# ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing *

**Smruti R. Sarangi**, **Wei Liu**, **Josep Torrellas**, and **Yuanyuan Zhou**
University of Illinois at Urbana-Champaign
{sarangi,liuwei,torrellas,yyzhou}@cs.uiuc.edu

## Abstract

As more data value speculation mechanisms are being proposed to speed-up processors, there is growing pressure on the critical processor structures that must buffer the state of the speculative instructions. A scalable solution is to checkpoint the processor and *retire speculative* instructions. However, in this environment, misprediction recovery becomes very wasteful, as it involves discarding and re-executing all the instructions executed since the checkpoint.

To speed-up execution in this environment, this paper presents a novel architecture (*ReSlice*) that selectively re-executes only the speculatively-retired instructions that directly depended on the mispredicted value, namely its *Forward Slice*. ReSlice buffers the (typically very few) instructions in the forward slice of the predicted value as such instructions initially execute. Then, potentially thousands of instructions later, ReSlice can quickly re-execute the slice if a misprediction is declared, and merge its state with the program state. In addition, this paper develops a sufficient condition for correct slice re-execution and merge. As one possible use of ReSlice, we apply it to recover from cross-task dependence violations in a chip multiprocessor with Thread-Level Speculation (TLS). ReSlice speeds up SpecInt applications *over* aggressive TLS by up to 33%, with a geometric mean of 12%. Moreover, $E \times D^2$ decreases by 20%. All this is obtained by saving on average 61% of the task squashes through slice re-execution. On average, a slice re-executes only 6.6 instructions, compared to the 210 that would be re-executed on a squash.

## 1. Introduction

With the advent of long memory and inter-processor communication latencies, data value speculation will increasingly assume a prominent role in processors. Rather than waiting for a long-latency event to produce a value, processors can predict the value and proceed speculatively. When the value is finally generated, if a misprediction is declared, the processor discards the speculative instructions and typically re-executes them.

Data value speculation is being considered for a wide variety of improvements, mostly related to hiding memory or communication latency. Following initial proposals for predicting load values from the L1 [17, 28] or data dependences [6, 22], there has been a flurry of new techniques that can be cast as long-latency data speculation. They include speculating on values loaded on L2 misses [4, 15, 37], independence of instructions following an L2 miss [31], parallelism of threads in Thread-Level Speculation (TLS) (e.g., [11, 16, 30, 33]), values of stale shared data [12], collision-free synchronization operations [20, 25], and ordering of accesses to shared memory [10].

As latencies continue to increase, the number of speculative instructions that need to be kept buffered until a prediction is verified increases as well. To reduce the resulting pressure on critical processor structures such as the instruction window or the register file, several recent proposals have opted for checkpointing the processor state and retiring these speculative instructions [1, 4, 7, 8, 15, 19, 20, 24, 25, 31]. This approach is already used in TLS systems (e.g., [11, 16, 30, 33]), and provides a scalable solution to the challenge of growing speculative state.

Unfortunately, in checkpointed systems, recovery from misprediction can be very expensive. For example, consider an L2 miss that returns a value different than speculatively used, or a TLS task that reads a variable and much later a predecessor task updates it to a different value, or a task executing speculatively beyond a raised barrier that detects a data collision with another task. In these cases, hundreds or thousands of instructions may have executed and *speculatively retired* since the prediction. There is no way to selectively re-execute only the instructions that used the incorrect data. Consequently, the processor rolls back to the checkpoint. The result is a tremendous waste.

Our goal is to selectively re-execute only the speculatively-retired instructions that depended on the mispredicted value, namely its *Forward Slice*. Since such instructions will be shown to be *very few* (Section 6.3), we reduce the amount of work to be redone by a dramatic 97%! The result is faster program execution.

To support selective re-execution of long-retired misspeculated instructions, we need three mechanisms. First, during the initial execution of the speculative instructions, we buffer some minimal state for potential re-execution. Second, after a misprediction, we use the buffered state to re-execute only the slice. Finally, we need to guarantee that the re-execution of the slice repairs the program state.

To address these issues, this paper makes three contributions:
1. It develops a sufficient condition under which re-executing only the slice is *guaranteed* to correctly repair the program. This condition is formulated in a way that can be checked in hardware.
2. It designs a novel generic architecture (*ReSlice*) that is able to selectively re-execute only the instructions in the forward slice of a mispredicted value. These instructions were speculatively retired. ReSlice has two components: one efficiently buffers the slice instructions as they execute; the other can quickly re-execute them with a new value and repair the program state.
3. As one possible application of ReSlice, we use it to recover from cross-task data dependence violations in TLS. ReSlice takes a Chip Multiprocessor (CMP) with an aggressive TLS system running SpecInt applications and further speeds it up by up to 33%, with a

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

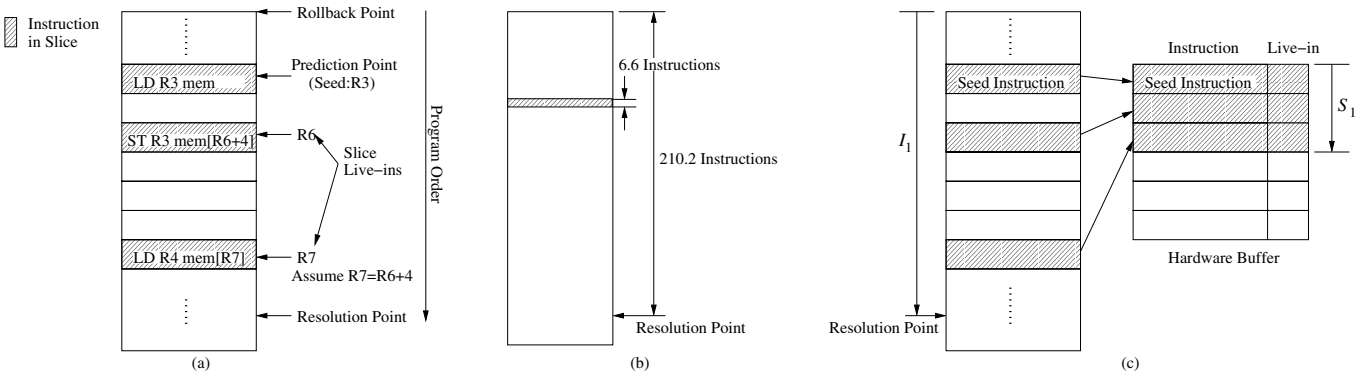


**Figure 1.** A forward slice: slice embedded in a task (a), experimental measures (b), and buffering it for later re-execution (c).

geometric mean of 12%. Moreover, ReSlice decreases the $E \times D^2$ of the system by 20%. All this is accomplished by saving on average 61% of the task squashes through slice re-execution. On average, a slice re-executes only 6.6 instructions, compared to the 210 instructions that would be re-executed on a squash.

This paper is organized as follows. Section 2 describes prior work; Section 3 presents the condition for correct slice re-execution; Section 4 describes ReSlice; and Sections 5 and 6 evaluate it.

## 2. Prior Work

To provide background, we describe prior work in the areas of data value speculation, selective re-execution, and slicing.

**Data value speculation** was used in the context of load values from the L1 [17, 28] and unknown data dependences [6, 22]. As memory and inter-processor communication latencies have grown, speculation has come to be regarded as a promising way to increase ILP and TLP. For example, one use is to speculate on the memory values at L2 misses [4, 15, 37]. Another use is to speculate on the independence of instructions following an L2 miss [31]. TLS (e.g. [11, 16, 30, 33]) speculates on the values of a task's live-ins. Coherence Decoupling [12] speculates that a cached word invalidated by the coherence protocol can still be used. SLE [25] and Speculative Synchronization [20] speculate that threads can skip sychronization operations and not suffer data collisions. Finally, [10] speculates that shared-memory accesses can be aggressively overlapped and not violate the perceived conservative memory model.

As latencies grow, the number of speculative instructions executed until the prediction is verified increases. If these instructions are buffered, they occupy size-critical structures such as the instruction window or register file. Consequently, many of the schemes described [4, 15, 20, 25, 31], including the TLS schemes [11, 16, 30, 33], checkpoint the processor and retire speculative instructions. The same strategy is proposed for checkpointed processors, which typically checkpoint and speculate to reduce stalls or overheads, such as Cherry [19], Virtual ROBs [8], Runahead [24], CPR [1], and Out-of-order commit processors [7]. In all of these proposals, a misspeculation is very costly: potentially many hundreds or thousands of retired instructions are discarded as the processor returns to the checkpoint. Our proposal is to discard and re-execute only a tiny fraction of them.

**Selective re-execution** of only the dependent instructions on a misspeculation has been widely studied in processors where speculative instructions are not retired. For superscalar processors, several designs to recover from load misspeculation have been proposed (e.g., [3, 14]), and implemented in processors such as the Intel Pentium 4 [21] and the AMD Opteron [13]. It has been shown [3] that dependence prediction with selective re-execution in processors with small instruction windows can approach the performance of perfect dependence prediction. For more parallel architectures such as the Trace Processor [27] and TRIPS [9], selective re-execution has also been analyzed. In these architectures, speculative instructions are not retired. Instead, the dependence information of the speculative instructions is kept. This information is used to re-issue only the dependent instructions. Unlike all these works, our design retires speculative instructions.

**Slicing** was first introduced by Weiser [34]. Software-based slicing is frequently used for compiler analysis and debugging [35]. This paper proposes a hardware-only solution to improve performance. Related hardware schemes include backward-slicing schemes. Specifically, Moshovos *et al.* [23] proposed a backward slicer that conceptually walks the dataflow graph in reverse. The backward slice is used by a helper thread to prefetch data. Other researchers have also proposed schemes to generate prefetching slices. Chappell *et al.* [5] use a similar backward-slicing mechanism to help predict branches. Note that backward slices are generated very differently than forward slices and are not useful for recovery.

