Transparent Control Independence (TCI)

Ahmed S. Al-Zawawi Vimal K. Reddy Eric Rotenberg Haitham H. Akkary*

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC

{aalzawa, vkreddy, ericro}@ece.ncsu.edu

*Digital Enterprise Group
Intel Corporation
Hillsboro, OR

haitham.h.akkary@intel.com

ABSTRACT

Superscalar architectures have been proposed that exploit control independence, reducing the performance penalty of branch mispredictions by preserving the work of future misprediction-independent instructions. The essential goal of exploiting control independence is to completely decouple future misprediction-independent instructions from deferred misprediction-dependent instructions. Current implementations fall short of this goal because they explicitly maintain program order among misprediction-independent and misprediction-dependent instructions. Explicit approaches sacrifice design efficiency and ultimately performance.

We observe it is sufficient to emulate program order. Potential misprediction-dependent instructions are singled out *a priori* and their unchanging source values are checkpointed. These instructions and values are set aside as a "recovery program". Checkpointed source values break the data dependencies with co-mingled misprediction-independent instructions – now long since gone from the pipeline – achieving the essential decoupling objective. When the mispredicted branch resolves, recovery is achieved by fetching the self-sufficient, condensed recovery program. Recovery is effectively transparent to the pipeline, in that speculative state is not rolled back and recovery appears as a jump to code. A coarse-grain retirement substrate permits the relaxed order between the decoupled programs. Transparent control independence (TCI) yields a highly streamlined pipeline that quickly recycles resources based on conventional speculation, enabling a large window with small cycle-critical resources, and prevents many mispredictions from disrupting this large window.

TCI achieves speedups as high as 64% (16% average) and 88% (22% average) for 4-issue and 8-issue pipelines, respectively, among 15 SPEC integer benchmarks. Factors that limit the performance of explicitly ordered approaches are quantified.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General.

General Terms Performance, Design.

Keywords Branch prediction, control independence, selective recovery, selective re-execution, checkpoints, speculation.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

1. INTRODUCTION

The performance of contemporary superscalar pipelines is profoundly affected by branch prediction accuracy. Even with modest issue widths of 3 to 6 instructions per cycle, the Intel Pentium-4 and IBM POWER5 processors form speculative instruction windows as deep as 126 and 200 instructions, respectively. A single branch misprediction may flush upwards of 100 in-flight instructions, causing extended retirement stalls as the pipeline gradually refills. Because of the large per-misprediction penalty, branch misprediction rates of 5-10% cause a disproportionate performance loss. Using our detailed cycle-level simulator of a 4-issue superscalar processor with a pipeline depth and memory hierarchy modeled after the Pentium-4, a state-of-the-art perceptron branch predictor [13] often achieves only half the performance of perfect branch prediction for SPEC integer benchmarks.

The performance penalty of mispredictions can be reduced by exploiting control independence [2][5][6][10][19][20][21][22], depicted in Figure 1. The figure shows a branch and instructions after it. Instructions between the branch and its reconvergent point are *control-dependent* (CD) on the branch, in that the outcome of the branch affects which CD instructions are fetched. Instructions after the reconvergent point are *control-independent* (CI) of the branch because they are fetched irrespective of the branch's outcome. Nonetheless, *control-independent data-dependent* (CIDD) instructions are influenced by the branch through data dependences (either register or memory dependences). For example, the consumer of R5 (after the reconvergent point) depends on the first production of R5 (above the branch) if the branch takes the left path or the second production of R5 if the branch takes the right path. Therefore, the consumer of R5 is influenced by the outcome of the branch and is consequently CIDD with respect to the branch (similarly, control-independent loads may be influenced by a prior branch through control-dependent stores). In the example, other instructions after the reconvergent point are not influenced by the branch in any way, referred to as *control-independent data-independent* (CIDI) instructions.

Conventional superscalar processors recover from a mispredicted branch by flushing all instructions after it and restarting from scratch at the branch. In contrast, superscalar processors that exploit control independence conceptually (i) selectively remove only the incorrect CD instructions from the window thus preserving the CI instructions in the window, (ii) insert the correct CD instructions in their place, and, (iii) among CI instructions, only the CIDD instructions are selectively re-executed. Thus, recovery is more selective and this reduces misprediction penalties. Specifically, the work of misprediction-independent instructions (CIDI) is saved.

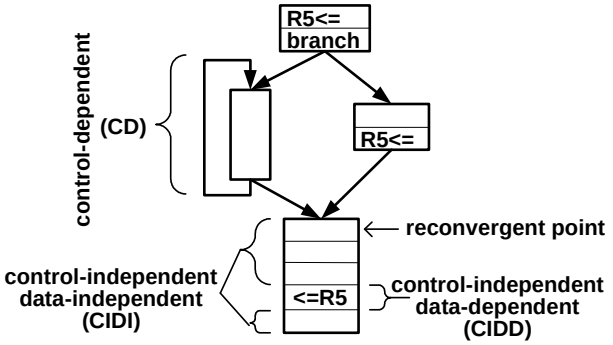

Figure 1. Example control-flow region.

The essential goal of selective recovery is to completely decouple the future misprediction-independent instructions (CIDI) from the deferred misprediction-dependent instructions (CD and CIDD). Existing solutions fall short of this goal because they still explicitly maintain program order among the misprediction-independent and misprediction-dependent instructions. They are order-constrained for two reasons in particular:

- They evolved from reorder buffer (ROB) based designs which require program order for fine-grained retirement. Ultimately this means the late-fetched correct CD instructions need to be reordered with respect to the early-fetched CI instructions.
- When CIDD instructions re-execute with changed values from the repaired CD region, they may also need to re-reference *unchanged* values from CIDI instructions. Ultimately this means dependence order needs to be maintained among co-mingled CIDI instructions and CIDD instructions.

Explicit program-ordered approaches sacrifice design efficiency and performance, because they fail to truly decouple misprediction-independent instructions from misprediction-dependent instructions.

We propose that it is sufficient to mimic the effect of program order between misprediction-independent and misprediction-dependent instructions. The key innovation is to single out CIDD instructions as they are fetched and checkpoint their CIDI-supplied source values, breaking dependences with the CIDI instructions. The CIDD instructions plus checkpointed source values are set aside in a FIFO re-execution buffer (RXB) for possible selective re-execution later. This is the first proposal for truly decoupling CIDI and CIDD instructions. Now, fine-grain retirement via a reorder buffer is the only reason for explicitly maintaining order. To emulate in-order retirement, we propose using a coarse-grain checkpoint-based retirement strategy [3][9][12][18] which relaxes ordering constraints between consecutive checkpoints.

When a branch is mispredicted, its incorrect CD instructions are fetched followed by CI instructions. All instructions – correct and incorrect – complete and speculatively release cycle-critical resources as they drain from the pipeline (issue queue entries, physical registers, etc.). When the mispredicted branch resolves, recovery is achieved by fetching a self-sufficient condensed "recovery program": the correct CD instructions (fetched from the instruction cache), the CIDD instructions (fetched from the RXB), and all input values needed to launch the correct CD and CIDD instructions (the branch's checkpoint and the checkpointed CIDI-supplied source values of CIDD instructions). Recovery is effectively transparent to the pipeline, in that speculative state is not rolled back and recovery appears as a jump to code. Transparent control independence (TCI) yields a highly streamlined pipeline that quickly recycles resources based on conventional speculation, enabling a large window with small cycle-critical resources, and prevents many mispredictions from disrupting this large window.

Figure 2 shows a high-level view of TCI. Dynamic instructions are shown from left to right in the order in which they are fetched (fetch time). Correctly fetched and executed instructions are shown in white and incorrectly fetched or executed instructions are shown in gray. Correctly fetched instructions are labeled with their order in sequential program order (incorrect CD instructions are labeled with x's instead). A branch is mispredicted at the beginning of the fetch timeline. Thus, incorrect CD instructions are fetched first followed by CIDI and CIDD instructions. The first correctly fetched instruction is instruction 4. Some time later, after fetching instruction 14, the misprediction is finally detected. At this point the independent (thanks to input values from the branch's checkpoint and RXB) recovery program is fetched. Notice the relaxed order: the recovery program's instructions 1, 2, 3, 6', 10', and 12' come after the speculative program's instruction 14 in the timeline. The pipeline does not differentiate between the speculative and recovery programs, as shown. The speculative state is not rolled back. Instead, the recovery program transparently repairs the speculative state.

