SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading

Angshuman Parashar[†] Sudhanva Gurumurthi[‡] Anand Sivasubramaniam[†]

[†]Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{parashar, anand}@cse.psu.edu

[‡]Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
gurumurthi@cs.virginia.edu

Abstract

Transient faults are expected a be a major design consideration in future microprocessors. Recent proposals for transient fault detection in processor cores have revolved around the idea of redundant threading, which involves redundant execution of a program across multiple execution contexts. This paper presents a new approach to redundant threading by bringing together the concepts of slice-level execution and value and control-flow locality into a novel partial redundant threading mechanism called *SlicK*.

The purpose of redundant execution is to check the integrity of the outputs propagating out of the core (typically through stores). SlicK implements redundancy at the granularity of backward-slices of these output instructions and exploits value and control-flow locality to avoid redundantly executing slices that lead to predictable outputs, thereby avoiding redundant execution of a significant fraction of instructions while maintaining extremely low vulnerabilities for critical processor structures.

We propose the microarchitecture of a backward-slice extractor called SliceEM that is able to identify backward slices without interrupting the instruction flow, and show how this extractor and a set of predictors can be integrated into a redundant threading mechanism to form SlicK. Detailed simulations with SPEC CPU2000 benchmarks show that SlicK can provide around 10.2% performance improvement over a well known redundant threading mechanism, buying back over 50% of the loss suffered due to redundant execution. SlicK can keep the Architectural Vulnerability Factors of processor structures to typically 0%-2%. More importantly, SlicK's slice-based mechanisms provide future opportunities for exploring interesting points in the performance-reliability design space based on market segment needs.

Categories and Subject Descriptors C.1.0 [*Processor Architectures*]: General

General Terms Reliability, Performance

Keywords Transient Faults, Redundant Threading, Backward Slice Extraction, Microarchitecture

1. Introduction

Technological trends point to the growing occurrence of transient faults in the future. In addition to protecting storage structures and communication paths, there is an increasing need to protect random logic within processors [23]. This has led to proposals to incorporate redundancy in processor pipelines [20, 29]. Contrary to earlier techniques that implemented complete hardware duplication such as lock-step processors [10], these recent proposals try to leverage existing microarchitectural resources, thus making them more cost-effective for a diverse set of market segments. Further, it is not necessary to provide complete fault coverage for all deployments, and the trade-offs between performance, implementation cost/complexity and fault coverage offer a rich space of operating points for different market segments.

Many of these recent proposals are based on the idea of Redundant Multithreading (RMT), where the instruction stream is redundantly executed on multiple execution contexts, with a subsequent comparison between the outputs of the two streams (the stores to the memory system in particular) to verify their integrity [20]. Implementations of the concept have been proposed for Simultaneous Multithreading (SMT) processors as well as Chip Multiprocessors (CMP). Contention for core resources (bandwidth and storage) between the multiple redundant threads leads to significant performance degradation relative to non-redundant execution for most of these implementations. In particular, previous studies have reported 20%-30% performance loss for SMT implementations.

Recent studies [7, 18] have attempted to improve redundant threading performance by reducing the resource contention between the two threads. The key idea behind these approaches is the notion of "partial redundant threading" where a subset of instructions of the redundant stream are executed, thereby reducing the resource pressure. Policies used for selecting instructions for redundant execution define the performance and reliability characteristics of these approaches.

The problem with the partial threading mechanisms proposed in these alternatives is twofold.

- The solutions are primarily narrow regions of operation in the performance-reliability design space and do not provide any convenient mechanism to explore a range of design choices.
- Maintaining correct dataflow for the redundant thread is a problem when subsets of instructions are arbitrarily picked for redundant execution. Copying state across threads is an option, but this is an expensive process with the potential to create redundancy violations.

With the intent to address these two issues, we present in this paper a novel partial redundant threading paradigm that is based on two main ideas, (i) Slice-based redundant execution, and (ii) Value and Control-flow Locality exploitation.

A redundant threading system can be viewed as a primary thread generating outputs – Stores – that emanate from the processor, and a redundant thread verifying the integrity of these outputs. The redundant thread can be further envisioned not as a sequence of dynamic instructions, but as intertwined dependency chains or slices

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

of instructions that ultimately lead up to these Stores. If a partial set of instructions needs to be chosen for redundant execution, then it makes sense to do this at the granularity of such *backward slices* of store instructions. This avoids expensive redundancy-violating techniques to copy state across threads. We propose the design of a simple and efficient backward slice extractor that is able to identify the set of instructions that lie on the backward slices of a selective set of stores, thereby enabling partial threading at the slice granularity.

With this mechanism in place, we next propose a policy for selecting slices for redundant execution. We use results from the extensively researched area of value locality for this purpose. Of particular interest to us is the fact that store addresses and data have been shown to exhibit good predictability [12]. We use predictors to verify store addresses and data, and redundantly execute only those slices where predictions fail. Thus, predictability/locality is itself being exploited as a redundancy mechanism for the non-selected slices. In addition to stores, we also track backward-slices of branches and use branch predictability to avoid redundant execution of certain slices. We show that even with very simple predictor configurations, a substantial number of slices can be eliminated while maintaining an extremely low error vulnerability.

We call our design that incorporates these two ideas **SlicK** (short for Slice-Kill), and present the detailed implementation and evaluation of SlicK on a Simultaneously and Redundantly Threaded (SRT) processor baseline. We evaluate the performance of SlicK on a cycle-accurate simulator using all applications from the SPEC CPU2000 suite. We evaluate the error coverage of our design using the Architectural Vulnerability Factor (AVF) [17] approach.

SlicK's locality-triggered slice-based execution paradigm enables it to:

- maintain an AVF of 0%-2% for critical processor structures while performing 10.2% better than SRT, buying back over 50% of the performance loss incurred due to redundant threading;
- achieve this with an efficient set of structures that are simple to implement and off the processor's critical paths;
- provide a useful baseline for further exploration (via *policies*) of the performance-reliability design space by providing the basic *mechanisms* for selective elimination of slices.

The next section covers related work. Detailed design of the SlicK microarchitecture is discussed in section 3. Section 4 presents an analysis of fault coverage of our mechanism, and how it can be quantified. Performance and coverage results are given in section 5. Finally, section 6 summarizes the contributions of this paper.

2. Related Work

Early work in developing highly resilient fault-tolerant systems used hardware duplication to implement lockstepping, as in the HP NonStop Himalaya [10] and IBM G5 [24]. The DIVA architecture [1] takes a slightly different approach wherein a special checker processor is used at the commit stage of the main processor core's pipeline to verify the correctness of the instructions being committed. Our work is more related to the emerging area of cost-effective transient fault detection/recovery techniques using redundant threading [16, 20, 29]. Sharing of processor resources between the threads has been shown to impact performance. There have been studies on efficiently sharing the instruction queue and reorder buffer capacities [26] and appropriately staggering the redundant threads [19] to minimize performance loss due to redundant execution.

Our previous work [18] shows that bandwidth pressure on the processor's functional units in a redundant threading system can be alleviated using instruction reuse (a form of locality), though this is done at the granularity of individual instructions. Fault coverage is the primary metric of concern in this work, with reuse being used to obtain as much performance as possible. Another way of approaching this problem is to treat performance as the primary metric with redundancy being incorporated opportunistically when it does not interfere with performance, as in [7]. Such opportunistic usage of redundancy provides better fault coverage than single-threaded execution but may not necessarily fall within the reliability budgets of stringent market segments. High vulnerabilities of processor structures have been observed with this approach.

Other works that attempt to adapt locality principles to reliability include [9], which observes that soft errors often manifest themselves as mispredictions in warmed-up predictors, and uses the misprediction recovery interval to perform more extensive error detection checks. In [30], the authors exploit the fact that soft errors could result in a range of observable symptoms such as ISA exceptions and branch mispredictions, and use these symptoms to trigger a rollback and re-execution from a previously stored checkpoint.