CFP stashes an instruction that misses in the cache plus its forward slice into a buffer [31]. These instructions are skipped and not executed until the requested data returns from memory. In the meantime, the processor executes independent instructions. Such scheme also relies on speculation, since it has to predict, possibly based on unknown register values, what instructions may or may not belong to the slice being skipped. If any such predictions are incorrect, the processor rolls back to a previous checkpoint. In our paper, we explore a different approach, namely continue executing using a prediction and, if the prediction is later shown incorrect, re-execute the slice.

## 3. Correct Slice Re-Execution

### 3.1. Model of a Forward Slice

The model we use for a forward slice is shown in Figure 1(a). A slice is embedded inside a section of code that we refer to as task. A slice starts with an instruction (typically a load) that sets a register. The register and instruction are called the *Seed* and seed instruction, respectively. The slice is formed with subsequent instructions that are data dependent on the seed through registers or memory locations. For example, in Figure 1(a), the seed instruction is a load into register *R3*. The instruction two positions later belongs to the slice because

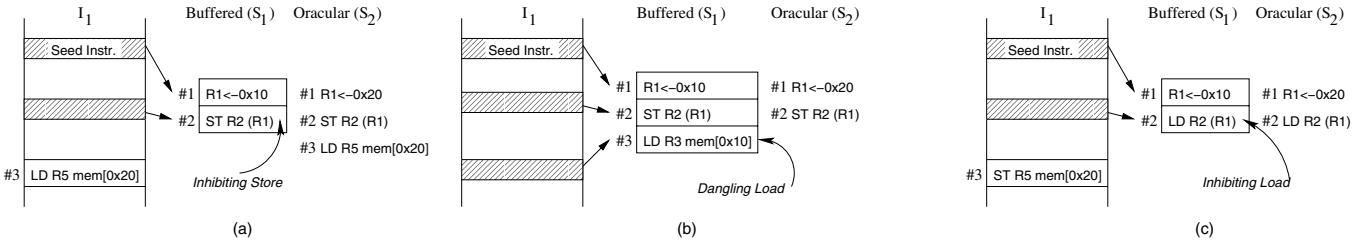


**Figure 2.** Examples of an Inhibiting store, Dangling load, and Inhibiting load. In the examples, the seed's original value was 0x10, while the new value is 0x20.

of a dependence through register *R3*. That instruction is a store to a location that is later read by a third instruction. As a result, the latter also belongs to the slice. In our design, control dependences do not propagate slice membership: the fact that a branch belongs to a slice does not imply that the target and fall-through instructions also belong to it.

A slice has *slice live-ins*, which are values that the slice uses without first defining them. They can be the contents of registers or the contents of memory locations. In Figure 1(a), the live-ins are the contents of registers *R6* and *R7*.

When a task executes, we identify three key points. The *Prediction Point* is where the prediction for the seed value is made. The *Rollback Point* is where the processor checkpoints. Sometimes, the Rollback and Prediction points coincide — for example if we predict the return value of a long-latency load and checkpoint at the same time. In other cases, the Rollback point precedes the prediction. For example, in TLS, the Rollback point is where the speculative task is spawned, while the Prediction point is at a load that may cause a violation. Finally, the *Resolution Point* is when the prediction is verified or rejected. In our examples, this is when the processor receives the requested datum from memory, or a cross-thread dependence violation is detected in TLS. Based on the seed value, a misprediction may be declared.

In conventional checkpointed systems, as soon as a misprediction is declared, execution returns to the Rollback point and the whole task is re-executed. With ReSlice, on a misprediction, the slice at the Prediction point is re-executed, and then execution resumes at the Resolution point.

Figure 1(b) shows the average conditions experimentally measured in the TLS system of Section 6.3 for SpecInt applications. The distance between the Resolution point (or end of task, whichever is earlier) and the Rollback point is 210.2 instructions, while the slice is only 6.6 instructions (not all contiguous). Consequently, in a dependence violation with misspeculation, TLS with ReSlice *only re-executes 3%* of the instructions that TLS would otherwise re-execute!

### 3.2. Why Correct Slice Re-Execution Is Challenging

A slice re-execution is *Correct* only if it repairs the state of the code, enabling the processor to resume execution at the Resolution point. To see why correct re-execution is challenging, consider Figure 1(c). When the task is executed, the instructions that form the dynamic slice of the seed are collected into a hardware buffer, together with the associated slice live-ins. Let $I_1$ denote all the dynamic instructions executed from the Rollback to the Resolution points, and $S_1$ the dynamic instructions in the slice (Figure 1(c)). Let $I_2$ and $S_2$ be entities similar to $I_1$ and $S_1$ for the case in which the task executed with the correct seed value. Since we do not have $I_2$ or $S_2$, we refer to them as *oracular*.

When a misprediction is declared, ReSlice proceeds to re-execute the slice with the correct seed value. Slice re-execution will be correct only if, by feeding the new seed value to the buffered $S_1$, we can transform the program state generated by $I_1$ as if $I_2$ had executed instead.

The main difficulty occurs because, since the new seed value induces changes to the contents of registers (potentially even registers used to generate addresses), re-executed instructions may read and write *different* memory locations than before. Since memory dependences propagate slice membership, the result may be that instructions that did not belong to the slice in the original run, they now do with the correct seed, and vice-versa (formally: $S_2 \neq S_1$). Unfortunately, as we try to re-execute the slice, only the $S_1$ instructions are buffered and available! Moreover, even if $S_2 = S_1$, it may be hard to supply the correct operand values to the buffered instructions. For example, a re-executed load may now read from a new memory location, and the correct value in that location may be unavailable because it was overwritten by a later write in $I_1$.

To identify a sufficient condition that guarantees correct slice re-execution, we isolate the conditions that ensure that: (i) $S_2 = S_1$ and (ii) all instructions in $S_2$ can actually read the correct operand values. For simplicity, we limit our analysis to cases where the new seed value does not change the control flow.

Let us first analyze what makes $S_2 \neq S_1$. We consider first what causes a new instruction to join the slice in the re-execution, and then what causes an old one to leave it. Figure 2(a) shows why an instruction may join. A store that belongs to both the buffered slice ($S_1$) and the oracular slice ($S_2$) writes to a different address as it re-executes. In the example, the store is instruction *#2*, which writes to address 0x10 in $S_1$ and 0x20 in $S_2$. If there was an instruction in the original run ($I_1$) that read from the second address, that load now belongs to $S_2$. In the example, that load is instruction *#3*. Unfortunately, the load read an incorrect value but, because the load is not buffered in our hardware, we have no easy way to re-execute it. We call the store that causes this problem *Inhibiting Store* (Appendix A presents the formal definition). If we find an Inhibiting store, slice re-execution fails, and we revert to the conventional recovery: roll back to the Rollback Point (Figure 1(a)) and re-execute the task fully.

Consider now when an instruction leaves the slice. An example is shown in Figure 2(b). Again, a store writes to a different address as it re-executes. In the example, the store is instruction *#2*, which writes to address 0x10 in $S_1$ and 0x20 in $S_2$. If there was a load that was in the buffered slice ($S_1$) because it read from the first address, then the load is not in the oracular slice ($S_2$). In the example, that load is instruction *#3*. Consequently, this load, while buffered by our hardware, does not belong to the correct slice. Moreover, it read an incorrect value in $S_1$ (an incorrectly modified location 0x10). Unfortunately, simply re-executing it would load the wrong value again — we would need expensive support to re-execute the load and its

forward slice correctly. We call this load a *Dangling Load* (Appendix A presents the formal definition). If we find a Dangling load, slice re-execution fails, and we revert to conventional recovery.

Finally, even if $S_2 = S_1$, slice re-execution fails if an instruction in the slice cannot read correct operand values. Instructions always get correct *register* values, since such values are propagated via the slice or from (unchanged) buffered slice live-ins. However, this is not necessarily true for memory operands. An example is shown in Figure 2(c). A load that belongs to both the buffered slice ($S_1$) and the oracular one ($S_2$) loads from a different address as it re-executes. In the example, instruction *#2* loads from address 0x10 in $S_1$ and 0x20 in $S_2$. If there was an instruction in the original run ($I_1$) that wrote to the second address (instruction *#3* in the example), that second address is now polluted. Re-execution of the buffered slice causes instruction *#2* to incorrectly read the value created by *#3*. We call the load that causes this problem *Inhibiting Load* (Appendix A presents the formal definition). If we find an Inhibiting load, slice re-execution fails, and we revert to conventional recovery.

### 3.3. Sufficient Condition for Correct Re-Execution and Merge

Theorems 3 and 4 in Appendix A prove that, if the control flow path in the slice re-execution is the same as in the original slice execution, and there are no Inhibiting stores, Dangling loads, or Inhibiting loads, then: (i) $S_2 = S_1$ and (ii) $S_2$ instructions can read their correct operands. This is a sufficient condition for correct slice re-execution. Note that it is still possible and acceptable that a given instruction in the slice accesses a different address in the original slice and in its re-execution.

After re-execution, we must merge the state generated by the slice with the up-until-now-speculative program state. The number of cases that can successfully be merged depends on the level of architectural support present. For example, some merge cases may require to undo incorrect cache updates performed in $S_1$. In our design, we log the values overwritten by every first update issued by slice instructions in $S_1$ to an address. In practice, it can be shown that the average number of such updates per slice is between one and two. With this support, Theorem 5 in Appendix A proves that, in the absence of Inhibiting stores, we can successfully merge as long as any cache location that needs to be restored to a value before the slice because it was incorrectly updated by $S_1$, received at most one update in $S_1$.

In the following, we present *ReSlice*, an architecture that is able to correctly buffer, re-execute and merge a slice as long as: (i) branch outcomes in the slice do not change, (ii) there are no Inhibiting stores, Dangling loads, or Inhibiting loads, and (iii) cache locations that need to be restored to a state before the slice, received at most one update in the slice. If this sufficient condition does not hold, recovery from misprediction involves rolling back to the checkpoint and re-executing the whole task from scratch.

