BranchTap: Improving Performance with Very Few Checkpoints Through Adaptive Speculation Control

Patrick Akl and Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
{pakl, moshovos}@eecg.toronto.edu

Abstract

Checkpoint prediction and intelligent management have been recently proposed for reducing the number of coarse-grain checkpoints needed to achieve high performance through speculative execution. In this work, we take a closer look at various checkpoint prediction and management alternatives, comparing their performance and requirements as the scheduler window size increases. We also study a few additional design choices. The key contribution of this work is BranchTap, a novel checkpoint-aware speculation strategy that temporarily throttles speculation to reduce recovery cost while allowing speculation to proceed when it is likely to boost performance. BranchTap dynamically adapts to application behavior. We demonstrate that for a 1K-entry window processor with a FIFO of just four checkpoints, our adaptive speculation control mechanism leads to an average performance degradation of just 1.49% compared to a processor that has an infinite number of checkpoints. This represents an improvement of 28.3% over using just prediction-based checkpoint allocation. Average performance degradation without BranchTap is 2.08%. For the same configuration, BranchTap decreases the worst case deterioration from 8.99% to 5.64%.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms

Performance, Design

Keywords

Branch misprediction, speculation control, state recovery, state checkpointing

1 Introduction

Modern processors use control flow speculation to improve performance. To preserve correctness, recovery mechanisms restore the machine's state on mispeculations. Modern processors utilize two such recovery mechanisms. The first is the re-order buffer (ROB) which allows recovery at any instruction including mispeculated branches. Recovering from the ROB amounts to *squashing*, i.e., reversing the effects of each mispeculated instruction, a process that requires time proportional to the number of squashed instructions. The second recovery mechanism uses a number of global checkpoints (GCs) which are allocated at decode time. A GC contains a complete snapshot of all relevant processor state. Recovery at an instruction with a GC is "instantaneous", i.e., it requires a fixed, low latency.

Ideally, a GC would be allocated at every instruction such that the recovery latency is always constant. In practice, only a limited number of GCs can be implemented without impacting the clock cycle significantly and thus reducing overall performance (we explain this issue further in Section 2.2).

For processors with relatively small scheduling windows, using a few GCs is sufficient. For example, the MIPS R10000 had a 32-entry window and used just four GCs which were allocated to branches in-order [24]. If no GC was available for a branch, decoding was stalled.

Recent work has demonstrated that for processors with larger scheduling windows a lot more checkpoints are needed [1,2,4,16]. Using checkpoint prediction was proposed for allocating few GCs judiciously to *low confidence*, or *weak* branches (i.e., those that are likely to cause a mispeculation) [1,2,16]. In addition, advanced GC management methods were proposed to further improve GC efficiency [16]. We review these proposals in more detail and other related work in Section 4. For many programs at least eight and often 16 GCs were needed to maintain performance within 2% of that possible with an infinite number of checkpoints even with a 256-entry window [16]. A method that improves GC efficiency further is highly desirable for the following three reasons: (1) it will reduce overall GC requirements thus improving scalability for future wider window processors, (2) it will reduce the overall cost, and thus power of the GC mechanism, and (3) more importantly, a method that allows us to use few checkpoints would permit embedding those checkpoints inside all relevant structures thus eliminating the need for expensive interconnects for checkpoint content transfers (see Section 2.2).

Accordingly, in this work we are concerned with methods that maximize performance with very few GCs (four or less) for processors with relatively large scheduling windows (i.e., 512 or 1K instructions). Existing GC allocation methods reduce the mispeculation recovery cost by allocating GCs to those branches[1] that are likely to be mispeculated. Unfortunately, this task becomes increasingly harder as the number of GCs is reduced and as the scheduler window size and thus the number of in-flight branches increases.

We observe that, in addition to checkpoint prediction, we can further reduce recovery costs by also trying to reduce the number of instructions that would need to be squashed. Accordingly, the key contribution of this work is *BranchTap*, a method that combines adaptive speculation control with prediction-based checkpoint allocation to maximize performance. Our method works by tracking the number of in-flight weak branches without a GC. The fetch stage is stalled while the aforementioned number exceeds a threshold. We demonstrate that using a fixed threshold is suboptimal across programs. Accordingly, we propose a method that dynamically adapts this threshold using short sampling intervals. BranchTap successfully improves performance for most programs even though the underlying trade-offs are complex. In particular, speculation control can significantly impair performance when used often, and can be ineffective in reducing recovery costs if used sparingly.

BranchTap requires little additional resources and its implementation is straightforward. BranchTap builds upon previous work in speculation control for power reduction, e.g., [9,13]. However, as we also explain in Section 4, there are three significant differences: (1) previous work assumed fixed recovery latencies at all mispeculations and thus ignored their impact on performance, (2) the metric, and (3) the policy used by BranchTap are very different. To the best of our knowledge, this is the first work that proposes combining speculation control with checkpoint prediction and that also proposes an adaptive method for adjusting speculation control.

The contributions of this work are:

- We propose BranchTap, a method that uses adaptive speculation control coupled with checkpoint prediction for improving over previous checkpoint allocation and management methods. Specifically, we demonstrate that our method reduces average mispeculation recovery cost by 28.3% compared to a previously proposed technique that uses checkpoint-prediction alone, when only four checkpoints are available for a 1K-entry scheduler.
- We also compare the previously proposed checkpoint prediction and allocation policies studying more closely the underlying design trade-offs. We discuss possible implementations of the more advanced checkpoint management methods and suggest a few simple to implement improvements.

The rest of this paper is organized as follows. In Section 2, we start by reviewing existing checkpointing alternatives and discuss the underlying performance trade-offs. In Section 3, we present BranchTap. In Section 4, we review related work. In Section 5, we present the results of our experimental analysis of BranchTap. We summarize our key findings in Section 6.

2 Checkpoint/Recovery Background

In this section, we start with a brief overview of existing checkpoint/recovery mechanisms. We then discuss the underlying performance trade-offs and how previously proposed techniques exploit these trade-offs to improve performance. In our discussion, without loss of generality, we focus on checkpointing for the register alias table (RAT) for clarity. The same concepts are applicable to other processor structures such as the register free list and the queue pointers and valid bits of the scheduler.

[1] Because of the high frequency of branch mispredictions relative to interrupts and exceptions we only consider control flow instructions as potential causes for machine state recovery.

2.1 RAT Checkpoint/Recovery

Register renaming eliminates false register dependencies, thus increasing instruction level parallelism. In our work, we focus on the register renaming implementation used in the MIPS R10000 where architectural registers are dynamically mapped onto a larger set of physical registers [24]. The current mapping is held in the Register Alias Table (RAT). This table is used by every instruction during the decode phase: the instruction's input operands are mapped to physical registers as specified in the RAT and then its destination register is mapped to a free physical register by updating the corresponding RAT entry. Detailed descriptions of RAT-based renaming can be found in [15,16,24]. To rename up to N instructions per cycle for an instruction set with two input and one output register operands, 3xN read ports and N write RAT ports are needed. Furthermore, for a W-entry window processor, W physical registers are typically used. Thus, each RAT entry contains lg(W) bits.