Slipstream [27, 11] is a performance-enhancing technique that runs a speculative prefetch A-thread ahead of the main R-thread. Instructions that are detected to be *ineffectual* (i.e., not required for correct forward progress) are removed from the A-thread. Slipstream achieves a certain amount of fault-tolerance due to the redundant computation being performed by the two threads. The "ineffectuality" of instructions in the A-thread is in fact a result of the locality that exists in the instruction stream.

SlicK differs from all of these approaches in its explicit use of Predictors to provide redundancy for entities that exit the Sphere of Replication: Stores. Our slice-based microarchitecture designed around this principle achieves significant performance benefits with extremely low vulnerabilities and provides an attractive baseline for further exploration of the performance-reliability design space.

Forward and backward-slice extraction and associated optimizations have been explored extensively in the past for a variety of applications [5, 6, 15, 21, 27, 32], both in hardware and software.

3. SlicK

We begin this section with a review of the concept of redundant threading and one of its previously proposed implementations called Simultaneous Redundant Threading (SRT).

3.1 Baseline SRT Microarchitecture

Central to any form of Redundant Multithreading is the concept of the Sphere of Replication (SoR). All components within the SoR are protected via redundant execution, and those outside it must be protected by other means such as spatial replication or ECC. Implementing the SoR involves two operations: input replication and output comparison. Redundancy is created at the input replication point and integrity verification is performed at the output comparison point before the effects of the computation are allowed to propagate outside the SoR.

Simultaneous Redundant Threading (SRT) [20] leverages the multiple contexts provided by a Simultaneous Multithreading (SMT) processor [28] to execute redundant copies of the same program. The L1 cache interfaces are used as the input replication and output comparison points. Thus, all of the core structures including the PC, fetch and decode logic, register files, the ROB, the issue logic and the functional units lie within the SoR. The two execution streams running on the SMT processor's contexts are referred to as the *leading* (primary) and *trailing* (redundant) threads. The fetch mechanism attempts to maintain a slack between the threads. To improve performance, SRT makes use of a Branch Outcome Queue (BOQ) which contains the targets for the resolved branches in the leading thread to prevent branch mispredictions in the trail-

ing thread. The leading thread places load values obtained from the data cache into the Load Value Queue (LVQ) for subsequent lookup by the trailing thread. The trailing thread issues loads in program order and absorbs values from the LVQ. At the output comparison point, the address and data of every store instruction is verified by comparing the respective outputs from the two redundant executions. To achieve this, retiring stores are sent out of the store queue into a Store Checking Buffer (SCB) where they wait until the trailing thread catches up. If no discrepancy is detected, a single (architected) copy of the store is retired into the memory system in program order. The SCB should also be capable of forwarding values to the loads in the leading thread. If any one of the instructions encounters an error during the course of its residence within the SoR, the error would be detected at the output comparison point.

SRT primarily focuses on the detection of transient faults, with recovery being treated as an orthogonal issue. We maintain the same focus in our proposal.

3.2 SlicK Overview

SlicK executes the leading thread in its entirety, exactly as in SRT. For the trailing thread, it uses a set of predictors to attempt to verify the outputs of the leading thread without re-execution. The only trailing thread instructions that are executed are those that belong to the backward slices of outputs that the predictors could not verify. This results in a partial redundant threading solution where all outputs from the SoR have been verified in some manner, but the actual number of instructions that are redundantly executed is significantly lowered, thereby reducing resource contention and enhancing performance.

In the following subsections, we progressively walk through the steps required to build a SlicK system. First, we define the notion of trigger instructions based on which slices are identified. Next, we describe the design details of a slice-extraction microarchitecture that identifies instructions belonging to slices of these triggers. Finally, we show how to integrate these structures into an SRT microarchitecture to complete the SlicK design.

3.3 Identifying Trigger Instructions

The first step in designing a slice-based redundant execution system is to identify *trigger* instructions based on which slices are to be marked and extracted. We define a trigger instruction to be one of the following: (i) a store instruction; or (ii) a branch instruction. A store trigger has a direct consequence on what leaves the SoR. A branch is also considered a trigger point for verification because the control flow path that leads to these stores needs to be verified as well. For instance, it is possible for both taken and not-taken paths of a branch to lead to the same store instruction, with each path generating a different address or data for the store. If there is an error in the control flow, both threads would execute this store erroneously.

If the integrity of a trigger can be "verified" (we will elaborate on this shortly), then the instruction itself and its backward-slice need not be redundantly executed and can be dropped or flushed from the trailing thread. We refer to such trigger instructions as *Flush Triggers (FT)*. On the other hand, when the integrity of a trigger instruction cannot be verified, then that instruction and its backward-slice need to be extracted and redundantly executed. We refer to such instructions as *Execute Triggers (ET)*. It is possible for an instruction to fall on the backward-slices of more than one trigger. If even one of these triggers is an ET, then the instruction would need to be redundantly executed. Note that there are two backward-slices to track for a store instruction, one for the address computation and the other for the store data. Both have independent backward slices that may or may not need to be executed based on the verifiability of the address and the data.

Our verification mechanism for triggers is based on the well-researched topics of branch [31] and store data and address [12] prediction. SlicK uses predictors to attempt to verify the outputs of trigger instructions and provide the required redundancy. The output (store data/address, branch target) of the trigger instruction of the leading thread is compared with that of the corresponding predictors. If the comparison is successful (a "Hit"), then the trigger is categorized as FT. If the comparison is either a "Mismatch", or the predictor is not able to predict due to insufficient confidence ("No-Prediction"), then the trigger is categorized as ET, requiring redundant execution of its backward-slice. Since predictors are essentially storage arrays, it is possible for errors to accumulate in them if entries remain untouched for a long time. Therefore, we recommend parity protecting these structures.

3.3.1 Store Predictor

We use independent predictors for Store Data and Store Addresses. Both predictors are accessed by the PC of the Store Instruction. We primarily employ a simple last-value predictor with 4-bit saturation counters, but also evaluate the benefits offered by a more extensive Finite Context Method predictor [4]. Our results show that a simple 1024 entry direct-mapped last value predictor gives reasonable performance.

3.3.2 Branch Confidence Estimator

Traditional Branch Predictors do not give a "No-Prediction" decision. Although modern branch predictors have high hit rates, we wish to convert as many Mismatches to No-Predictions as possible (this enhances fault coverage and is explained further in Section 4). We use a simple pattern-based filter for confidence estimation on branch predictions [8]. The filter is constructed from an array of resetting saturation counters and is indexed with the current branch prediction appended to the global branch history.

3.4 Extracting Backward Slices

Given a dynamic sequence of instructions comprised of ETs, FTs and non-trigger instructions, we now need an online technique to identify the set of instructions that lie on the backward slices of ETs and FTs and mark them as requiring execution or being flushable. We assume that (i) instructions arrive at the slice extractor in non-speculative (correct path) program order, (ii) decoded architected register identifiers are available for these instructions, and (iii) memory dependencies are handled independently and do not need to be taken into account by the slice extraction mechanism.

3.4.1 Slice Extractor Design Goals

We identified the following requirements for a backward-slice extractor for SlicK:

1. **Adequate Bandwidth.** The leading thread in SlicK executes in the traditional SRT-like manner. In order to not adversely affect performance, it is critical for the slice extractor to be able to support a bandwidth equal to the average commit bandwidth of the leading thread, thereby enabling a smooth flow of instructions through the pipeline without interruptions.
2. **Low Latency.** Inordinate delays in trailing thread slice extraction would lead to increased pressure on leading thread buffers. By definition, backward slice instructions are older than their corresponding triggers in program order, and therefore any slice extractor would have no way of immediately identifying an instruction the moment it arrives. It has to buffer it for a certain amount of time before its "status" can be determined. This inherent latency is an unavoidable property of the instruction sequence itself.