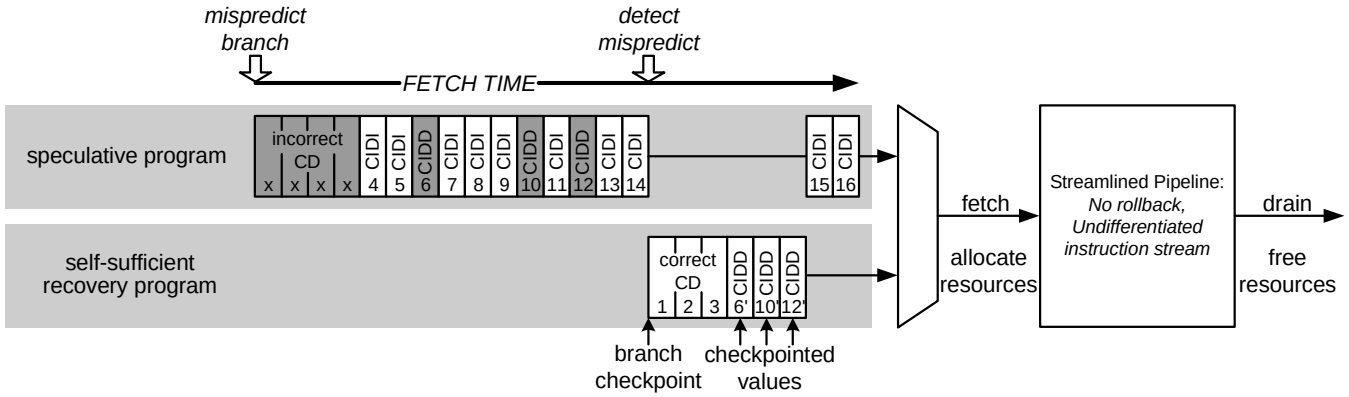

Figure 2. Transparent Control Independence (TCI).

This paper makes the following chief contributions:

- *TCI concept and microarchitecture*. We propose a new approach that fully decouples misprediction-independent instructions from misprediction-dependent instructions, yielding a highly streamlined microarchitecture for exploiting control independence. The key insight is checkpointing CIDI-supplied source values of CIDD instructions. Another important aspect is using a relaxed, coarse-grain retirement substrate.
- *Identifying CIDD instructions*. Novel mechanisms are developed for assembling the CIDD instructions: the control-flow stack (CFS) for detecting arbitrary and nested reconvergent points, predicting the influenced register set (IRS), poisoning registers for identifying CIDD instructions, branch-sets for identifying CIDD loads, etc.
- *RXB reconstruction*. Since CIDD slices of multiple branches are co-mingled within the RXB, servicing a branch misprediction may require repairing CIDD slices of other branches and selectively removing CIDD instructions of the resolved branch. A simple unified solution – identify CIDD instructions in the recovery program itself, as was done the first time for the speculative program – enables arbitrary adjustments to the RXB while preserving its simple FIFO policy.
- *Renaming partial programs*: We propose a novel technique for renaming the recovery program despite its CIDI gaps.
- *Comparing resource and bandwidth overheads for repairing CIDD instructions*. We analyze factors that reduce the performance of explicit program-ordered approaches and measure the impact of these factors. We show TCI uses fewer resources and less bandwidth for repairing CIDD instructions.

Section 2 provides a high-level overview of the proposed TCI microarchitecture. Section 3 discusses closely related work and identifies factors that reduce performance of previous approaches. Section 4 presents the TCI microarchitecture in detail. Section 5 covers the simulator and methodology. Results are presented in Section 6. Additional related work is discussed in Section 7. Finally, the paper is summarized in Section 8.

2. HIGH-LEVEL OVERVIEW OF TCI MICROARCHITECTURE

Figure 3 shows our transparent control independence (TCI) architecture. The shaded region highlights a resource-streamlined pipeline that aggressively releases resources based on conventional speculation. Correct and incorrect instructions alike flow through the pipeline as fast as they would with conventional speculation, aggressively freeing issue queue entries and physical registers [3][9][18][23] on the assumption that branch predictions are correct. Instructions drain from the pipeline as soon as they complete – there is no reorder buffer (ROB) and precise exceptions are achieved via checkpoints [3][9][12][18][23].

When a branch is encountered in the fetch unit, its predicted CD instructions are fetched from the instruction cache (I-cache), highlighted in Figure 3 with Step-1. These are soon followed by the branch's CI instructions, corresponding to Step-2 in the figure. Both the predicted CD and CI instructions are renamed with the speculative rename map and sent down the pipeline. The branch's CIDD instructions are identified in the dispatch stage and duplicates of these instructions are set aside in a FIFO buffer, the Selective Re-execution Buffer (RXB), as shown. When these instructions issue and read their source operands from the physical register file, copies of the source values are also set aside with the corresponding instructions in the RXB. If, when the branch executes, a misprediction is detected, control is temporarily transferred to the correct target of the branch. Corresponding to Step-3 in the figure, the branch's correct CD instructions are fetched from the I-cache and renamed using the repair rename map, which is initialized from a corresponding branch checkpoint thus ensuring the correct CD instructions have values in the physical register file to begin execution with. When the reconvergent point is encountered again, control is transferred to the branch's CIDD instructions in the RXB, corresponding to Step-4 in the figure. These are also renamed using the repair rename map to establish linkages with producer instructions prior to the reconvergent point. A key point is that the branch's CIDD instructions residing in the RXB do not tie up cycle-critical resources (issue queue entries and physical registers) and are allocated resources only when control is transferred to the RXB, just like instructions that are dispatched from the I-cache. Another key point is that CIDDs' source operands that depend on CIDI instructions cannot be resolved by the repair rename map because the CIDIs' values were most likely freed from the physical register file already and those that have not been freed are inaccessible by the repair rename map anyway; fortunately the source values were individually checkpointed previously and are in the RXB with the CIDD instructions.

Loads issue aggressively and are speculative with or without branch mispredictions [7]. Store-load dependences are also resolved correctly, as we explain in Section 4.5.

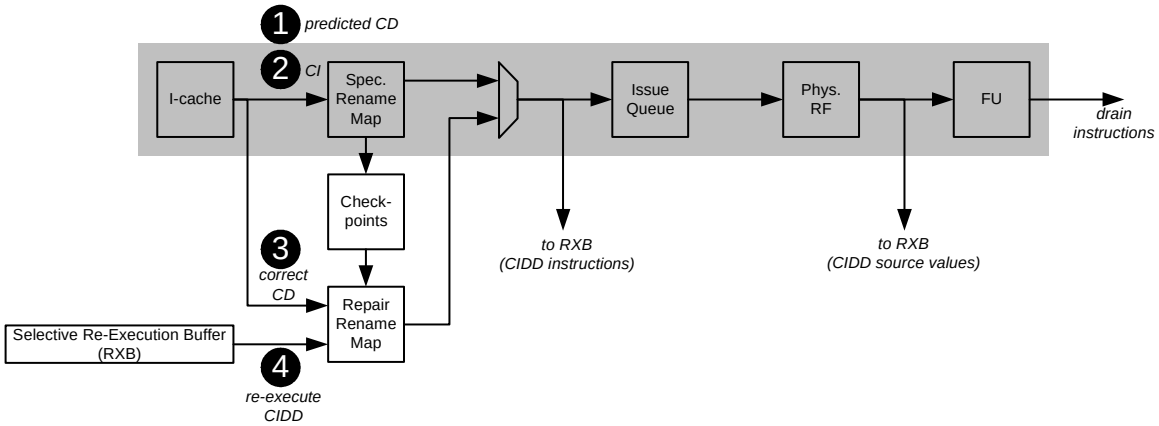

Figure 3. Transparent control independence (TCI) architecture.

3. UNDERSTANDING RELATED WORK

3.1 Qualitative Comparisons

There have been a number of proposals for exploiting control independence in superscalar [5][6][10][19][20][21] and speculatively multithreaded processors [2][22]. Exploiting control independence involves three key implementation issues, and different architectures address these issues in different ways and with various compromises.

