Breaking In-Order Branch Miss Recovery

Stijn Eyerman, Wim Heirman, Sam Van den Steen, Ibrahim Hur

Abstract—Despite very accurate branch predictors, branch misses remain an important source of performance limiters, especially for irregular applications. To ensure in-order commit, branch miss recovery is done in-order: all instructions after the oldest branch miss are flushed, even if they eventually reconverge with the correct path. We propose a technique to limit flushing to real wrong-path instructions only, allowing the resolution of newer branch misses while an older one is not yet resolved. Our technique involves minimal additions to a conventional out-of-order processor, by reusing existing checkpoint mechanisms and relying on programmer/compiler inserted hints to detect data and control independence. We evaluate the technique on graph benchmarks, resulting in up to 2× increase in performance.

Index Terms—Superscalar execution, branch prediction, speculative execution

1 Introduction

Branch mispredictions break the continuous flow of instructions through the pipeline by unconditionally flushing all instructions following the branch miss and restarting fetch at the correct branch target. The pipeline is unable to know which instructions will eventually end up on the correct path in case of branch reconvergence and whether these instructions have data dependences with the correct or false path, so conservatively flushing all instructions ensures correctness. This branch miss recovery approach also keeps the instructions in the reorder buffer in sequential program order, which is important to enable precise and immediate interrupts through in-order commit. However, it wastes compute time and resources, because branch code paths reconverge often, and many flushed instructions turn out to be correctly fetched and executed. These instructions not only need to be fetched and decoded again, memory fetches need to be redone and branches are predicted again, while they could have already been resolved speculatively before the flush.

The existence of control and data independent instructions after a mispredicted branch has been recognized before, and a lot of prior work proposes mechanisms to exploit branch reconvergence to improve performance [8]. However, most of these proposals either add complex hardware to detect these instructions and to reuse them on branch recovery [2], which increases design complexity and cancels some of the energy saving benefits, or they only reuse a small part of the converging instructions [6]. Our proposal targets adding minimal extra hardware to a conventional out-of-order pipeline, while maximally reusing converged instructions.

This paper has the following contributions:

- We propose three novel instructions to hint the pipeline which instructions are control and data independent. We believe that in general, programmer/compiler – hardware cooperation will be crucial to improve energy efficiency.
- We show how existing out-of-order mechanisms, such as renaming and checkpoints, can be used to implement reuse of converged instructions. The only additions are a linked list reorder buffer and a fetch redirect queue.
- We estimate the performance impact of our technique on graph benchmarks, resulting in 1.3× to 2× performance increase for branch intensive applications, and no performance loss for applications with low branch miss rates.

• *All authors are with Intel Corporation, {stijn.eyerman, wim.heirman, sam.van.den.steen, ibrahim.hur}@intel.com*

2 Related Work

Branch convergence and selective flushing have been recognized as potential performance optimizations for out-of-order pipelines more than 20 years ago [8]. More recently, Al-Zawawi et al. [2] propose a novel ROB-less design, based on re-execution buffers to execute control and data dependent instruction after a mispredicted branch. Naresh et al. [6] propose to only reuse convergent instructions in the frontend pipeline, flushing all dispatched instructions. These two proposals are examples of two extreme approaches: the first advocates a dramatic redesign of the processor pipeline to extract the largest performance benefit, while the latter has a very small impact on the pipeline design, but only reuses a small fraction of the convergent instructions. Our proposal lies somewhere in the middle: minimal changes to the architecture by using hints in the code, and maximal reuse of convergent instructions.

Alternatively, multithreading can be used to spawn threads for control and data independent regions [1], which avoids flushing instructions of other regions on a misprediction. Creating and spawning threads has a high overhead, especially when the independent regions are small. Furthermore, optimized parallel code already uses the available hardware thread contexts.

Malik et al. [7] discuss the performance benefit of parallel branch resolution, highlighting the importance of maximizing branch level parallelism (BLP) in control independent architectures. Our proposal supports BLP: branches in different slices execute concurrently and are not flushed due to misses in other slices.

3 Selective Flushing Mechanism

For developing the selective flush mechanism, we put forward two guidelines: (1) Minimize hardware additions by reusing existing mechanisms and relying on programmer/compiler support, and (2) retain observable pipeline behavior, in particular in-order commit, to not jeopardize the compute ecosystem outside the core.

We split the discussion of our mechanism in two parts: detecting control and data independent instructions, and the branch miss recovery mechanism to reuse these instructions. Next, we discuss further details of our proposal as well as an estimation of the performance impact.

