Enabling Branch-Mispredict Level Parallelism by Selectively Flushing Instructions

Stijn Eyerman
Wim Heirman
Sam Van den Steen
stijn.eyerman@intel.com
wim.heirman@intel.com
sam.van.den.steen@intel.com
Intel Corporation
Belgium

Ibrahim Hur
ibrahim.hur@intel.com
Intel Corporation
USA

ABSTRACT

Conventionally, branch mispredictions are resolved by flushing wrongly speculated instructions from the reorder buffer and refetching instructions along the correct path. However, a large part of the misspeculated instructions could have reconverged with the correct path and executed correctly. Yet, they are flushed to ensure in-order commit. This inefficiency has been recognized in prior work, which proposes either complex additions to a core to reuse the correctly executed instructions, or less intrusive solutions that only reuse part of the converged instructions.

We propose a hardware-software cooperative mechanism to recover correctly executed instructions, avoiding the need to refetch and re-execute them. It combines relatively limited additions to the core architecture with a high reuse of reconverged instructions. Adding the software hints to enable our mechanism is a similar effort as parallelizing an application, which is already necessary to extract high performance from current multicore processors. We evaluate the technique on emerging graph applications and sorting, applications that are known to perform poorly on conventional CPUs, and report an average 29% increase in performance.

ACM Reference Format:
Stijn Eyerman, Wim Heirman, Sam Van den Steen, and Ibrahim Hur. 2021. Enabling Branch-Mispredict Level Parallelism by Selectively Flushing Instructions. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480045

1 INTRODUCTION

Processor development has historically been driven by technology improvements, providing more transistors and frequency boosts without increasing energy density [13, 31]. Architecture design mainly had to focus on extracting as much performance as possible out of the ever growing amount of transistors, with little focus on efficiency. This has led to wide and deep out-of-order architectures, relying on prediction and speculation to increase performance, even though mispredictions lead to a high amount of uselessly fetched and executed instructions. Nowadays, the halt of frequency and power efficiency scaling puts an increasing focus on efficiency, with the advent of specialized architectures and accelerators as the extreme examples [21].

Branch prediction is an important contributor to the performance of out-of-order processors: if the prediction is correct, the core does not have to stall fetch until a branch is executed. However, branch predictor misses have an increasingly higher power and performance penalty as the core's instruction window gets larger to exploit more parallelism: the amount of wrong-path instructions that need to be flushed increases with window size. Furthermore, CPU applications are becoming more irregular, both in terms of memory behavior and branch predictability. This is because regular code is now increasingly executed on architectures with a higher peak compute rate, such as GPUs, leaving the less regular code for the CPU [5]. Additionally, emerging big data analysis algorithms, such as graph analysis [17], sparse neural networks [44] and graph neural networks [42], operate on sparse and irregular data, resulting in high branch miss penalties.

In this paper, we propose a mechanism to increase the efficiency and performance of a conventional out-of-order processor, by not flushing all instructions after a mispredicted branch. The mechanism detects instructions in the speculative stream that reconverge to the correct path and only flushes those instructions that need to be refetched and re-executed to ensure correct execution. This increases performance, as the non-flushed instruction do not need to be refetched and re-executed, and also saves energy for the same reason.

Not flushing data and control independent instructions to increase efficiency has been extensively explored in prior work, as discussed in the next section. Our proposal differs from these proposals by its software-hardware cooperation: software indicates independent code fragments, while hardware uses this information to only flush instructions that truly depend on the branch miss. The goal is to maximally exploit instruction reuse while minimizing the additions to the core micro-architecture.

The novel instructions to indicate independent code fragments might seem to limit usefulness, as the application needs to be annotated and recompiled to use the mechanism. However, it is our

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480045

opinion that software-hardware cooperation is key in increasing processing efficiency, for two reasons:

(i) A hardware only solution often involves detection and/or prediction structures, setting off part of the efficiency gains of the proposed mechanism. By using software to indicate independent code fragments, no complex detection, prediction and/or rollback hardware needs to be implemented.

(ii) Programmers and compilers are already forced to cooperate with hardware to extract the highest possible performance, for example by implementing parallel applications to exploit the compute power of multicore machines. Parallelizing an application includes thinking about synchronization, which is a form of dependence analysis.

Regarding the second point, we show how independent loop iterations can be easily detected using OpenMP "parallel for" pragmas, a commonly used parallelization technique.

In this paper, we make the following contributions:

- We propose a hardware-software cooperative technique to increase performance and efficiency with minimal changes to the hardware and software.
- We describe different implementation options that provide different performance/complexity tradeoffs.
- We evaluate our proposal using simulation on graph benchmarks, showing an average 29% and up to 54% performance increase.

We first discuss prior work and refresh memory on the branch recovery mechanism in conventional out-of-order processors. Next, we elaborate on the software and hardware aspects of our proposal. We then explain our experimental setup and show results, analyzing the performance impact of different software and hardware options. Finally, we conclude and discuss future work.

2 PRIOR WORK ON BRANCH MISS OPTIMIZATIONS

Branch convergence and selective flushing have been recognized as potential performance optimizations for out-of-order pipelines more than 20 years ago by Rotenberg and Smith [35]. They implement a selective flush mechanism in a trace processor, an architecture that is not commonly used nowadays.

Since then, a lot of proposals have been made to exploit branch convergence to improve performance. We can roughly divide them into two categories: the first advocates a dramatic redesign of the processor pipeline to extract the largest performance benefit, while the second category proposals have a very small impact on the pipeline design, but only reuse a small fraction of the convergent instructions.

Skipper [11] belongs to the first category: it skips instructions that are control and data dependent on a hard-to-predict branch, and fetches and executes these instructions after the branch is resolved. It adds multiple predictors and roll-back mechanisms to the core for a relatively low (10%) performance gain. Furthermore, correctly predicted hard branches (50% probability) also cause out-of-order fetches, potentially delaying forward progress. Collins et al. [12] propose a more accurate reconvergence predictor, which can be used in control-independent architectures. Pajuelo et al. [34] use convergence detection and a vectorization scheme to issue multiple iterations of control independent code while a hard-to-predict branch is being resolved. Al-Zawawi et al. [4] propose a novel ROB-less design, based on re-execution buffers to execute control and data dependent instruction after a mispredicted branch. Mao et al. [30] describe a mechanism to implement the reuse of converging instructions in composable multi-processors.

An example of the second category (small additions, limited benefit) is the proposal by Roth and Sohi [36]. They keep outcomes of squashed instructions and reuse the values if the inputs are still valid. This has a limited impact on performance, as reconvergent instruction still need to be re-fetched and dispatched. Selective Branch Recovery (SBR) [18] can be applied when the recovergence point is the start of the correct path. By transforming wrong-path instructions into move instructions and re-executing them, data dependences are preserved. SBR only works for a subset of branch misses, i.e., those for which the start of the correct path is already fetched, while our mechanism can handle all branch misses. Naresh et al. [27] propose to only reuse convergent instructions in the frontend pipeline, flushing all dispatched instructions.

Our proposal lies somewhere in the middle between these two categories: we target minimal changes to the architecture, and maximal reuse of convergent instructions. We avoid expensive predictors, dependence checkers and roll-back mechanisms by relying on compiler hints to denote control and data independent regions. NOREBA [19] uses a similar strategy: compiler inserted instructions inform the hardware which branch independent instructions can commit out of order. Furthermore, we reuse the existing out-of-order mechanisms for register renaming, flushing instructions and redirecting fetch. Our mechanism does involve some non-trivial changes to the micro-architecture, but our goal is to keep these additions as limited as possible.

As an alternative to reusing converging instructions within a thread, multithreading can be used to spawn threads for control and data independent regions [1, 2], 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 all available hardware thread contexts.