We do however wish to minimize the *processing latency* of the slice extractor. When a trigger instruction arrives at the extractor, all buffered instructions whose status can be determined by this trigger must be identified by the extractor in as little time as possible.

3. **Support for Early Deletion.** Certain situations may require the oldest buffered instructions in the extractor to be forcibly deleted (popped), regardless of the fact that their status was unknown. This could happen if, for example, the leading thread has stalled due to resource shortage and the trailing thread is required to proceed forcibly in order to clear the stall. In such situations, the slice extractor needs to be able to pop the oldest instruction(s) with minimal latency and update its internal state to maintain correct dependencies.

3.4.2 Inadequacy of Traditional Mechanisms

A straightforward technique for backward-slice extraction is to buffer the sequence of instructions until a trigger arrives, and then perform a reverse program-order traversal of all buffered instructions in order to determine which of them lie on the backward slice of the trigger. Many traditional hardware and software slice extraction techniques have used variations of this approach [5, 15].

Unfortunately, this approach works well only if triggers are infrequent, slice extraction is required on rare occasions, and a certain amount of processing latency is tolerable. In our case, triggers are quite frequent (potentially every store and branch), slices need to be identified constantly without interrupting the instruction flow, and processing latencies could severely affect performance.

It is possible to maintain a smooth instruction flow by using additional storage to buffer incoming instructions while another set is undergoing traversal [15], but our attempts to build a slice extractor for SlicK with this technique resulted in a complex design that also turned out to be sub-optimal. Despite the additional buffering, instructions can only be processed in non-overlapping batches (or "windows") at a time, and dependencies across these windows are lost. To solve this, instructions could be "spilled" from one window into the next, but this makes the design even more complex to implement. Further, processing latencies remain high with this approach.

3.4.3 Our Solution – the SliceEM

We make the following key observations in order to come up with a simple and efficient backward-slice extraction mechanism for SlicK.

- In the presence of multiple triggers, we do not need to associate each instruction individually with a trigger. Instead, we only need to know if an instruction lies on the backward slice of one or more ETs or FTs, or both.
- Complete slices need not be identified. It is acceptable for an old instruction to exit the slice extractor's buffers with an "unknown" status, as long as dependencies are correctly maintained.
- The number of distinct dependency identifiers (i.e., architected register identifiers) in the decoded instruction sequence is finite and typically small (in the range of 8-32 for most ISAs).

Our solution, the Slice Extraction Matrix (SliceEM), borrows from several previous works on backward slicing [15], forward slicing [21], and matrix based instruction scheduling logic [2, 14]. The SliceEM constantly updates and keeps track of all dependency chains that exist between the instructions currently residing within its buffer, and it employs a simple set of structures to achieve this. This makes it possible to *instantly* identify all the instructions (among those that are currently buffered) lying on the backward slice of a trigger that has just arrived. No reverse program order traversal is necessary, and all our design goals are satisfied.

We call the window of buffered instructions that the SliceEM is operating on at any point in time as the Slice Analysis Window (SAW). A larger window results in a higher probability that the oldest entries in the window have their status determined. As our later experiments will show, a window size of 256 is able to determine the status of a majority of the instructions. The SAW does not need to store the instructions themselves. It is simply a mask that indicates which instructions need to be executed/dropped and which are still with an unknown status.

The SliceEM is composed of three main structures (Figure 1) that are all initialized to zero:

- A *Bit Matrix* with columns corresponding to each instruction in the SAW and rows corresponding to each architecturally visible register. The Bit Matrix at any given instant indicates which registers are "live" (if there is at least one non-zero column for a given row), and which instructions in the window are on the backward-slice of that register (the corresponding columns of that row which are non-zero). For example, in Figure 1, instructions in columns 3, 5 and 9 lie on the backward dependency chain of register r3, implying that any future instruction using r3 as an input will require the services of instructions 3, 5 and 9 (and no other instruction) in order to execute correctly.
- A *Live Mask* with one bit corresponding to each entry in the SAW. At any instant this mask indicates that the corresponding instruction has written into at least one "live" register.
- An *Execute Mask* with one bit corresponding to each entry in the SAW. This marks the instructions in the SAW that need to be redundantly executed because they could not be verified.

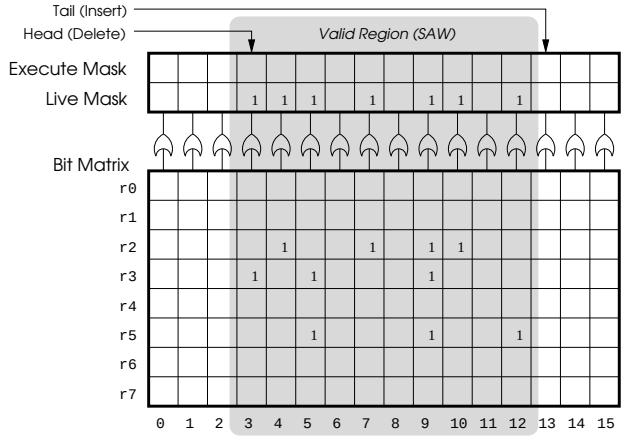

Figure 1. The Slice Extraction Matrix. Column indices are indicated at the bottom of the matrix.

3.4.4 SliceEM Operations

We now describe how the SliceEM handles all Insertion and Deletion operations. Instructions are always inserted and deleted (popped) in FIFO order, and a circular queue algorithm is used.

Inserting a non-trigger instruction When a **Non-Trigger** instruction, e.g., an `add $r5 := $r2 + $r3` register-arithmetic instruction is inserted into the window, the SliceEM does the following (starting from the SliceEM status in Figure 1, the process of inserting this instruction is shown in Figure 2):

1. Allocate a new column index k (k = 13 in this example) for this instruction in the SAW.

2. Read the rows corresponding to the inputs of this instruction (rows for r2 and r3 in this example), and compute the bitwise-union of these rows.
3. Set the k-th bit of this union to 1. It is guaranteed that this bit was previously a 0, otherwise this column could not have been allocated for this new instruction.
4. Write the modified union vector into the row in the Matrix corresponding to the instruction's output register (r5 in this example), completely overwriting any values that may be already present.
5. Update the Live Mask vector by taking a wired-OR *of each of the columns*. The Live Mask of the new instruction would get set to 1. It is possible that some of the previous 1s in this vector could now get reset to 0 if this new output register row (r5 in the example) does not have a 1 in the corresponding position, and no other row has a 1 in that column either.

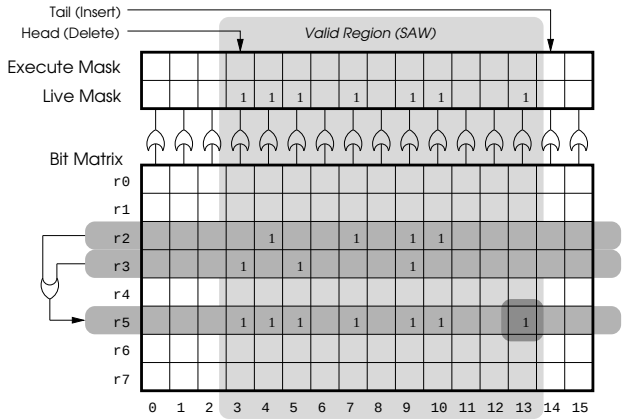

Figure 2. Process of inserting the instruction `add $r5 := $r2 + $r3` **into the SliceEM. Notice that the instruction in column 12 ceases to be Live after this insertion. The new instruction is inserted in column 13, and a new 1 is entered into the matrix entry (**`r5`**, 13) to indicate this.**

Inserting an Execute Trigger When an ET instruction (for instance an effective address calculation instruction for a store `computeEA $r2 + 5`) enters the window, we do the following (starting from the Matrix in Figure 2, the process is shown in Figure 3):

1. Allocate a new column index k (k = 14 in this example) for this instruction.
2. Read all rows corresponding to the inputs of this instruction (r2 in this example), and compute the bitwise-union of these rows. Note that this would correspond to all the instructions on the backward-slice of this ET instruction.
3. Set the k-th bit of this union to 1.
4. Take a bitwise-union of this resulting vector with the contents of the Execute Mask, and overwrite the Execute Mask with the result.