## 4. Architecture for Collecting, Re-Executing, and Merging Slices

### 4.1. Overview

To selectively re-execute long-retired, misspeculated instructions, ReSlice needs to support two actions: (i) as the task initially executes, ReSlice buffers the forward slice of the seed and, (ii) later, if needed, it re-executes the slice and merges the resulting state with the program state.

For the first action, ReSlice must be able to detect the seed instruction. The latter could be detected at different locations in the pipeline. For simplicity, in this paper, we assume that we detect it no later than rename — e.g., at instruction fetch, when we see a PC that has historically corresponded to a very costly operation (e.g., an L2-missing load or a load that resulted in a TLS violation).

After the seed is detected, its forward slice is collected. For this, ReSlice follows register and memory dependences as instructions execute, and buffers the instructions of the slice in the *Slice Buffer*. Dependences are followed by tagging physical registers and buffers with a *SliceTag*, which indicates if they hold data belonging to the slice. For data generated by the slice that are stored in the cache, instead of tagging cache lines, ReSlice keeps the addresses with their SliceTags in a small buffer called *Tag Cache*. Finally, ReSlice also buffers the live-in operands of the slice; they are needed to support re-execution and would not be otherwise available at re-execution time.

The second action, slice re-execution, is triggered when the correct seed value becomes available and differs from the predicted one. At this point, task execution is suspended. Then, the buffered slice instructions and their live-ins are fetched and re-executed in the *Re-Execution Unit*. During re-execution, ReSlice checks the condition for correct re-execution described in Section 3.3. If the condition is met then, after re-execution completes, ReSlice merges the state generated by the slice with the program state, and the task is resumed from the Resolution Point. Otherwise, the task is rolled back to the Rollback Point (Figure 1(a)).

ReSlice may need to buffer multiple, possibly overlapping slices per task, which result from different seeds. Consequently, a SliceTag is organized as a bit vector, where bit *i* is set if the corresponding datum or instruction belongs to slice *i*. If the datum or instruction belongs to multiple slices, multiple bits in the SliceTag are set.

Figure 3 shows a flowchart of the actions in ReSlice, while Figure 4 shows the main components of ReSlice. The challenges presented to ReSlice are: (i) efficient slice collection, (ii) fast re-execution and correctness check, (iii) correct merge of the slice state, and (iv) support for overlapping slices. The following subsections describe our solution.

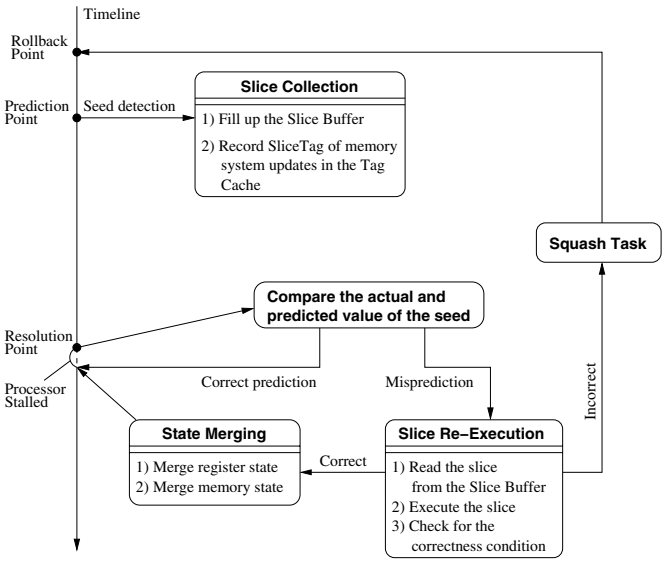


**Figure 3.** Flowchart of the actions in ReSlice.

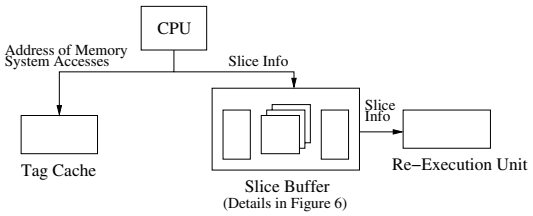


**Figure 4.** Main components of ReSlice.

### 4.2. Slice Collection

During slice collection, ReSlice buffers several pieces of information. First, to be able to re-execute the slice, it buffers the slice instructions and the slice live-ins. Secondly, to be able to check the condition for correct slice re-execution later, it records the outcomes of the branches in the slice and the memory addresses accessed in the slice. Finally, it also needs to buffer the few memory system values that are overwritten by the slice. We will see that these values may be needed to correctly merge the slice state with the program state.

In the following, we describe the key times for slice collection, namely when a seed is detected, an instruction reads operands, and an instruction retires.

#### 4.2.1. Seed Detection

ReSlice uses a high-coverage predictor that can predict seed instructions at or before rename — for example, at instruction fetch based on their PC. Once a seed instruction is detected, it is marked as such. When it is renamed, the processor checkpoints the registers, and ReSlice sets the instruction's SliceTag to a currently-unused slice ID. A slice ID has as many bits as the number of concurrently-supported slices, and only one bit set.

#### 4.2.2. Operand Read

At this time, ReSlice generates two pieces of information: (i) which slice(s) this instruction belongs to, and (ii) which of its two operands are live-ins of which slice(s). To obtain this information, ReSlice reads the SliceTags of its two operands. SliceTags are stored beside the register file, load/store queue and, for operands in the cache, in the Tag Cache. The Tag Cache is a set-associative structure with about 32 entries. It is so small because it can be shown that each slice updates on average between one and two memory locations.

To determine which slice(s) this instruction belongs to, ReSlice logically ORs together the two source operands' SliceTags (Figure 5(a)). If the instruction is a seed, its SliceTag was set to a slice ID. In this case, ReSlice also ORs-in the SliceTag of the instruction, since the instruction also belongs to such slice. In all cases, the result of this OR operation is stored in the SliceTags of the destination operand and of the instruction (Figure 5(a)).

If the result of the OR is zero, no further action is taken because the instruction belongs to no slice. Otherwise, ReSlice needs to determine if any of the two source operands is a slice live-in and, if so, of which slice. This is done as follows. The left source operand is a live-in for all the slices that are in the SliceTag of the right source operand and not in the SliceTag of the left source operand. This can be computed with a logical NOT and AND operation (Figure 5(b)). The result of this operation is a mask that identifies the slices for which the left source operand is a slice live-in. Similar logic is used for the right source operand.

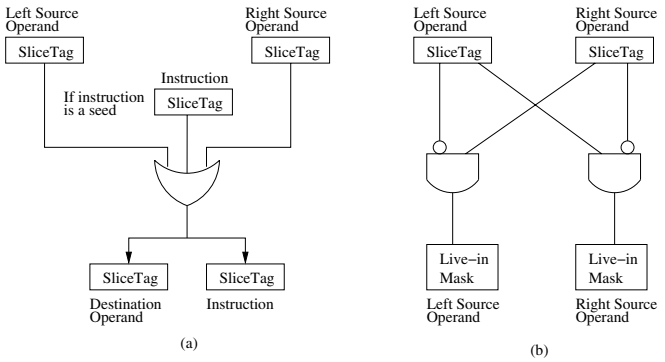


**Figure 5.** Logic used to determine slice membership of instructions and destination operands (a), and slice live-in membership of source operands (b).

#### 4.2.3. Instruction Retirement

The ReSlice state of an instruction is buffered together with the rest of the temporary instruction state until the instruction commits. If the instruction is squashed due to a branch misprediction, its ReSlice state is discarded too. When the instruction retires, its ReSlice state is finally stored in the Slice Buffer and Tag Cache.

The Slice Buffer is shown in Figure 6. It contains several Slice Descriptors (SDs), each of which buffers one slice with instructions in program order. As a result, multiple slices can be buffered concurrently. Each SD entry contains information for one instruction: a pointer (SD.IB) to the decoded instruction in the Instruction Buffer (IB) and, if one of the instruction's source operands is a live-in for this slice, a pointer (SD.SLIF) to the operand's value in the Slice Live-In File (SLIF). In addition, each SD entry has three bits. One indicates if the instruction is a taken branch; the other two indicate which source operand (if any) is stored in the SLIF. For a given instruction in a given slice, at most one of its two source operands can be a slice live-in. This is because at least one of the source operands must be produced by another instruction in the same slice — thereby passing-on slice membership. Note that since multiple slices can share an instruction, multiple SDs may point to the same IB or SLIF entry.

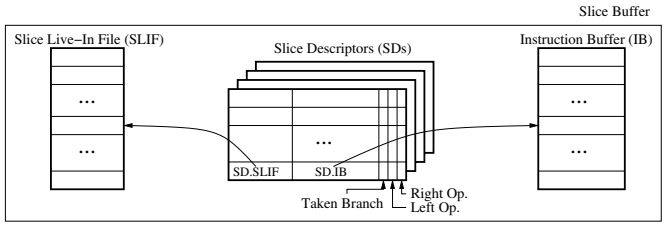


**Figure 6.** Structure of the Slice Buffer.

The IB stores the instructions of all the buffered slices in decoded form and in program order. We use a RISC ISA, where ALU, store, and branch instructions have two registers as source operands, and loads have one register and one memory location as source operands. Indirect branches are unsupported and abort slice buffering.

When an instruction retires, the Slice Buffer is updated as follows. If the instruction is a seed, a new SD is allocated. If the instruction's SliceTag shows that it belongs to one or more slices, the instruction is buffered in the corresponding SD(s). This is done by first saving the retiring instruction in the IB. Then, the live-in masks of its source operands (Figure 5(b)) are checked to see if any of the operands is a live-in of any slice. If so, ReSlice saves the corresponding source

operand(s) in the SLIF. Then, for *each* of the slices to which the instruction belongs, ReSlice fills one SD entry as follows: (i) the *SD.IB* field is set to point to the instruction in the IB, and (ii) the *SD.SLIF* field is set to point to the slice live-in for this instruction (if any) in the SLIF. Recall that, for each slice, an instruction can at most have a single slice live-in. ReSlice also sets the SD entry's *LeftOp*, *RightOp*, and *TakenBranch* bits (Figure 6) appropriately.