Another way of reducing the performance degradation due to hard-to-predict branches is predication [10, 23]. Branch paths following branches that miss often are turned into predicated instructions, fetching both the taken and not taken path. Depending on the branch condition, the results of these instructions are committed or invalidated. Instructions after the convergence point are not predicated and can be executed while the branch resolves. No instructions need to be flushed. The performance benefit of predication is a subtle balance between adding data dependences, fetching more instructions and avoiding flushing instructions. Therefore, a performance monitoring system is needed to avoid performance inversion of predicating instructions [10, 24]. Additionally, predication options are limited in current architectures (x86 only has `cmov`, 64-bit ARM instruction set removed most of the predicated instructions), and the compiler has strict conditions for applying if-conversion, such as no function calls and a limit on the number of instructions. As a result, none of the applications we evaluate can use predication to reduce the branch penalty. Our technique does not add dependences, has no restrictions on the type and number

of instructions, and is only triggered when the branch is effectively mispredicted, while predication is done for all occurrences of the converted branch. Interestingly, Chauhan et al.[10] report that 72% of the hard branches converge for the benchmarks they evaluate, which is a distinct set from the benchmarks we used. This shows that convergence can be exploited in a large set of applications.

Farzad et al. [37] found that wrong-path control and data independent code accounts for 6% to 12% of the total power consumption in embedded processors, which could potentially be saved by not flushing them. Malik et al. [29] discuss the performance benefit of parallel branch resolution, highlighting the importance of maximizing branch-miss level parallelism (BLP) in control independent architectures. Our proposal implements BLP: branches in different slices execute concurrently and are not flushed due to misses in other slices.

Branch resolution is often delayed by long-latency cache misses on which the branch depends, as is the case for our evaluated applications. The indirect memory prefetcher [43] reduces the latency of irregular indirect memory accesses, speeding up branch resolution and thus indirectly reducing the branch penalty. Pipette [32] exploits pipeline parallelism in irregular applications and adds a mechanism for quickly switching between stages when a stage is blocked due to long memory accesses. These techniques are orthogonal to our proposal.

3 OUT-OF-ORDER EXECUTION AND BRANCH PREDICTION

Ultimately, the goal of out-of-order processing is to approach data flow execution: instructions are executed as soon as their inputs are ready, independent of the status of instructions that appear earlier in the instruction stream. However, to enable precise interrupts and to ease the implementation of synchronization between threads, instructions appear to be executed in order, by ensuring in-order commit.

Branch prediction increases the continuous flow of instructions into the reorder buffer, to provide a large window of instructions to select multiple ready-to-execute instructions. Branch mispredictions are, however, very costly: because we need to ensure in-order commit, all instructions after a detected branch miss need to be flushed and re-fetched along the correct path. The total penalty of a single branch misprediction equals the time spent fetching instructions along the wrong path, which could be many in a deeply pipelined processor [16]. For the branch miss heavy applications evaluated in our study (see Section 5.1) using a state-of-the-art TAGE branch predictor [38], wrong path instructions account for on average 53% more dispatched instructions, branch miss resolution takes 47% of the execution time and oracle branch prediction improves performance by 60%.

Many branches have convergent paths: the taken and not-taken path eventually converge. In this case, flushing all instructions after a branch miss breaks the data flow execution model: the execution of converging instructions that are independent of the branch outcome is cancelled or delayed because of the in-order commit model, not because of true dependences. Our technique improves on this by keeping convergent instructions that are known to be independent of the branch paths in the reorder buffer and by continuing to execute them while the correct path is being fetched. Using our proposal, the penalty of a branch miss reduces to the fetch time of only those instructions that are control and data dependent on the mispredicted branch. Furthermore, independent instructions do not need to be refetched, resulting in less energy consumption.

For a better understanding of our proposal, we briefly recapture the conventional branch miss resolution mechanism in an out-of-order processor. If a branch is executed and it turns out to be mispredicted, all instructions that are younger are flushed from the ROB, and the issue, load and store queue entries they occupy are released, as well as the renamed physical registers. Before fetching and dispatching instructions along the correct path, the rename table has to be restored to its state just after the branch was dispatched, undoing all renamings along the mispredicted path. Conceptually, there are two main methods to restore the rename table [3]:

(a) *Using checkpoints*: At each branch instruction, the rename table is checkpointed. When a branch turns out mispredicted, the checkpoint is looked up and restored. This is a quick mechanism, but requires substantial storage to keep all checkpoints.

(b) *Using rollback logs*: The renaming operations (architectural to physical register mappings) of all instructions in the ROB are logged, and rolled back one by one until the mispredicted branch is reached. This mechanism has lower storage requirements, but more time is needed to restore the rename table, because rename operations need to be undone one by one.

The combination of both is currently the most chosen option to balance storage requirements and speed: regular checkpoints (but not at all branches) and a rollback log. When a branch is mispredicted, the closest checkpoint is looked up and rolled back to the mispredicted branch. In the remainder of this paper, we assume a checkpointing mechanism at each branch. However, our proposal also supports a rollback or hybrid mechanism, because a rollback mechanism ensures that rename tables at dispatch of all in-flight instructions can be restored.

4 SELECTIVE FLUSH MECHANISM

Our proposal to selectively flush instructions after a branch miss and to not waste resources on re-processing instructions on the convergent path consists of combined software and hardware additions. The software part is responsible for indicating control and data independent regions by means of 3 additional instructions. The hardware part takes this information to selectively flush instructions on a branch miss.

4.1 Denoting Independent Regions

Selective flush should only flush instructions that are dependent on the instructions before the branch miss. To simplify the mechanism, we only consider sequential lists of instructions, i.e., we do not extract individual (in)dependent instructions. We define a *slice* as a sequence of instructions such that all instructions following the slice are control and data independent[1] of the instructions in the slice, up to a certain point in the application, called the *slice*

[1]Considering only true RAW dependences, WAR and WAW dependences are removed by register renaming.

Listing 1: Example loop

```
1  loop:  slice_start
2         load A[r1],r2
3         jl r2,0,else
4         mul r2,r2,r2
5         j end
6  else:  mov 0,r2
7  end:   store r2,A[r1]
8         slice_end
9         inc r1
10        jl r1,N,loop
11        slice_fence
12        load A[0],r1
```

region end, after which instructions can depend on sliced instructions. When a branch misprediction occurs within a slice, only the remaining instructions in the same slice should be flushed, while newer instructions can continue to be executed (until the end of the slice region).

The typical use case is a loop of independent iterations, see Listing 1. The slice is the body of the loop, indicated by the `slice_start` and `slice_end` instructions, and the number of slices in the region equals the number of iterations. The region is ended after the loop using a `slice_fence` instruction, when data calculated in the loop is used.

Not all instructions within a region should belong to a slice. For example, the iterator increment and the loop branch (instructions 9 and 10) are always control and data dependent on the previous iteration increment and branch. Therefore, they are left out of the slice, although they still belong to the slice region. If a branch that is not within a slice is mispredicted, all instructions following that branch should be flushed, as in the regular case. For example, if the loop branch is predicted taken after the last iteration, we should flush all wrongly speculated next iterations of the loop.

Instructions in a slice can depend on instructions out of a slice in the same region. In the example, the loop body depends on the value of `r1`, i.e., the iterator, and is also control dependent on the loop branch. The only requirement is that all instructions following the slice should be control and data independent of the instructions in the slice.

Figure 1 shows a typical sliced loop, with its (potential) dependences. The iterator increment and branch are outside the slice, but within the slice region, i.e., all code before the slice fence. All slices are independent of each other, except for reduction variables, which we discuss in Section 4.5. Furthermore, none of the non-sliced code depends on sliced instructions, up to the slice fence. Code after the fence depends on the sliced code (otherwise the sliced code would be dead code), but only through memory, as we explain in Section 4.4.