Inserting a Flush Trigger When an FT instruction enters the window, we only need to allocate a new column for this instruction. Since we are not updating the SliceEM in any other way, any instruction in its backward-slice would automatically be *killed* when the registers whose values they produced are overwritten by any future instruction (treated exactly the same as a dynamically dead

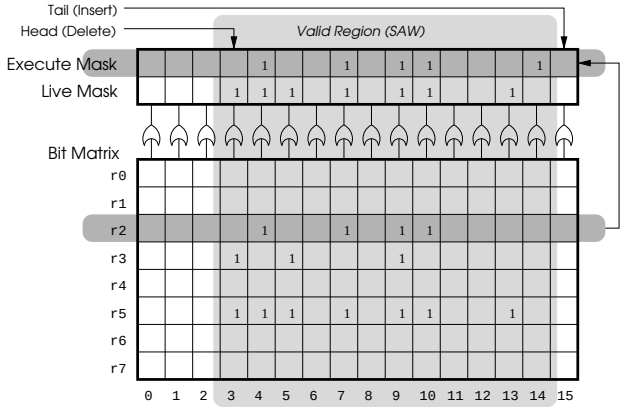

Figure 3. Process of inserting the ET store effective address calculation instruction `computeEA $r2 + 5` **into the SliceEM. Notice the updated Execute Mask.**

instruction). Thus our mechanism is intended to eliminate instructions in the redundant thread (i) that do not lie on the backward slice of any store or branch, and (ii) that lie on the backward slice(s) of only verifiable (FT) stores and/or branches. The Execute Mask and Live Mask bits are both 0 for such instructions.

Deleting (Popping) Entries At any instant, the entry at the Head of the SAW can be examined and deleted from the SliceEM, and either sent for redundant execution or flushed. If the Execute Mask and Live Mask bits for the entry are both 0, then the instruction corresponding to this entry is guaranteed flushable. If the Execute Mask bit is 1, then the instruction requires redundant execution. If the Execute Mask bit is 0, but the Live mask bit is 1, then the instruction's status is unknown, and the logic that is removing this entry could choose to either conservatively execute the instruction redundantly, or allow it to stay in the SliceEM for a little longer. If the entry is deleted, then the Execute Mask bit, Live Mask bit and the entire column of the Bit Matrix for that instruction are reset to 0.

Observe that the SliceEM does not literally "extract" slices, but only identifies and marks instructions lying on the backward slices of certain triggers. It also does not necessarily identify complete slices. If the SAW is not wide enough to accommodate all instructions of a slice, then the oldest instructions can start spilling out with an "unknown" status. No additional logic is required to ensure that dependencies are correctly maintained in such scenarios.

The Bit Matrix and Live Mask together keep track of the "liveness" of each entry in the SAW. The Execute Mask identifies a set of instructions on the backward slice of a certain *class* of triggers. SlicK has only one such class: Execute Triggers. However, the SliceEM can also be used in scenarios where slices belonging to multiple classes of triggers need to be identified simultaneously. In such cases, a distinct Execute Mask needs to be used for each class, but the Bit Matrix and Live Mask can be shared.

3.4.5 Hardware Implementation Concerns

The key component of the SliceEM is the bit-matrix, and it primarily needs to perform 3 operations: (i) Read and write a given row, (ii) Compute the wired-OR of all bits in a column, and (iii) Reset all bits in a column. Since the number of rows (equal to the number of architected registers) is expected to be in the range of 8-32, we do not expect wire delay problems associated with operations (ii) and (iii). Further, as we will show later in our results, window sizes (columns) of 256 instructions suffice, and operation (i) is not expected to have significant overheads either. Its read/write ports need

to be enough to support the commit bandwidth of leading thread instructions, which is typically less than 3 per cycle.

3.5 Integrating the Predictors and SliceEM into an SRT processor

Armed with a set of predictors and a slice extraction mechanism, we now need to integrate these into an SRT pipeline and complete the SlicK design. We were faced with multiple design choices with various aspects of this process. In the interest of brevity, we refrain from analyzing the pros and cons of each of these choices. Instead, we present the final design incorporating our design decisions along with brief explanations of the rationale behind these decisions.

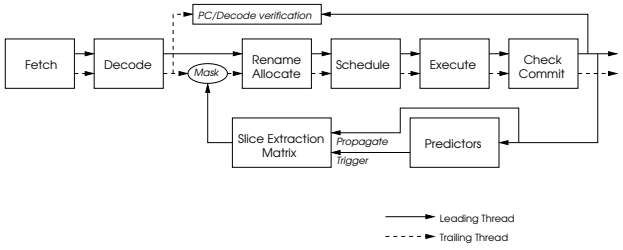

Figure 4. Overview of SlicK Pipeline

Most of the core pipeline remains unchanged from the SRT design (Figure 4). As in SRT, the two PCs of a two-context SMT processor are provided for the redundant threads (leading and trailing), each independently fetching their instruction stream from the L1-cache into the SoR with a temporal slack separating the two. The trailing thread uses the Branch Outcome Queue (BOQ) for obtaining branch targets into which the leading thread would have previously placed the results of resolved branches.

Our first design decision for SlicK was to *make the leading thread send its committed instructions to look up the predictors and update the SliceEM*. The leading thread's runahead slack gives the SliceEM ample opportunity to discover the status of older instructions in the SAW by the time the trailing thread catches up. "Inserting" a committed leading thread instruction into the SliceEM simply involves updating the Bit Matrix and the two masks, and as far as slice identification is concerned, no additional information about the instruction need be maintained. All that the trailing thread needs to do is examine the Execute and Live mask bits in order to determine which instructions it needs to execute, and this can be done immediately after its Fetch stage.

There is a redundancy issue here that needs to be handled. Since it is the leading thread that looks up the predictors and constructs the execution masks to select the trailing thread's instructions, these operations themselves need to be verified by the trailing thread. This can be achieved by verifying all fetched PCs and the decoded architected register identifiers. Therefore, *we require every instruction in the trailing thread to go through the Fetch and Decode stages* (our second design decision), regardless of whether they are ultimately executed or flushed. Techniques such as fingerprinting [25] can be used to significantly reduce the storage and bandwidth overheads associated with buffering this data from the leading thread until the trailing thread catches up and performs the comparison.

After Decode, trailing thread instructions can look up the SliceEM masks and decide to either execute or flush themselves. Since the SliceEM supports early removal, the trailing thread can choose to delete an entry even if its status was unknown and execute it conservatively. This brings us to our third design decision: *if the trailing thread has decoded instructions available, then it always pops/deletes SliceEM entries, regardless of whether their status were known or unknown*. In our experience, ensuring the smooth flow of the pipeline is a more critical factor for performance than removing a few additional instructions by stalling the thread for several cycles.

3.5.1 Memory Ordering for the Trailing Thread

Trailing thread Loads always pick up their operands from the LVQ, and trailing thread Stores have only one purpose: to provide redundancy for leading thread Stores waiting in the SCB. Therefore, as long as the LVQ and SCB semantics are correct, no additional ordering needs to be enforced between trailing thread Loads and Stores. We ensure that all trailing thread Loads and Stores perform the necessary synchronization with their associated structures (LVQ/SCB) regardless of whether they are executed or flushed.