3.1 Detecting Convergent Instructions

Instead of spending hardware (and thus chip area and energy) to automatically detect branch miss control and data independent (CDI) instructions, we propose to use code hints to the processor pipeline.

Listing 1. Example in pseudo assembler. `A`, `B` are lists of instructions, `c` is a condition, `i` is the iterator and `N` the iteration count.

```
loop:   slice_start
        A
        brc c, end
        B
end:    slice_end
        inc i
        brl i, N, loop
        slice_fence
```

Instructions inserted by the programmer or compiler delineate CDI regions, called slices. Compiler inserted instructions may limit the general applicability of the technique, but it minimizes extra hardware and energy consumption, and the compiler (in cooperation with the programmer) has a much more global view on the program. Adding new instructions has a non-negligible impact on the architecture, however, only three instructions with no operands are needed, whose coding can be selected from the wide range of no-op instructions in most instruction sets.

We define a *slice* as a sequence of instructions in a program such that all slices within a region (delineated by a slice fence) are CDI with respect to each other. Slices do not need to be consecutive to each other, instructions outside a slice can occur between two slices. These intermediate instructions are CDI of the instructions in the slices, but the instructions in the slices can depend on the instructions outside the slices.

To illustrate these definitions, assume a loop with independent iterations, containing a conditional branch. Listing 1 shows pseudo assembler code, where `A` and `B` are a list of instructions, and `c` is a condition. The code already contains the three novel instructions: `slice_start`, `slice_end` and `slice_fence`. The slices consist of code blocks A and B and the branch, for each iteration. The increment and loop branch are outside of slices, but within the slice region. The region is ended after the loop with a `slice_fence`.

The slices are independent of each other, so a misprediction of the branch in the slice has no impact on other iterations. This means that only the instructions until the next `slice_end` need to be flushed. The slices do depend on the code outside the slice, i.e., they depend on the value of `i` and they are control dependent on the loop branch. However, the increment and branch do not depend on the code within the slices, so they can execute out of order with the slices, but need to be executed in order w.r.t. themselves. On a misprediction of the loop branch (e.g., after the last iteration), all instructions fetched after the branch need to be flushed, whether in a slice or not, because no new iterations are needed. The `slice_fence` indicates that instructions following the fence might depend on the data generated in the slices, so all branch misses within the slices should be recovered before executing instructions after the fence. Additionally, we require that dependences between slices and the code after a `slice_fence` are through the memory and not through registers. This ensures that register renaming has impact only within a slice, simplifying the rename table recovery.

These instructions are inserted by the compiler, which usually has a global view on data and control dependences between instructions, or by the programmer. Performance aware programmers already need to think about parallelism to exploit the compute power of multi-core processors. For example, a popular parallelization framework is OpenMP. A `parallel for` loop means that iterations can be spread over multiple threads and are therefore inherently independent. However, it is often inefficient to schedule each iteration to a different thread, so chunks of multiple iterations are assigned to each thread. The iterations within a chunk are also independent, and can be sliced by inserting the slice instructions. Furthermore, because iterations can be spread across threads, there will be no register dependences between the iterations and the code following the parallel for loop since registers are not shared between threads.

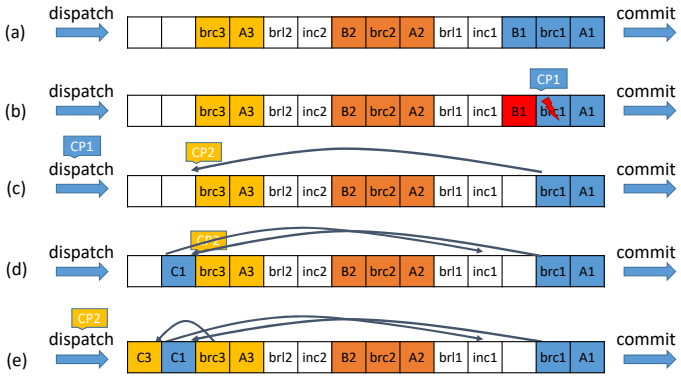

Fig. 1. Illustration of the recovery mechanism and linked ROB. Instructions in slices are colored, with a different color for each slice. `brc1` is mispredicted.

Special care must be taken for reduction variables, which are explicitly mentioned in the `parallel for` construct. When implemented through a register update instead of an atomic memory operation for performance reasons, they cause a loop carried dependence. We will discuss how to handle reduction variables in Section 3.5.