2.2 ROB and GC Checkpoint/Restore

There are two commonly used methods for RAT checkpointing and recovery. The first is the reorder buffer (ROB) and the second uses a set of global checkpoints. The ROB is a circular buffer. Instructions allocate an ROB entry in program order as they are decoded and release it upon commit. The ROB entry contains sufficient information for reversing the effects of the corresponding instruction on a mispeculation. For the RAT, it is sufficient to keep the previous mapping for the instruction's destination register. The ROB is a *fine-grain* checkpoint mechanism as it allows recovery at any instruction as follows: starting from the most recently decoded instruction, we traverse the ROB in reverse order while writing back the previous mappings into the RAT until we reach the branch that was mispeculated. The number of cycles required for ROB recovery is proportional to the number of instructions being squashed. It is reasonable to assume that up to N instructions can be squashed per cycle if the machine is capable of decoding N instructions per cycle. Because the ROB is *fine-grain,* it can be used to recover from any exception in addition to branch mispeculations.

The second method uses global checkpoints (GCs) which contain complete RAT content snapshots. Conceptually, as shown in Figure 1(a), the GCs form a queue of complete RAT replicas. As implemented in the R10000 processor, each RAT bit has embedded next to it a small queue (shown in Figure 1(b)). A GC is taken by shifting into the queue a copy of the corresponding RAT bit (all RAT bits are copied in parallel into their own little queues). Recovery amounts to copying the contents of one of the queue elements back to the corresponding RAT bit. This can be done at a fixed latency which is independent of the number of instructions that are squashed. Thus, the more instructions are squashed, the more preferable it is to use GC over ROB recovery. GC is a *coarse-grain* checkpoint/restore mechanism as it allows recovery only at some instructions.

Ideally, we would allocate a GC to every instruction. Unfortunately, implementing more GCs can impact RAT latency and thus overall performance. If the GCs are embedded inside the RAT next to each RAT bit cell, embedding more GC bits will

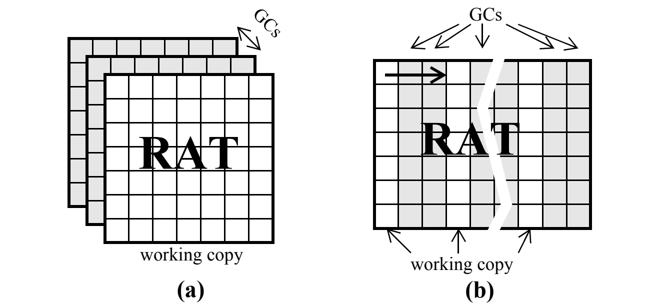

Figure 1: GC RAT Checkpointing. (a) Conceptual organization. (b) Actual implementation.

elongate either the wordlines or the bitlines or both. If we implement the GCs separately, additional bit lines would be needed to communicate *all* RAT content from or to the GC store. This will also elongate the RAT bitlines or wordlines since additional wires will be needed to transfer the RAT content. For example, for a 64-entry RAT and a 512-entry physical register file, 64x9 wires are needed to checkpoint the complete RAT. Elongating the wordlines or the bitlines increases their capacitance, access latency, and power. It may also require the use of larger transistors in the RAT cells to maintain stability. Accordingly, it is desirable to maintain the number of GCs as small as possible. In this work we target four or less GCs for processors with 512- or 1K-entry windows (the MIPS R10000 used four GCs too but it had a window of 32 instructions).

2.3 GC Prediction

Previous work has observed that checkpoint prediction can be used to improve efficiency over naive checkpoint allocation [16]. Specifically, as originally used in R10000, GCs were allocated to all branches at the decode stage. If no GC was available, decode stalled. GCs were released in-order as the branches were resolved. Previous work showed that many GCs would be needed for larger window processors if this *conventional* policy was used [1,2,16].

Accordingly, rather than allocating GCs to all branches previous work suggested using a confidence mechanism to allocate GCs only for low confidence (weak) branches. Moshovos used *anyweak*, a simple confidence mechanism relying on the bias of existing branch predictors [16], while Akkary et. al. [1,2] used the dedicated confidence estimator proposed by Jacobsen et. al. [11].

2.4 Performance Trade-offs with GC Prediction

When GCs are allocated only to some branches there are two possible recovery scenarios. In the first, the mispeculation occurs at a branch with a GC. In this case recovery latency is fixed and independent of the number of squashed instructions. In the second, the mispeculation occurs at a branch without a GC. In this case two possible recovery policies are possible. In the first used by Moshovos [16] and shown in Figure 2, recovery proceeds into two phases. First, the closest subsequent GC, if any exists, is used to partially recover the machine state at that instruction, and then the ROB is used to complete the recovery. In this case, the recovery latency is proportional to the number of instructions in-between the mispeculated branch and the closest subsequent branch with a GC (or the end of the ROB if no such branch exists). In the second policy, used by Akkary et. al. [1,2], the closest *preceding* branch with a GC is located and used to restore machine state. Instructions following the GC and up to the mispeculated branch are re-executed[1]. The advantage of this method is that it can be used without an ROB. In this work, we focus on the first recovery model. However, BranchTap can potentially also be used with the second recovery model. In this case, BranchTap can help when a branch that would cause a mispeculation is delayed until a GC becomes available. An investigation of this aspect is beyond the scope of this work.

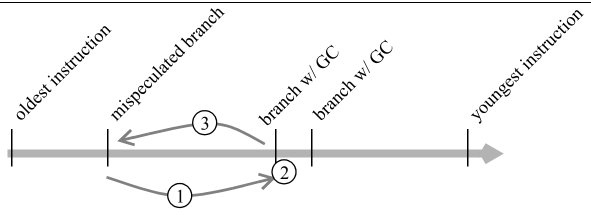

Figure 2: Checkpoint recovery cost when not all branches have a GC. (1) First we find the next branch with a GC if any. (2) Then we recover at that branch using the GC. (3) Finally, we use the ROB to roll-back to the mispeculated branch.

Under this recovery model, the recovery cost can be expressed as the sum of the number of cycles required to perform three tasks in sequence:

$$RecoveryCost = \begin{array}{c} CyclesToLocateNextGC + \\ CyclesToRecoverAtNextGC + \\ CyclesToRecoverFromROBStartingAtNextGC \end{array}$$

As we explain in Section 5.2.5, when GCs are allocated in-order, it is straightforward to locate the next GC. However, when out-of-order allocation is used it may not be possible to locate the next GCs in a single cycle [16]. Specifically, when there are just four GCs it is reasonable to assume that locating the next GC from any instruction can be done with a fixed latency (we calculate the distance to all four GCs in parallel and select the minimum). When more GCs are used, a tree-like structure similar to that used for selecting instructions for wake-up in schedulers can be used [17] (see Section 5.2.5).

3 BranchTap

In this work, we are interested in improving overall performance for processors with relatively large windows (i.e., 512 and 1K instructions), and with very few GCs (e.g., four or less). We can improve performance by trying to improve how we allocate the GCs so that more mispeculated branches are given a GC. Unfortunately, trying to identify the branches that will be mispeculated becomes increasingly more difficult as the number of instructions in the window increases, and as the number of GCs is reduced. As we demonstrate in Section 5, using a larger and thus more accurate confidence estimator does improve performance. However, BranchTap offers a significantly better performance vs. cost tradeoff. BranchTap requires few counters and comparators whose cost is negligible. For most programs, at least an additional 12Kbits are needed for the confidence table to provide similar benefits.