When the retiring instruction being copied to the IB is a load or a store, ReSlice also stores the address being read/written in the subsequent IB entry. This simplifies slice re-execution (Section 4.3).

Finally, if the slice instruction is a store, two operations are performed as the datum is speculatively written to the cache[1]. First, the datum's SliceTag is saved in the Tag Cache. Secondly, if this is the first update to the location in the slice, the contents of the location before the update are saved in a small *Undo Log* buffer. The Undo Log helps state merging (Section 4.4).

### 4.3. Slice Re-Execution

After a misprediction is detected (Figure 3), ReSlice initiates recovery. The processor's pipeline is flushed and stalled. The Re-Execution Unit (REU) takes over. Given that a slice re-execution involves, on average, only 6.6 instructions (Section 6.3), there are several options for the REU design. It can be a simple core with a small register file or it can be a piece of firmware that uses the resources of the processor — this is implementation dependent.

The REU starts with a clean register file and re-executes a slice by processing, in order, the entries in the target SD. From an SD entry, it follows the pointer to fetch the decoded instruction from the IB and, if either the *LeftOp* or the *RightOp* bit is set, a slice live-in from the SLIF. As it executes, the REU may access memory system locations, although the cache is not modified until the re-execution process completes.

During re-execution, the REU checks for the sufficient condition for correct slice re-execution (Section 3.3): unchanged branch directions and the absence of Dangling loads, Inhibiting loads, and Inhibiting stores. Branch directions are checked against the *TakenBranch* bit in the SD. The other requirements are checked by comparing, for each load and store in the slice, the address accessed in the re-execution and the one accessed in the initial run. Recall that the latter is stored in the IB after the entry for the load or store.

More specifically, on a store, the REU compares the new and old addresses. If they are the same, the current store is not Inhibiting. Otherwise, the REU checks if the new address was accessed in the initial run of the task ($I_1$ in Section 3.2). This is done by checking if the referenced word is in the cache with the Speculative Read or Write bits set. These bits were set for all the words read or written, respectively, in the initial, speculative run of the task, as is typical in TLS systems. If any of these two bits is set, this is an Inhibiting store (see Section 3.2 and its formal definition in Appendix A).

On a load, the REU compares the new and old addresses. If they are different, the REU checks if the new address was written in the initial run (again, by checking the Speculative Write bit in the cache). If so, this is an Inhibiting load (Section 3.2). If, instead, the old and new addresses read are the same, the REU checks for a Dangling load. For that, the REU searches backwards the stores in the original execution of the slice ($S_1$) — recall that it has just read each of them. If it finds a matching address, it means that this particular store instruction produced the data for the load in the initial run. If this same store in the new run ($S_2$) generates a different address, then the load we are processing is a Dangling load (Section 3.2).

In practice, these checks are very fast. Indeed, Section 6.1 shows that, even if we do not restrict the types of slices that we support, the average re-executed slice reads only one word and has a two-word update footprint.

### 4.4. State Merging

After the slice is proven to have re-executed correctly, the REU merges the register and memory state that it has generated with the main program state. Consider registers first. Let us call $R_1$ the set of architectural registers defined in the initial execution $S_1$ of the slice, and $R_2$ the set defined in the re-execution $S_2$. Since a correct slice re-execution involves executing the same instructions as in the original execution, $R_1$ is equal to $R_2$. To perform the register merge, the REU takes its $R_2$ registers and, using the rename table of the processor stalled at the Resolution Point, accesses their corresponding current physical registers. Then, it checks for liveness of the register updates performed in the initial slice execution, i.e., if in the SliceTag of these physical registers, the bit corresponding to the re-executed slice is still set. For those where it is set, the physical register in the processor is updated with the value in the REU.

Consider now memory state. Let us call $M_1$ the set of memory locations defined in the initial slice execution and $M_2$ those defined in the re-execution. The addresses of $M_1$ and $M_2$ are collected by the REU as it re-executes a slice (Section 4.3). To create the correct memory state at the Resolution Point, we require: (i) for locations that are in $M_1$ and not in $M_2$, potentially undo the initial execution's update, and (ii) for locations that are in $M_2$, potentially perform the re-execution's update.

To perform action (i), the REU takes each address in $M_1$ that is not in $M_2$ and checks its SliceTag in the Tag Cache. If the bit corresponding to the slice is still set, then the update is still alive. To undo it, the REU takes the corresponding value from the Undo Log and writes it to the cache[2]. To perform action (ii), the REU considers each address in $M_2$ in turn. The REU updates the cache with the new update only if the update is live at the Resolution Point. The latter is true in two cases. First, if the Tag Cache contains an entry for this address and, in its SliceTag, the bit corresponding to this slice is still set. Second, if the Tag Cache does not contain an entry for this address.

Note that, in TLS, the cache updates performed during state merging may prompt the coherence protocol to propagate invalidations to other caches, and possibly cause the re-execution of slices in successor speculative tasks.

### 4.5. Supporting Overlapping Slices

The algorithm described supports re-executing a given slice multiple times. This is useful in TLS, where the location read by a seed instruction may receive multiple updates; after each update to a new value, ReSlice runs the algorithm described. It is also useful when slices overlap, i.e., they share instructions. Given two overlapping slices, when we detect the first misprediction of a seed value (say,

[1] Like in many TLS systems, we assume that the cache can store the data read or written by the speculative task, and that it marks them with Speculative Read and Write bits. Other designs are possible.

[2] As proven in Theorem 5 of Appendix A, this undo will repair the state unless this address had already been undone before, or the address was updated multiple times in the initial slice execution. In these rare cases, re-execution is aborted.

seed *i*), we re-execute the slice (slice *i*) as usual. However, when we later detect the misprediction of the other seed value (say, seed *j*), we have to re-execute both slices (slices *i* and *j*) together. The reason is that the first slice re-execution may have changed the live-ins for the second slice and, therefore, rendered the live-in values of the second slice in the SLIF stale.

As an example, Figure 7(a) shows code with two overlapping slices: one with instructions 1-3-4, and the other with 2-3-4. Figure 7(b) shows that each slice has its own SD and one live-in in the SLIF, namely the contents of R3 for the first slice and the contents of R4 for the second one. If *Address2* receives a new value, slice 2-3-4 re-executes, using as live-in the value of R4 saved in the SLIF. If, later, *Address1* receives a new value, re-executing only slice 1-3-4 is incorrect: we would use as live-in the value of R3 in the SLIF, which is stale. Instead, we need to re-execute both slices concurrently. Next, we show how ReSlice detects and handles overlap.

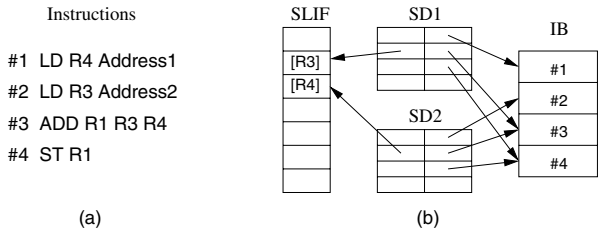


**Figure 7.** Supporting overlapping slices.

#### 4.5.1. Detecting Overlap

Slice overlap can occur at two different times. Typically, it is detected as the slices are collected. In the example of Figure 7(a), when instruction *3* retires, its SliceTag shows that it belongs to two slices. Consequently, the instruction is buffered in two SDs. These SDs are marked with an Overlap bit.

It is also possible that two slices that did not overlap as they were collected end up overlapping when one of them re-executes. This can occur when a re-executing slice accesses different memory locations than in its initial execution, and these locations overlap with another slice. This case, however, is uninteresting because it results in a failed re-execution according to our condition in Section 3.3. The intuitive reason is that the re-execution ends up adding additional instructions to one slice and, therefore, $S_1 \neq S_2$. A proof can be given in terms of Inhibiting stores and Dangling loads.

#### 4.5.2. Handling Overlap

To handle overlap, ReSlice supports the concurrent execution of multiple slices. When a slice that has the Overlap bit set is to be re-executed, ReSlice also triggers the concurrent execution of all the other slices in the task that have the Overlap bit set and have already re-executed.

To support the concurrent execution of multiple slices, ReSlice has two features. First, it enforces the *in-order* execution of the instructions of the combined slice. Specifically, as it examines SD1 and SD2 in Figure 7(b) top down, it always selects the pointer with the smallest offset as the next instruction to execute.

Second, when ReSlice is about to execute an instruction that is shared by multiple re-executing slices, it only takes a live-in from the SLIF if all the re-executing slices "agree" (i.e., all SDs point to the *same* SLIF entry). Otherwise, the REU uses instead the current value in its register file. For example, for instruction *3* in Figure 7(a), ReSlice reads R3 and R4 from the REU register file because neither is a live-in common to both slices. Indeed, as shown in Figure 7(b), the second entry in each SD points to a different SLIF entry. ReSlice would take the same action if one of the two pointers were nil.

Theorem 6 in Appendix A proves that, by combining the slices in this way, the condition for correct re-execution is the same as for a single slice (Section 3.3). For simplicity, ReSlice supports the concurrent execution of at most three slices.

Section 6.3 compares ReSlice to a scheme that does not support the concurrent execution of multiple slices (*NoConcurrent*). In such scheme, when a slice with the Overlap bit set needs re-execution, if there is another slice in the task with the Overlap bit set that has already re-executed, we squash the task.

## 5. Evaluation Methodology

As one application of ReSlice, we use it to recover from cross-task dependence violations in a TLS CMP. For this, we use a TLS compiler that produces a binary with tasks [18, 26]. We also use an execution-driven cycle-accurate simulator with detailed models of out-of-order superscalars, a memory subsystem, and a TLS protocol. It includes the models of dynamic power from Wattch [2] and Cacti [29], and static power from HotLeakage [36].