A `parallel for` is not the only construct where slices can be used. Any code block with a conditional branch of which the next few code blocks are independent (delimited by a slice_fence) can be sliced, avoiding flushing the independent code blocks on a branch miss in the slice.

3.2 Selective Flush and Reuse

The three slice instructions define which instructions should be flushed on a branch miss. When within a slice, only the instructions until the next slice_end should be flushed and refetched. When outside a slice, we should flush all instructions after the branch miss, as in normal execution. Through the renaming scheme in out-of-order processors, independent iterations use separate sets of physical registers, so all non-flushed instructions can continue executing while fetch is redirected.

Selectively flushing and refetching instructions breaks the program order of the ROB entries: older (refetched) instructions can appear after newer (non-flushed) instructions, and flushed instructions can leave holes in the ROB. To ensure in-order commit, we propose to implement the ROB as a linked list [8], where the next instruction is indicated using a pointer, rather than the next in line. A linked list ROB is used in the IBM POWER [9] to enable sharing the ROB across SMT threads. Similar to POWER, the pointer overhead can be reduced by dividing the ROB into blocks of a few consecutive instructions, and use pointers between blocks.

Figure 1 illustrates the mechanism for the example in Listing 1, assuming `A` and `B` consist of one instruction and leaving out the slice instructions for clarity. In fact, slice instructions could be left out of the ROB by adding a single bit to each instruction whether or not it is in a slice. The numbers indicate the different iterations. To increase generality, assume a third code block `C` in the loop body that is executed when `B` is not and vice versa (an if-then-else construct).

Initially, instructions are dispatched in program order (Fig. 1a). When the misprediction of branch `brc1` is detected, instructions of that slice (instruction `B1`) are flushed (Fig. 1b). The next ROB entry pointer after the branch is set to the next free entry (Fig. 1c). Meanwhile, a rename table checkpoint (CP2) is taken at the newest instruction. Conventional branch recovery uses checkpoints of the rename table at the mispredicted branch to rewind the execution. We

use the same mechanism to also checkpoint the rename table at the most recent non-flushed instruction.

The branch checkpoint (CP1) is used to redirect fetch to the correct path (Fig. 1d), and when that slice is ended, the ROB next pointer points back to the first non-flushed instruction after the mispredicted branch (`inc1`). Next, we start fetching again from CP2, after letting the ROB entry of the newest instruction (`brc3`) point to the next free ROB entry (Fig. 1e). Because we require that there are no register dependences across slices and the rest of code, renamings done in the correct path after CP1 do not have an impact on CP2. Note that in this case, instruction `C1` could be put in the flushed `B1` slot, avoiding pointer redirects. In general, we cannot assume that the correct path is as long as or shorter than the wrong path, so pointer redirects are required to implement this scheme.

Concerning the load/store queues, we assume that ROB entries containing memory operations have a pointer to the corresponding load/store queue entries, ensuring that these are also committed/retired in program order. Because we ensure that there are no memory dependences between slices, memory address aliasing checks and store-to-load forwarding can be done on the out-of-order sequence of memory operations in the load/store queues, avoiding a linked list load/store queue implementation. Note that memory operations within a slice will always appear in order (potentially with other non-dependent operations in between).

3.3 Concurrent Branch Misses

While recovering from a branch miss in our selective flush mechanism, other branch misses in newer instructions might be detected. When we are still recovering from an older branch miss, it would harm performance to interrupt the old branch miss recovery to recover from the newer branch miss, and it would also complicate the implementation to restart the older recovery afterwards. Therefore, we postpone the recovery from the newer miss until the recovery of the older miss is done.

To implement this behavior, we propose to add a *fetch redirect queue* (FRQ) to the frontend of the pipeline. This FIFO queue holds all pending fetch redirects due to branch misses. When a branch miss within a slice is detected, instructions are selectively flushed, and redirect data is pushed in the FRQ. This data includes the correct path instruction address, a pointer to the checkpoint after the mispredicted branch and a pointer to the ROB entry of the mispredicted branch. At the fetch stage, the FRQ is checked when we are in normal fetch mode (i.e., not recovering from a branch miss) or when the previous branch miss has been fully recovered (i.e., a slice_end instruction is encountered). If there is a redirect entry at the head of the FRQ, fetch is redirected by restoring the checkpoint, setting the instruction pointer to the correct path address, and setting the next pointer of the ROB entry of the branch to the next free ROB entry. If we were in normal fetch mode, additionally a checkpoint of the current state is saved in a separate place (e.g., CP2 in Fig. 1, called the 'normal fetch' checkpoint hereafter). As long as there are entries in the FRQ, we continue to recover from the pending misses after recovering from the previous miss. When the FRQ is empty, normal fetch mode is restored by redirecting fetch to the saved 'normal fetch' checkpoint.