In developing BranchTap, we observe that rather than trying to allocate GCs more effectively, we can instead try to minimize the number of instructions that have to be recovered from the ROB

[1] The closest earlier checkpoint can be detected early at decode time for all instructions if in-order checkpoint allocation and release is used as in [1,2].

when it was not possible to allocate a GC. This is the last factor in the aforementioned recovery cost model. BranchTap achieves this goal by temporarily stalling the fetch stage while the number of preceding unresolved weak branches that do not have a GC exceeds a threshold WT. The insight behind this approach is as follows: when the aforementioned condition holds true, then the delayed instructions follow a relatively long sequence of in-flight, unresolved weak branches. The probability that those instructions will not be squashed is thus relatively small and becomes smaller the more weak branches are in-flight. If we were to allow more instructions to proceed, then we are most likely increasing the number of instructions that will have to be squashed. Since all GCs are currently allocated to earlier branches, these instructions will have to be squashed from the ROB and hence recovery latency will only become longer.

BranchTap uses an adaptive policy for dynamically adjusting its threshold WT. This allows it to adapt across and within applications. Figure 2 illustrates how BranchTap is integrated into the pipeline and the sampling process it uses. We have experimented with many different adaptive policies. The policy that performed well across all benchmarks works in two repeating phases. During the first phase, execution proceeds for a relatively long time (we use one million cycles) with the current threshold. During the second phase, sampling is used to determine whether the threshold should change. The second phase consists of a number of relatively small sampling sub-phases of 25000 cycles each where we count the number of committed instructions for three possible threshold values: Current, Current-4 and Current+4. We then update WT with the threshold value that resulted in the largest number of committed instructions. We use very short sampling intervals to minimize their effect on overall performance.

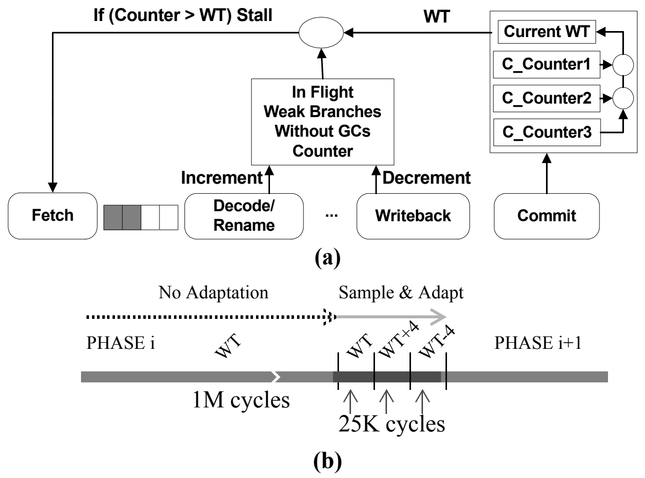

Figure 3: (a) How BranchTap is integrated into the pipeline (threshold = WT). (b) How BranchTap adapts its threshold WT.

BranchTap requires very few resources (a few counters and comparators, and the ability to temporarily stall the decode stage). Thus, its use may be justified even when it does not improve performance drastically.

4 Related Work

BranchTap is orthogonal to checkpoint prediction methods proposed by Moshovos [16] and Akkary et. al. [1,2]. A contribution of this work is that it compares these two proposals under the recovery model used in [16]. Another contribution of this work is that it analyzes the impact of several design choices on overall performance. BranchTap can be used with an ROB as in [16] or without one[1] as in [1,2]. In this study we focus only on using BranchTap with an ROB.

BranchTap builds upon previous work on speculation control for power reduction [9,13]. With BranchTap there are three key differences. First, previous work assumed a fixed recovery cost from mispeculations. Accordingly, the performance trade-offs we are interested in were not accounted for in previous work where speculating more could never increase recovery cost, and hence hurt performance. Second, previous work relied on counting the number of unresolved, weak branches currently in flight. We instead rely on those branches that do not have a GC. Finally, the most important difference is that previous work used a fixed threshold. We demonstrate that using a fixed threshold is suboptimal across programs and that the performance differences can be significant for some programs.

BranchTap relies on a confidence estimator for identifying weak branches. We study both the anyweak estimator [16] and the confidence estimator proposed by Jacobsen et. al. [11]. Jimenez and Lin studied composite branch confidence estimators that are more accurate but require more resources [12]. BranchTap is orthogonal to the choice of the confidence estimator; thus it can be used to boost performance as needed with little additional cost.

Another approach to reducing the cost of mispeculations is to execute multiple control flow paths at hard to predict branches using predication techniques, e.g., [3] or dynamically, e.g., [23]. Also, control independence can be exploited to avoid complete re-execution of some instructions that are squashed [5,6,8,18,21]. These approaches are orthogonal to BranchTap and require additional support.

Modern checkpoint/recovery mechanisms have evolved out of earlier proposals for supporting speculative execution [10,19,22].

Other proposals for improving scalability for wider window processors target early reclamation of processor resources [4,14]. As we have seen, allowing fetch to proceed in parallel with RAT recovery reduces the performance loss on mispeculations. Zhou et. al. proposed a more aggressive method where some of the instructions down the correct control path may be renamed while RAT recovery is still in progress [25]. It may be possible to use BranchTap with these techniques. However, this investigation is beyond the scope of this paper.

5 Evaluation

In Section 5.1 we detail our experimental methodology. In Section 5.2 we compare previously proposed checkpointing alternatives and study some of the design space parameters in more detail. In Section 5.3 we demonstrate that once a good checkpoint prediction policy is used, most mispeculations occur on branches with a GC but most of the cycles lost to recovery are caused by branches without a GC. This result illustrates that significant potential exists for BranchTap. In Section 5.4 we show that different programs require different speculation control thresholds (WT) to perform best. This result motivates the use of adaptive speculation control in BranchTap. Finally, in Section 5.5 we study performance with BranchTap.

[1] In this case, BranchTap could improve performance if it delays a branch long enough for a GC to become available. Different performance trade-offs apply in this case.

5.1 Methodology

We used Simplescalar v3.0 [7] to simulate the processor detailed in Table 1. We compiled the SPEC CPU 2000 benchmarks for the Alpha 21264 architecture using HP's compilers and for the Digital Unix V4.0F using the SPEC suggested default flags for peak optimization. All benchmarks were ran using a reference input data set. It was not possible to simulate some of the benchmarks due to insufficient memory resources. As a result the following SPEC CPU 2000 benchmarks are included in our experiments: *ammp, applu, apsi, art, bzip2, crafty, eon, equake, facerec, fma3d, galgel, gap, gcc, gzip, lucas, mcf, mesa, mgrid, parser, swim, twolf, vortex, vpr* and *wupwise*. To obtain reasonable simulation times, samples were taken for one billion committed instructions per benchmark. We first skipped 100 billion committed instructions prior to collecting measurements for all benchmarks except for *art* and *parser* for which we only skipped 20 billion instructions.