- *Insertion/removal of CD instructions*. The key challenge with regard to CD instructions is arbitrary insertion/removal of CD instructions in the middle of the window. The FIFO reorder buffer (ROB) of conventional superscalar processors is structurally incompatible with removing incorrect CD instructions and inserting correct CD instructions (except the special case in which the two branch paths are equal in length), because the ROB can no longer be managed as a simple circular FIFO. Among ROB-based implementations, the ROB can be managed as a linked-list at the level of individual instructions [19], segments [19], or processing elements [20][22], but at the cost of significant complexity. Instruction reuse (or squash reuse) [21] and a related dual ROB implementation [6] avoid this complexity by re-fetching/re-dispatching all instructions after the mispredicted branch, emulating the simple control-flow repair model of full flushing, but this yields a significantly weakened implementation that only saves on re-execution bandwidth of CIDI instructions. Two more recent ROB-based implementations (Skipper and Exact Convergence) either pre-pad the ROB to make room for expansion [5], potentially underutilizing the ROB or overdesigning the ROB, or exploit control independence only for the special case in which the correct branch target is the reconvergent point itself (thus not requiring insertion of any CD instructions) [10]. The expandable/contractable window of speculative multithreading implementations [2][22] are structurally compatible with arbitrary insertion/removal of CD instructions, but this is achieved as a byproduct of departing from the familiar superscalar paradigm, moreover, mispredictions cause full flushing within threads thus not exploiting arbitrary reconvergent points.
- *Selectively re-renaming CIDD instructions*. After repairing the CD region of the mispredicted branch, CIDD instructions with stale source names have to be identified and re-renamed to establish linkages with their correct producers (e.g., the R5 consumer in Figure 1). Some implementations resequence through all CI instructions to locate CIDD instructions that require re-renaming [2][19][20], expending time on scanning through all instructions whether or not their source names require fixing. As mentioned, instruction reuse [21] and the dual ROB implementation [6] flush and re-fetch/re-dispatch all instructions after the mispredicted branch, thus needlessly re-renaming all instructions. Skipper [5] and Exact Convergence [10] inject proxy move instructions at the reconvergent point for logical registers influenced by the branch (e.g., R5 in Figure 1), insulating CIDD instructions from source name changes. Re-renaming is localized to proxy instructions, at the cost of increased physical register pressure, issue queue pressure, and ROB pressure for the proxy instructions of each protected branch.
- *Selectively re-executing CIDD instructions*. After repairing the CD region and fixing stale source names of CIDD instructions, all CIDD instructions must be selectively re-executed. The drawback of the above superscalar based implementations is that selective re-execution of CIDD instructions (or deferred execution of CIDD instructions in the case of Skipper [5]) increases pressure on cycle-critical resources. Issue queue entries and source and destination physical registers cannot be released for completed CIDD instructions because they may need to selectively re-execute. Release is delayed until corresponding branches resolve. This is inefficient compared to conventional speculation, which releases resources aggressively (issue queue slots and even physical registers in some designs [3][9][18][23]) since misprediction recovery involves rolling back to the mispredicted branch anyway.

TCI compares favorably with previous control independence architectures. The choice of a checkpoint-based, ROB-free superscalar substrate with counter-based physical register allocation/deallocation is meaningful, as this substrate is structurally compatible with arbitrary insertion/removal of instructions in an order-agnostic way as long as producer-consumer dependences are respected. Thus, TCI avoids the complexity of linked-list ROB management [19][20], the performance degradation of full flushing [21], and the underutilization of ROB padding [5]. And unlike speculative multithreading [2][22], TCI maintains a familiar (superscalar) execution model and exploits arbitrary reconvergent points. With TCI, CIDD instructions are selectively re-renamed using the compressed (CIDD instructions only) RXB, unlike implementations that re-inspect all CI instructions [2][19][20]. Moreover, selective re-execution of CIDDs does not come at the price of tying up precious cycle-critical resources such as issue queue slots and physical registers. In contrast, previous superscalar implementations increase resource pressure even in the case of correct speculation, potentially impeding performance in the common case (either by degrading IPC for an undersized resource or increasing cycle time for an oversized resource).

3.2 Resource and Bandwidth Overheads

Different methods for re-renaming and re-executing CIDD instructions result in different resource and bandwidth overheads, influencing performance. In this section, we compare the resource and bandwidth overheads for repairing CIDD instructions, for different generalized models on a common substrate. The common substrate is a 4-issue ROB-free checkpointed processor with aggressive register reclamation (described in Section 5). Due to a common high-performance flexible-window substrate, we do not capture the performance differences among different CD insertion/removal methods (for example, the penalty of reservation based approaches [5] is not captured).

Three generalized re-renaming models are considered. *Proxy* uses proxy move instructions to insulate CI instructions from source name changes, and only the proxies are re-renamed. *Seq CI* sequences through all CI instructions to update stale source names. *Seq CIDD* re-renames only the CIDD instructions, like in TCI. *Seq CIDD* requires TCI's mechanisms (specifically, poisoning) for distinguishing a CIDD instruction's source operands as coming from either CIDI or CD/CIDD instructions – only sources coming from CD or other CIDD instructions should be re-renamed.

Two models are considered for selective re-execution of CIDD instructions. *Hold IQ* is conservative, as it holds all instructions in the issue queue. *Drain IQ* is aggressive, as it drains instructions from the issue queue when they issue. For the *Drain IQ* model,

selective re-execution is achieved differently for *Proxy*, *Seq CI*, and *Seq CIDD*. *Proxy* holds proxy and CIDD instructions in the issue queue, signified as *Drain IQ (partial)*. *Seq CI* uses a re-execution buffer (RXB) containing all CI instructions. *Seq CIDD* uses a compressed RXB containing only CIDD instructions.

Table 1 compares the resource and bandwidth requirements for repairing CIDD instructions, for *Base* (conventional recovery), *Proxy*, *Seq CI*, and *Seq CIDD*, on both *Hold IQ* and *Drain IQ* re-execution substrates (*Base* always drains). In addition, the last column in Table 1 cites specific implementations of these approaches from the literature. Resources are further divided into registers, issue queue entries, and RXB entries. Bandwidth is further divided into re-renaming and re-execution bandwidth. Note that TCI (*Drain IQ/Seq CIDD*) is qualitatively the best or tied for best in every category. TCI may re-rename fewer or more instructions than *Proxy*, depending on the number of proxy instructions and CIDD instructions.

Table 1. Resource and bandwidth usage for repairing CIDD instructions.

	Model	Hold resources until branch resolves			CI resequencing bandwidth		Related work
		Registers	Issue Queue	RXB	Re-renaming	Re-execution	
	Base	none	none	none	CIDD + CIDI	CIDD + CIDI	
Hold IQ	Proxy	all	all	none	proxy	CIDD + proxy	[5][a],[10]
Hold IQ	Seq CI	all	all	none	CIDD + CIDI	CIDD	[19],[20]
Hold IQ	Seq CIDD	all	all	none	CIDD	CIDD	
Drain IQ	Proxy	some CIDI + CIDD + proxy	CIDD + proxy	none	proxy	CIDD + proxy	
Drain IQ	Seq CI	none	none	all	CIDD + CIDI	CIDD	[2],[6]
Drain IQ	Seq CIDD	none	none	CIDD	CIDD	CIDD	TCI

[a]Cited for the use of proxy inst., and not skipper style control independence.

Figure 4 shows the harmonic mean of IPCs for 14 of the SPEC integer benchmarks listed in Table 3, for the seven models. Benchmark mcf has been excluded from the harmonic mean because its extremely low IPC drowns out trends. The issue queue size is varied to understand resource pressure. The resource inefficiency of the *Hold IQ* re-execution substrate is a major bottleneck with small issue queues. In fact, *Base* outperforms all *Hold IQ* models, for issue queues with fewer than 256 entries. This is because the issue queue limits the overall window size when all instructions are held in the issue queue.

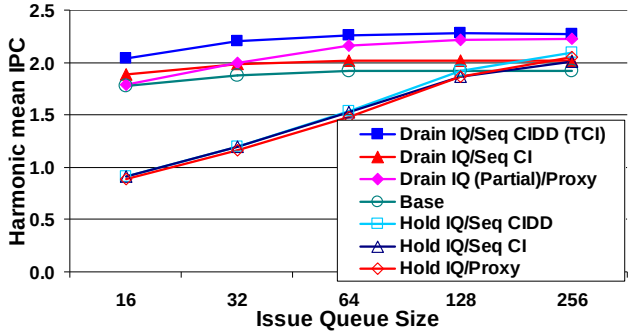

Figure 4. Performance of different CIDD repair models.

Ideally, all instructions should free all cycle-critical resources speculatively, allowing for a bigger window, and CIDD instructions should only be re-allocated resources when selective re-execution is required after a branch misprediction. *Drain IQ* strives for this goal. However, *Proxy* falls short of this ideal scenario because proxy and CIDD instructions remain in the issue queue for possible selective re-execution. The residual issue queue pressure is evident in Figure 4: *Proxy* is unique in its sensitivity to issue queue size compared to other models with *Drain IQ*. In fact, for a 16-entry issue queue, *Proxy* has no performance advantage over conventional recovery (*Base*) let alone the other selective recovery approaches. On the other hand, a 64-entry issue queue enables *Proxy* to overtake *Seq CI*. Overall, *Seq CIDD* (TCI) performs the best due to its combined bandwidth and resource efficiency. Figure 5 shows similar trends for the benchmark *gzip*, individually.

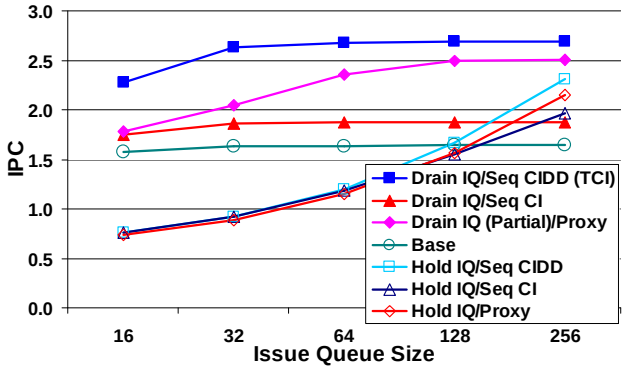

Figure 5. Performance of different CIDD repair models (*gzip*).