The baseline architecture modeled is a 4-core CMP with TLS support (*TLS*). We are interested in the impact of adding ReSlice support (*TLS+ReSlice*). As a reference, we also consider a non-TLS, single-superscalar chip (*Serial*).

The parameters of the architectures are shown in Table 1. We model 3-issue out-of-order cores. In *TLS*, each core has a private L1 that buffers speculative state. The TLS cache coherence protocol is similar to [16]. To account for any complexity that TLS adds to the L1, we set its access time to 3 cycles. The L2 is shared and does not store speculative data. Each processor also has a cross-task dependence and value predictor (Section 5.1).

*TLS+ReSlice* extends *TLS* with the ReSlice parameters shown in the rightmost column of Table 1. In particular, ReSlice can buffer up to 16 slices at a time of 16 instructions each. The ReSlice hardware adds up to about 2.4 Kbytes per core, about 512 bytes in the predictor chip wide, and the REU. The REU is a tiny in-order core.

*Serial* has one core, L1, and L2 in a chip. We do not change the cache sizes to avoid affecting the cycle time. However, since TLS is not supported, we reduce the L1 access time by one cycle to 2.

These architectures run the SpecInt 2000 codes with the *Ref* input data set. The exceptions are *eon* (written in C++), *gcc*, and *perlbmk*, which our TLS compiler does not fully support. *Serial* runs unmodified, uninstrumented binaries, while *TLS* and *TLS+ReSlice* run the binaries generated by our TLS compiler. In our simulations, we skip the initialization (1-6 billion instructions), and then execute the applications by about 0.75-1.50 billion *Serial* instructions.

### 5.1. Dependence and Value Predictor

Both *TLS* and *TLS+ReSlice* have a cross-task dependence and value predictor like the one in [32]. The dependence predictor is built with a per-core 4-entry CAM called Temporary Dependence Buffer (TDB), and a 4-way 512-entry shared (but distributed) global Dependence and Value Predictor (DVP). The DVP is PC-indexed and each entry has a 2 bit confidence count.

When a dependence violation occurs, the address that caused it is inserted in the TDB. As the squashed consumer task is immediately

| Processor | Cache | D-L1 | I-L1 | L2 | ReSlice Parameters | | | |
|---|---|---|---|---|---|---|---|---|
| Frequency: 5.0 GHz @ 70 nm | Size: | 16KB | 16KB | 1MB | | #Units | #Entries | Width (Bits) |
| Fetch/issue/comm width: 6/3/3 | RT: | 3 cyc | 2 cyc | 10 cyc | | | | |
| I-window/ROB size: 68/126 | | (2 cyc in no TLS) | | | IB | 1 | 160 | 40 |
| Int/FP registers: 90/68 | Assoc: | 4-way | 2-way | 8-way | SD | 16 | 16 | 18 |
| LdSt/Int/FP units: 1/2/1 | Line size: | 64B | 64B | 64B | SLIF | 1 | 80 | 32 |
| Ld/St queue entries: 48/42 | Pend ld/st: | 16 | - | 64 | Tag Cache | 1 | 32 | 48 |
| Branch penalty: 13 cyc (min) | DVP: 512 entries, 4-way assoc | | | | Undo Log | 1 | 32 | 80 |
| BTB: 2K entries, 2-way assoc. | Confidence bits: 2 (+2 to predict buffering in ReSlice) | | | | | | | |
| Branch predictor (spec. update): | Bus & Memory: DDR-2 | | | | REU (in-order core) | | | |
| bimodal size: 16K entries | Bus frequency: 533MHz; Bus width: 128bit | | | | Registers: 16 | | | |
| gshare-11 size: 16K entries | DRAM bandwidth: 8.528GB/s; memory RT: 98ns | | | | | | | |

**Table 1.** Parameters of the architectures modeled. In the table, DVP and RT stand for Dependence and Value Predictor, and minimum Round-Trip time from the processor, respectively. Cycle counts are in processor cycles.

| App. | #Insts per Slice | #Branches per Slice | #Insts Seed to End | #Insts Roll to End | #Insts per Task | #Live Ins per Slice | | Update Footprint per Slice | | #Slices per Task | %Tasks with Overlapping Slices | Buffering Predictor Coverage |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Reg | Mem | Reg | Mem | | | |
| bzip2 | 3.9 | 0.05 | 138.0 | 185.9 | 983.6 | 1.90 | 0.04 | 1.12 | 0.81 | 1.20 | 0.4 | 0.98 |
| crafty | 8.0 | 0.97 | 290.4 | 382.0 | 913.7 | 4.66 | 0.25 | 2.31 | 1.65 | 1.59 | 14.7 | 0.88 |
| gap | 27.9 | 2.20 | 193.7 | 251.6 | 1755.2 | 8.33 | 1.92 | 3.64 | 4.16 | 3.56 | 24.0 | 0.65 |
| gzip | 4.9 | 0.13 | 31.5 | 118.4 | 661.4 | 1.91 | 0.01 | 1.24 | 1.35 | 1.27 | 15.0 | 0.97 |
| mcf | 20.1 | 4.59 | 33.1 | 58.9 | 53.8 | 5.97 | 6.43 | 4.73 | 3.06 | 1.01 | 0.0 | 0.99 |
| parser | 10.5 | 0.44 | 135.2 | 232.1 | 303.8 | 5.64 | 0.31 | 2.18 | 2.23 | 2.08 | 34.2 | 0.95 |
| twolf | 10.0 | 1.08 | 98.8 | 194.6 | 406.8 | 6.20 | 0.00 | 2.40 | 1.27 | 1.37 | 18.3 | 0.95 |
| vortex | 6.5 | 0.13 | 200.9 | 295.4 | 1846.7 | 5.03 | 0.03 | 1.89 | 2.42 | 1.00 | 0.0 | 0.60 |
| vpr | 1.8 | 0.03 | 175.3 | 362.1 | 453.5 | 0.57 | 0.03 | 0.15 | 0.40 | 1.47 | 28.0 | 0.99 |
| Avg. | 10.4 | 1.07 | 144.1 | 231.2 | 819.8 | 4.47 | 1.00 | 2.18 | 1.93 | 1.62 | 15.0 | 0.89 |

**Table 2.** Characterizing the slices that are re-executed. The data assumes ReSlice structures of unlimited size.

re-executed, its load addresses are checked against the TDB. When there is a match, the PC of the load is inserted into the DVP, and its confidence is set to the maximum value. The assumption is that this PC is likely to be involved in future dependences. At 100K cycle intervals, the counters are decremented; if one goes below zero, it is invalidated.

The DVP includes a value predictor that uses a combination of hardware and software to provide a value when a dependence is predicted [32]. It is a hybrid predictor that combines a last-value predictor and an incremental predictor, with confidence counters to select between the two. If the PC of a load hits a valid DVP entry, the load uses the predicted value.

In *TLS+ReSlice*, to predict *when to buffer* a slice, we use the same DVP but extend it with two additional bits per entry to give it higher coverage. This is because coverage is important for buffering. *Coverage* is the fraction of violations that, thanks to the predictor, find the corresponding slice buffered. Using 4 bits per entry gives us an average coverage of 89% (Section 6.1), at a cost of a modest increase in size (2 bits per entry), management overhead, and energy consumption. With this support, if a load hits a valid DVP entry, the load is marked as a seed and slice buffering begins. If, in addition, the confidence is high enough to predict a dependence (the two most significant bits in the DVP entry are both set), the predicted value is used; otherwise, the current value is used. This gives *TLS+ReSlice* the same accuracy of value prediction as in plain *TLS*, and at the same time high buffering coverage.

## 6. Evaluation

In this section, we first characterize forward slices, then assess ReSlice's impact on performance and energy, and finally analyze various ReSlice architectural parameters.

### 6.1. Characterizing Forward Slices

For this section only, we assume that the ReSlice structures of Figures 4 and 6 have unlimited size. Table 2 characterizes the forward slices of loads that cause violations — namely, the slices that are re-executed. We observe that, even with unlimited resources, the slices are small, containing 10.4 dynamic instructions per slice on average (Column 2). Secondly, the number of branches per slice is just 1.07 on average (Column 3). This data matches the observation in [31] that only around 10% of the instructions in a slice are branches. Columns 4 and 5 show the distance from the seed to the end, and from the Rollback Point to the end, respectively. The end point is either when the seed value becomes available, or when the task ends, whichever is earlier. The average distance between the rollback and the end points is 231.2 instructions, which is *22 times* larger than the slice size. Hence, if we can re-execute slices correctly, we can eliminate substantial wasted work. Column 6 shows the average size of committed tasks[3].

Columns 7-8 and 9-10 show the average number of slice live-ins and average update footprint size, respectively, organized into register and memory variables. Note that live-ins for the seed instruction are not included. On average, these slices have about four registers and one memory location as live-ins, and two registers and two memory locations as update footprint. Columns 11 and 12 focus on the tasks that have slices. These tasks have on average 1.62 slices. Moreover, on average 15% of these tasks have overlapping slices. This data motivates supporting overlapping slices per task. Finally, the last column shows the buffering coverage of the DVP. We can see that, other than in *gap* and *vortex*, coverage is very high. The average coverage across all programs is 0.89.

[3]Note that, in *mcf*, the average size of the committed tasks is smaller than the average distance between rollback and end for the tasks with violations.