It can occur that a branch miss in a slice is flushed itself, because an older non-slice branch is detected as a misprediction. In that case, flushed ROB entries are checked whether they contain a sliced branch miss, and the corresponding entries in the FRQ are removed. These entries are detected by comparing the ROB pointers in the FRQ. The FRQ is used only for branch misses within a slice. If a branch miss outside a slice is detected, the 'normal fetch' checkpoint is replaced by the correct path target. If the FRQ still contains entries (after removing the flushed branch misses), these are recovered first before fetching the correct path.

3.4 Slice Fence

A slice_fence instruction indicates that instructions following it might depend on the data produced in the slices preceding it. This means that all slice branch misses need to be recovered before starting to execute instructions after the fence. By prioritizing FRQ entries upon fetch, we ensure that all detected misses are resolved first. However, undetected branch misses may still reside in the ROB. Therefore, instructions after the fence should not be executed until all branches in slices are resolved.

Instead of stalling the pipeline until all sliced branches are resolved, which would incur a performance penalty, instructions after the fence can also be executed speculatively. Thereto, we store a checkpoint at the fence instruction. When a branch miss in a preceding slice is detected, the instructions within the slice are flushed, as well as all instructions after the fence. The 'normal fetch' checkpoint is replaced by the checkpoint at the fence, such that all instructions after the fence are refetched after the branch miss is recovered.

3.5 Reduction Variables

Reduction variables within parallel for loops have the following properties: (a) they can be (atomically) updated in any order and (b) none of the instructions in the loop body depend on them. They are usually implemented as thread local register updates for the iterations on one thread, after which they are reduced globally across all threads. This implementation will break when using slices and selective flushing: when restoring the 'normal fetch' checkpoint, the old value of the reduction variable is restored and the updates (or rollbacked updates) in the recovered branch path will be lost.

Storing these variables in memory and doing atomic updates through memory solves this issue, but harms performance. Instead, we propose to postpone their execution until commit. Committed instructions are never flushed, so the final value will be correct. To implement this, we propose to add an 'execute-at-commit' flag to common reduce instructions (add, increment, multiply). Because no other instructions depend on the reduction variable, delaying their update until commit will not delay the execution of the other instructions.

3.6 Performance Impact

To reason about the performance benefit of selective flushing, we use interval analysis [5], and count penalties at the dispatch stage. On a branch miss, wrong-path instructions are dispatched until the next slice_end instruction. Thereafter, correct path instructions outside the slice and from the next slices are dispatched. When the branch miss is detected, dispatch is not stalled, because the correct path instructions are fetched back-to-back to the 'normal fetch' instructions. So the penalty equals the time between dispatching the branch miss and the next slice end instruction. The branch penalty of the conventional mechanism equals the full branch resolution time (executing the dependence path to the branch) and frontend pipeline refill time [5], which is considerably longer. Furthermore, branch misses will now also be resolved in parallel, further reducing the penalty.

The performance benefit can be reduced when there are not enough free ROB entries to hold the correct path after flushing the wrong-path instructions in a slice. In that case, we need to wait until the first correct path instructions commit and free their entries to continue dispatching the next correct path instructions. To avoid this performance hit when the ROB is almost full, the processor can also flush instructions newer than a recent checkpoint, and use that checkpoint as the next 'normal fetch' point instead of taking a checkpoint at the newest instruction. This leaves more space for the correct path, while never performing worse than the conventional flush mechanism, which flushes all instructions.

Listing 2. BFS backward update loop.

```
1  #pragma omp parallel for reduction(+:awake_count)
       schedule(dynamic, 1024)
2  for (NodeID u=0; u < g.num_nodes(); u++) {
3    if (parent[u] < 0) {
4      for (NodeID v : g.in_neigh(u)) {
5        if (front.get_bit(v)) {
6          parent[u] = v;
7          awake_count++;
8          next.set_bit(u);
9          break;
10       }
11     }
12   }
13 }
```

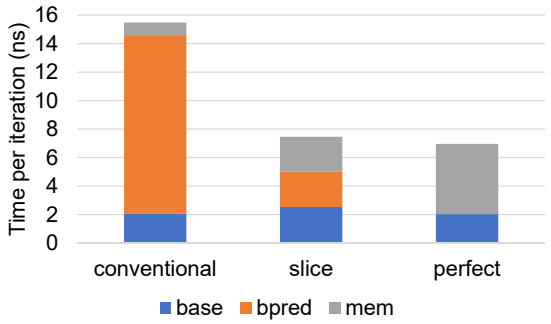

Fig. 2. Time per BFS backward update iteration for a conventional pipeline, our slice mechanism and perfect branch prediction, divided into base (instruction execution), branch miss stalls and memory stalls.