Figure 6 shows the performance sensitivity of *Drain IQ/Seq CI* and *Drain IQ/Seq CIDD* to the RXB size. *Seq CI* is very sensitive to RXB size: all instructions are inserted into the RXB, therefore, the RXB limits the overall window size (like a ROB). In contrast, *Seq CIDD* is much less sensitive to the RXB size: only CIDD instructions are inserted into the RXB, therefore, the RXB does not limit the overall window size.

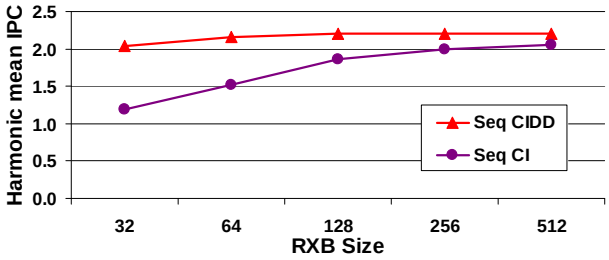

Figure 6. Sensitivity to RXB size.

Figure 7 focuses on the re-rename bandwidth of *Drain IQ/Seq CIDD*. The RXB contains CIDD instructions for multiple branches in program order. Thus, other branches' CIDD instructions may increase the time to re-sequence, compared to re-sequencing only the CIDD instructions of the mispredicted branch. The latter model is labeled *Drain IQ/Seq Br CIDD* in the graph. The graph shows little performance difference between the two models. Moreover, there is little performance difference even with *Ideal*, which re-sequences in 0 cycles.

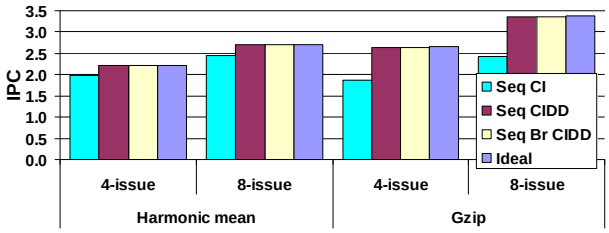

Figure 7. *Seq CIDD* bandwidth.

4. TCI MICROARCHITECTURE

4.1 Identifying and Inserting CIDD Instructions into RXB

This section explains how CIDD instructions are identified and inserted into the RXB by the speculative rename map, in a process called poisoning.

4.1.1 Reconvergent Point and Influenced Register Set (IRS) Predictor

The compiler or a hardware predictor can be used to identify branches' reconvergent points. In this paper, we use the dynamic reconvergence predictor proposed by Collins et al. [8]. We augment the predictor to provide additional information for each branch. First, the predictor keeps track of the maximum path length through a branch's control-dependent (CD) region, among paths that were traversed. This information is useful for guiding when to apply control independence. We select a maximum CD path length above which it is not worthwhile to exploit control independence due to the sheer number of incorrect CD instructions. Second, we add a learning mechanism to collect a branch's influenced register set (IRS). As the predictor monitors retired instructions for reconvergence, it keeps track of logical registers written to after the branch and before reconvergence is detected. The use of confidence ensures repetition, so that enough different paths are traversed through a branch's CD region to yield a representative IRS.

4.1.2 Control-Flow Stack (CFS)

When a branch is dispatched, we must detect its reconvergent point among later instructions as they are dispatched. The reconvergent point marks the beginning of CI instructions, so it is at this point that we need to mark, or "poison", influenced registers (indicated by the branch's IRS) in the speculative rename map.

A novel hardware mechanism called the *control-flow stack* (CFS) detects reconvergent points in the dispatch stage. When a checkpointed branch is dispatched, its reconvergent PC and checkpoint tag (to identify the branch) are pushed onto the CFS top-of-stack.

The next reconvergent point in the dynamic instruction stream is detected by comparing the PCs of newly dispatched instructions to the reconvergent PC at the top-of-stack. If there is a match, then the branch corresponding to the current top-of-stack has reconverged. We know which branch this is via the checkpoint tag at the current top-of-stack. Since the beginning of control-independent instructions has been reached, the branch's IRS is used to poison influenced registers at this time. Poisoning registers is explained in the next section. Finally, the CFS top-of-stack is popped (removed), re-exposing the next reconvergent point to search for.

The CFS can detect cases in which multiple branches have the same dynamic reconvergent point. If the reconvergent PC of a newly dispatched branch matches the reconvergent PC at the CFS top-of-stack, then the new branch and the branch corresponding to the CFS top-of-stack have the same dynamic reconvergent point.[1] In this case, the new branch does not push a new entry onto the CFS, implicitly "merging" with the CFS top-of-stack.

There are three cases in which a branch is forced to inherit the reconvergent point of its encompassing branch region: if the branch does not have a predicted reconvergent PC, if there are no free checkpoints, or if the branch is confidently predicted. The branch corresponding to the CFS top-of-stack is the closest encompassing branch. Thus, the new branch inherits the reconvergent point of its encompassing branch simply by not pushing onto the CFS and instead merging as explained above.

The CFS only needs as many entries as there are checkpoints (16 entries in this paper). CFS entries of branches that resolve before they reconverge are collapsed away (since they are not popped).

4.1.3 Poison Vectors

After a branch's CD region is fetched and its reconvergent point is detected by the CFS, we are ready to use the branch's IRS to poison registers and thereby identify CIDD instructions. Each influenced register specified in the IRS must be poisoned.

We provide a 16-bit *poison vector* per entry in the speculative rename map. A logical register is poisoned if one or more bits are set in its poison vector. Moreover, which bits are set indicates which branches a logical register is influenced by. A checkpointed branch is identified by its checkpoint tag. A non-checkpointed branch is identified by the checkpoint tag of the branch from which it inherited its reconvergent point (discussed in Section 4.1.2). Since we use 16 checkpoints in this paper, a poison vector has 16 bits.

When a branch reconverges, the poison vector of each influenced register, specified by the IRS, is updated in the speculative rename map. In particular, the poison bit corresponding to the branch's checkpoint tag is set.

CIDD instructions can now be identified during renaming. When an instruction's logical source registers are renamed, the corresponding poison vectors are ORed together. If the ORed vector has any bits set, the instruction is CIDD with respect to one or more branches. Also, the ORed vector overwrites the poison vector of the logical destination register, in the speculative rename map. This propagates poison status for identifying indirect CIDD instructions.

When a checkpoint is freed, the corresponding poison bit is cleared in all poison vectors. Given that all branches associated with the checkpoint are now resolved, no future instructions should be considered CIDD with respect to these branches.

Only the speculative rename map, repair rename map, and checkpoints have poison vectors. Poison vectors in the repair rename map and checkpoints are discussed in Section 4.2.

4.1.4 Inserting CIDD instructions into the RXB

CIDD instructions are inserted into the RXB in program order at the dispatch stage. When a CIDD instruction issues and reads its source values from the physical register file, it replaces its source mappings in its entry in the RXB with the source values (a bit is

[1] They do not have the same dynamic reconvergent point if the call depths of the two branches are different, e.g., due to recursion. We make the test definitive by tracking call depth in the dispatch stage and including call depths in CFS entries. If the new branch's reconvergent PC and call depth match the CFS top-of-stack, then the branches have the same dynamic reconvergent point.

set within its entry in the RXB to signify that source values have replaced source mappings).

4.2 Misprediction Recovery

When a misprediction is detected, the fetch unit temporarily redirects fetching to the correct target of the mispredicted branch. Correct CD instructions are fetched from the instruction cache and renamed using the repair rename map initialized from a checkpoint at the branch. The repair rename map, like the speculative rename map, has its own CFS to detect the reconvergent point again that marks the end of the correct CD region (its CFS also identifies new nested branch regions). At this point, the branch's CIDD instructions are fetched from the RXB, re-renamed using the repair rename map, and re-injected into the pipeline. Finally, the repair rename map is used to fix up the speculative rename map and checkpoints.

4.2.1 Reconstructing the RXB

The RXB contains CIDD instructions with respect to all unresolved branches. This means the RXB must be reconstructed when recovering from a branch misprediction, as follows.

- *Case A*. There may be instructions from the branch's incorrect CD path in the RXB, that were thought to be CIDD with respect to other prior branches. These have to be removed from the middle of the RXB.
- *Case B*. New instructions from the correct CD path may be CIDD with respect to other prior branches. These have to be inserted into the middle of the RXB.
- *Case C*. Instructions in the RXB that are only CIDD with respect to the branch being serviced should be selectively removed from the RXB, since they will not be revisited again. Instructions in the RXB that are CIDD with respect to other branches (whether or not they are also CIDD with respect to the current branch) must remain in the RXB. Note that these two types of instructions are co-mingled in the RXB.

