errors are shrinking and devices may act correctly under certain condition and not under others. As such, hard errors may exist but are only exercised by very specific machine state and signal pathways. In addition new complications with burn-in may mean that latent hard errors are not exposed in the fab and reach the customer before becoming active.

BlackJack provides a safety net that detects hard errors (in addition to soft errors) that are exposed by a program at runtime.

Averaged across all benchmarks, BlackJacks incurs a 15% performance penalty when compared to SRT. In exchange for this degradation, BlackJack provides 97% instruction coverage of pipline hard errors compared to SRT's 34%.

8 References

[1] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, 1999.

[2] D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd. Fault-tolerant design of the IBM pSeries 690 system using the POWER4 processor technology. *IBM Journal of Research and Development*, 46(1), 2002.

[3] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–208, 2005.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, 1996.

[5] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: A defect-tolerant cmp switch architecture. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, February 2006.

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

[7] W. Maly, A. Gattiker, T. Zanon, T. Vogels, R. D. Blanton, and T. Storey. Deformations of IC structure in test and yield learning. In *International Test Conference (ITC)*, 2003.

[8] M. Meterelliyoz, H. Mahmoodi, and K. Roy. A leakage control system for thermal stability during burn-in test. In *International Test Conference (ITC)*, November 2005.

[9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, 2002.

[10] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36. ACM Press, 2000.

[11] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 160–171, 2005.

[12] F. F. Sellers, M. yue Hsiao, and L. W. Bearnson. *Error Detecting Logic for Digital Computers*. McGraw-Hill, 1968.

[13] O. Semenov, A. Vassighi, M. Sachdev, A. Keshavarzi, and C. F. Hawkins. Effect of cmos technology scaling on thermal management during burn-in. *IEEE Transactions on Semiconductor Manufacturing*, 16(4), 2003.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[15] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[16] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 520–531, 2005.

[17] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 87–98, 2002.

[18] E. Weglarz, K. Saluja, and T. M. Mak. Testing of hard faults in simultaneous multi-threaded processors. In *Proceeding of the 10th IEEE International On-Line Testing Symposium*, 2004.