We adhere to the SRT mechanisms for handling uncached operations and precise interrupt delivery.

4. Fault Coverage

4.1 Error Model

Transient errors can lead to either Silent Data Corruption (SDC) or Detected Unrecoverable Errors (DUE) [17]. The focus of this paper is to avoid SDC errors in a Single Event Upset (SEU) fault model. Upon the occurrence of a single transient error, our mechanisms are intended to detect its occurrence before an erroneous store propagates into the memory system.

4.2 Identifying Points of Vulnerability in SlicK

The speculative nature of the predictors in SlicK can cause certain rare redundancy violations. This is best illustrated through an example (Figure 5).

For 100 iterations		101^{st} iteration	
Lead. Thr.:	`store 0x4000`	Lead. Thr.:	`store 0x4100`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (mismatch – trigger Trail. Thr.)
		Trail. Thr.:	`store 0x4100` (no error)

(a)

For 100 iterations		101^{st} iteration	
Lead. Thr.:	`store 0x4000`	Erroneous Lead. Thr.:	`store 0x4101`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (mismatch – trigger Trail. Thr.)
		Trail. Thr.:	`store 0x4100` (error detected)

(b)

For 100 iterations		101^{st} iteration	
Lead. Thr.:	`store 0x4000`	Erroneous Lead. Thr.:	`store 0x4000`
Predictor:	`store 0x4000` (match – commit)	Predictor:	`store 0x4000` (match – commit) **SDC**

(c)

Figure 5. SlicK Vulnerability Example, showing the execution of a Store instruction through 101 iterations of a loop in which the data value is 0x4000 for the first 100 iterations and 0x4100 in the final iteration. This is shown for (a) the Correct Execution, (b) one possible Erroneous Execution (caught by SlicK), and (c) another possible Erroneous Execution (not caught by SlicK).

In this example, a store instruction, having produced the same data value 0x4000 for several iterations, warms up the data predictor. Consequently, the trailing thread does not execute the backward slice for this store data. However, the next iteration for some reason now produces the value 0x4100. In the error-free execution (Case

(a)), this causes a mismatch with the predictor, which causes the redundant slice to be extracted and executed. In Case (b), the leading thread is corrupted by an error, but the error is detected. However, it is possible for a single-bit error to corrupt the leading thread data in such a way that it exactly matches with the predictor's output and result in SDC (Case (c)). From this example, we can observe the following key facts:

- The scenario we are observing involves a Mismatch between the leading thread and the predictor in the error-free execution.
- Both Cases (b) and (c) involve a fault in the *leading thread*.
- SDC only occurs in Case (c) – if the Mismatch in the error-free case is transformed into a Hit (or Match) in the erroneous case.
- Only certain patterns of faults in the leading thread could result in Case (c); most faults will result in Case (b)-like situations.

In general, with a single-error model and with full Fetch and Decode redundancy as described in the previous section, any errors in either the predictor or SliceEM structures or in pipeline structures being occupied by trailing thread instructions cannot cause SDC, since any such errors in this model automatically implies that there are no errors in the leading thread's execution.

On the other hand, a fault in the pipeline structures occupied by *leading thread* instructions in SlicK can lead to SDC. As the example illustrated, an error in a leading thread Store instruction (or its backward slice) that causes a predictor Mismatch in the error-free execution could potentially transform the Mismatch into a Hit and result in SDC.

We refer to these stores and their backward-slices (which lead to a Mismatch in the prediction structures) as being *unguarded*. Any arbitrary fault in an unguarded instruction would not necessarily cause such a Mismatch-to-Hit conversion; the fault has to be such that the resulting value of the Store exactly aliases with the value that the Predictor produces. However, this is tricky to quantify due to architectural and program level masking effects [13, 17], and we conservatively assume that any fault in an unguarded instruction would cause SDC. In terms of the example, we assume that any single-bit error in the leading thread *will* convert all Case (a) scenarios into Case (c) scenarios.

Note that instructions could be in more than one backward-slice, with different prediction outcomes for each slice. In our analysis, we count an instruction as being unguarded if the earliest trigger (in program order) of its forward slice is a Mismatch.

Identifying unguarded instructions requires analyzing register dependencies over a very large instruction window since the forward-slice trigger can occur millions of instructions later. However, our simulations were excessively slow with very large windows. We found that using a moderate window size of 16K instructions provided reasonable simulation speeds while still keeping the number of instructions with unknown status sufficiently small. In our results, we show coverage as ranges assuming worst (all unguarded) and best (all guarded) cases for these unknown instructions.

4.3 The Common Case – How SlicK detects soft errors

SlicK's predictor lookups can have three outcomes: No-Prediction, Hit and Mismatch. Of these, only Mismatches create vulnerability. Therefore, the predictors are configured so as to minimize the occurrence of Mismatches, with No-Predictions and Hits forming the majority of cases. No-Predictions result in full redundant execution as in SRT. Hits cause instructions to be flushed, gaining performance, while redundancy is maintained by the predictor output. In case an error corrupts a leading thread trigger value that would have been a Hit in the error-free case, it is guaranteed that it will be a Mismatch in the actual erroneous execution, triggering redundant execution.

4.4 Quantifying the Fault Coverage of SlicK

We use the Architectural Vulnerability Factor (AVF) [17] approach to quantify the error coverage of our design. Calculating AVF requires identifying bits in a pipeline structure that are necessary for Architecturally Correct Execution (ACE). The ACE-ness of every physical bit changes from cycle to cycle, and is deduced from the ACE-ness of the logical entity (such as some component of an instruction) that was holding that bit in that cycle. Several factors contribute to making an instruction ACE or un-ACE, all of which are outlined in [17]. Once the ACE-ness of all bits in a structure are established on a cycle-by-cycle basis, the AVF of the structure is defined as the average fraction of ACE bits in the structure over the entire execution of a suite of benchmarks. The "unguarded" instructions defined above are ACE, and we consequently only need to track the bits of these instructions to quantify the AVF. Note that we are automatically discounting wrong-path instructions and dynamically dead instructions, since they will not be unguarded. However, we do not track dynamically dead instructions via memory dependencies, which again makes our estimation conservative. We perform AVF analysis for several non-speculative processor structures that contribute significantly to the total chip area, namely, the ROB and Physical Register File (which are coupled together into a unified RUU in our model), the Load Store Queue, and the Issue Queue.

5. Results

Our experiments were conducted via execution-driven simulation using an extended version of the SimpleScalar 3.0 toolset [3], where we implemented the SRT and SlicK models. We evaluate the schemes using all 26 applications from the SPEC CPU2000 benchmark suite. The benchmarks were compiled for the Alpha ISA and use the reference input set. We measured the statistics for detailed simulation of 100 million instructions after fast-forwarding to the first SimPoint [22] with a weight of at least 1% for each benchmark. The parameters of our baseline hardware model are shown in Table 1.

Table 2 gives the IPCs for single-thread and SRT execution. IPC losses due to SRT range from as low as 3.67% (in *mcf*) to as high as 39.73% (in *sixtrack*), with an average loss of 18.01% across these 26 benchmarks. In order to estimate an optimal size for the slice analysis window (number of columns in the SliceEM) for SlicK, we ran simulations with relatively large window sizes to find out how many younger instructions need to appear in the dynamic instruction sequence before the status (whether to execute or not) of each instruction is known. We then plotted this information as a CDF graph, and in Figure 6 we show representative results for a few applications. For most benchmarks, the status of 90-95% of the instructions is known within a window size of 256 instructions. Results for most other applications are similar except for *wupwise* and to a certain extent *galgel* where the status of around 75-80% of the instructions is known within a window size of 256. We consequently fix the number of columns in the SliceEM at 256.