Table 1. Base processor configuration

Branch Predictor	**Fetch Unit**
8K-entry GShare and 8K-entry bi-modal 16K selector 2 branches per cycle	Up to 8 instr. per cycle 64-entry Fetch Buffer Non-blocking I-Cache
Issue/Decode/Commit	**Scheduler**
any 8 instr./cycle	512- or 1K-entry/half size LSQ
FU Latencies	**Main Memory**
Default simplescalar values	Infinite, 200 cycles
L1D/L1I Geometry	**UL2 Geometry**
64KBytes, 4-way set-associative with 64-byte blocks	1Mbyte, 8-way set-associative with 64-byte blocks
L1D/L1I/L2 Latencies	**Cache Replacement**
3/3/16 cycles	LRU
Fetch/Decode/Commit Latencies	
4 cycles + cache latency for fetch	

Unless otherwise noted all performance results are normalized over an identical configuration that has an infinite number of GCs. We refer to this configuration as "perfect checkpointing", or PERF. However, in some cases having fewer GCs results in better performance. We explain why this is possible in Section 5.2.4.

5.2 Existing Checkpointing Alternatives

Before investigating BranchTap we study existing checkpoint alternatives to better understand the underlying trade-offs and their relative performance. To the best of our knowledge no previous study compared the performance of the recently proposed checkpoint-prediction-based techniques of [1,2,16]. The checkpoint design parameters that we consider in this work are the following:

- **Checkpoint count**: While having more GCs makes checkpoint allocation easier, it also increases complexity, latency, and power.
- **Checkpoint prediction**: It is desirable to utilize the limited number of available checkpoints efficiently. We consider using a confidence estimator as in [1,2] or the anyweak estimator of [16]. The latter is less accurate but requires virtually no additional resources.
- **Checkpoint release stage**: Checkpoints can be released at commit or at writeback. Release at commit requires no additional support while release at writeback requires additional support for tracking the release status of preceding branches.
- **Checkpoint release order**: If GCs are released at writeback there are two possibilities: release them in-order or out-of-order. In the first case, a GC is held until all preceding branches have been resolved, whereas in the second case a branch can release its GC as soon as it decides its direction. With out-of-order checkpoint release, additional hardware support is needed for tracking the order of GCs and for locating the nearest subsequent checkpoint on a mispeculation.
- **Checkpoint stealing**: This option can only be used with out-of-order checkpoint release. In particular, as proposed in [16], GCs are initially allocated to all branches in order. A subsequent weak branch can steal the GC of an earlier strong branch if all GCs have been allocated.
- **Stages stalled during recovery:** While a recovery is underway we can chose to stall the whole front-end (fetch and decode stages) or just the decode stage. In the latter case, the fetch stage can be redirected a single cycle after the mispeculation is detected and thus start fetching instructions while the decode stage is stalled for RAT recovery.

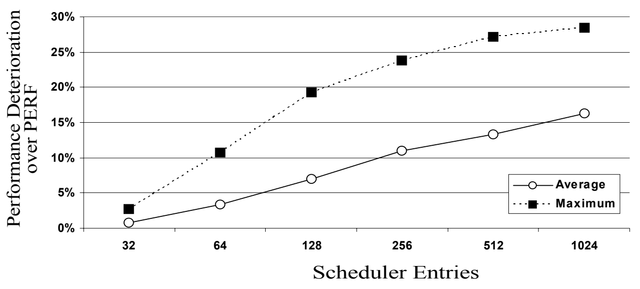

Figure 4: Average and maximum performance degradation as a function of the scheduler window size for ROB_Only recovery.

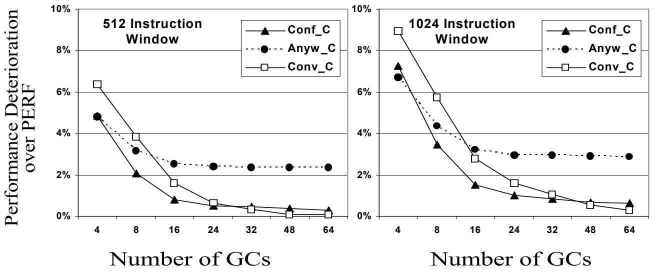

Figure 5: Average performance deterioration for three checkpoint allocation policies as a function of the number of available GCs and for instruction window sizes of 512 (left) and 1024 (right).

5.2.1 ROB-Only Checkpointing

Before considering any of the recently proposed checkpoint enhancement techniques, we validate that ROB-only checkpointing is not a viable alternative for wide-window processors. Figure 4 reports the average and the maximum performance degradation of ROB-only with respect to PERF as a function of scheduler window size (32-entry to 1K-entry windows) and over all benchmarks we studied. While this is not shown on the graph, a few programs (*swim*, *mgrid*, *applu,* and *lucas*) performed well even with ROB-only checkpointing. We do not

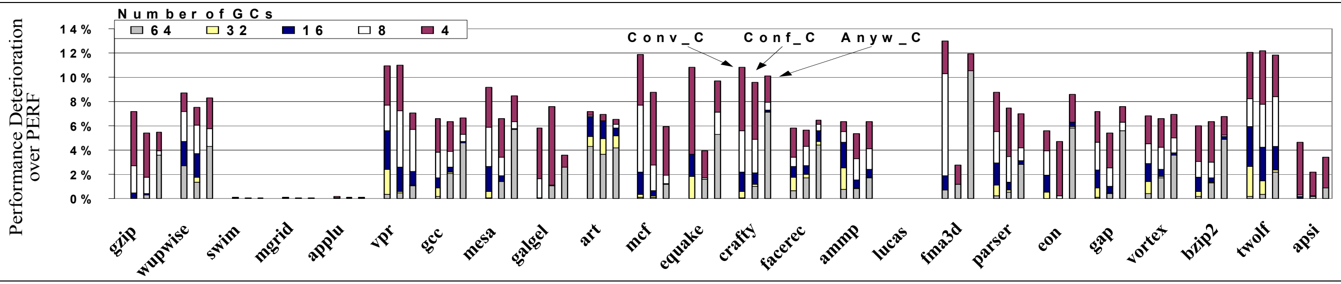

Figure 6: Per-benchmark performance deterioration with Conv_C, Conf_C and Anyw_C (left to right respectively) as a function of the number of GCs for a 1K-entry window processor.

omit these programs from the rest of this evaluation. However, none of the techniques we discuss can improve performance for these programs.

5.2.2 *Checkpoint Prediction*

For the time being, we assume that the GCs are allocated in-order and released at commit, and we compare the performance of conventional checkpointing (Conv_C), anyweak-based GC allocation (Anyw_C), and confidence-based GC allocation (Conf_C). The Conv_C policy is a variation of the policy implemented in MIPS R10000 in that it does not stall a branch at decode when no GC is available.

Figure 5 reports average performance deterioration for the three policies as a function of the number of GCs (X-axis) for two processors with instruction window sizes of 512 and 1K. With a high number of GCs, Conf_C and Conv_C outperform Anyw_C, and Conv_C approaches the performance possible with an infinite number of checkpoints. This is because Conv_C is able to allocate GCs to all branches. Conf_C and Anyw_C allocate GCs only to weak branches.

With eight or less GCs, Conf_C and Anyw_C outperform Conv_C. This corroborates the results of previous work that proposed using checkpoint prediction to improve performance when there are few checkpoints. When there are just four GCs, Anyw_C performs the same (512-entry window) or slightly better than Conf_C (1K-entry window).