There is only one solution and it is simple, because it is analogous to initial CIDD identification and insertion into the RXB described in the previous section. The recovery program for the current branch is comprised of the correct CD instructions from the instruction cache and all instructions in the RXB logically after the resolved branch's reconvergent point. (The recovery program is not as efficient as it could be because it has CIDD instructions of other branches that are not also CIDD with respect to the current branch.) *Poisoning of the recovery program via the repair rename map can once again construct the RXB contents*. As a preliminary step, the RXB tail pointer is moved back to the branch (even though the branch may not be in the RXB physically, the branch knows its logical position in the RXB). This naturally takes care of any incorrect CD instructions in the RXB since they will get overwritten by the adjusted tail pointer (*case A*). Then, poisoning the recovery program using the repair rename map will naturally (1) insert new CIDD instructions with respect to prior branches from among the correct CD instructions (*case B*), and (2) insert old CIDD instructions only if they are CIDD with respect to remaining unresolved branches (*case C*).

Since CIDD instructions are concurrently fetched from the RXB (while fetching the recovery program) and inserted into the RXB (while constructing a new recovery program), we need a mechanism to prevent overwriting CIDD instructions in the RXB before they are fetched. We set up a pre-read pointer into the RXB, that points to the first CI instruction with respect to the resolved branch. Since we moved the tail pointer to the branch, the pre-read pointer is logically after the tail pointer. The pre-read pointer is where fetching of CIDD instructions is supposed to begin. If we wait until the correct CD path is fetched, some of the CIDD instructions beginning at the pre-read pointer could get clobbered by the advancing tail pointer. Therefore, using the pre-read pointer, we begin pre-reading CIDD instructions from the RXB right away so that they cannot get clobbered. They are transferred to a Temp Buffer, from which fetching of CIDD instructions will eventually begin (after the correct CD instructions are fetched from the instruction cache).

Figure 8 shows a detailed RXB reconstruction example with two branches, B1 and B2, and respective reconvergent points R1 and R2. Logical positions of B1/R1 and B2/R2 with respect to RXB instructions are indicated with wide black arrows. RXB instructions are labeled with their position # in the dynamic instruction stream. Noncontiguous numbers merely highlight that CIDD instructions are noncontiguous. Instruction x is not numbered because it is an incorrect CD instruction of mispredicted branch B2. Furthermore, instructions are marked with either a rectangle or oval: rectangles are CIDD with respect to B1, ovals are CIDD with respect to B2, rectangle+oval are CIDD with respect to both B1 and B2. Below we step through each of the frames (a)-(g).

(a) Frame (a) shows the initial state of the RXB. B1 has no CD instructions in the RXB since there are no branches prior to it. B1 has four CIDD instructions after R1: 9, x, 16, 20. B2 has one (incorrect) CD instruction, x. Instruction x is not in the RXB because of B2 but rather because it is CIDD with respect to B1. B2 has two CIDD instructions after R2: 18, 20.

(b) In frame (b), mispredicted branch B2 is detected, causing the RXB tail to rollback to just after B2 (instruction x), and the RXB pre-read pointer to initiate at the first CIDD instruction past B2's reconvergent point R2 (instruction 16).

(c) In frame (c), new instructions 11 and 12 – correct CD instructions with respect to B2 – are fetched from the instruction cache (I$) and dispatched for the first time to the issue queue (To IQ). Moreover, instruction 12 is inserted into the RXB because it is CIDD with respect to B1. Instruction 12 is inserted at the RXB tail (which then advances) thereby replacing instruction x. Note also that pre-reading has begun: instruction 16 is transferred to the Temp Buffer so that it is not clobbered by B2's incoming correct CD instructions.

(d) Similarly, in frame (d), we continue fetching and dispatching the remainder of B2's correct CD instructions (13 and 14). Both 13 and 14 are dispatched to the issue queue but only 14 is inserted into the RXB, since 14 is CIDD with respect to B1. Meanwhile we continue pre-reading instructions (18) into the Temp Buffer.

(e) In frame (e), no more instructions are fetched from the I$ because B2's reconvergent point R2 has been reached from the correct CD path. We begin reinjecting and/or recirculating CIDD instructions from the Temp Buffer. Frame (e) shows instruction 16 leaving the Temp Buffer only to be recirculated back to the RXB (CIDD on unresolved B1). It is not reinjected into the issue queue because it is not CIDD on B2 (the mispredicted branch).

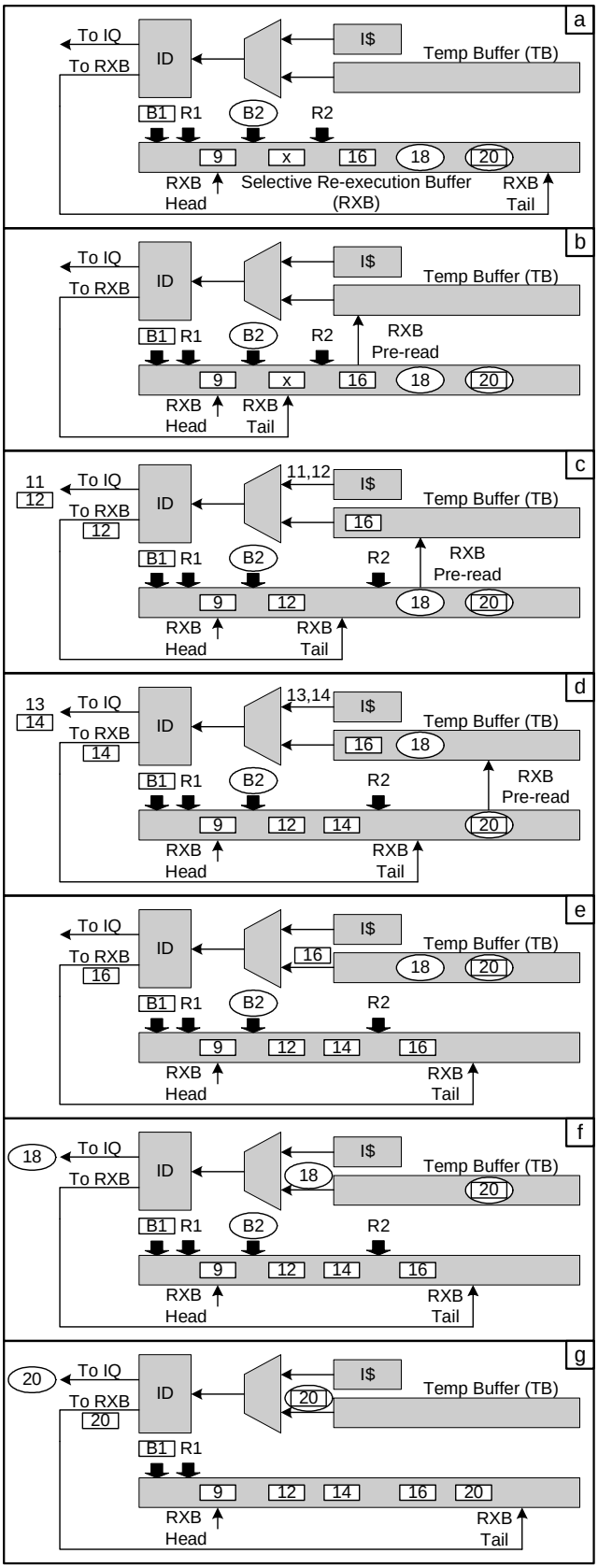

Figure 8 . RXB reconstruction example.

(f) However, instruction 18 in frame (f) is reinjected into the issue queue (CIDD on resolved B2) and not recirculated back to the RXB since it is not CIDD on B1.

(g) Finally, in frame (g), instruction 20 is both reinjected into the issue queue and recirculated to the RXB from the Temp Buffer, because it is CIDD on both B1 and B2. Since the Temp Buffer is empty, we are done servicing B2.

4.2.2 Poisoning via Repair Rename Map

The repair rename map's poison vectors are initialized from the mispredicted branch's checkpoint. While fetching the correct CD instructions from the instruction cache and CIDD instructions from the RXB, the poison vectors are managed the same way as described for the speculative rename map (Section 4.1.3), except for a subtle modification. The poison vectors of logical registers that would have been updated by CIDI instructions, simply are not, because they are not observed by the repair rename map. These logical registers represent "holes" in the repair rename map and their poison vectors cannot be referenced by an instruction's source registers. Fortunately, we know two things: (1) the poison vector generated by a CIDI instruction is all 0's because it is not CIDD with respect to any unresolved branch, and (2) a CIDI instruction *is* observed *once* (and only once) in either the speculative rename map (CIDI immediately) or repair rename map (CIDI eventually). So, when a source register of a CIDD instruction references a CIDI production for the first and only time (signaled by an all-0 poison vector in the rename map), a sticky bit ("CIDI_supplied") associated with the source register in the RXB is set to indicate that the source register's poison vector is by definition all 0s. Once CIDI_supplied=1, in future passes, an all-0 poison vector is used instead of referencing an absent poison vector in the repair rename map.