4 Performance Evaluation

We evaluate our proposal first on the backward update loop of breadth-first graph search (BFS), see Listing 2 (taken from the GAP benchmark suite [3]). The parallel for loop indicates independent loop iterations, with one reduction variable. The loop body contains 3 hard-to-predict data dependent branches (lines 3 and 5, and the loop condition on line 4 that depends on the variable in-degree), leading to a large penalty due to branch misses: the CPI profile generated by our simulator indicates that 81% of the cycles is spent resolving and recovering from branch misses. Note that the inner for loop (line 4) cannot be sliced: it is exited as soon as a match is found, so the iterations are control dependent. So we add a slice_start between lines 2 and 3, and a slice_end between lines 12 and 13.

To evaluate performance, we use an in-house simulator based on Sniper [4], configured as a 4-wide out-of-order processor pipeline, with state-of-the-art TAGE branch predictor. The slice instructions are implemented as Sniper 'magic' instructions that are captured by the timing model. When a branch miss occurs within a slice, only the instructions up to the next slice_end are marked as wrong path instructions and are eventually flushed.

Fig. 2 shows the time per iteration of one thread of the BFS backward update loop with the conventional pipeline, our selective flush technique and oracle branch prediction. Perfect branch prediction has 2.2× lower execution time, and is now bottlenecked by the memory bandwidth. Our technique improves performance 2.1×, reducing the gap with oracle branch prediction to 7%.

The division of the execution time shows that the conventional execution is limited by branch misses. Our mechanism still has a considerable branch miss component, but it is reduced a lot because 67% fewer instructions are flushed and branch misses are resolved in parallel (the number of branch misses remains the same). The reduction in branch miss stalls decreases the time between memory operations, increasing bandwidth usage and thus memory stalls. Bandwidth usage becomes close to the available bandwidth, which explains why further reducing branch miss penalty (perfect branch prediction) does not improve performance. The base component for the slice mechanism is slightly higher, because of the extra slice instructions.

TABLE 1
Performance increase of selective flush and perfect branch prediction versus a conventional processor core for all GAP benchmarks.

	bc	bfs	cc	pr	sssp	tc
Slice	1.30×	1.74×	1.71×	1.00×	1.34×	1.30×
Perfect	1.41×	1.80×	1.82×	1.03×	1.34×	1.74×

Next, we evaluate our mechanism on all GAP benchmarks, where we add slice instructions at the beginning and end of the main parallel for loops. Table 1 shows for the six benchmarks the performance gain of our mechanism versus a conventional core, as well as the impact of perfect branch prediction. All benchmarks but one (pr) result in substantial performance improvements, close to perfect branch prediction. Pagerank (pr) has a low branch miss rate, limiting the potential gain. Triangle count (tc) has many branch misses, but they resolve quickly, limiting the net gain of selective flush.

5 Conclusions and Future Work

Branch mispredictions remain an important source of performance loss. Efficiency gains can be obtained by not flushing and continuing to execute control and data independent converging instructions. We propose a mechanism to implement selective flushing, reusing most of the existing checkpoint infrastructure, while still ensuring in-order commit. Initial evaluations show up to 2× performance benefit for applications with high branch penalties. As future work, we will investigate multiple variations of selective flushing (fewer checkpoints, freeing more ROB entries for the correct path) and quantify the performance impact.

References

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank, "Exploiting postdominance for speculative parallelization," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb 2007, pp. 295–305.

[2] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary, "Transparent control independence (TCI)," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, 2007, pp. 448–459.

[3] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[4] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[5] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009.

[6] V. R. Kothinti Naresh, R. Sheikh, A. Perais, and H. W. Cain, "SPF: Selective pipeline flush," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 152–155.

[7] K. Malik, M. Agarwal, S. S. Stone, K. M. Woley, and M. I. Frank, "Branch-mispredict level parallelism (BLP) for control independence," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb 2008, pp. 62–73.

[8] E. Rotenberg and J. Smith, "Control independence in trace processors," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, Nov 1999, pp. 4–15.

[9] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, Mar 2017.