Slices and regions are indicated using three new instructions: `slice_start`, `slice_end` and `slice_fence`. These need to be inserted by the programmer or compiler, who is responsible for ensuring that the slice independence conditions are met. This alleviates hardware from dependence checking, but it puts more burden on the software. However, dependence checking is already a main task

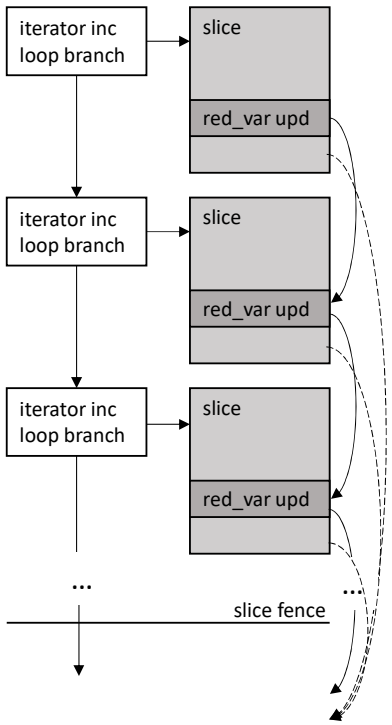

Figure 1: Typical sliced loop. Full line arrows are register dependences, dashed arrows are dependences through memory (store to load).

of the compiler, and increasingly also of the programmer to enable thread parallelism.

Enabling selective flush is a similar effort as parallelizing an application. For example, consider the popular OpenMP (omp) framework for parallelizing applications. The iterations of an omp parallel for loop can be divided among threads and are therefore inherently independent. These loop bodies can be safely encapsulated in a slice, exploiting the selective flush mechanism for iterations executed in the same thread. In fact, by putting parallel for iterations into slices, we further exploit the parallelism within a thread: instead of enforcing sequential execution after a branch miss, newer iterations can be executed in parallel with the branch miss resolution. Furthermore, branch reconvergence and instruction dependence compiler analysis have been discussed in prior work [6, 19], which means an automatic compiler implementation is realistic.

4.2 Selective Flush Mechanism

The first addition to the hardware is the ability to decode the three new slice instructions. Note that these instructions can be taken from the set of no-op instructions, such that a binary compiled with slice instructions can also be executed on a processor that does not support selective flush. The core keeps track of which instructions are within a slice and which not, e.g., by adding one bit to each ROB entry. When a misprediction on a non-sliced branch is detected, the regular branch recovery mechanism is used, i.e, flushing all newer instructions and restarting fetch at the correct path.

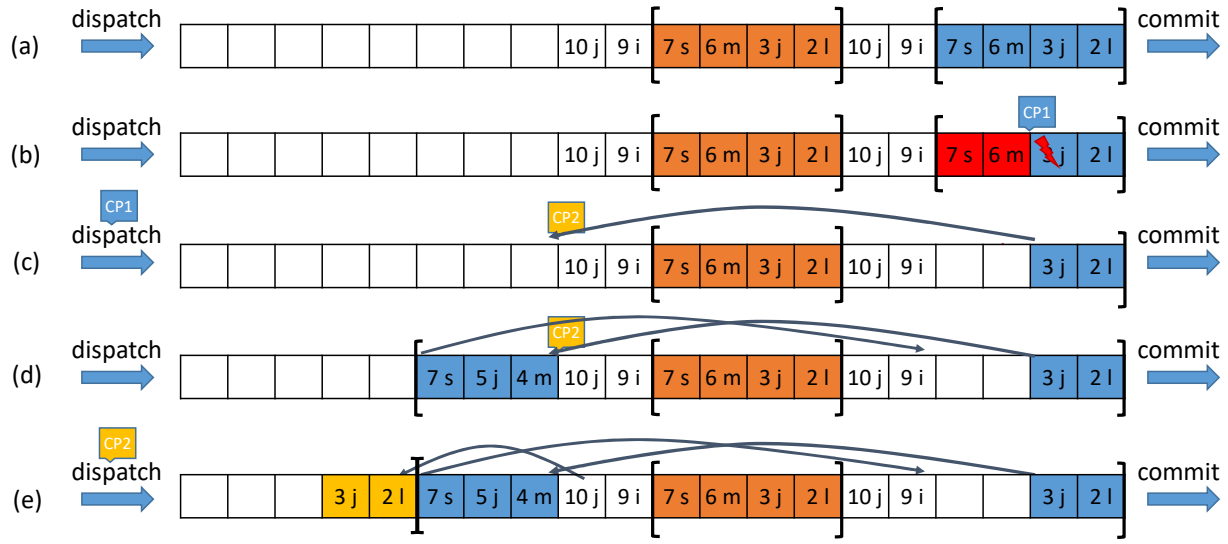

Figure 2: Selective flush mechanism example. Instructions from Listing 1: line number and first letter of instruction (j=branch). Different colors denote different slices/iterations, slice instructions are shown as large brackets. (a) Situation after fetching 2 iterations. (b) Misprediction detected in instruction '3 j', 2 instructions flushed (red). (c) ROB relinked to fetch correct path, dispatch set to checkpoint CP1, checkpoint CP2 taken at 'regular fetch'. (d) Fetching correct path until end of slice and pointing back to next out-of-slice instruction (9 i). (e) Resuming regular fetch from CP2.

When a misprediction on a sliced branch is detected, only the instructions after the branch in the same slice are flushed, see Figure 2(b). Additionally, we take a checkpoint (CP2) of the current rename table, i.e., at the point where instruction fetch should continue after fetching the correct path within the slice. We call this checkpoint the 'regular fetch checkpoint'. Newer instructions that do not belong to the slice (rightmost '9 i' and further to the left) are kept in the ROB, and continue to execute. Fetch is redirected at the correct path, and the mispredicted branch rename table checkpoint (CP1) is restored (c). When the correct path execution reaches a `slice_end` instruction (d), the correct path within the slice is complete. The instructions after the `slice_end` are already fetched and potentially executed. At that point, the 'regular fetch checkpoint' (CP2) is restored and fetch continues where it left off (e).

When a branch misprediction in a slice is detected and the end of the slice is already in the ROB, the front-end (fetch-decode-rename) contains correct path instructions (because they are outside the current slice). These front-end instructions should not be flushed and can continue to proceed through the pipeline. The regular fetch checkpoint is taken when the last of these instructions is renamed. In the meantime, the front-end is filled with the resolved correct path instructions within the affected slice, so there is no disruption in the flow of instructions in the front-end. Redirecting fetch after the detection of the miss might introduce a one cycle bubble, which also ensures that no regular fetch and resolved path instructions end up in the same rename batch and a clean regular fetch checkpoint can be taken.

After the correct path is resolved, i.e., a slice end instruction is fetched, the fetch stage continues at the regular fetch point. To enable a smooth flow, we assume that slice ends are detected early in the fetch pipeline (similar to branches), such that we do not need to wait until they are decoded to redirect fetch to the regular fetch point. To avoid deadlocks, we propose to reserve some resources for resolving correct paths, see Section 4.7.

The regular fetch checkpoint (CP2 in the example) does not contain the renamings made after dispatching the resolved path. Because there are no dependences between instructions in the slice and the next instructions, all registers renamed inside a slice are dead at the end of the slice and the regular fetch checkpoint does not depend on the novel renamings along the correct path. An exception is a reduction variable, which we discuss in Section 4.5. We assume that the rename table is used only at the rename stage. The architectural/physical renaming for each instruction is also encoded in its ROB entry, this information is used at the commit stage to write back the architectural register and free the physical register.