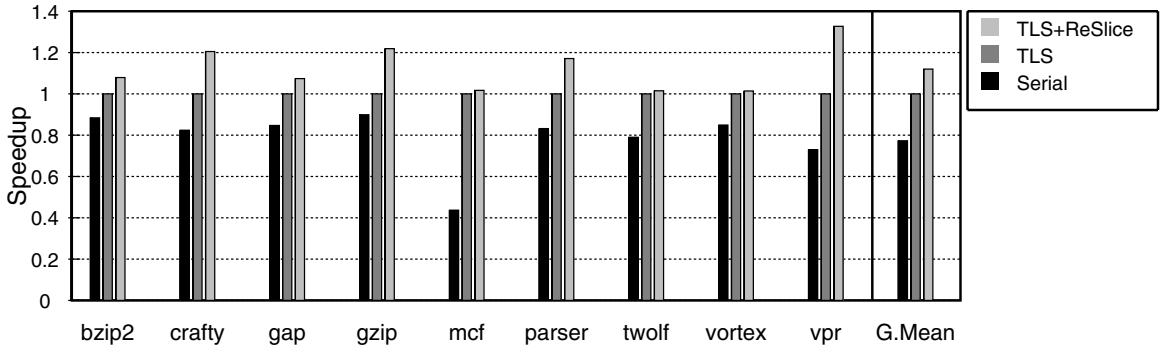


**Figure 8.** Speedup of *TLS+ReSlice* over *TLS*.

Figure 9 characterizes slice re-executions. A slice re-execution can be Successful or Failed. A successful re-execution is one that satisfies the sufficient condition of Section 3.3. Successful re-executions are divided into two classes, based on whether or not all the loads and stores in the re-execution access the same addresses as in the initial execution. Failed slices are classified according to the first instruction that causes a failure: a different branch outcome, a Dangling load, an Inhibiting load, or an Inhibiting store.

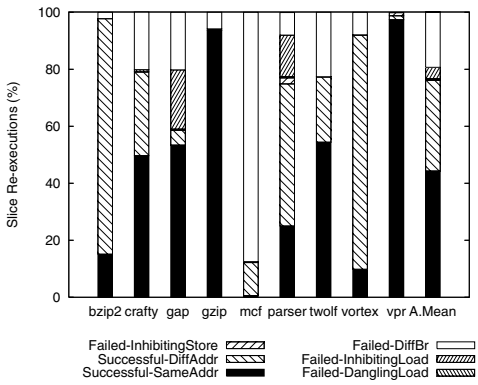


**Figure 9.** Characterizing slice re-executions.

As shown in Figure 9, most re-executions are successful. On average, 76% of the re-executions are successful. Of them, slightly more access the same memory locations as before (44% of all re-executions) than not (32% of all re-executions). This data justifies using a model in Section 3.3 that tries to salvage re-executions that access different addresses than the initial execution. We also observe that the major reason why re-executions fail is control flow changes.

A task with slices can complete without squash only if all re-executions of its slices are successful. Figure 10 considers all tasks with slice re-executions and groups the tasks depending on how many slice re-executions they have (1, 2, 3, or more). Then, tasks are classified into Salvaged (all re-executions succeed) or Squashed (at least one fails). To see the results better, consider the average bars in the figure. We see that about 20% of these tasks have two or more re-executions. Moreover, if we add up all the black sections of the bars, we see that about 70% of the tasks avoid squashes by always re-executing slices successfully. This is an encouraging result.

### 6.2. Performance and Energy Impact

To evaluate ReSlice, we assess its impact on performance and energy. We now use the limited resources of Table 1. Figure 8 shows

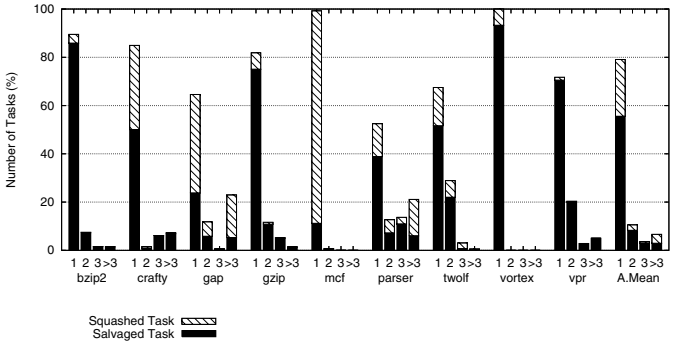


**Figure 10.** Tasks with slice re-executions. The numbers below the bars are the number of slice re-executions per task.

the speedups of *TLS+ReSlice* over *TLS*. As a reference, we also show *Serial*. The figure shows that *TLS+ReSlice* outperforms *TLS* in all applications. The speedups are often significant; they have a geometric mean of 1.12, and reach up to 1.33. Note also that our *TLS* architecture is highly optimized. It is on average 29% faster than *Serial* for these SpecInt codes (not just loops). This number compares favorably to previously published numbers. Overall, therefore, our ReSlice support builds on top of an aggressive TLS system to deliver a total average speedup of 45% over *Serial*.

To understand these results, we break down the number of cycles $n_{app}$ taken by the execution of an application as follows:

$$n_{app} = \underbrace{\left(\frac{n_{app}}{\sum_{i=1}^{ncores} n_i}\right)}_{\frac{1}{f_{busy}}} \times \underbrace{\left(\frac{\sum_{i=1}^{ncores} n_i}{\sum_{i=1}^{ncores} I_i}\right)}_{\frac{1}{IPC}} \times \underbrace{\left(\frac{\sum_{i=1}^{ncores} I_i}{I_{req}}\right)}_{f_{inst}} \times I_{req}$$

$$= \frac{f_{inst}}{f_{busy} \times IPC} \times I_{req}$$

In this formula, $n_i$ is the number of cycles during which core $i$ is busy. $I_i$ is the number of instructions retired by core $i$, including those of squashed tasks and re-executed slices. $I_{req}$ is the total number of instructions retired in the program assuming no task squashes or slice re-executions; it is the same for *TLS* and *TLS+ReSlice*. We can then re-write the formula using three factors: the average number of busy CPUs ($f_{busy}$), the average IPC of the cores ($IPC$), and the ratio of retired to required($I_{req}$) instructions ($f_{inst}$).

As we move from *TLS* to *TLS+ReSlice*, these three factors change (Table 3). To see why, recall that ReSlice's goal is to eliminate squashes. Columns 2-3 of Table 3 show the number of tasks squashes per task commit in *TLS* and *TLS+ReSlice*. Since the number of task

commits does not change, we clearly see the impact of ReSlice: on average, it reduces the number of squashes per commit by 61%, from 0.80 to 0.31. The reduction occurs across all the applications, and is very significant for *bzip2*, *gap*, and *vpr*.

| App. | # Task Squashes per Commit | | $f_{inst}$ | | $f_{busy}$ | | *IPC* | |
|---|---|---|---|---|---|---|---|---|
| | *TLS* | *T+R* | *TLS* | *T+R* | *TLS* | *T+R* | *TLS* | *T+R* |
| bzip2 | 1.34 | 0.01 | 1.26 | 1.13 | 1.65 | 1.60 | 1.23 | 1.22 |
| crafty | 0.75 | 0.22 | 1.29 | 1.16 | 1.72 | 2.01 | 1.46 | 1.35 |
| gap | 2.99 | 1.98 | 1.69 | 1.51 | 1.99 | 1.97 | 1.21 | 1.18 |
| gzip | 0.08 | 0.04 | 1.01 | 1.12 | 1.20 | 1.81 | 1.21 | 1.08 |
| mcf | 0.19 | 0.14 | 1.04 | 1.04 | 2.88 | 2.91 | 0.49 | 0.49 |
| parser | 0.23 | 0.07 | 1.34 | 1.28 | 2.27 | 2.56 | 0.83 | 0.83 |
| twolf | 0.22 | 0.06 | 1.07 | 1.01 | 1.61 | 1.60 | 0.45 | 0.43 |
| vortex | 0.29 | 0.22 | 1.07 | 1.13 | 1.34 | 1.53 | 1.39 | 1.31 |
| vpr | 1.12 | 0.02 | 1.52 | 1.06 | 2.31 | 2.40 | 1.08 | 0.96 |
| Avg. | 0.80 | 0.31 | 1.25 | 1.16 | 1.89 | 2.04 | 1.04 | 0.98 |

**Table 3.** Characterizing the run-time impact of ReSlice. *T+R* represents *TLS+ReSlice*.

This reduction in squashes helps in two ways: (i) it reduces the number of instructions executed by the processors and (ii) it exposes more parallelism, as the serializing operation of squashing a task and all its successors, and gradually re-spawning them occurs less frequently.

The reduction in instructions can be seen in Columns 4-5 of Table 3, which show $f_{inst}$. On average, ReSlice reduces $f_{inst}$ from 1.25 to 1.16. As shown in the formula above, this reduction directly reduces the number of cycles taken by the application. There are a few applications where $f_{inst}$ remains constant or increases slightly. These applications (*gzip*, *mcf*, and *vortex*), are those with the smallest reduction in squashes (Columns 2-3). In these applications, it sometimes saves instructions to squash a task early-on with *TLS* rather than to postpone the squash with *TLS+ReSlice*. Indeed, an early squash may avoid later violations in the task. With *TLS+ReSlice*, this early squash is avoided, but new violations that cannot be fixed with re-execution appear later on.

The second benefit of squash reduction, namely exposing more parallelism, can be seen in Columns 6-7 of Table 3. The data shows that, on average, ReSlice increases $f_{busy}$ from 1.89 to 2.04.

Finally, ReSlice induces a slight reduction in the average $IPC$. This can be seen in Columns 8-9. The main reason is the higher contention for chip resources induced by *TLS+ReSlice*'s higher parallelism. Overall, the reasons for *TLS+ReSlice*'s speedups are a lower $f_{inst}$ and a higher $f_{busy}$, which offset the small losses in $IPC$.

Next, we examine the energy consumption and the $E \times D^2$ of ReSlice. ReSlice adds new hardware structures that consume additional energy. On the other hand, it reduces the total number of instructions and squashes, and saves time.

Figure 11 compares the energy consumption in *TLS+ReSlice* and *TLS*, normalized to *TLS*. The *TLS+ReSlice* bars are broken down into non-ReSlice structures (*Base*), and ReSlice structures for slice logging, dependence prediction, and slice re-execution. We see that, on average, the new structures add about 7% to the original energy consumption, while the instruction reduction saves 5% of the original energy. Overall, ReSlice adds a negligible 2% energy overhead. Across applications, the bars are loosely correlated with $f_{inst}$ (Table 3). For example, in *gzip*, *TLS+ReSlice* consumes significantly more than *TLS* because it executes more instructions, while the opposite occurs in *vpr*.