Average performance can be misleading. Accordingly, Figure 6 shows the per-benchmark performance for the three policies (Conv_C, Conf_C, and Anyw_C from left to right in that order) as a function of the number of GCs and for a 1K-entry window processor. Behavior varies significantly across benchmarks. In *swim*, *mgrid*, *applu*, and *lucas*, branch mispeculation recoveries are infrequent and do not impact performance noticeably. For most programs, Conf_C and Anyw_C outperform Conv_C when there are few checkpoints. In some cases, however, Conv_C performs slightly better (e.g, *bzip2*). For some benchmarks, Anyw_C performs better than Conf_C (e.g., *mcf*) while for others the opposite is true (e.g., *fma3d*). Ultimately, this result illustrates that the underlying performance trade-offs are complex and minor changes in policy can impact overall performance significantly. We can also observe that some benchmarks suffer from significant performance loss even with eight checkpoints (e.g., 7.8% for *twolf* and Conf_C). In the rest of the paper, due to space limitations, we restrict our attention primarily to policies that use the confidence estimator of [11]. However, the confidence estimator requires a lot more resources than the anyweak estimator.

5.2.3 *Checkpoint Management*

In this section, we compare in-order at commit (Conf_C), in-order at writeback (Conf_I), out-of-order at writeback (Conf_O) GC release, and lazy checkpoint allocation with out-of-order release at writeback and GC stealing (Conf_L). Figure 7 shows average performance deterioration with these four policies as a function of the number of GCs and for processors with 512- (left) and 1K-entry (right) window sizes.

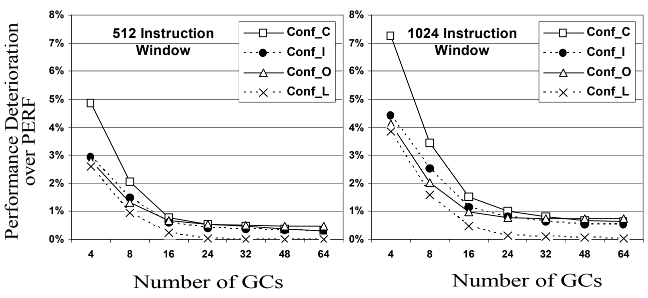

Figure 7: Average performance deterioration with various checkpoint management policies as a function of the number GCs for a 512 (left) and 1024 (right) instruction window processor. See text for a description of the four policies.

Releasing GCs in-order at writeback (Conf_I) is much better than releasing them at commit time (Conf_C) when there are few GCs. This result is expected as releasing a GC early allows us to reutilize it for another low confidence branch. Releasing GCs out-of-order (Conf_O) further improves performance albeit not significantly. The tradeoff between Conf_I and Conf_O is between temporarily keeping the checkpoint until it is certain that earlier branches do no longer need it for recovery (Conf_I), and releasing the checkpoint as early as possible so that it is allocated to newly decoded branches (Conf_O). Performance improves further with Conf_L. This method differs in that it initially allocates free GCs to all branches including non-weak branches. Such branches can never be checkpointed by the other methods. This is why Conf_L is the only policy whose performance approaches that of PERF as the number of checkpoints increases.

Figure 8 presents the per-benchmark performance deterioration for a 1K-entry window processor. The graphs indicate that even with Conf_L, some benchmarks still perform poorly with few checkpoints. *Twolf*, for example, suffers from 11.7% and 7.7% performance deterioration with four and eight checkpoints respectively. This result motivates the need for further improvement. In the rest of this study we restrict our attention to Conf_I due to space limitations and because this policy avoids the complexities of out-of-order checkpoint management (except for

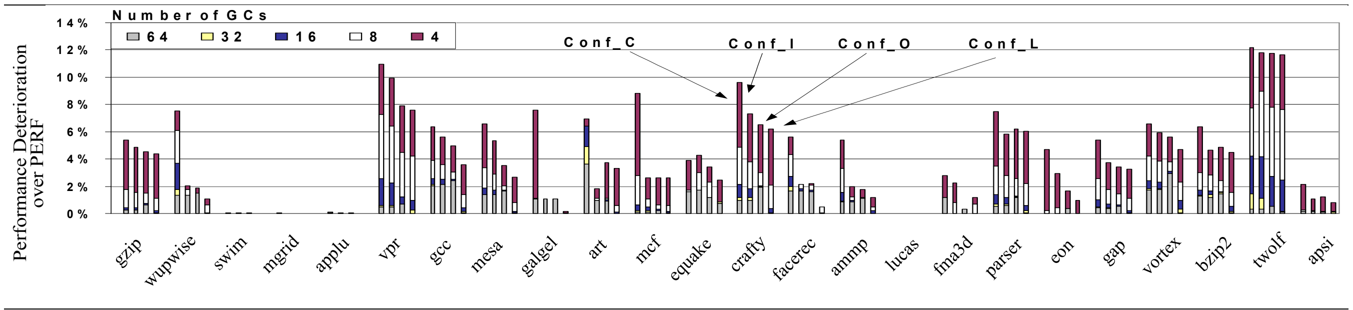

Figure 8: Per-benchmark performance deterioration as a function of the number of Checkpoints and checkpoint management (left to right: Conf_C, Conf_I, Conf_O, Conf_L) for a 512-entry window processor.

Section 5.2.5 where we model the latency effect of finding the next GC on a mispeculation). However, we note that we observed similar trends with Conf_L which is a viable, slightly better alternative when there are very few checkpoints.

5.2.4 *Front-End Stalling Alternatives*

In this section, we analyze the effect of decoupling the fetch stalling mechanism from the decode/rename stalling mechanism during recovery. Figure 9 shows performance deterioration for Conf_I when both fetch and decode are stalled (Conf_I bars) or when only the decode stage is stalled (Conf_I-F bars). In the latter case, fetch is stalled for a single cycle. Conf_I-F reduces performance deterioration by about 40% on average. When only the decode stage is stalled during recovery, the fetch stage can proceed in parallel to fetch the instructions down the right control path. As a result, the overall penalty is reduced. In the rest of the paper we restrict our attention to policies that only stall the decode stage.

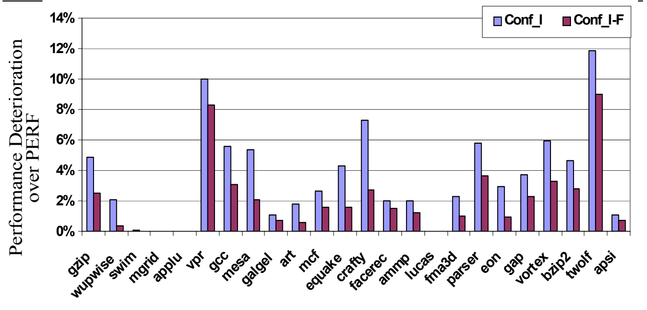

Figure 9: Per-benchmark performance deterioration when both the fetch and decode stages are stalled for the duration of the recovery (Conf_I) or when only the decode stage is stalled (Conf_I-F) for a 1K-entry window processor with 4 GCs.