4.3 Linked List ROB

Selectively flushing and restoring the wrong path within a slice breaks the in-order appearance of instructions in the ROB: instructions of the correct path of one iteration are inserted after instructions of later iterations. However, we still want to support in-order commit to avoid impact on off-core mechanisms that rely on in-order commit, such as precise interrupts and memory consistency mechanisms. Therefore, we propose to implement the ROB as a linked list, which enables removing and adding instructions in the middle of a stream. On a branch miss in a slice, the next pointer of the ROB entry containing the branch is set at the next free ROB entry, where the correct path will be fetched, see Figure 2(c). After finishing the correct path in the slice, the pointer is set to the instruction that logically follows, but that has already been fetched earlier (d). This pointer is saved together with the 'regular fetch checkpoint'. Next, the last instruction fetched when detecting the branch miss points to the next free entry, i.e., after the correct path

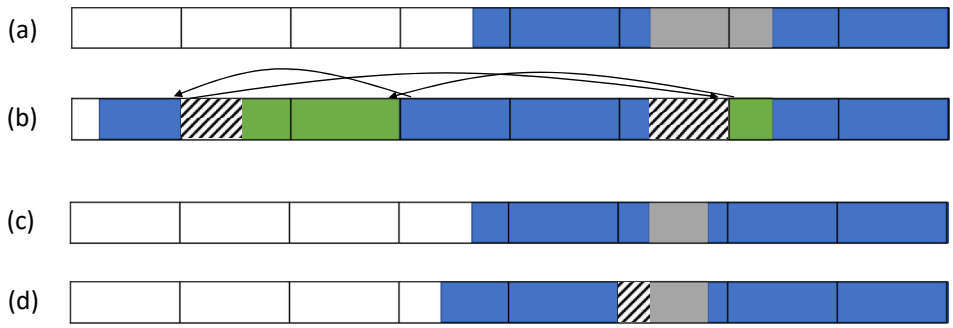

Figure 3: A blocked linked list ROB. Each block consists of multiple instructions; there is one pointer at the end of each block. In blue are correct path instructions, flushed instructions are in gray, green is the resolved correct path and shaded refers to gaps in the ROB. (a) Flushed instructions cross block boundary. (b) Situation after resolving the branch miss. (c) Flushed instructions do not cross block boundary. (d) Inserting a gap at dispatch to enable correct redirection to the resolved path.

(e), and dispatch continues. At the commit stage, the pointers are followed to ensure in-order commit of instructions. Note that most instructions will still be stored sequentially, each branch miss in a slice causes at most 3 redirections. The number of cycles where commit is stopped prematurely because of a pointer indirection is therefore limited.

Depending on the implementation of the load and store queues in the baseline core, they should or should not also be changed to a linked list. For example, if there are pointers to/from the corresponding ROB entries or some kind of sequence numbers, the relative order for checking address aliasing, load-store forwarding and ordering for memory consistency, can be determined without a linked list implementation. If a linked list implementation is necessary, it involves a small extra overhead, because these queues are smaller than the ROB.

The linked list ROB is arguably the most impacting change to the regular core design. It involves adding a pointer to each entry, which adds $n\lceil \log_2 n \rceil$ bits for a ROB of n entries (e.g., 256 byte for a 256 entry ROB). Linked list ROBs have been used in the IBM POWER5 [22] and POWER8 [40] architecture to enable dynamically sharing ROB entries between threads in SMT mode.

To reduce the overhead of pointers, the ROB can be partitioned into blocks (e.g., of 8 entries), with only one pointer per block. Inside a block, instructions are consecutive, while blocks can be dispatched out of order. Blocks also simplify committing multiple instructions per cycle by avoiding following pointers. This reduces the overhead (e.g., to 20 byte for a 256-entry ROB and 8-entry blocks), but it also leaves holes in the ROB, as out-of-order paths can only start at block boundaries.

If a branch miss in a slice is detected, the instructions in the slice are flushed, see Figure 3. If the `slice_end` is in a different block as the branch miss (or it is the last instruction of the same block), see Figure 3(a), we start fetching the correct path after the branch miss to fill gaps and to keep instructions consecutive as much as possible (rightmost green area in Figure 3(b)). When the last block before the end of the flushed slice is filled, we use its pointer to point to the next free block to continue fetching the correct path. This means that the flushed entries in the partially flushed block will remain empty. Once the correct path is fetched, the pointer of the last correct path block points back to the block with the next instruction, i.e., the first instruction after the flushed instructions. The remainder of the last correct path block also has to remain empty, because the pointer is used to point back to the next block. As soon as all instructions in a block with a gap are committed, these gaps can be reclaimed.

However, if the missed branch and `slice_end` are in the same block (and `slice_end` is not the last instruction) (Figure 3(c)), there is no available pointer between the flushed instructions and the first out-of-slice correct path instruction. To avoid this, we propose the following mechanism:

- At dispatch, a 'slice branch' bit is kept.
- If a conditional branch within a slice is dispatched, this bit is set.
- If we cross the boundary of a block, the bit is reset.
- When we dispatch a `slice_end` and the bit is set, we pad the rest of the block with empty slots, such that the next instruction after the `slice_end` starts at the next block, see Figure 3(d).

Note that this addition is only needed when the ROB is divided into blocks with multiple instructions. We evaluate the impact of a blocked implementation in the results section.

4.4 Slice Fence

A `slice_fence` denotes the end of a slice region. After a `slice_fence`, instructions can depend on instructions in slices. If an in-slice branch miss is detected after fetching a `slice_fence`, we should therefore also flush the instructions after the `slice_fence`. Thereto, when dispatching a `slice_fence`, we also take a rename table checkpoint, as for branches. When an in-slice branch miss is detected and the `slice_fence` is already dispatched, we also flush the instructions after the `slice_fence`, in addition to flushing the instructions in the slice. We store the rename table checkpoint at the `slice_fence` as the 'regular fetch checkpoint', i.e., the point where fetch should restart after dispatching the correct path in the slice. Like this, instructions after a `slice_fence` can be executed speculatively and are only flushed when a branch miss is detected within a slice.

Instructions after the `slice_fence` can depend on instructions within a slice, but the register rename information of instructions in a recovered slice can be lost in the rename table (see the last

paragraph of Section 4.2). Therefore, the dependences between the slices and the code after the fence should be implemented through memory and not through registers, by storing and loading data from memory. In the envisioned use case of a parallel loop, this is the case: because iterations are executed by different threads, no data can be transferred through registers, only through memory. Reduction variables are a special case, which are discussed in the next section.

4.5 Reduction Variables

Reduction variables are variables that are updated and accumulated in the loop iterations, such as counting the occurrence of certain conditions. An important characteristic of a reduction variable is that it can be updated in any order, i.e., its operation is commutative (e.g., addition, taking maximum, etc.). Additionally, the other calculations in the loop do not depend on these variables, they only depend on each other between different iterations.

In OpenMP, reduction variables have to be explicitly defined, such that they are correctly updated when the loop is executed by multiple threads. Typically, a local reduction variable is instantiated in a register, and after all iterations are finished, the local variables of all threads are reduced through shared memory to obtain the final value. This implementation breaks when applying our selective flush mechanism: if the reduction variable is wrongly updated speculatively, the next iteration will take the wrong value, and this will not be corrected when the correct path is executed because the newer instructions are not re-executed.

To solve this, we propose to execute reduction variable operations only when they are not speculative anymore, i.e., when they are at the head of the ROB. Reduction variable updates are kept in the reservation stations until they are at the head of the ROB, similar to what is currently done for atomic operations. Delaying their execution until commit does not delay other instructions, because they only depend on themselves, no other instructions depend on them. Additionally, reduction variables should not be renamed and should read from and write to architectural registers, which is correct because they are the oldest instruction when executed. This is needed because reduction variable update instructions may be re-fetched on a branch miss in a slice, and an output register renaming would not be transferred to already fetched non-flushed slices. We propose to introduce a new prefix to indicate instructions that cannot be executed speculatively (similar to the 'lock' prefix in x86). This prefix is added by the compiler, guided by the programmer's annotations (e.g., the reduction keyword in OpenMP).