5.1 Performance Results

Figure 7 gives the IPC results (normalized with respect to single thread performance) of our SlicK mechanism using both Last-Value (LV) and Finite-Context Method (FCM) store predictors, and compares them with the normalized IPCs of the SRT execution. FCM provides only marginal improvements over LV but at a much higher hardware cost, and we consequently focus on the LV store predictors for the rest of this paper. Table 3 shows the actual

Parameter	Value
Fetch/Decode/Issue/Commit Width	8
Pipeline Stages	15
Fetch Queue Size	16
Load Value Queue (LVQ) Size	128
Branch Outcome Queue (BOQ) Size	128
Store Checking Buffer (SCB)	64
Branch-Predictor	Combined predictor with 16K-entry meta-table. 2-lev predictor with 16K-entry L1, 16K-entry L2, 14-bit history XORed with address
RAS Size	64
BTB Size	2K-entry 4-way
RUU Size	128
LSQ Size	64
Integer ALUs	6 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	4 (2)
FP Mult./Div./Sqrt.	2 (4,12,24)
L1 D-Cache Ports	4
L1 D-Cache	64KB, 4-way with 32B block (2)
L1 I-Cache	64KB, 4-way with 32B block (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
I-TLB	512-entries 4-way set-associative
D-TLB	1K-entries 4-way set-associative
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles
SliceEM Columns	256
Store Address and Data Predictors	– Last-Value, 1024-entry – FCM, 1024/8192-entry L1/L2, Order 3
Branch Conf. Estimator	Gshare, 14-bit Global BHR, 16K L2

Table 1. Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root.

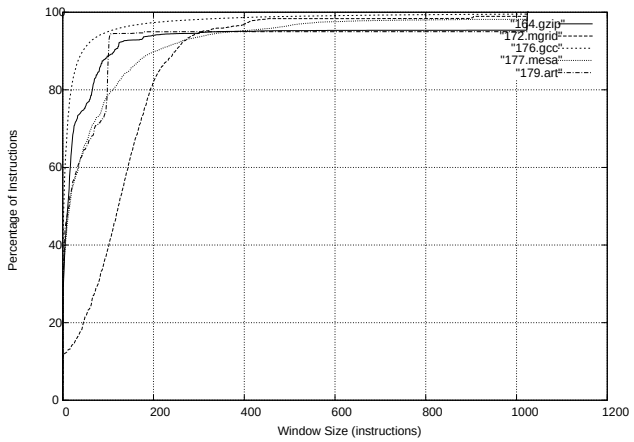

Figure 6. Percentage of instructions whose status (whether to execute or not) becomes known after x instructions in the dynamic instruction stream.

IPC values and some other statistics about the SlicK execution, including the percentage of instructions that are flushed, number of branch FTs (and as a percentage of number of branches), and the number of store address/value FTs (and as a percentage of the number of stores).

From Figure 7, we see that SlicK can provide a significant bridge for the performance gap between SRT and Single Thread executions. For instance, in *vortex*, *mesa* and *eon*, where SRT incurs about 29%, 20% and 17% IPC loss respectively, SlicK is able to cut this loss by 42.47%, 32.65%, and 37.81% respectively by eliminating a substantial number of instructions (57%, 43% and 41%) in the trailing thread.

Benchmark	Single-Thread IPC	SRT IPC	% IPC Loss
164.gzip	1.9614	1.7924	8.62
168.wupwise	0.6664	0.4818	27.70
171.swim	0.8704	0.6046	30.54
172.mgrid	1.1261	1.0245	9.02
173.applu	1.1093	0.8398	24.29
175.vpr	0.5936	0.5578	6.03
176.gcc	1.5293	1.4369	6.04
177.mesa	2.5172	2.0216	19.69
178.galgel	2.4665	1.8210	26.17
179.art	0.6030	0.4039	33.02
181.mcf	0.1715	0.1652	3.67
183.equake	0.6657	0.5628	15.46
186.crafty	1.9653	1.7603	10.43
187.facerec	2.6073	1.7662	32.26
188.ammp	0.8191	0.6903	15.72
189.lucas	0.7389	0.6982	5.51
191.fma3d	2.3089	2.0712	10.29
197.parser	1.0874	0.9984	8.18
200.sixtrack	3.6739	2.2143	39.73
252.eon	2.3617	1.9697	16.60
253.perlbmk	1.5293	1.4704	3.85
254.gap	0.2606	0.2257	13.39
255.vortex	2.7971	1.9827	29.11
256.bzip2	1.8440	1.5976	13.36
300.twolf	0.7790	0.7479	3.99
301.apsi	3.2301	2.0835	35.50

Table 2. IPC of single-thread (non-redundant) execution of the benchmarks and redundant execution using SRT. The loss of IPC with SRT is also given, with the average loss across the applications being 18.01%. All the data is for the execution of 100 million instructions from the first SimPoint with a weightage at least 1% for each benchmark.

	IPC	Flushed	Branch FT	St. Addr FT	St. Data FT
gzip	1.8855	34.48%	7.64M (81%)	3.17M (45%)	0.24M (3%)
wupwise	0.5957	9.32%	3.25M (99%)	0.00M (0%)	0.00M (0%)
swim	0.8167	12.27%	1.16M (99%)	0.75M (6%)	0.00M (0%)
mgrid	1.0485	5.49%	0.27M (96%)	2.85M (59%)	0.02M (0%)
applu	1.0154	12.51%	0.20M (97%)	2.90M (27%)	3.71M (35%)
vpr	0.5850	33.67%	9.10M (88%)	4.71M (43%)	3.87M (35%)
gcc	1.4696	36.25%	4.78M (30%)	2.00M (19%)	3.53M (34%)
mesa	2.1834	43.35%	7.11M (83%)	7.69M (63%)	6.54M (54%)
galgel	2.1094	28.26%	4.32M (93%)	2.51M (50%)	0.01M (0%)
art	0.5765	33.59%	9.08M (81%)	4.28M (55%)	0.76M (9%)
mcf	0.1712	35.07%	14.26M (61%)	0.06M (1%)	2.68M (40%)
equake	0.6246	24.97%	3.11M (97%)	2.14M (51%)	2.97M (71%)
crafty	1.8712	38.44%	5.48M (49%)	2.33M (40%)	1.88M (32%)
facerec	2.0351	12.97%	3.12M (89%)	0.85M (6%)	0.52M (3%)
ammp	0.7505	11.44%	3.50M (90%)	2.47M (37%)	0.31M (4%)
lucas	0.7133	1.00%	0.53M (99%)	0.00M (0%)	0.00M (0%)
fma3d	2.2679	52.46%	14.82M (82%)	2.15M (28%)	3.74M (50%)
parser	1.0110	35.64%	9.58M (61%)	4.06M (50%)	3.00M (37%)
sixtrack	2.3898	10.59%	2.21M (98%)	1.19M (22%)	0.00M (0%)
eon	2.1179	40.70%	9.16M (83%)	8.36M (49%)	7.70M (45%)
perlbmk	1.5021	43.09%	10.01M (71%)	5.25M (33%)	6.03M (38%)
gap	0.2282	19.51%	9.04M (69%)	4.71M (40%)	2.56M (21%)
vortex	2.3286	57.57%	15.56M (87%)	3.09M (24%)	6.86M (55%)
bzip2	1.7241	45.46%	9.39M (84%)	3.92M (48%)	2.47M (30%)
twolf	0.7778	28.35%	7.27M (59%)	2.50M (35%)	1.54M (21%)
apsi	2.2848	11.94%	3.00M (91%)	1.87M (14%)	0.65M (4%)

Table 3. Execution Statistics for SlicK with LV predictor. We show the percentage of flushed/eliminated instructions, the number (and as a percentage of total branches) of Branch FTs, the number (and as a percentage of total stores) of Store FTs.