5.2.5 *Modeling the Latency of Finding the Nearest GC*

Thus far we assumed that finding the nearest checkpoint under out-of-order checkpoint management is possible in a single cycle. As we explained in Section 2.4, this may be reasonable when there are just four checkpoints. For a larger number of checkpoints a search mechanism will probably be needed. As per the discussion of Section 2.4, we modeled the delay of a search mechanism that uses a tree-like structure with a fan-in/fan-out of four. In this model it takes log4(N) cycles to find the nearest subsequent branch with a checkpoint, where N is the number of instructions in-between the mispeculated branch and the branch with the GC. When in-order checkpoint allocation and release are used, it is straight-forward to locate the next GC in a single cycle as follows: as instructions are decoded, they locate the nearest *preceding* GC. On a mispeculation we use that checkpoint to locate the one after it in the GC queue. This is the next nearest GC we should be using.

Figure 10 compares the average performance deterioration for Conf_I (the best performing in-order policy) and two variations of Conf_L for various number of GCs (X-axis) and for a 512-entry window processor. In the first variation (Conf_L) we assume that it takes a single cycle to locate the next GC, while in the second (Conf_L_L4) we use the aforementioned log4(N) model. With four checkpoints Conf_L_L4 performs worse than Conf_I, however, as we explained Conf_L should be feasible in this case. With more checkpoints, Conf_L_L4 performs slightly better than Conf_I. However, the performance improvements are not necessarily significant enough to justify the complexity of out-of-order checkpoint management with more than four checkpoints.

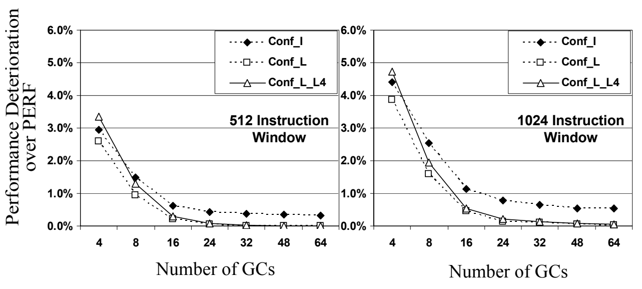

Figure 10: Average performance deterioration when it takes multiple cycles to locate the next GC under out-of-order checkpoint management (Conf_L_L4) as a function of the number of GCs for a 512-entry and 1K-entry instruction window processor. See text for a description.

5.2.6 *Impact of the Confidence Estimator*

Performance could improve if a better confidence estimator is used. In the previous experiments, we used a confidence table consisting of 1K, 4-bit resetting counters (e.g., the equivalent of 16 bits of history depth [11]).

Figure 11 shows the relative to perfect checkpointing average performance deterioration as the number of the confidence table entries is varied under Conf_I-F. We restrict our attention to processors with either four or 16 GCs (different curves). As expected, with a smaller number of checkpoints, performance is more sensitive to table size. Performance generally improves with large confidence tables. However, performance improvements are minor for tables with more than 4K entries for both the 512-entry and 1K-entry window processors. Obtaining a significant gain in performance requires a significantly larger table whose implementation and integration in the pipeline might hurt overall

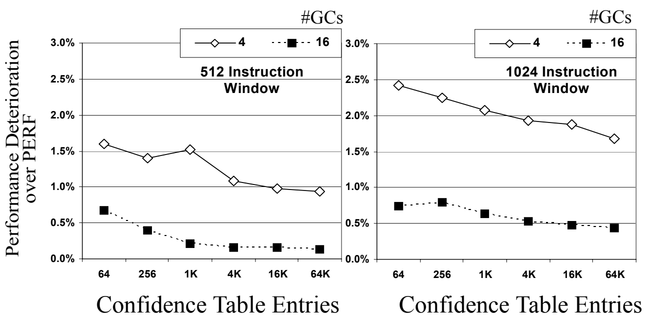

Figure 11: Average performance deterioration as a function of the confidence table number of entries (X-axis) and for different number of GCs (curves) with Conf_I-F and for 512-entry (left) and 1K-entry (right) window processors.

performance. In Section 5.5 we show that using BranchTap with a 1K-entry confidence table results in better average performance than using a 4K-entry confidence table alone. The latter confidence table requires an additional 12Kbits compared to the 1K-entry confidence table.

5.3 Where is Performance Lost

To motivate BranchTap we demonstrate first that with a checkpoint prediction mechanism in place most mispeculations occur on branches that have a GC but most of the performance loss is caused by branches that do not have a GC. In the experiments that follow and due to space limitations we focus on Conf_I-F, the best in-order checkpoint management policy we studied thus far and on a 1K-entry window processor with four checkpoints. Very similar results were obtained for a 512-entry window processor.

Figure 12 shows the percentage of mispredictions that lead to indirect recoveries (left bar), as well as the percentage of the total recovery cycles that are the result of an indirect recovery (right bar). An *indirect recovery* is a recovery on a branch that does not have a GC and hence uses the ROB and the nearest subsequent GC if any exists. Focusing on the left bar, we observe that for most benchmarks, most mispeculated branches do have a GC as the percentage of indirect recoveries is small. Focusing on the right bar, we observe that many and often most of the cycles lost on recovering from a mispeculation are caused by indirect recoveries. This result demonstrates that there is significant potential for reducing recovery cost and thus for improving performance with BranchTap.

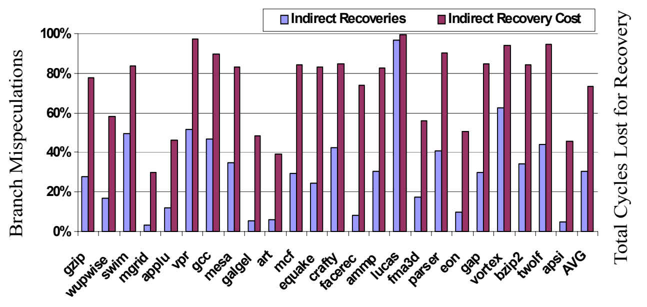

Figure 12: Per-benchmark and average percentage of indirect recoveries and percentage contribution of indirect recovery latency to total recovery cost for a 1K-entry window processor with 4 GCs.

5.4 Fixed-Threshold Speculation Control

In this section, we demonstrate that a fixed threshold speculation control policy is suboptimal across programs. To do so, we implemented a fixed threshold speculation control policy and ran each benchmark with all possible fixed threshold values (WT) from zero up to 16 in steps of two, for WT of 32, and finally with no threshold (i.e., 11 runs per benchmark). We refer to the method that uses no threshold as "full speculation" as it never stalls the fetch stage intentionally. All methods we studied up to this point used full speculation. The underlying full speculation method we used in these experiments is Conf_I-F. Speculation control is added on top of this method.

For each run we took measurements at intervals of one million committed, consecutive instructions (thus we obtained 1000 measurements per run of one billion instructions). At each measurement, we counted the number of cycles required to commit the corresponding one million instructions. Thus, each measurement if divided by 1M corresponds to the observed CPI for that 1M instruction interval. Comparing the corresponding CPI measurements across all 11 runs per benchmarks allows us to identify the best fixed threshold for that program interval. Figure 13 shows a subset of these measurements for *mcf* and *twolf,* and for a 1K-entry window processor with four GCs. Along the X-axis we report the number of committed instructions since the beginning of the run and along the Y-axis we report the number of cycles (in thousands) that were required to commit each 1M instruction sample. For clarity, we report only a small subset of the 1000 measurements per benchmark and for WT values of zero, four and full.