4.6 Multiple Concurrent Branch Misses

A consequence of the selective flush mechanism is that branch misses in newer slices can occur when an older branch miss is still being resolved, because newer instructions continue to execute. To support multiple concurrent branch misses, we propose to add a 'fetch redirect queue' (FRQ), that contains all detected branch misses in slices that should be resolved before continuing to fetch at the 'regular fetch checkpoint'. The FRQ is a FIFO queue where each entry contains the following fields:

- The ROB entry of the mispredicted branch, to set the ROB pointers when dispatching the correct path.
- The program counter of the correct path instruction after the mispredicted branch, to indicate the start of the correct path.
- A pointer to the rename table checkpoint of the mispredicted branch.

Additionally, there is a separate entry (not part of the queue) containing this data for the 'regular fetch checkpoint'.

On a branch miss in a slice, this data is pushed in the FRQ, and if the FRQ is empty, the 'regular fetch checkpoint' is also set. At the fetch stage, the FRQ is checked, and if there is an entry at the head, the correct path is fetched until a `slice_end`. If another branch miss in a slice is detected while another slice is being resolved, its data is pushed in the FRQ, but fetch continues at the currently resolving slice. When the slice is resolved, the head of the FRQ is removed and the FRQ is again checked. If there is another entry at the head, this slice is resolved first. Only if the FRQ is empty, fetch is resumed from the 'regular fetch checkpoint'. This ensures that branch misses are resolved in the order they occur, and the oldest instructions are executed first, such that commit is not needlessly blocked.

When instructions are flushed after a non-slice branch miss, the FRQ is checked if it contains entries that point to flushed instructions. If so, the corresponding FRQ entry is removed from the queue. Because all newer instructions are also flushed, all newer FRQ entries will also be removed, preserving a simple FIFO ordering.

4.7 Freeing/Reserving Resources

Selectively flushing and refetching instructions can lead to a deadlock if there are no free resources for fetching the correct path. For example, assume a mispredicted branch that jumps over a section with store operations, i.e., the correct path is to execute these store operations. Assume further that before detecting this miss, all store queue entries have been occupied by newer instructions. Since the wrong path contains no store instructions, no store queue entries are freed when the wrong path is flushed. The correct path cannot dispatch because of the lack of store queue entries, and the store queue entries of the newer instructions cannot be released because they cannot commit before the older slice is resolved. A similar deadlock can occur when there are more regular fetch instructions in the front-end than the number of free ROB entries after flushing the wrong path.

To prevent deadlocks, we propose to reserve a certain number of resources for resolving correct paths when there are in-slice instructions in the ROB. These cannot be used for the regular fetch. We identified three resources to reserve: reservation stations (RS), load queue (LQ) entries and store queue (SQ) entries. By reserving these resources, we also ensure that enough ROB entries are free to fetch the correct path. Note that reserving a single resource of each suffices to prevent deadlocks: eventually, the resolved path instructions will become the oldest and can commit, freeing resources to dispatch the next correct path instruction(s).

In case there are still not enough free resources to hold the regular fetch instructions in the front-end, we flush (part of) the front-end instructions. This has no impact on the checkpoints (they are not renamed yet), it only has a small extra performance penalty.

Reserving resources does not only prevent deadlocks, it can also improve performance. The more resources are reserved, the quicker we can execute the correct path. On the other hand, reserving resources slows down the progress in the regular fetch path.

We evaluate the impact of freeing fewer or more resources in the evaluation section.

4.8 Summary of Additions

In summary, the following additions to a conventional out-of-order core are needed to support selective flush:

- 3 new slice instructions (no arguments).
- A linked list ROB, which can be block partitioned to limit the overhead.
- One bit per ROB entry to indicate if an instruction belongs to a slice.
- The FRQ with a few (e.g., 8) entries, to support concurrent branch misses. If the FRQ is full, new misses can be resolved using the conventional scheme, i.e., flushing all newer instructions.
- A prefix for the common reduce instructions to execute at commit.
- A controller to correctly set the ROB linked list pointers on a branch miss and after resolving the correct path.

These are non-trivial changes, but they mainly relate to ROB bookkeeping. Critical pipeline stages, such as wakening and selecting instructions to issue in the reservation stations, are not impacted by our mechanism.

5 EXPERIMENTAL SETUP

5.1 Benchmarks

For evaluating the performance, we selected the GAP benchmark suite[7]. The GAP benchmarks are optimized CPU implementations of 6 basic graph kernels: betweenness centrality (bc), breadth first search (bfs), connected components (cc), pagerank (pr), single source shortest path (sssp) and triangle count (tc). Graph applications, and more generally unstructured sparse applications, are gaining importance to analyze relations within big data sets [33]. Recently, several graph accelerators [15, 20, 26, 41] and graph analysis software platforms [28] have been proposed, and a US government project to optimize graph analysis [39] was launched.

As input graphs, we use synthetically generated RMAT graphs [9]. Synthetic graphs have the advantage that we can control their size, and measure the impact of graph size on performance. The execution time of the six benchmarks as a function of input graph size differs considerably. In order to have a similar execution time, the baseline graph input size differs per application: RMAT-18 for `tc`, RMAT-20 for `bc`, `cc`, `pr` and `sssp`, and RMAT-22 for `bfs` (for single core evaluation). The memory footprint of these graphs is 47 MB (RMAT-18) to 283 MB (RMAT-22), which is much larger than the 1.4 MB LLC cache per core. The miss rate in the LLC varies between 45% and 70%. We also evaluate our technique on larger graphs.

Graph applications typically have data dependent hard-to-predict branches, which depend on loads from main memory, because of the high cache miss rate due to the lack of locality. Furthermore, they are often highly parallel, applying a function on each vertex, which means iterations are independent. Lastly, the small code footprint of the GAP benchmarks makes it easy to add slice instructions at meaningful places.

Additionally, we also evaluate our proposal on merge sort (ms) [25], which also suffers from a high branch miss rate because of the unpredictable comparisons between elements. Sorting is a common kernel for many applications. For example, the SPEC 2017 benchmark mcf spends 40% of its execution time in sorting. Mcf uses the quicksort algorithm, for which is it hard to find or create independent slices. Merge sort is particularly fit for parallelization, and thus also for selective flush, because distinct sections of the array can be merged in parallel. For the baseline results, we sort a list of 10 million random integers.

Although we only evaluate our proposal on graph applications and sort, we believe that it will be also beneficial for other branch miss heavy workloads. Note that our mechanism mostly favors short parallel sections, whereas conventional thread parallelism performs best with long parallel sections to reduce the threading overhead. This is why existing code (such as the SPEC benchmarks) is difficult to convert without heavily reworking parts of the code. However, emerging domains, such as machine learning, require new code which can be implemented with this mechanism in mind. Furthermore, the advent of domain specific processors in search for more efficiency, provides a choice to processor designers to implement this mechanism only for domains where it has a substantial impact (such as graph analysis).

5.2 Simulator

We implement the selective flush mechanism into an in-house version of the Sniper multicore simulator [8]. We configure the simulator to resemble an Intel® Xeon® Platinum 8180 processor, codenamed Skylake [14], see Table 1. We extend Sniper with a wrong-path engine, which models the flow of wrong-path instructions through the core pipeline until they are flushed. This is needed to accurately model the impact of fetching and flushing only the wrong-path instructions within a slice. We use Pin's code cache and decoder to reconstruct the wrong path, so when both paths are in the code cache (which is soon for hard-to-predict branches) we can reconstruct them both in case of a misprediction. Because selective flush is an internal core mechanism, most of our results are measured on a single core. For these simulations, we scale down shared resources (LLC and memory bandwidth) proportionally (i.e., 28 times smaller).