The outcome of poisoning by the repair rename map indicates what to do with each instruction. For correct CD instructions from the instruction cache, the choices are: insert or do not insert into the RXB. For CIDD instructions from the RXB, the choices are: reinject only, insert (i.e., recirculate) only, reinject and insert, or discard. An instruction is inserted into the RXB if poisoning indicates that it is CIDD with respect to any unresolved branches. An instruction is reinjected into the pipeline if poisoning indicates that it is CIDD with respect to the mispredicted branch being serviced.

4.2.3 Reinjecting CIDD Instructions

Only CIDD instructions from the RXB that are CIDD with respect to the branch being serviced are reinjected into the pipeline. These are re-renamed to bind physical registers and thereby facilitate re-execution.

CIDI instructions are absent from re-renaming, just as they were absent from poisoning. Now, additionally, CIDD instructions from the RXB that are not reinjected are also absent from re-renaming. The latter instructions are CIDD with respect to other branches but not with respect to the branch being serviced. They are tantamount to CIDI instructions with respect to the branch being serviced ("implicit" CIDI instructions), and need not be re-executed. As such, they are not re-allocated storage and do not participate in re-renaming.

When re-renaming a source register of a reinjected CIDD instruction, we need to determine if it depends on an explicit or implicit CIDI instruction (the two cases outlined above) versus a

CD or reinjected CIDD instruction. If it depends on an explicit or implicit CIDI instruction, then the source value (if available) or source mapping from the RXB is used in lieu of re-renaming, because the repair rename map has a stale name. Otherwise, the correct mapping is obtained from the repair rename map.

The source register depends on an explicit CIDI instruction if its CIDI_supplied bit in the RXB is set. The source register depends on an implicit CIDI instruction if its poison vector in the repair rename map does not have the current branch's bit set. Note, it is safe to reference the poison vector because all CIDD instructions in the RXB undergo poisoning. It is only unsafe to reference the poison vector in the case of explicit CIDI instructions, which is why the CIDI_supplied bit is checked first.

The reinjected CIDD instruction is allocated a new physical destination register and updates the repair rename map accordingly.

If a CIDD instruction is both inserted (i.e., recirculated) into the RXB and reinjected into the pipeline, its source registers may be updated in the RXB, analogous to what was described in Section 4.1.4. Specifically, when it redispatches, a re-renamed source register updates the corresponding source mapping in the RXB. When it reissues, it reads values from the physical register file for source registers that did not reuse values from the RXB. These new values replace corresponding source mappings in the RXB.

4.2.4 Merging Repair/Speculative Rename Maps

When RXB reconstruction is completed, the repair rename map is logically at the same point in the dynamic instruction stream as the speculative rename map. Some mappings in the speculative rename map have to be repaired using the repair rename map. Specifically, any speculative mapping whose poison vector has the branch's bit set may be incorrect (it may have changed due to the control-flow adjustment). We simply copy the corresponding mapping from the repair rename map to the speculative rename map. All poison vectors in the repair rename map are copied.

Checkpoint maps are repaired the same way, as the repair rename map resequences through the RXB and reaches checkpoints along the way.

4.3 Conventional Recovery

If a branch misprediction is detected before the fetch unit has reached the branch's reconvergent point, then there is no need to transfer control to the repair rename map and RXB, as there are no CI instructions with respect to the branch yet. This scenario is easily detected by checking if the mispredicted branch has not yet popped the CFS (not reconverged). In this case, the speculative rename map is simply restored to the checkpoint corresponding to the mispredicted branch as in conventional recovery.

4.4 Servicing Multiple Branch Mispredictions

TCI supports servicing new mispredictions concurrently with the one being serviced, if the new mispredictions are logically after the repair rename map. A new misprediction will begin servicing when the repair rename map logically reaches it, in a natural continuation of RXB reconstruction. After fetching the correct CD instructions of the new misprediction, CIDD instructions of both the initial and new mispredictions are reinjected concurrently. If a new misprediction is logically before the repair rename map, we wait until the initial RXB reconstruction completes before servicing the new misprediction; however, an earlier misprediction that has not reconverged is serviced immediately via conventional recovery.

4.5 Store/Load Queues and CIDD Loads

Loads issue speculatively and dependence violations are detected by comparing completed stores against the load queue. Also, stores commit in order. A key issue is that loads and stores may need to be inserted and removed from the middle of the load/store queues as mispredicted branches alternate CD paths, so that loads and stores remain ordered. Rather than do this literally, we apply the same technique that we used to adjust the RXB. Tail pointers are moved back, and control independent loads/stores are pre-read and recirculated into shifted positions. Note that this is equivalent to what a conventional superscalar does, only it refetches the control independent loads and stores from the instruction cache instead of recirculating them from the load/store queues themselves.

Exploiting control independence increases load violations, due to mispredicted branches that fetch incorrect CD stores (false memory dependences) or delay correct CD stores (true memory dependences). We define CIDD loads – these are CI loads influenced by stores within prior branches' CD regions. CIDD loads are predicted at dispatch by accessing the store-set predictor (indexed by load PC). Predicted CIDD loads are then copied into the RXB like normal CIDD instructions. Accordingly, a CIDD load and its poisoned CIDD descendants will be re-injected from the RXB when a branch misprediction is detected (if the CIDD load depends on it), eliminating an exception if the branch misprediction removes or inserts an influencing store.

A conventional store-set predictor works for stores currently in the window, but stores fetched late have no way to convey their influence to the CIDD loads. To improve the accuracy of the store-set predictor, we augment it with branch proxies for potential stores (called branch-sets). Hence, the modified store-set predictor will predict if the window contains any potentially violating stores, both current stores and potential late stores.

4.6 Reconvergence Predictor Misinformation

The reconvergence predictor may provide a flawed reconvergent PC, incomplete IRS, or misleading CD path length for a branch. Inaccuracies are detected when fetching CD instructions of the branch. If inaccurate information is detected during the first pass through the CD region, it can be amended. If detected during the second pass (repairing mispredicted branch), it is handled by forgoing control independence. We call the latter "downgrades" (downgrade to conventional recovery). The frequency of downgrades is reported in results.

5. SIMULATION METHODOLOGY

We implemented the TCI microarchitecture on a detailed cycle-level simulator. Table 2 shows microarchitecture parameters. A functional simulator is run concurrently with and independently of the timing simulator, to confirm correctness. For comparison, the baseline is TCI with the dynamic reconvergence predictor disabled, which ensures conventional (full) recovery for all branch mispredictions. Thus, the baseline is a checkpoint-based superscalar processor with aggressive register reclamation [3].

We use 11 SPEC2K integer benchmarks and 4 SPEC95 integer benchmarks compiled with the Simplescalar gcc compiler [4] for the PISA ISA with -O3 optimization. Reference inputs are used.

Table 2. Microarchitecture.

L1 I & D caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle
L2 unified cache	2MB, 8-way, 64B line, LRU, L2hit = 10 cycles, L2miss = 200 cycles
Branch predictor	perceptron (128KB)
Memory dependence prediction	store/branch sets
Physical registers	256
Checkpoints	16
CFS	16 entries
Issue width	4 or 8
# pipeline stages	20
Issue queue	32 or 64
Load/store queue (LSQ)	512
Re-execution buffer (RXB)	256
Temp buffer (TB)	128

Table 3. Benchmarks.

Benchmarks	SimPoint 3.2 (100m)	L2 load miss/1k inst		Branch misp. /1k inst		Base IPC				Perfect %IPC improvement			
						4-issue		8-issue		4-issue		8-issue	
		Base	TCI	Base	TCI	IQ32	IQ64	IQ32	IQ64	IQ32	IQ64	IQ32	IQ64
bzip2-program-ref	406	2.73	2.74	12.74	12.17	1.57	1.60	1.83	1.91	115%	124%	168%	208%
compress95-bigtest-ref	374	0.31	0.31	10.01	9.92	1.60	1.62	1.80	1.89	98%	120%	119%	171%
crafty-ref	1466	0.06	0.06	5.67	6.17	2.41	2.43	3.11	3.33	55%	61%	81%	108%
gap-ref	1619	0.99	1.04	2.18	2.27	2.86	2.95	3.62	3.96	20%	24%	26%	33%
gcc-expr-ref	89	0.11	0.12	4.99	5.60	2.36	2.38	3.02	3.13	46%	50%	66%	81%
go95-5stone21-ref	138	0.02	0.02	20.65	21.21	1.21	1.21	1.32	1.33	186%	205%	254%	342%
gzip-graphic-ref	774	0.73	0.73	10.42	10.56	1.63	1.64	1.89	1.94	102%	113%	128%	178%
ijpeg95-specmun-ref	84	0.63	0.63	4.67	4.83	2.51	2.54	3.37	3.59	42%	44%	66%	76%
li95-ref	329	0.00	0.00	5.24	6.42	2.42	2.45	3.05	3.22	52%	59%	79%	96%
mcf-ref	441	128.13	128.87	5.02	4.75	0.10	0.10	0.10	0.11	1%	2%	1%	1%
parser-ref	2803	0.04	0.04	7.69	7.66	1.75	1.85	1.99	2.19	53%	71%	61%	90%
perlbmk-diffmail-ref	117	0.04	0.04	2.34	2.40	2.97	3.00	4.21	4.45	25%	28%	38%	46%
twolf-ref	1075	0.02	0.03	13.43	16.59	1.36	1.41	1.49	1.58	86%	116%	101%	149%
vortex-two-ref	407	0.97	0.99	0.29	0.30	3.54	3.63	5.18	5.66	3%	3%	4%	5%
vpr-route-ref	528	5.48	6.91	9.98	9.62	1.18	1.24	1.32	1.44	73%	95%	73%	97%