For most applications, we find an expected correlation between the percentage of instructions flushed to the improvements in IPC for the SlicK executions. As noted above, *vortex*, *mesa* and *eon* are representative of those with significant improvements due to removal of instructions based on locality. At the other end, *sixtrack*, despite the high IPC loss (40%) due to SRT, SlicK can only provide

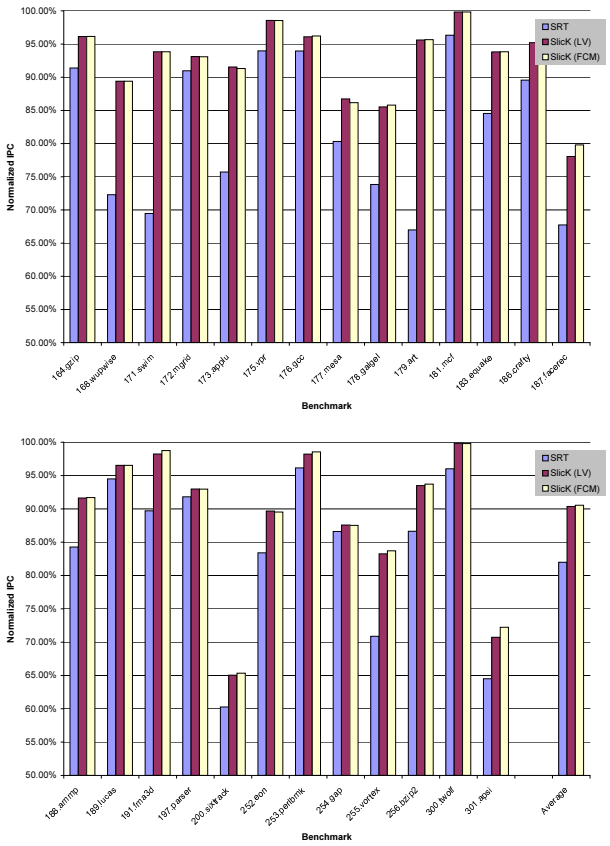

Figure 7. IPC Results for SlicK executions with a simple Last Value (LV) and a more extensive Finite-Context Method (FCM) store address/value predictor, shown in comparison with SRT. All values are normalized with respect to Single Thread (non-redundant) execution.

8% improvement due to the low number of flushed instructions. This reduction in flushed instructions can be attributed to the lower locality of both store addresses and data for this application – even though branch prediction accuracy rates are high (98.46%), the number of branches themselves (2.2M) are not as high as the number of Store ETs (9.5M).

mgrid experiences good locality (visible from the high FTs for this application), but the number of flushed instructions for this application is not as high as for others. We found that this was because of our in-order extraction of entries from the SAW, wherein we send trailing thread instructions corresponding to these entries into the pipeline if these instructions have arrived and pipeline resources are available, even if their status is unknown in the SAW. Although our maximum window size is 256, the SAW rarely fills up to its full capacity. For most applications, this is not a problem, but *mgrid* can discover the status of a very small number of instructions with a smaller window of information (16 or lower) as is evident from its low initial slope in Figure 6.

There are cases (as in *parser*) where the SRT performance loss is itself not very high (around 8%) since the datapath resource contention between the threads is not very significant. In such cases, the effect of eliminating 35% of the instructions for redundant execution by SlicK does not produce substantial savings over SRT performance (only 1.26%). SlicK seems to perform well even for memory-bound applications such as *mcf, art* and *swim* compared to SRT, but we are slightly less interested in them since these are very low IPC applications to begin with, and SlicK's focus is on alleviating datapath resource contention for the trailing thread.

Overall, SlicK provides around 10.2% performance improvement over SRT, and buys back over 50% of the performance loss incurred by SRT with respect to Single Thread execution.

5.2 Fault Coverage Results

	RUU	Issue Queue	LSQ
164.gzip	0.19% – 5.66%	0.08% – 2.16%	0.07% – 2.40%
168.wupwise	0.01% – 27.23%	0.01% – 26.81%	0.01% – 14.74%
171.swim	0.00% – 0.70%	0.00% – 0.68%	0.00% – 0.00%
172.mgrid	0.00% – 0.00%	0.00% – 0.00%	0.00% – 0.00%
173.applu	0.00% – 0.38%	0.00% – 0.37%	0.00% – 0.00%
175.vpr	0.48% – 0.49%	0.37% – 0.38%	0.71% – 0.71%
176.gcc	0.21% – 0.21%	0.11% – 0.11%	0.18% – 0.18%
177.mesa	0.14% – 1.28%	0.08% – 1.19%	0.09% – 0.10%
178.galgel	0.14% – 0.17%	0.13% – 0.15%	0.01% – 0.02%
179.art	0.35% – 2.56%	0.35% – 1.65%	0.00% – 0.16%
181.mcf	0.18% – 0.44%	0.12% – 0.39%	0.16% – 0.16%
183.equake	0.16% – 0.27%	0.11% – 0.22%	0.13% – 0.13%
186.crafty	0.41% – 0.41%	0.24% – 0.24%	0.32% – 0.32%
187.facerec	0.04% – 1.53%	0.03% – 1.32%	0.01% – 0.88%
188.ammp	0.45% – 1.96%	0.21% – 0.75%	0.22% – 2.10%
189.lucas	0.00% – 0.05%	0.00% – 0.04%	0.00% – 0.00%
191.fma3d	0.27% – 0.27%	0.14% – 0.14%	0.14% – 0.14%
197.parser	0.40% – 0.40%	0.16% – 0.17%	0.18% – 0.18%
200.sixtrack	0.00% – 0.04%	0.00% – 0.04%	0.00% – 0.00%
252.eon	0.59% – 0.60%	0.42% – 0.42%	0.64% – 0.64%
253.perlbmk	0.24% – 0.24%	0.12% – 0.12%	0.22% – 0.22%
254.gap	0.16% – 1.03%	0.03% – 0.89%	0.10% – 0.10%
255.vortex	0.20% – 1.60%	0.15% – 1.28%	0.21% – 0.22%
256.bzip2	0.20% – 0.98%	0.13% – 0.86%	0.11% – 0.11%
300.twolf	0.94% – 0.94%	0.47% – 0.47%	1.08% – 1.08%
301.apsi	0.02% – 0.02%	0.02% – 0.02%	0.00% – 0.00%

Table 4. Architectural Vulnerability Factors for important structures in a SlicK processor with LV store predictors. Vulnerability comes from unguarded instructions due to predictor Mismatches (0.58% on the average), while performance comes from flushed instructions due to predictor Hits (46% on the average).

Table 4 gives our measured AVF numbers for important structures in a SlicK processor with LV store predictors. As mentioned previously, we provide AVF ranges instead of single values due to simulation time limitations. For most benchmarks these ranges are fairly small, except for *wupwise*, which has a very large number of instructions whose status cannot not be determined within the finite window sizes that our simulators can handle.

Barring *wupwise*, we notice from Table 4 that the AVF numbers are very low when compared against the structural AVFs of either a processor that is running in non-redundant mode [17], or one that is employing a more performance-oriented redundancy mechanism [7]. With SlicK, maximum RUU (where instructions usually have the highest residency), AVFs are less than 1% for 18 of the 25 benchmarks, between 1% and 2% for 5 benchmarks, and greater than 2% for only 2 benchmarks. Minimum RUU AVFs are less than 1% for all 26 benchmarks (including *wupwise*), and less than 0.5% for 24 of them. Issue Queue and LSQ numbers are similarly low. These low AVFs are the result of very few unguarded instructions (0.61% on the average). Unguarded instructions are low due to the extremely low rate of Mismatches (0.58%).