6 RESULTS AND DISCUSSION

6.1 Placing Slice Instructions and Single Core Speedup

All of the GAP benchmarks have nested loops, of which the outer loop is parallelized with OpenMP. Typically, the outer loop iterates over all vertices of the graph, while the inner loop iterates over the neighbors of these vertices. For some benchmarks, the iterations of this inner loop are also independent, meaning that we can choose to put the slice instructions around the iterations of the outer loop or the inner loop (slices cannot be nested). Putting the slices in the outer loop means that on a branch miss, all of the iterations

Table 1: Simulated processor configuration

Dispatch/commit width	4
Reorder buffer (ROB)	224 entries
Reservation stations	97
Load/store queue entries	72/56
Branch predictor	TAGE [38]
L1 I/D-cache	32 KB/32 KB
L2 private cache	1 MB
LLC NUCA	1.375 MB/core
Core count	28
Network on chip	mesh
Memory latency	50 ns
Memory bandwidth	115.2 GB/s

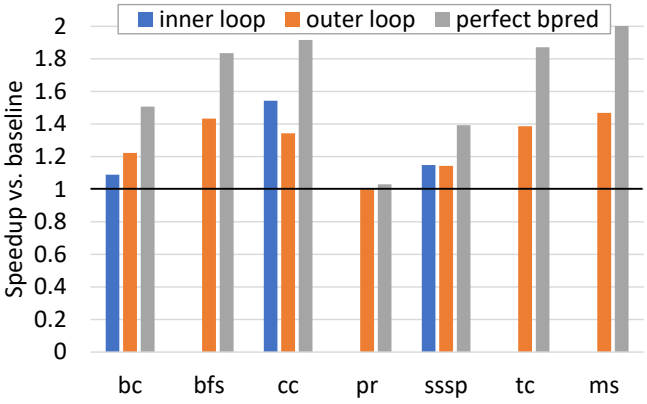

Figure 4: Speedup of the selective flush mechanism versus baseline for slicing the inner (where possible) and outer loop, and speedup for perfect branch prediction.

of the inner loop after the branch miss will be flushed and re-executed, while slicing the inner loop only flushes instructions within one inner loop iteration. However, putting them in the inner loop prevents the use of selective flush for branches outside the inner loop, falling back to the default mechanism of flushing all instructions. In particular, the branch ending the inner loop is often mispredicted, because the number of neighbors is variable.

The decision where to put the slice instructions depends on the iteration count of the inner loop, and where most branch misses occur. We need to make this decision for three of the six GAP benchmarks. `bfs` and `tc` have inner loops with control dependent iterations (they break out of the loop prematurely), and `pr` has no conditional branch in its inner loop. Merge sort's inner loop (merging the subarrays) also contains dependent iterations, so only the outer loop can be sliced. Figure 4 shows the single core speedup of the selective flush mechanism versus the baseline core, for slicing the inner and outer loop for the three other benchmarks, and the outer loop slicing for the rest.

For `bc`, slicing the outer loop gives better performance, while for `cc`, slicing the inner loop is better. For `sssp` it does not matter much. Therefore, if there is a choice, we propose to test a few options to determine which one performs best.

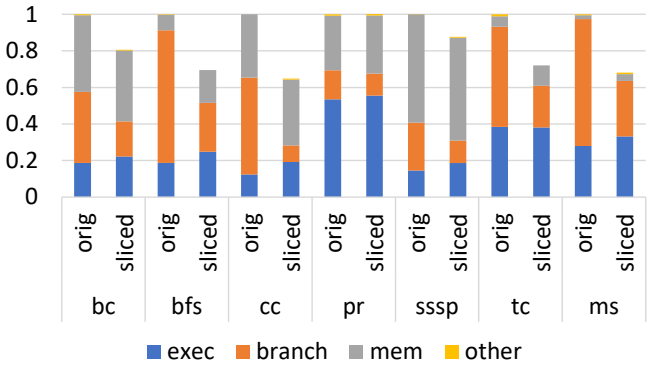

Figure 5: Cycle stacks of the baseline (orig) and sliced execution.

Taking the best performing options, the overall average speedup is 1.29 (harmonic mean). `pr` has no speedup, because it has no conditional branches in its loop, other than the inner loop branch. Without `pr`, the average speedup is 1.35.

Figure 4 also shows the speedups for an oracle branch predictor, i.e., the maximum achievable speedup for branch optimization. It confirms the low margin for `pr`. For the other applications, our mechanism closes a large part of the gap between the baseline and a utopic perfect branch predictor. Perfect branch prediction has an average speedup of 1.60, meaning that selective flush reaches almost 50% of the potential gain.

Note that the branch miss rate in the baseline and the sliced execution is exactly the same. Selective flush decreases the penalty per miss. There is still an unavoidable penalty, namely the flush and refetch of miss dependent instructions, so perfect branch predictor performance can never be reached. In addition, if the slice is large (e.g., multiple iterations of the inner loop), a large number of instructions need to be flushed, and the penalty will be closer to that of the conventional branch miss resolution mechanism.

Figure 5 shows the (simplified) cycle stacks of the baseline and sliced execution. The stacks are normalized to the cycle count of the baseline execution. The 'exec' component refers to the time needed to execute all instructions assuming all cache hits and no branch mispredictions. The 'branch' component is the time lost due to branch misses, and the 'mem' component is the time the core is stalled waiting for memory operations that miss in the L1 cache. The 'other' component contains other stall cycles, such as decode bottlenecks.

The evaluated benchmarks are branch miss and memory bound. The graphs do not fit into the LLC, and traversing a graph leads to low locality, resulting in a large miss rate. Branches are data dependent, and often depend on loads that miss in the cache, explaining their large miss penalty.

Our mechanism clearly reduces the branch miss component, explaining the execution time reduction. The memory component increases slightly, because some cache misses along the wrong path turn out to be useful prefetches for the correct path, and in the baseline execution, their latency is therefore partly hidden by the branch resolution time. In the sliced execution, this penalty is exposed more, because execution continues during branch resolution.

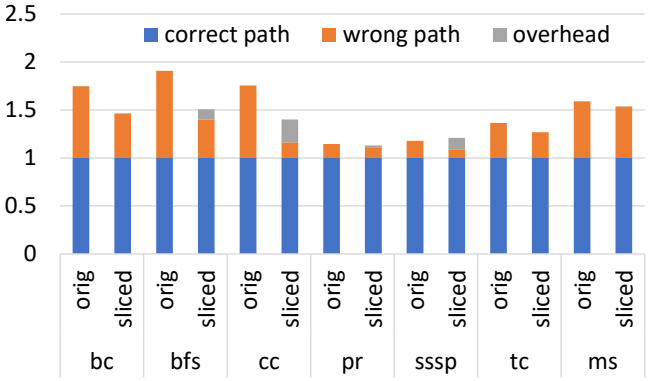

Figure 6: Dispatched instruction count, normalized to the number of correct path instructions.

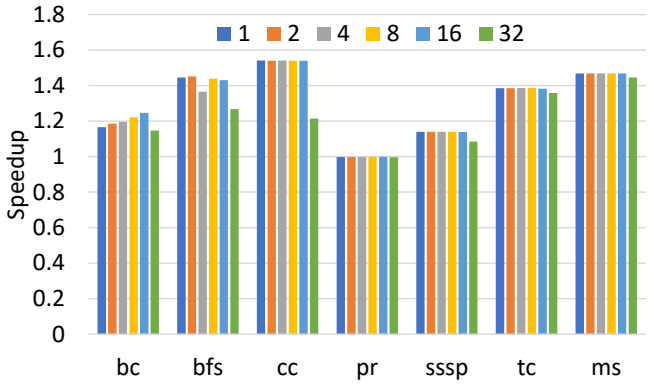

Figure 7: Speedup of the selective flush mechanism with reserving 1 to 32 RS/LQ/SQ entries for resolving correct paths.