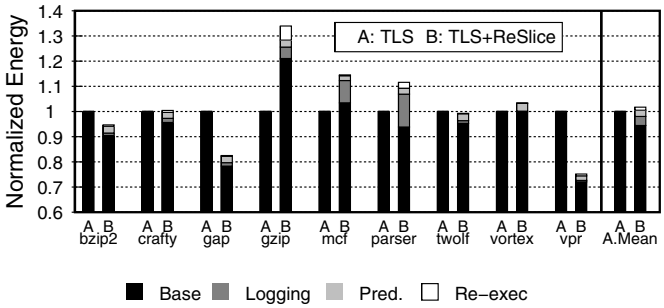


**Figure 11.** Comparing the energy consumed.

Figure 12 compares $E \times D^2$ for *TLS+ReSlice* and *TLS*. We see that *TLS+ReSlice* reduces $E \times D^2$ in 6 out of 9 applications. The geometric mean reduction is 20%.

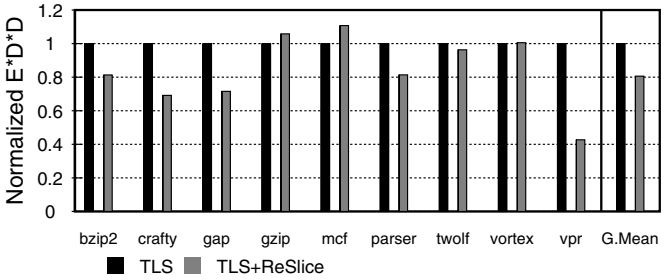


**Figure 12.** Comparing $\mathrm{Energy} \times \mathrm{Delay}^2$.

### 6.3. Architectural Analysis

**Structure Utilization.** Table 4 shows the utilization of the ReSlice structures. In the table, utilization numbers are computed as follows. For each successfully committing task that has at least buffered one slice (*buffering task*), we measure the number of entries that the task has used in each ReSlice structure. Then, we average these numbers across all buffering tasks.

From Columns 2 and 3, we see that, on average, a buffering task buffers 9.7 slices, where each slice has 6.6 instructions. Note that this average slice size is smaller than the average slice size shown in Table 2, which was 10.4 instructions. The reason is that Table 2 assumed unlimited resources. In Table 4, slices over 16 instructions are discarded.

Column 4 shows that, on average, the distance between the rollback and end points is 210.2 instructions. Consequently, ReSlice on average *re-executes only about 3.1%* of these instructions! The rest of the table shows the utilization of IB and SLIF. Note that IB contains both instructions and addresses. The *NoShare* column shows the number of IB entries used if sharing of entries between slices was

| App. | # SDs | # Inst./ SD | #Insts Roll to End | # IB Entries | | # SLIF Entr. |
|---|---|---|---|---|---|---|
| | | | | Total | No-Share | |
| bzip2 | 11.4 | 6.4 | 189.5 | 91.8 | 99.7 | 45.3 |
| crafty | 15.3 | 7.2 | 396.8 | 126.9 | 145.7 | 76.0 |
| gap | 14.7 | 6.0 | 189.7 | 120.9 | 131.4 | 42.2 |
| gzip | 11.5 | 7.6 | 120.8 | 95.9 | 114.5 | 43.3 |
| mcf | 4.0 | 12.0 | 78.1 | 72.8 | 73.3 | 18.9 |
| parser | 8.8 | 5.7 | 206.4 | 66.0 | 77.3 | 31.6 |
| twolf | 10.6 | 4.7 | 161.9 | 57.8 | 68.1 | 31.5 |
| vortex | 5.0 | 3.2 | 317.8 | 24.7 | 25.0 | 10.6 |
| vpr | 6.4 | 6.4 | 230.6 | 47.8 | 47.9 | 22.4 |
| A.Mean | 9.7 | 6.6 | 210.2 | 78.3 | 87.0 | 35.8 |

**Table 4.** Utilization of the ReSlice structures.

not allowed. We see that around 11% of the entries are shared. Consequently, our structures save space by leveraging sharing.

**Supporting Overlapping Slices.** ReSlice supports the concurrent re-execution of overlapping slices. Figure 13 compares ReSlice to a scheme that can only re-execute one slice at a time (*NoConcurrent* from Section 4.5.2), and a scheme where only one slice per task is ever re-executed (*1slice*). We can see that, on average, *1slice* only delivers a speedup of 1.08, the conservative *NoConcurrent* only manages 1.09, and ReSlice delivers 1.12. This data motivates our choice of the ReSlice scheme.

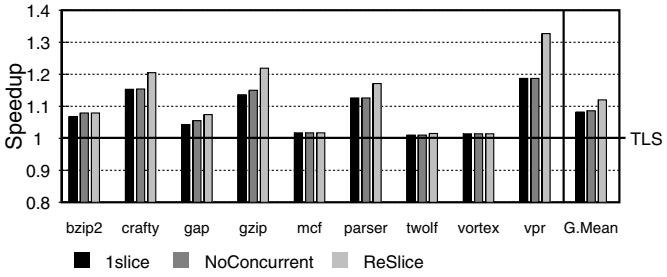


**Figure 13.** Impact of fully supporting overlapping slices.

**Perfect Environments.** Figure 14 evaluates the performance possible in three perfect environments. Perfect coverage (*Perf-Cov*) means that, when a violation is detected, the corresponding slice is always found buffered. Perfect re-execution (*Perf-Reexec*) means that the re-execution of every buffered slice is always correct. These two schemes improve performance by 3%, and their combination (*Perfect*) improves it by 6%. This data shows that ReSlice delivers most of the potential for selective re-execution.

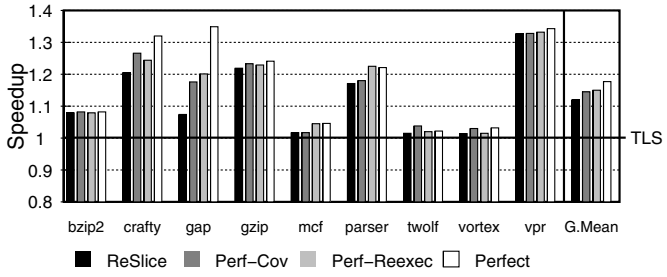


**Figure 14.** Comparison with perfect coverage and/or re-execution.

## 7. Conclusion

In TLS and other checkpoint-based architectures, data value mispredictions are often detected only after hundreds or thousands of speculative instructions have retired. At that point, all these instructions are re-executed. To reduce this waste, this paper made three contributions. First, it developed a sufficient condition under which slice re-execution and merge are guaranteed to correctly repair the state of a program. Secondly, it proposed *ReSlice*, a broadly-usable architecture that (i) buffers the forward slice as it is executed with a predicted value, and (ii) can quickly re-execute the slice with the correct value much later and merge the resulting state with the program state. Thirdly, this paper evaluated ReSlice. ReSlice sped-up a CMP with already aggressive TLS running SpecInt applications by up to 33%, with a geometric mean of 12%. Moreover, $E \times D^2$ decreased by 20%. These improvements were accomplished by saving on average 61% of the task squashes through slice re-execution. On average, a slice re-executed only 6.6 instructions, compared to the 210 instructions that would be re-executed on a squash.

With the number of possible avenues for enhancing ILP or TLP in SpecInt-type codes decreasing, and different forms of speculation becoming more prominent, recovering wasted work as ReSlice does is a promising approach to boost speedups.

## References


[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO*, Dec. 2003.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, June 2000.

[3] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2000.

[4] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 misses with checkpoint assisted value prediction. *Computer Architecture Letters*, Dec. 2004.

[5] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-path branch prediction using subordinate microthreads. In *ISCA*, May 2002.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA*, June 1998.

[7] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *HPCA*, Feb. 2004.

[8] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002-39, Universitat Politecnica de Catalunya, July 2002.

[9] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler. Scalable selective re-execution for edge architectures. In *ASPLOS*, Oct. 2004.

[10] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, May 1999.

[11] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, Oct. 1998.

[12] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *ASPLOS*, Oct. 2004.

[13] J.B. Keller, R.W.Haddad, and S.G.Meier. Scheduler which discovers non-speculative nature of an instruction after issuing and reissues the instruction. *United States Patent 6,564,315*, May 2003.

[14] I. Kim and M. Lipasti. Understanding scheduling replay schemes. In *HPCA*, Feb. 2004.

[15] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed early load retirement. In *HPCA*, Feb. 2005.

[16] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. on Comp*, Sep. 1999.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS*, Oct. 1996.

[18] W. Liu, J. Tuck, L. Ceze, K. Strauss, J. Renau, and J. Torrellas. POSH: A profiler-enhanced TLS compiler that leverages program structure. In *IBM Watson P=AC2 Conference*, Sep. 2005.

[19] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO*, Nov. 2002.

[20] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, Oct. 2002.

[21] A. Merchant, D. Sagger, and D. Boggs. Computer processor with a replay system. *United States Patent 6,163,838*, Dec. 2000.

[22] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA*, 1997.

[23] A. Moshovos, D.Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *ICS*, June 2001.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, Feb. 2003.

[25] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *MICRO*, Dec. 2001.

[26] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS*, June 2005.

[27] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *MICRO*, Dec. 1997.

[28] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO*, Dec. 1997.

[29] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, Aug. 2001.

[30] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *ISCA*, June 1995.

[31] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS*, Oct. 2004.

[32] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, Feb. 2002.
[33] J. G. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA*, Feb. 1998.
[34] M. Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering*, 1981.
[35] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI*, June 2004.
[36] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, Univ. of Virginia, March 2003.
[37] H. Zhou and T. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS*, June 2003.