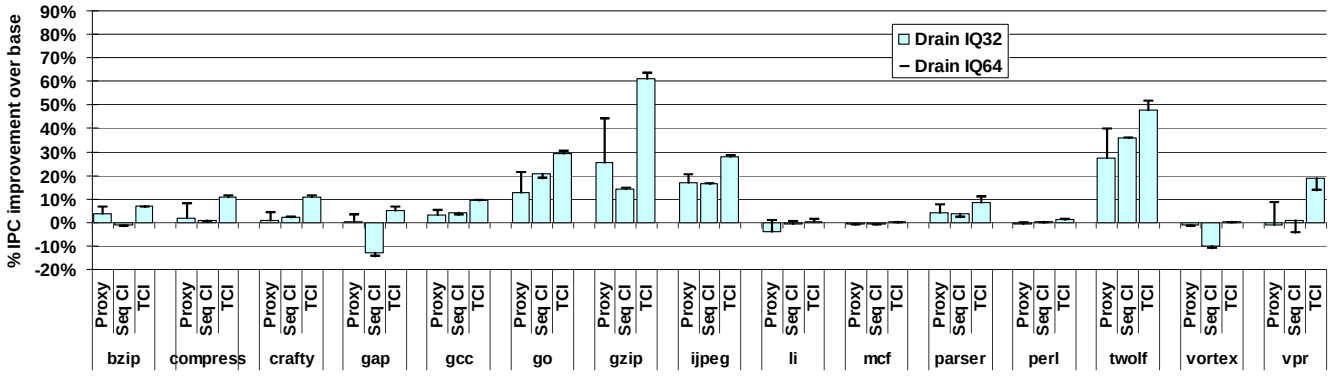

Figure 9. Performance improvement for 4-issue pipeline.

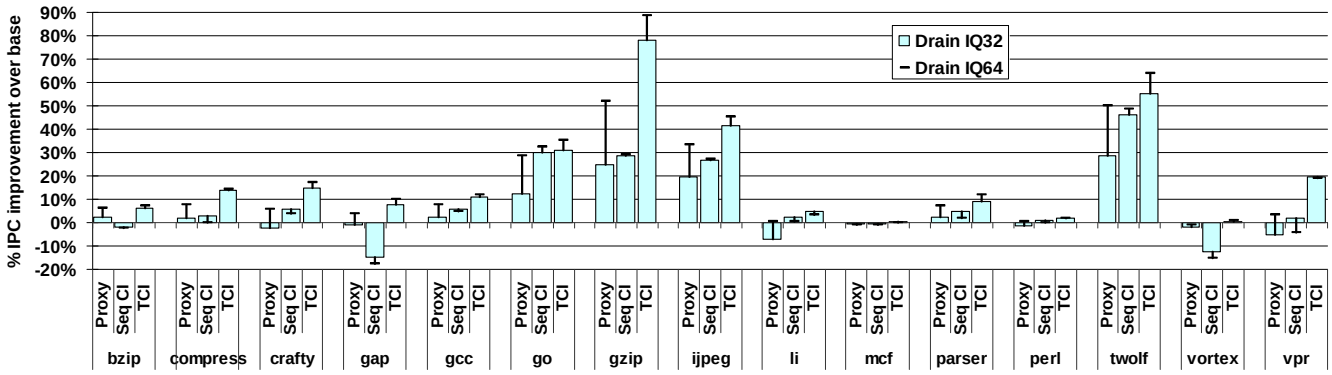

Figure 10. Performance improvement for 8-issue pipeline.

For all benchmarks, a single simulation point of 100 million instructions was selected using the SimPoint 3.2 [27] toolkit. In addition, predictors and caches are warmed up for 10 million instructions prior to starting the simulation point. Table 3 shows benchmarks, inputs, and selected simulation points.

6. RESULTS

We present performance results for five models: *Base* (the baseline described in Section 5), *Proxy*, *Seq CI*, *TCI*, and *Perfect* (the baseline with perfect branch prediction). *Proxy*, *Seq CI*, and *TCI* leverage the *Drain IQ* re-execution substrate (see Section 3). Table 3 shows the IPCs for *Base* for 4-issue and 8-issue pipelines with 32-entry and 64-entry issue queues. IPC improvement of *Perfect* over *Base* is also shown in Table 3.

6.1 Performance and Analysis

Figure 9 shows the performance improvement of the various models over *Base*, for 4-issue pipelines with 32-entry and 64-entry issue queues. The 64-entry issue queue results are shown as error bars with respect to the 32-entry bars. *TCI* improves IPC by up to 61% (64%) over *Base* with a 32-entry (64-entry) issue queue. The average IPC improvement of *TCI* over *Base*, across all benchmarks, is 16% for both issue queue sizes.

Figure 10 shows corresponding IPC improvements over *Base* for 8-issue pipelines. The maximum improvement of *TCI* over *Base* increases to 78% (88%) for a 32-entry (64-entry) issue queue, as the opportunity cost of mispredictions is higher for the wider pipeline. On average, *TCI* achieves 20% (22%) IPC improvement over *Base* for a 32-entry (64-entry) issue queue.

TCI consistently and significantly outperforms *Seq CI*, making clear that resequencing all CI instructions after a misprediction does not fully capitalize on control independence opportunity. Furthermore, as a consequence of limiting the window to the size of the RXB, *Seq CI* degrades performance on some benchmarks with respect to the ROB-free *Base*.

Proxy is not resource efficient. As seen in Figure 9 and Figure 10, for the 32-entry issue queue, *TCI* outperforms *Proxy* in all benchmarks. In some benchmarks (e.g., li, vpr), *Proxy* degrades with respect to *Base* as a result of issue queue pressure caused by proxy and CIDD instructions. The average gain for *Proxy* drops from 11% to 6% on a 4-issue pipeline when the issue queue size is reduced from 64 to 32. In contrast, *TCI* and *Seq CI* are less sensitive to the issue queue size.

To understand the performance improvements of *TCI*, we refer to measurements in Table 3 (L2 load misses per 1000 instructions, branch mispredictions per 1000 instructions) and Figure 11. The latter provides a breakdown of branch mispredictions. Some mispredictions are not covered because they have a maximum CD path length that exceeds our chosen threshold of 256 (Non-CI Br) or they resolve before reconverging. For some mispredictions, control independence is attempted (CI Br) but it fails due to downgrade scenarios, two of which are (i) incomplete IRS (IRS downgrade) and (ii) exceed temp buffer (TB downgrade) thereby preventing RXB expansion. Control independence cannot be exploited in these cases. Due to this, in some benchmarks where branch misprediction rates are fairly high, *Perfect* shows great promise but *TCI* cannot exploit enough control independence resulting in more modest performance gains (e.g.,bzip,compress).

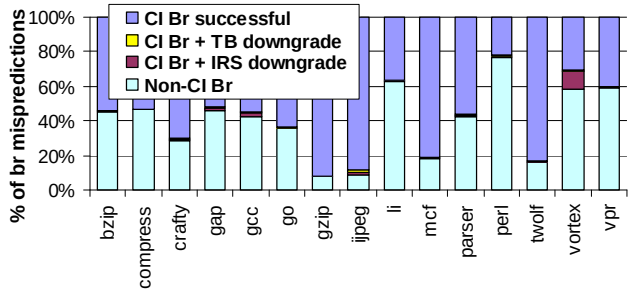

Figure 11. Breakdown of branch mispredictions.

To not artificially favor misprediction-tolerance, we chose the high quality perceptron predictor [13]. Notice in Table 3 branch misprediction rates for *TCI* are typically higher than for *Base*. This is mainly due to gaps in global history (branches in mispredicted CD regions are omitted from global history used by future branches). We found the perceptron predictor to be relatively more resilient to history gaps than gshare. Further, *TCI* can tolerate some extra mispredictions.