To understand the impact of selectively dropping slices based on locality on vulnerability, we present an anecdotal analysis comparing locality-based slice removal versus an approach that removes slices randomly. We show results from experiments where we use the SliceEM to (a) drop only dynamically dead instructions (Dead), (b) drop dynamically dead instructions and slices for 20% randomly-selected triggers (Random-20), and (c) drop dynamically

dead instructions and slices for 30% randomly-selected triggers (Random-30). Figure 8 shows the normalized IPC and maximum RUU-AVF results for these three experiments and compares them with SRT and SlicK (locality-based removal) results. For these experiments, we focus on a small subset of high-IPC benchmarks that are primarily datapath-limited, and suffer high performance degradation due to SRT.

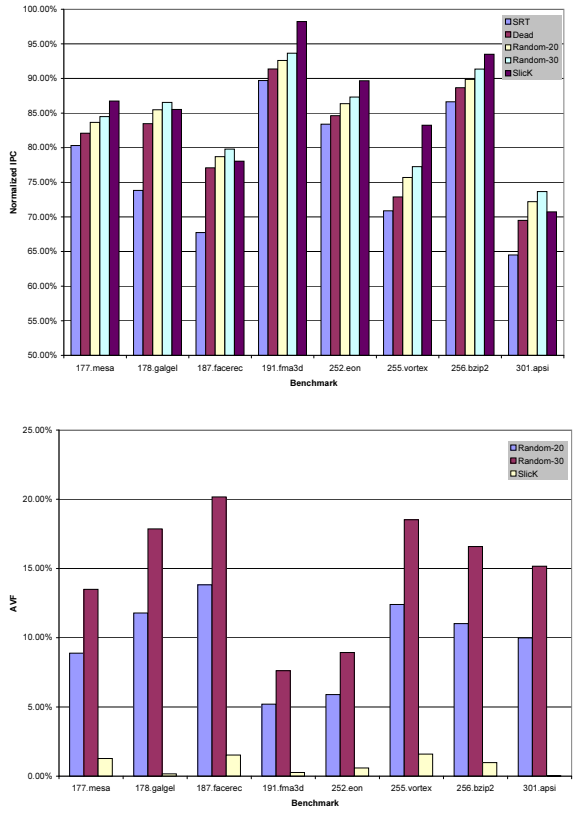

Figure 8. Normalized IPC and maximum RUU-AVF that were run with a subset of trigger instructions randomly chosen to be dropped together with a redundant execution that drops only dynamically dead instructions. Note that the AVFs of SRT and Dead experiments are 0.

At one end, baseline SRT provides full coverage (0% AVF due to full dual modular redundancy) but suffers a relatively high performance impact (65-90%). Starting from SRT, dropping only dynamically dead instructions provides a moderate improvement in performance, while maintaining 0% AVF. From here, if we start randomly selecting slices to not execute redundantly, while the performance does improve, vulnerability increases rapidly as the fraction of dropped instructions increases. As we reach 30% dropped instructions (which is approximately the same as the SlicK average drop rate), we achieve IPCs between 74-94% of single thread execution, but with RUU AVFs that can range anywhere between 7.5% to 20%. This is where the advantages of SlicK's value locality exploitation becomes apparent. At a performance level slightly better than Random-30 (71-97% of single thread IPC), SlicK brings the AVF down to 0.02-1.59%. Value locality is a redundancy mechanism by itself, thus allowing the option of avoiding redundant execution of the corresponding slices.

5.3 Discussion

The previous experiment illustrates the reason why SlicK is very attractive as a *baseline* for exploring the performance-reliability tradeoff space via partial threading.

On the one hand, designers may find SlicK's vulnerability baseline too conservative and want to improve on performance. Slice-based execution simplifies the decision making (of what instructions to execute and what to drop) by allowing heuristics to focus directly on entities that exit the processor's Sphere of Replication and affect the rest of the system. For instance, one heuristic could be to avoid redundant execution of register-spill ET Stores (FTs are always dropped), indicating that the value would probably undergo several levels of masking before it propagates to an I/O device or another processor. Another heuristic could be to increase the threshold of dropping if the processor resources are being overly stressed, and to lower it if there is spare execution bandwidth available (similar to [7]). The slice-based approach avoids any complications related to copying state from one thread to another.

On the other hand, mission-critical systems may desire even more reliability than that afforded by SlicK's baseline. In this case, a certain subset of FT slices could be redundantly executed (recall that in a real world scenario, an error can convert an ET-mismatch into an FT). Once again, different heuristics could be used to select these.

Our framework can also allow a compiler to mark critical stores for redundant execution to over-ride the predictor outcomes, or mark them for avoiding redundant execution. Exploring these compiler issues and evaluating the effectiveness of different runtime heuristics on a SlicK platform is part of our future work.

6. Concluding Remarks

This paper has proposed the idea of using slice-level redundant execution of instructions based on the value and control-flow locality in the program. Redundant execution at the granularity of slices is a novel paradigm that enables partial threading policies to focus directly on entities that exit the processor and affect the outside world. We have demonstrated the effectiveness of using control-flow and value locality as a slice-selection policy. With predictors being used to verify the execution of the leading thread, redundant execution of a large fraction of instructions can be avoided without significantly compromising on the vulnerability of processor structures. With a detailed cycle level model of our SlicK architecture, we show that SlicK outperforms SRT by 10.2%, buying back over 50% of the performance loss incurred by SRT due to redundant execution. At the same time, elimination of redundant execution of a significant fraction of the slices has only marginal effect on the AVF of processor structures, keeping them typically in the 0%-2% range.

The backward-slice extractor proposed in this paper is is able to track the instruction lying on slices of several trigger instructions simultaneously and does not need to stall the instruction flow while tracking is occurring. It is simple to implement, and though we have adapted it for redundant threading, there are several other applications for backward slice extraction.

More importantly, SlicK provides a set of basic mechanisms that can be used as a starting point for designing policies to avoid redundant slice executions, taking us to interesting points in the performance-reliability space. Designing and evaluating such policies is part of our future work.

Acknowledgments

We would like to thank Kristen Walcott and Moinuddin Qureshi for their helpful comments. We would also like to thank the anonymous reviewers for their detailed comments which helped us improve the

quality of the presentation. This research was funded in part by NSF grants 0103583, 0509234, 0325056 and 0429500.

References

[1] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 196–207, November 1999.

[2] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 204–213, December 2001.

[3] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com.

[4] M. Burtscher. An Improved Index Function for (D)FCM Predictors. *ACM SIGARCH Computer Architecture News*, 30(3):19–24, June 2002.

[5] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 306–317, December 2001.

[6] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511, 1992.

[7] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–183, 2005.

[8] D. Grunwald, A. Klauser, S. Manne, and A. R. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 122–131, 1998.

[9] S. Gurumurthi, A. Parashar, and A. Sivasubramaniam. SOS: Using Speculation for Memory Error Detection. In *Proceedings of the Workshop on High Performance Computing Reliability Issues (held in conjunction with HPCA)*, February 2005.

[10] HP NonStop Himalaya. http://nonstop.compaq.com/.

[11] J. J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in slipstream processors. *IEEE Transactions on Computers*, 53(4):399–413, 2004.

[12] K. Lepak, G. Bell, and M. Lipasti. Silent Stores and Store Value Locality. *IEEE Transactions on Computers*, 50(11):1174–1190, November 2001.

[13] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Softarch: An architecture level tool for modeling and analyzing soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 496–505, 2005.

[14] E. Morancho, J. Labia, and A. Olive. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[15] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 321–334, 2001.

[16] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 99–110, May 2002.

[17] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, December 2003.

[18] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 376–386, June 2004.

[19] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 434–443, 2005.

[20] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[21] S. R. Sarangi, J. T. Wei Liu, and Y. Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 257–270, 2005.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[23] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[24] T. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2), March 1999.

[25] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 224–234, October 2004.

[26] J. Smolens, J. Kim, J. Hoe, and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 257–268, December 2004.

[27] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268, 2000.

[28] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[29] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.

[30] N. J. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 30–39, 2005.

[31] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[32] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.