Twolf performs best with a threshold of four while *mcf* performs best with full speculation. The performance difference between full-speculation and restricted speculation with a threshold of four for *twolf* can be as high as 8% on certain samples. For *mcf,* full speculation can perform as much as 11% better than the next best policy shown. Other benchmarks have different optimal fixed thresholds, and some benchmarks exhibit intra-benchmark variation of their optimal threshold. This result demonstrates that a fixed threshold policy is suboptimal and that depending on the benchmark significant performance improvements may be possible if the threshold can be changed at runtime.

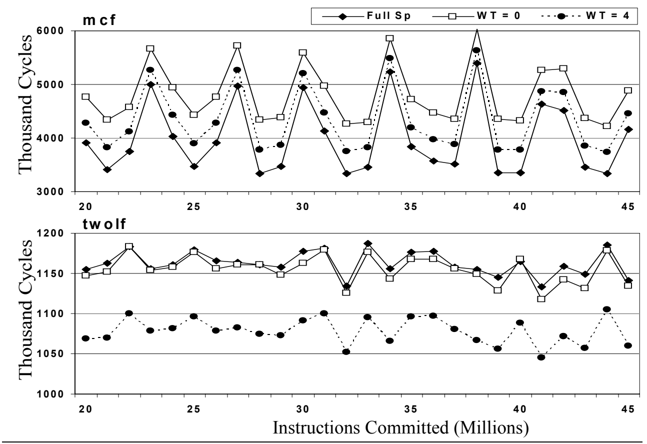

Figure 13: Number of cycles required to commit one million instructions under various fixed threshold speculation control policies for mcf and twolf. See text for an explanation.

5.5 BranchTap Performance

Figure 14 reports the per-benchmark performance with full speculation (Full Sp — which uses the Conf_I-F checkpoint management and recovery scheme described previously), fixed speculation control with a threshold of four (FixedSC-4 — a policy that was on average better than all other fixed-threshold policies) and BranchTap for processors with 512-entry and 1K-entry windows. The fixed threshold policy can be thought as representative of existing methods for power reduction through speculation control [9].

Focusing on the 1K-entry window results, all three policies perform well and close to each other for *wupwise, swim*, *mgrid*, *applu*, *galgel*, and *lucas*. This is because these benchmarks exhibit relatively accurate branch prediction or because they execute few branches. FixedSC-4 performs much better than full speculation for some (e.g., *gzip* and *parser*) but not all (e.g., *mcf*) benchmarks. In some cases FixedSC-4 deteriorates performance significantly compared to full speculation. Specifically, in *mcf* and *bzip2* FixedSC-4 results in slowdowns of 9.18% and 5.76% respectively, while full speculation results in slowdowns of only 1.56% and 2.77%. Overall, we observe that while FixedSC-4 offers relatively low performance loss on the average, its behavior varies significantly across benchmarks.

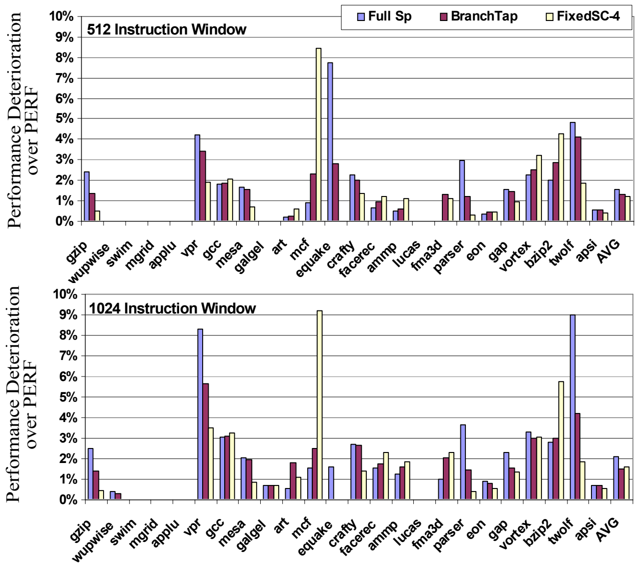

Figure 14: Per-benchmark performance deterioration with four GCs under: (1) full speculation, (2) BranchTap, and (3) fixed speculation control with a threshold of four.

Contrary to full speculation and FixedSC-4, performance with BranchTap varies less across programs. Specifically, for most programs, BranchTap improves performance over full speculation. For *equake*, performance with BranchTap exceeds that of PERF (not shown in the figure) by 4.43%. This could be caused by decreased contention in the memory system due to less aggressive speculation. BranchTap does not always improve performance over full speculation. This is because BranchTap has to dynamically discover that these benchmarks prefer full speculation. However, even in these cases, the performance deterioration is not as dramatic as it is with FixedSC-4 (e.g. *mcf* and *bzip2*). Overall, BranchTap either improves over full speculation or performs slightly worse. On average, BranchTap improves performance deterioration by 28.35% compared to full speculation. Average performance deterioration drops from 2.08% to 1.49% and worst case deterioration drops from 8.99% to 5.64% for the 1K-entry window processor. Similar observations apply to the 512-entry window processor where the absolute differences are smaller but the trends remain the same.

We also report performance when two, one, or no GCs are available in Figure 15. When no GCs are available, BranchTap simply keeps track of the number of unresolved branches in flight. Shown is the per-benchmark performance deterioration with full speculation (left bar), BranchTap, and FixedSC-4 (right bar) for a 1K-entry window processor. The same trends observed with a processor with four GCs are also observed here, but the absolute performance benefits become higher as less GCs are available.

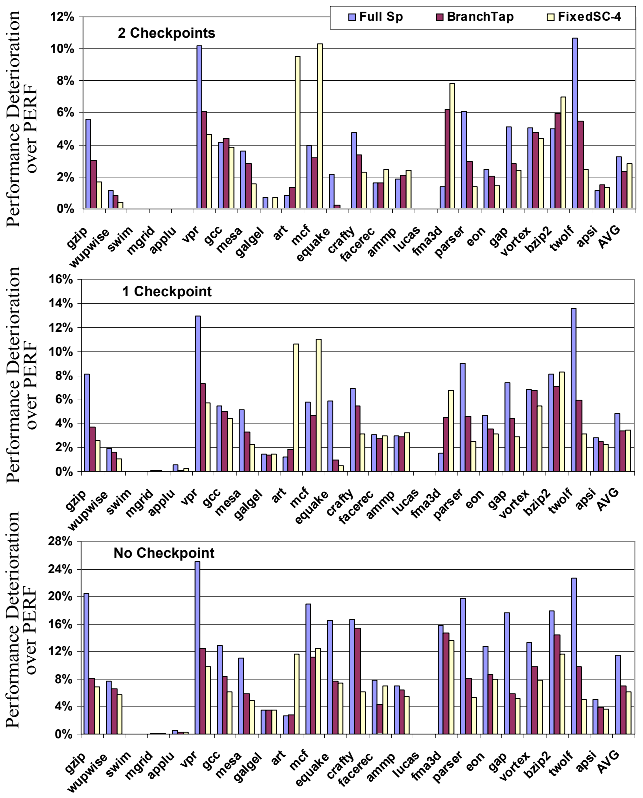

Figure 15: Per-benchmark performance deterioration with full speculation (left bar), BranchTap, and FixedSC-4 (right bar), with two, one, or no GCs and for a 1024-entry window processor.