We analyze the 64-entry issue queue results by grouping benchmarks based on branch misprediction rates (Table 3) and control independence coverage (CI coverage) (Figure 11):

- Group A (bzip, compress, go, gzip, twolf, and vpr): High misprediction frequency (9 to 21/1K inst.). Gzip and twolf post significant speedups due to high CI coverage (92% and 83%): 64% and 52% on 4-issue, and 88% and 64% on 8-issue. Go posts a medium speedup: 30% for 4-issue and 35% for 8-issue. Though it has the highest branch misprediction frequency, benefits are limited by medium CI coverage (64%), leaving about 7.6 mispredictions uncovered per 1000 instructions. For bzip, compress, and vpr, CI coverage is moderate (54%, 54% and 40%) leading to moderate speedups: 7%, 11%, and 14% for 4-issue, and 7%, 14%, and 19% for 8-issue.
- Group B (crafty, gcc, ijpeg, li, and parser): Moderate misprediction frequency (4 to 8/1K inst.). For crafty, gcc, ijpeg, and parser, CI coverage is medium to high (55%-88%), yielding modest speedups: 11%, 10%, 28%, and 11% on 4-issue, and 17%, 12%, 45% and 12% on 8-issue. Li shows low speedups (1-3%) due to its low CI coverage (37%). In li, most branch mispredictions resolve before fetching their reconvergent points.
- Group C (gap, perl, and vortex): Low misprediction frequency (less than 3/1K inst.). Group C does not benefit from *TCI* due to excellent accuracy in the simulated regions, yielding performance close to *Perfect*.
- Group D (mcf): Moderate misprediction frequency, but very high L2 miss rate. For mcf, the simulated region is dominated by a high frequency of serialized L2 misses, as shown in the third column of Table 3. Despite high CI coverage (81%), the penalty of branch mispredictions is masked since they occur in the shadow of L2 misses. This is confirmed by the negligible gains for *Perfect*.

6.2 Instruction Breakdown

Figure 12 characterizes retired instructions in the context of branch mispredictions. SBM ("shadow of branch misprediction") refers to control independent instructions that are logically in the window when a prior misprediction is detected. (In *TCI*, these are preserved whereas *Base* squashes and re-fetches them.) In contrast, instructions before mispredictions or instructions fetched after a misprediction has initiated servicing, are not considered to be in the shadow of a branch misprediction (Non-SBM). SBM instructions represent control independence opportunity, Non-SBM do not.

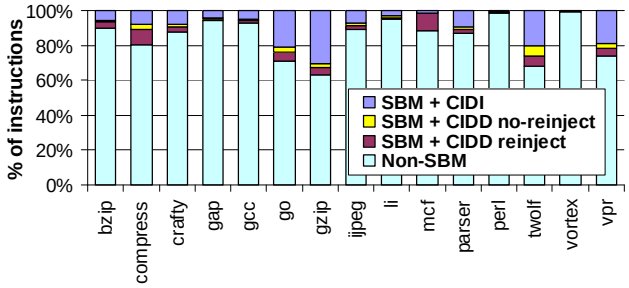

Figure 12 . Breakdown of all instructions.

SBM instructions are broken down further into those that were inserted into the RXB (CIDD) and those that were not (CIDI). Among those that were inserted into the RXB, we indicate if they had to be reinjected (CIDD reinject) or not (CIDD no-reinject). SBM+CIDD reinject occurs when the instruction is CIDD with respect to the mispredicted branch (must re-execute). SBM+CIDD no-reinject occurs when the instruction is not CIDD with respect to the mispredicted branch, but rather a different correctly predicted branch. Thus, SBM+CIDD no-reinject is tantamount to SBM+CIDI with respect to the misprediction.

Summing up, the top two classes in Figure 12 (SBM+CIDD no-reinject, SBM+CIDI) represent savings compared to conventional (full) recovery. Benchmarks in Group A and Group B have the

largest percentages of these misprediction-independent instructions (7%-33% for Group A and 4%-11% for Group B). Their speedups in Figure 9 and Figure 10 correlate well with their percentages of saved instructions.

7. ADDITIONAL RELATED WORK

We already compared and contrasted TCI with the following control independence architectures in Section 3 and, in the interest of space, that discussion is not repeated here: speculative multithreading architectures such as Multiscalar [22] and DMT [2], trace processors [20], and superscalar based implementations including instruction reuse [21], dual ROBs [6], Skipper [5], exact convergence [10], and a generic implementation [19].

ReSlice [24] uses slice re-execution to selectively recover from data misspeculation. Correct repair is guaranteed by checking for sufficient slice conditions. In general, ReSlice is designed for any data misspeculation handling including control-flow influenced data misspeculation, but it was studied only for thread-level speculation (TLS). ReSlice aborts slice re-execution if there are branches (whether in the slice or not) that change the slice's instructions. As we illustrated with the example in Section 4.2.1 of two co-mingled CIDD slices, RXB reconstruction allows slices to change, moreover, the co-mingled slices can resequence in any order, with correct results.

The continual flow pipeline (CFP) [23] is related to our work in that CFP takes an analogous approach for releasing resources of L2 miss dependent instructions. However, CFP does not exploit control independence.

Multipath execution [1][11][14][25][26] reduces misprediction penalties, but also decreases performance and increases power consumption when both paths of a correctly predicted branch are fetched/executed. Predication (e.g., [15][17]) has the same drawback of consuming excess resources by fetching/executing multiple paths, and also delays forwarding of correct speculative values outside of predicated blocks.

8. SUMMARY

For misprediction-inflicted workloads running on deep superscalar pipelines, exploiting control independence is an effective means for reducing the performance penalty of branch mispredictions.

The essential goal of exploiting control independence is to completely decouple future misprediction-independent instructions from deferred misprediction-dependent instructions. Previous implementations fall short of complete decoupling because they still explicitly maintain order among all instructions. TCI is successful because it enforces order indirectly, by breaking dependences between co-mingled misprediction-independent and misprediction-dependent instructions. TCI facilitates truly selective recovery in terms of burning a minimum amount of extra resources and bandwidth on a condensed recovery stream, yielding higher performance than all previous approaches and presenting a qualitatively compelling streamlined design.

9. ACKNOWLEDGMENTS

This research was supported by NSF grant No. CCR-0429843, NSF CAREER grant No. CCR-0092832, and generous funding and equipment donations from Intel. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

10. REFERENCES

[1] P. Ahuja, K. Skadron, M. Martonosi, D. Clark. Multipath Execution: Opportunities and Limits. *ICS*, 1998.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. *MICRO-31*, 1998.

[3] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. *MICRO-36*, 2003.

[4] D. Burger, T. Austin, S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. July 1996.

[5] C-Y. Cher,T. Vijaykumar. Skipper: A Microarchitecture for Exploiting Control-flow Independence. *MICRO-34*,2001.

[6] Y. Chou et al. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. *ICS*, 1999.

[7] G. Chrysos and J. Emer. Memory Dependence Prediction Using Store Sets. *ISCA-25*, 1998.

[8] J. D. Collins et al. Control Flow Optimizations Via Dynamic Reconvergence Prediction. *MICRO-37*, 2004.

[9] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. *HPCA-10*, 2004.

[10] A. Gandhi et al. Reducing Branch Misprediction Penalty via Selective Branch Recovery. *HPCA-10*, 2004.

[11] T. Heil and J. Smith. Selective Dual Path Execution. Tech. Report, ECE Department, UW-Madison, 1996.

[12] W.-M. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. *IEEE Transactions on Computers*, 36(12):1496-1514, Dec. 1987.

[13] D. A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. *HPCA-7*, 2001.

[14] A. Klauser, A. Paithankar, D. Grunwald. Selective Eager Execution on the Polypath Architecture. *ISCA-25*, 1998.

[15] A. Klauser et al. Dynamic Hammock Predication for Non-predicated Instruction Set Architectures. *PACT*, 1998.

[16] A. R. Lebeck et al. A Large, Fast Instruction Window for Tolerating Cache Misses. *ISCA-29*, 2002.

[17] S. Mahlke et al. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *ISCA-22*, 1995.

[18] M. Moudgill et al. Register Renaming and Dynamic Speculation: an Alternative Approach. *MICRO-26*, 1993.

[19] E. Rotenberg, Q. Jacobson, J. Smith. A Study of Control Independence in Superscalar Processors. *HPCA-5*, 1999.

[20] E. Rotenberg and J. Smith. Control Independence in Trace Processors. *MICRO-32*, 1999.

[21] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. *ISCA-24*, 1997.

[22] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *ISCA-22*, 1995.

[23] S. T. Srinivasan et al. Continual Flow Pipelines. *ASPLOS-XI*, 2004.

[24] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Reslice: Selective Re-execution of Long-Retired Misspeculated Instructions Using Forward Slicing. *MICRO-38*, 2005.

[25] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. *ISCA-25*, 1998.

[26] S. Wallace, D. Tullsen, B. Calder. Instruction Recycling on a Multiple-Path Processor. HPCA, 1999

[27] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. *ASPLOS-X*, 2002.