The execution component also increases, because more instructions need to be fetched, i.e., the slice instructions.

In that respect, Figure 6 shows the number of dispatched instructions for both executions, normalized to the number of correct path instructions in the baseline execution. Wrong path instructions are fetched after a branch miss until the miss is detected, or in case of sliced execution, until a slice end is fetched. Overhead refers to the slice instructions, which are discarded after dispatch but still take up slots in the frontend. Slicing reduces the number of dispatched wrong path instructions, and for all but one application, the overhead of slice instructions is smaller than the reduction of wrong path instructions. This means we can also claim better energy efficiency, because fewer instructions need to be processed.

For `sssp`, the inner loop code is small, resulting in a large overhead of slice instructions, and an overall increase in total dispatched instruction count. However, since the slice instructions are processed in the frontend only and are discarded at dispatch, they only have a small impact on total latency, explaining the net positive performance gain.

The small reduction in wrong-path instructions for `ms` might seem strange, given the high performance benefit. Figure 6 shows the number of wrong-path instructions that are dispatched, which does not include the instructions flushed in the front-end pipeline. `ms` has a small memory component, because the sequential accesses can be efficiently prefetched. As a result, the branch miss is resolved quickly and the number of dispatched wrong-path instructions is low. Selective flush does flush slightly fewer instructions, but most of the non-flushed instructions are in the core front-end, which are not counted here. In fact, because selective flush can continue fetching and dispatching instructions while the mispredicted branch is recovering, there is no interruption in the front-end, yet another performance benefit of our mechanism.

6.2 Freeing/Reserving Resources

Figure 7 shows the impact of reserving 1 to 32 entries (of each type) in powers of two (out of 97 reservation stations, 72 load queue entries and 56 store queue entries). It shows that until 16 reserved entries, the performance remains pretty constant, or even improves (for `bc`). `bc` has its outer loop sliced, so the correct path can potentially be long (multiple inner loop iterations), which is why it benefits from the extra resources. Reserving 32 entries has a clear negative impact on performance, because it limits the progress of regular fetch, while not providing much extra performance to the resolve paths. Note that reserving 32 entries still has a speedup (or neutral for `pr`) versus the baseline: due to the branch misses, high cache miss rate and indirect memory operations, there is not much instruction-level parallelism to exploit, and therefore these resources are not used efficiently in the baseline. In the results in the previous section, we always assume 8 reserved entries.

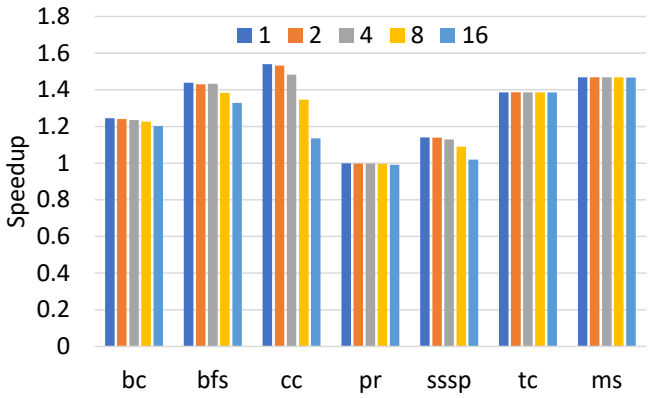

Figure 8: Speedup of the selective flush mechanism with a block linked list from blocks of size 1 (no blocks) to 16.

The small dip for `bfs` at 4 reserved entries is due to a second-order interference with the data prefetcher, making the prefetcher perform slightly worse for this configuration. Disabling the prefetcher for all configurations does not show this dip, but the overall performance is lower.

6.3 Blocked Linked List ROB

A blocked linked list ROB reduces the overhead of pointers and simplifies committing multiple instructions per cycle. However, as discussed in Section 4.3, it creates gaps in the ROB when flushing instructions, reducing the ROB capacity. We model these gaps in our simulator. Figure 8 shows the resulting performance for different block sizes, from 1 (no blocks) to 16. Up to a block size of 4, we see negligible impact, because the number of empty slots is small. For

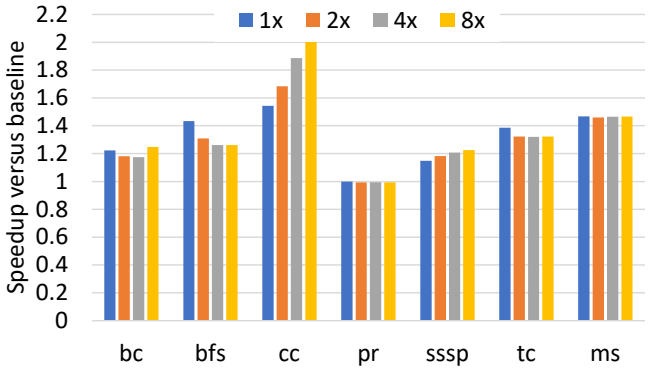

Figure 9: Sensitivity to input size.

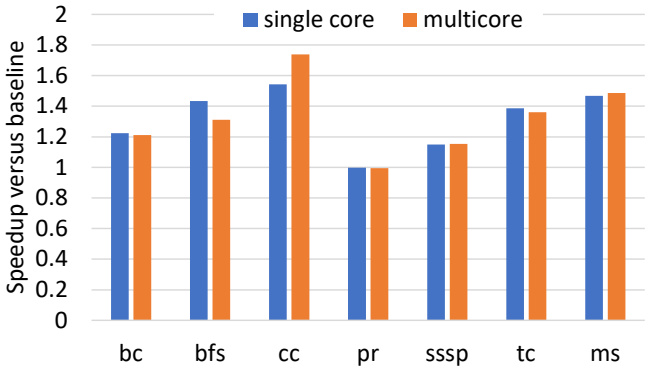

Figure 10: Multicore speedups versus single core speedups.

a block size of 8, we see an average 4.1% reduction in performance, increasing to 9.5% for blocks of 16 entries.

6.4 Sensitivity Studies

The sensitivity of our mechanism to the input size is evaluated in Figure 9. We increased the input size by 2×, 4× and 8× versus the baseline results. There is no clear trend: for `cc` and `sssp`, the speedup increases as the graph gets bigger, while for `bfs` and `tc`, the speedup decreases. We find that the gain is highly correlated with the branch miss fraction in the cycle stack: for some applications, this fraction decreases because the memory fraction increases, while for others, the branch miss fraction also increases because branches are dependent on memory operations that miss in cache. The higher the branch miss fraction, the higher the potential gain of selective flush. The average speedup varies between 1.27 (2× larger graph) and 1.31 (8× larger graph).

We also evaluated the selective flush mechanism on a full 28-core configuration, see Figure 10. We increased the input size by a factor of 16, to have a similar memory footprint per core. In this experiment, we both exploit the thread parallelism (using OpenMP threads) as the branch parallelism within a thread (using selective flush). As for the input size sensitivity, speedups either decrease or increase versus a single core evaluation. This is again dependent on the branch miss fraction in the cycle stack. The average speedup of 1.29 shows that the benefit of our technique is orthogonal to that of thread parallelism.