## Appendix A: Correctness Proofs

We use $l$ and $s$ to represent a load and store instruction, and $A(l)$ and $A(s)$ their addresses. We define $s$ to be an *Inhibiting Store* (Figure 2(a)) if: $s \in S_1$; $s \in S_2$; $s$ stores to different addresses in $S_1$ and $S_2$; and $s$ in $S_2$ stores to an address that is read *or written* in $I_1$. We define $l$ to be a *Dangling Load* (Figure 2(b)) if: $l \in S_1$; $l$ loads from the same address in $I_1$ and $I_2$; and if $s$ is the store that produces the value for $l$ in $S_1$, then in $I_2$, the store has changed its address and, therefore, now $A(s) \neq A(l)$. We define $l$ to be an *Inhibiting Load* (Figure 2(c)) if: $l \in S_1$; $l \in S_2$; $l$ loads from different addresses in $S_1$ and $S_2$; and $l$ in $S_2$ loads from an address that is written to in $I_1$.

Let ReSlice buffer the forward slice $S_1$ of a seed, along with slice live-ins, and keep the instructions in program order. Let ReSlice also log the values overwritten at every first update to an address by instructions in $S_1$. Let $I_1$ and $I_2$ *have the same control flow*.

Consider the case when the PCs of the sequence of instructions in slices $S_1$ and $S_2$ are different. Let us scan $S_1$ and $S_2$ together from top to bottom until we find the first position $i$ where the PCs of the corresponding instructions in $S_1$ and $S_2$ differ.

**Theorem 1** *At least one of the two instructions in position $i$ must be a load. Moreover, this load loads from the same address in both $I_1$ and $I_2$.*

**Proof:** Suppose that none of the two instructions in position $i$ is a load. Then, the reason why they belong to $S_1$ or $S_2$ must be because of a register dependence with an earlier instruction $i_x$ in the slice. Since $i_x$ is there in both slices, the two instructions in position $i$ must be part of both slices by the definition of register dependence, and be the same. This is not true. Therefore, at least one of the two instructions must be a load, and be part of one slice because it reads the value from a previous store from that one slice. Let us call that load $\lambda$, and write it as LD $R_{dst}$, Disp($R_{src}$). Note that $\lambda$ cannot be register-dependent on an earlier instruction in its slice because, otherwise, $\lambda$ would belong to both slices $S_1$ and $S_2$. Consequently, $R_{src}$ is not updated by any earlier instruction in the slice, and it has the same contents in both $I_1$ and $I_2$. Therefore, $\lambda$ loads from the same address in both $I_1$ and $I_2$. The reason why $\lambda$ belongs to only one of the slices is that a preceding store $\sigma$ that belongs to both $S_1$ and $S_2$ writes to different addresses in $S_1$ and $S_2$. Only one of such addresses is equal to the one that $\lambda$ loads from.

**Theorem 2** *If $S_1 \neq S_2$ (i.e., they have a different sequence of PCs), there must be a Dangling load or an Inhibiting store in $S_1$.*

**Proof:** There are two cases, depending on which slice has $\lambda$:

*Case 1: $\lambda \in S_1 \wedge \lambda \notin S_2$ -* In this case, $\sigma$ in $S_1$ writes to address $A(\lambda)$, while in $S_2$, it writes to a different address. Moreover, no instruction in $S_2$ preceding $\lambda$ writes to $A(\lambda)$ — otherwise $\lambda$ would be in $S_2$. Thus, by definition $\lambda$ is a Dangling load.

*Case 2: $\lambda \notin S_1 \wedge \lambda \in S_2$ -* In this case, $\sigma$ in $S_2$ writes to address $A(\lambda)$, while in $S_1$, it writes to a different address. Note that $\sigma$ is in both $S_1$ and $S_2$ because $\lambda$ is the first instruction that is in one slice and not in the other. Since Theorem 1 showed that $A(\lambda)$ is the same in both slices, $\sigma$ is by definition an Inhibiting store.

**Theorem 3** *If there are no Inhibiting stores and no Dangling loads, then $S_1 = S_2$.*

**Proof:** This theorem is the contrapositive of Theorem 2. Recall that we are always considering only the case when the branches take the same directions in $I_1$ and $I_2$.

We therefore found a sufficient condition to ensure $S_1 = S_2$. Now, we need to prove that all the instructions in $S_2$ get the correct operand values. For that, we need to examine all the live-ins to the slice. Register live-ins pose no problem, since they remain the same in the second run. On the other hand, memory live-ins (i.e., the values obtained by loads), may change. We need to prove their correctness.

**Lemma 1** *All stores in $I_2$ that store to different addresses than they did in $I_1$ belong to $S_2$.*

**Proof:** We have set $S_1 = S_2 = S$. In the non-slice instructions (i.e., $I_1 - S$), stores store to the same addresses in $I_1$ and $I_2$. Therefore, any stores in $I_2$ that store to different addresses in $I_1$ and $I_2$ must be in slice $S_2$.

**Theorem 4** *If $S_1 = S_2$ and we disallow Inhibiting stores, Dangling loads, and Inhibiting loads, the loads in $S_2$ get the correct memory values.*

**Proof:** Let $l_1$ be a load in $S_1$ and $l_2$ the corresponding load in $S_2$. There are two cases.

*Case 1: $A(l_1) = A(l_2)$ -* There are two possible subcases here: $l_1$ got the value from a store in $S_1$ or from a store outside $S_1$. In the first subcase, assume that $l_1$ got its value from store $s_x$. In $I_2$, $l_2$ must also get its value from the same store $s_x$ — otherwise $l_1$ is a Dangling load. Thus, $l_2$ gets the correct memory value when the slice re-executes. Consider now the other subcase. In this case, $l_2$ cannot get its value from a store $s_x$ in $S_2$. If it did, $s_x$ would be an Inhibiting store, which is not allowed. Consequently, $l_2$ must get its value from a store outside $S_2$. The saved live-in is correct when the slice is re-executed.

*Case 2: $A(l_1) \neq A(l_2)$ -* If the load is an Inhibiting load (and therefore $l_2$ reads something that $I_1$ wrote), we cannot guarantee correct re-execution. However, if it is not Inhibiting ($A(l_2)$ is not written in $I_1$), then from Lemma 1, $A(l_2)$ is either not written in $I_2$ or it is written in $S_2$. In either subcase, the load gets the correct value when the slice re-executes: either from the memory system because the address was never written in the task, or from the values generated by $S_2$ respectively.

We have just proved a sufficient condition for correct slice re-execution. Now, we consider state merging. We assume ReSlice automatically logs the values overwritten by every first update issued by $S_1$ to an address. The problematic merging case is when a store $s_1$ in $S_1$ and its corresponding store $s_2$ in $S_2$ write to different addresses. If both values are live at the Resolution point, merging requires undoing the update of $s_1$ and applying the update of $s_2$.

**Theorem 5** *If we disallow Inhibiting stores, no other store in $S_1$ beyond $s_1$ has written to $A(s_1)$, and no update to $A(s_1)$ in the current task has yet been undone, then the merging operation is correct.*

**Proof:** Under these conditions, it is correct to apply the update of $s_2$ and to undo the update of $s_1$. First, since the store cannot be Inhibiting, $A(s_2)$ is neither read nor written in $I_1$. As a result, applying the update of $s_2$ (if live) is always correct. Secondly, since $A(s_1)$ was only updated by $s_1$ in $S_1$, and no previous slice in the task has yet induced an undo on the contents of $A(s_1)$, the ReSlice log contains the correct value that needs to be restored to $A(s_1)$. As a result, using the log to undo the update of $s_1$ (if live) is correct.

**Theorem 6** *Given slices $SL_1, \ldots, SL_n$ that are potentially overlapping. Let $SL$ be the union of all the instructions in the slices in program order ($SL = SL_1 \cup \ldots \cup SL_n$). The condition for correct re-execution of $SL$ is the same as that for a single slice.*

**Proof:** $SL$ is different from a single slice because it contains several seeds. Let the seed instructions load from addresses $LX_1, \ldots, LX_n$. Here, we consider only those slices that are not a strict subset of another slice. Note that $LX_1, \ldots, LX_n$ are the same in $I_1$ and in $I_2$. Let us add a piece of hypothetical code $C$ preceding all the slices in the task (Figure 15(a)).

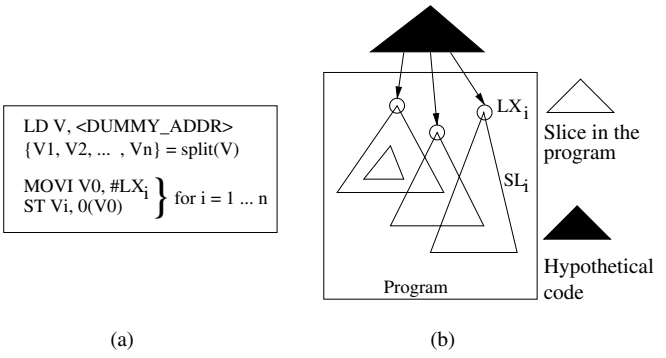


**Figure 15.** Unifying the overlapping slices.

In the code, $V$ and $V0, V1, \ldots, Vn$ are hypothetical registers. Moreover, while the $Vi$ registers contain word-sized quantities, $V$ is wide enough to hold the concatenated value of $n$ $Vi$ registers. Let us assume that the hypothetical address $< DUMMY\_ADDR >$ stores the concatenated value of all $n$ seeds; that we execute a load that loads such value into $V$; and that we execute an instruction that splits the contents of $V$ into registers $V1, \ldots, Vn$. Then, the code stores the contents of $Vi$ into the $LX_1, \ldots, LX_n$ locations. The purpose of $C$ is to unify through dependences the code of all the slices into a single slice $S = SL \cup C$ with a single seed instruction (Figure 15(b)). Now, the condition for correct re-execution of $S$ is like that of any other slice. Finally, since $C$ does not have side effects, we can remove $C$ and the condition for correct slice re-execution applies to $SL$.