Although processors with BranchTap and FixedSC-4 performed almost equally on the average, BranchTap was able to avoid the dramatic performance deterioration of FixedSC-4 observed in certain benchmarks.

We also studied BranchTap in combination with the *Anyweak* confidence estimator. Due to space limitations, we only show the results for a 1K-entry window processor with 4 GCs in Figure 16. While its overall performance loss is higher, BranchTap-A still improves performance over full speculation and offers less variability compared to FixedSC-4.

An added benefit of BranchTap is that, similar to previous work on speculation control, it reduces the work lost to squashed

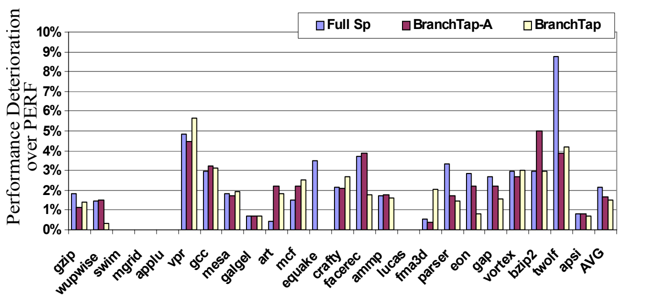

Figure 16: Per-benchmark performance deterioration with full speculation (left bar), BranchTap with the *Anyweak* estimator, and BranchTap with the explicit confidence estimator (right bar), for a 1K-entry window processor with four GCs.

instructions. We observed a 10% decrease in the number of fetched instructions on average for a 1K-entry window processor with four GCs. Thus we expect that BranchTap reduces overall power dissipation. Further investigation of this aspect is beyond the scope of this paper.

6 Conclusion

We have presented BranchTap, a technique that combines adaptive speculation control and checkpoint-prediction to improve performance when very few global checkpoints are available. Prediction-based methods improve performance by trying to locate those branches that are likely to cause a mispeculation and thus use a checkpoint in the future. However, we observed that this task becomes increasingly harder as the window increases and as the number of available GCs is kept low. We have shown that there is significant potential for improvement over checkpoint-prediction methods by trying to instead reduce the number of instructions that would have to be squashed. We have seen that for a 1K-entry window processor and with just four checkpoints BranchTap reduces average and worst case performance degradation by 28.3% and 37.2% respectively. Since BranchTap requires little additional resources and is relatively straightforward to implement, it is preferable over improving the accuracy of checkpoint-prediction by increasing the size of the underlying confidence estimator. We have observed that BranchTap is better than fixed threshold policies in that it offers lower variability in the performance loss Finally, we compared previous prediction-based checkpoint allocation methods and investigated several design choices.

Acknowledgments

The authors would like to thank Ioana Burcea, Jason Zebchuk, and the anonymous reviewers for their valuable comments. This research was supported by a CFI equipment grant, an Intel equipment donation, funds from the University of Toronto and an NSERC Discovery Grant.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan, *An Analysis of Resource Efficient Checkpoint Architecture,* ACM Transactions on Architecture and Code Optimization (TACO) Volume 1, Issue 4, Dec. 2004.

[2] H. Akkary, R. Rajwar, and S. Srinivasan, *Checkpoint Processing and Recovery: Towards Scalable Instruction Window Processors*, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Nov. 2003.

[3] D. I. August, W. Hwu, and S. Mahkle, *A Framework for Balancing Control Flow and Predication*, Proceedings of the 30th International Symposium on Microarchitecture. Nov. 1997.

[4] A. Cristal, D. Ortega, J. Llosa and M. Valero. *Kilo-Instruction Processors.* Proceedings The 5th International Symposium on High Performance Computing (ISHPC-V), Oct., 2003.

[5] C. Y. Cher and T. N. Vijaykumar, *Skipper: A Microarchitecture for Exploiting Control-Flow Independence*, Proceedings of the 34th International Symposium on Microarchitecture, Nov. 2001.

[6] Y. Chou, J. Fung, and J. P. Shen, *Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection*, Proceedings of the 13th International Conference on Supercomputing, June 1999.

[7] D. Burger and T. Austin. *The Simplescalar Tool Set v2.0*, Technical Report UW-CS-97-1342. Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[8] A. Gandhi, H. Akkary, and S. T. Srinivasan, *Reducing Branch Misprediction Penalty via Selective Branch Recovery*, Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04), Feb. 2004.

[9] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. *Confidence Estimation for Speculation Control*, in Proceedings of the 25th Annual International Symposium on Computer Architecture, June 1998.

[10] W. W. Hwu and Y. N. Patt, *Checkpoint Repair for Out-of-Order Execution Machines*, Proceedings of the 14th Annual Symposium on Computer Architecture, June 1987.

[11] E. Jacobsen, E. Rotenberg, and J. E. Smith, *Assigning Confidence to Conditional Branch Predictions*, Proceedings of the 29th Annual International Symposium on Microarchitecture, Dec. 1996.

[12] D. A. Jimenez and C. Lin, *Composite Confidence Estimators for Enhanced Speculation Control*, Technical Report TR-02-14, Department of Computer Sciences, The University of Texas at Austin, Jan. 2002.

[13] S. Manne, A. Klauser, and D. Grunwald, *Pipeline Gating: Speculation Control for Energy Reduction*, Proceedings of the 25th Annual International Symposium on Computer Architecture, June 1998.

[14] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic and J. Torrellas, *Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors*, Proceedings of the 35th International Symposium on Microarchitecture, Nov. 2002.

[15] A. Moshovos and G. Sohi, *Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling*, Proceedings of the IEEE, 89(11):1560-1575, Jan. 2001.

[16] A. Moshovos, *Checkpointing Alternatives for High Performance, Power-Aware Processors*, Proceedings of the IEEE International Symposium Low Power Electronic Devices and Design (ISLPED), Aug. 2003.

[17] S. Palacharla, N. P. Jouppi, and J. E. Smith. *Complexity-effective Superscalar Processors*, Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[18] E. Rotenberg, Q. Jacobsen, and J. E. Smith, *A Study of Control Independence in Superscalar Processors*, Proceedings of the 5th International Symposium on High Performance Computer Architecture, Feb. 1999.

[19] J. Smith and A. Pleszkun. *Implementing Precise Interrupts in Pipelined Processors*, IEEE Transactions on Computers, 37(5), May 1988.

[20] J. E. Smith and G. Sohi, *The Microarchitecture of Superscalar Processors*, Proceedings of the IEEE, vol. 83, Dec. 1995.

[21] A. Sodani and G. S. Sohi, *Dynamic Instruction Reuse*, Proceedings of the 24th Annual International Symposium on Computer Architecture, June 1997.

[22] G. S. Sohi. *Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers*. IEEE Transactions on Computers, March 1990.

[23] G. Tyson, K. Lick, and M. Farrens, *Limited Dual Path Execution*, CSE-TR 346-97, University of Michigan, 1997.

[24] K. C. Yeager, *The MIPS R10000 Superscalar Microprocessor*, IEEE MICRO, 1996.

[25] P. Zhou, S. Onder, S. Carr, *Fast Branch Misprediction Recovery in Out-of-Order Superscalar Processors,* Proceedings of the International Conference on Supercomputing, June 2005.