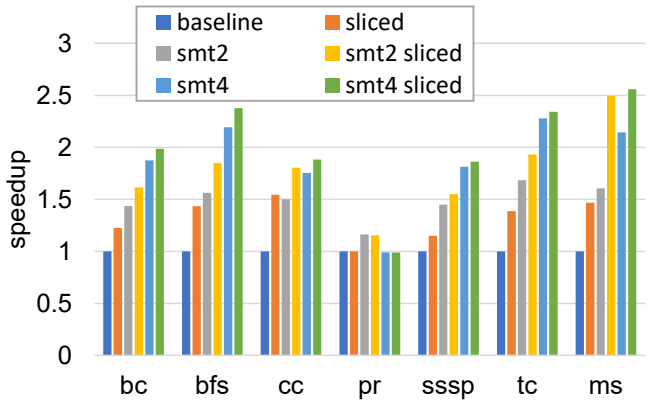

Figure 11: Speedups of SMT (2 and 4 threads), sliced execution and combinations (single core).

Lastly, Figure 11 shows the impact of simultaneous multithreading (SMT) and combining SMT and slicing. We execute 2 and 4 application threads on a single core, for SMT2 and SMT4, respectively. SMT reduces the penalty of branch misses: fewer speculative instructions are fetched, because instructions of other threads are also fetched during the branch resolution time. Furthermore, SMT also hides more memory latency by executing instructions of other threads during the memory latency. Therefore, we see a larger speedup for SMT than for slicing. However, combining slicing and SMT still provides additional speedup. Compared to perfect branch prediction, slicing still provides 50% of the potential speedup with SMT2 or SMT4.

For some applications (`cc` and `ms`), slicing performs even better than adding SMT threads. For SMT, multiple threads execute concurrently, which increases the current working set and puts more pressure on caches and TLBs. This leads to more conflict misses and lower per-thread performance, in most cases compensated by the larger throughput (but not always, e.g., for `pr` at SMT4). Slicing does not require more threads, and therefore has no impact on cache miss rates.

7 CONCLUSIONS AND FUTURE WORK

Branch mispredictions remain an important source of performance limiters, which gets worse with deeper pipelines and increasingly irregular applications. Reusing the execution of data and control independent instructions after a branch miss instead of refetching them increases performance and energy efficiency. We propose a novel hardware-software cooperative technique to select reconverging instructions and not flush them after a branch miss. Independent slices are denoted by 3 novel slice instructions, inserted by the (performance expert) programmer or compiler. The reorder buffer is reorganized as a linked list to enable removing and inserting instructions in the middle of a stream, enabling the insertion of the part of the correct path that truly depends on the branch misprediction. An evaluation on emerging graph applications shows an average performance increase of 29%.

This paper serves as a proof of the efficacy of this technique. Further research is needed to implement automatic insertion of slice instructions by the compiler, and to study the impact of the novel ROB organization on low level (RTL and circuit) design.

REFERENCES

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. 2007. Exploiting Postdominance for Speculative Parallelization. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*. 295–305.

[2] Mayank Agarwal, Nitin Navale, Kshitiz Malik, and Matthew I Frank. 2008. Fetch-Criticality Reduction through Control Independence. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[3] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 423–434.

[4] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham H. Akkary. 2007. Transparent Control Independence (TCI). In *34th Annual International Symposium on Computer Architecture (ISCA)*. 448–459.

[5] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen. 2012. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro* 32, 6 (2012), 4–16. https://doi.org/10.1109/MM.2012.57

[6] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 513–526.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[8] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. 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)*.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SIAM International Conference on Data Mining*. 442–446.

[10] Adarsh Chauhan, Jayesh Gaur, Zeev Sperber, Franck Sala, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. 2020. Auto-predication of critical branches. In *ACM/IEEE 47th International Symposium on Computer Architecture (ISCA)*. 92–104.

[11] Chen-Yong Cher and TN Vijaykumar. 2001. Skipper: a microarchitecture for exploiting control-flow independence. In *34th ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 4–15.

[12] Jamison D Collins, Dean M Tullsen, and Hong Wang. 2004. Control flow optimization via dynamic reconvergence prediction. In *37th International Symposium on Microarchitecture (MICRO)*. 129–140.

[13] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.

[14] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation Intel Core: New microarchitecture code-named Skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[15] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms (IA^3 '16)*. 2–9.

[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A Mechanistic Performance Model for Superscalar Out-of-order Processors. *ACM Transactions on Computer Systems (TOCS)* 27, 2 (May 2009), 3:1–3:37.

[17] S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur. 2018. Many-Core Graph Workload Analysis. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 282–292. https://doi.org/10.1109/SC.2018.00025

[18] Amit Gandhi, Haitham Akkary, and Srikanth T Srinivasan. 2004. Reducing branch misprediction penalty via selective branch recovery. In *International Symposium on High Performance Computer Architecture (HPCA)*. 254–264.

[19] Ali Hajiabadi, Andreas Diavastos, and Trevor E Carlson. 2021. NOREBA: a compiler-informed non-speculative out-of-order commit processor. In *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 182–193.

[20] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.

[21] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *37th International Symposium on Computer Architecture (ISCA)*. 37–47.

[22] Ron Kalla, Balaram Sinharoy, and Joel M Tendler. 2004. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro* 24, 2 (2004), 40–47.

[23] Hyesoon Kim, Jose A Joao, Onur Mutlu, and Yale N Patt. 2006. Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. In *39th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 53–64.

[24] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N Patt. 2005. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *38th International Symposium on Microarchitecture (MICRO)*.

[25] Donald E Knuth. 1998. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional.

[26] Andrew Kopser and Dennis Vollrath. 2011. Overview of the next generation Cray XMT. In *Cray User Group Proceedings*. 1–10.

[27] V. R. Kothinti Naresh, R. Sheikh, A. Perais, and H. W. Cain. 2018. SPF: Selective Pipeline Flush. In *IEEE 36th International Conference on Computer Design (ICCD)*. 152–155.

[28] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 411–428.

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

[30] Mengjie Mao, Hong An, Tao Sun, Qi Li, Bobin Deng, Xuechao Wei, and Junrui Zhou. 2012. Distributed Control Independence for Composable Multi-processors. In *2012 IEEE/ACIS 11th International Conference on Computer and Information Science*. 124–129.

[31] Gordon E Moore. 1965. Cramming more components onto integrated circuits. , 114—117 pages.

[32] Quan M Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *53rd International Symposium on Microarchitecture (MICRO)*. 596–608.

[33] M. U. Nisar, A. Fard, and J. A. Miller. 2013. Techniques for Graph Analytics on Big Data. In *2013 IEEE International Congress on Big Data*. 255–262.

[34] Alex Pajuelo, Antonio González, and Mateo Valero. 2005. Control-flow independence reuse via dynamic vectorization. In *19th IEEE International Parallel and Distributed Processing Symposium*.

[35] E. Rotenberg and J. Smith. 1999. Control independence in trace processors. In *32nd ACM/IEEE International Symposium on Microarchitecture (MICRO)*. 4–15.

[36] Amir Roth and Gurindar S Sohi. 2000. Register integration: a simple and efficient implementation of squash reuse. In *33rd ACM/IEEE international symposium on Microarchitecture (MICRO)*. 223–234.

[37] Farzad Samie and Amirali Baniasadi. 2011. Power and frequency analysis for data and control independence in embedded processors. In *2011 International Green Computing Conference and Workshops*. 1–6.

[38] André Seznec. 2011. A new case for the TAGE branch predictor. In *44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 117–127.

[39] Wade Shen. [n. d.]. Hierarchical Identify Verify Exploit (HIVE). ([n. d.]). https://www.darpa.mil/program/hierarchical-identify-verify-exploit

[40] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.

[41] William S Song, Vitaliy Gleyzer, Alexei Lomakin, and Jeremy Kepner. 2016. Novel graph processor architecture, prototype system, and results. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[42] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[43] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *48th International Symposium on Microarchitecture (MICRO)*. 178–190.

[44] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *51st IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28.