Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches

Stephen Pruett
stephen.pruett@utexas.edu
University of Texas at Austin
United States

Yale N. Patt
patt@ece.utexas.edu
University of Texas at Austin
United States

ABSTRACT

High performance microprocessors require high levels of instruction supply. Branch prediction has been the most important driver of this for nearly 30 years. Unfortunately, modern predictors are increasingly bottlenecked by hard-to-predict data-dependent branches that fundamentally cannot be predicted via a history based approach. Pre-computation of branch instructions has been suggested as a solution, but such schemes require a careful trade-off between timeliness and complexity. This paper introduces Branch Runahead: a low-cost, hardware-only solution that achieves high accuracy while only performing lightweight pre-computation. The result: a reduction in branch MPKI of 47.5% and an average improvement in IPC of 16.9%.

KEYWORDS

Branch Prediction, Pre-computation, Control Independence

ACM Reference Format:
Stephen Pruett and Yale N. Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In *MICRO'21: 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/10.1145/3466752.3480053

1 INTRODUCTION

Despite significant improvements in branch prediction over the years, branch mispredictions remain a key performance limiter. Unfortunately, it is getting harder and harder to improve traditional branch predictors due to the existence of fundamentally unpredictable data-dependent branches. Data-dependent branches present serious challenges for history-based predictors[17–19, 24, 30, 32]as their outcome is not correlated to any previous outcomes in the branch history [12]. Rather, the outcome is based on a value recently loaded from memory.

As branch predictors continue to improve, their accuracy on data-dependent branches remains roughly constant. Over time, data-dependent branches have become responsible for a larger share of the total remaining mispredictions. To illustrate this, Figure 1 shows the misprediction rate of the 32 most hard-to-predict branches from each benchmark. The left bar is the accuracy of a 64KB TAGE-SC-L [32], winner of the 2016 Champion Branch Prediction competition (CBP-2016) limited storage category, and the middle bar is the accuracy of MTAGE-SC [33], winner of CBP-2016 unlimited storage category. On average, MTAGE-SC is only able to reduce misprediction from 11% (TAGE-SC-L) to 9%, an improvement of only 18%. We propose using dependence chains— a short sequence of operations that can *pre-compute* the result of the branch before it is needed in the fetch stage. Figure 1 also shows the accuracy of using dependence chains to pre-compute branch outcome (right bar). On average, dependence chains decrease mispredictions to 5%, reducing misprediction by 55% (TAGE-SC-L) and 44% (MTAGE-SC). This result clearly demonstrates that pre-computation can improve branch prediction for branches that TAGE fundamentally cannot predict accurately.

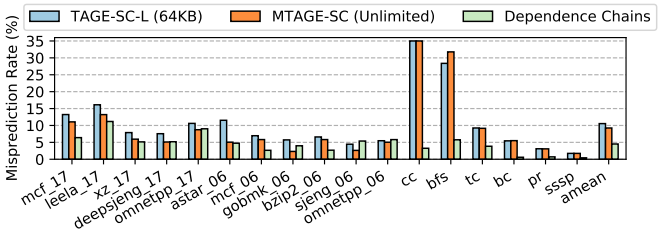

Figure 1: Misprediction Rate: TAGE-SC-L (64KB) vs MTAGE-SC (Unlimited) vs Dependence Chains for Hard-to-Predict Branches

Unfortunately, prior work in pre-computation for branch prediction has primarily focused on heavy-weight, compile-time approaches. Here, the compiler creates a filtered version of the original program, only containing instructions necessary to compute the result of hard-to-predict branches. The filtered thread, or "helper" thread, is executed asynchronously on another core [21, 36], Simultaneous Multi-threading (SMT) context [8, 9, 31, 38], or on a dedicated unit within the core [34]. We argue these approaches are fundamentally more costly as they require re-executing most instructions in the program, and thereby require expensive resources to pre-compute the branch. Meanwhile, light-weight runtime approaches [9] are not able to run continuously, limiting their ability to provide timely predictions.

We propose Branch Runahead, a system that continuously executes lightweight dependence chains to pre-compute the result of hard-to-predict, data-dependent branches. Dependence chains, which are extracted from the program at runtime, are far simpler than the helper threads proposed by prior work, which allows

MICRO'21, October 18–22, 2021, Athens, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480053

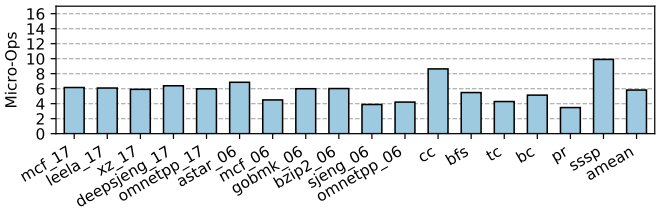

Figure 2: Average Length of Dependence Chains

them to be accelerated using less hardware, rather than requiring another core or SMT context. In particular, Branch Runahead improves state-of-the-art in four key areas.

Light-weight Dependence Chain. The dependence chain is a short sequence of operations necessary to produce the result of a branch instruction. [1] Unlike helper threads, dependence chains are guaranteed to be simple. All dependence chains have fewer than 16 micro-operations, do not contain expensive operations such as integer divide or floating point operations, and do not contain any control flow instructions. Figure 2 shows that the average length of a dependence chain is fewer than 8 micro-operations. Branch Runahead dynamically filters the instruction stream to produce the exact sequence of operations needed to compute the branch outcome. The result is a light-weight sequence of instructions that can be accelerated efficiently.

Continuous Execution. Prior techniques using light-weight dependence chains [9, 16, 26] have struggled to execute continuously, as dependence chains do not contain control flow instructions. This causes them to diverge from the main thread quickly, reducing the accuracy of predictions. Branch Runahead solves this problem using our new merge point predictor. The merge point predictor detects control and data dependencies between branch instructions, which allows Branch Runahead to properly order dependence chains.

Timeliness. Pre-computation is only effective if the results are ready before the prediction is needed. Branch Runahead maximizes chain level parallelism by predicting the next dependence chain, allowing it to issue as early as possible.

Dependence Chain Engine. Branch Runahead executes dependence chains using a dedicated unit, the Dependence Chain Engine (DCE). We propose 3 variants of the DCE: an unlimited storage *Big* engine, a 17KB *Mini* engine, and a 9KB *Core-Only* engine, which shares reservation stations, physical registers, and functional units with the core. The Dependence Chain Engine consumes just 2.2% of the area of a typical out-of-order core (or only 1.4% in the case of the Core-Only model). As branch outcomes are produced, they are inserted into prediction queues, which override predictions supplied by the traditional branch predictor. Dependence chains begin executing in the DCE as soon as their live-ins are known. Live-ins are copied from the physical register file during a branch misprediction and loaded into the DCE. This initializes the register file for the chain, effectively synchronizing it with the core. Dependence chains are not guaranteed to be correct, and occasionally diverge from the main thread. Once this is detected, the DCE is synchronized again by repeating the copy of the physical registers.

The contributions of this paper are:

- To our knowledge, this paper is the first work to evaluate dynamically generated, light-weight dependence chains that are run continuously as a means to pre-compute the results of branch instructions.
- We demonstrate the importance of accurately identifying affector and guard dependencies between branches (section 4.4).
- We introduce a new method for merge point prediction, which is 92% accurate, compared to prior work which is only 78% accurate [29] (section 4.4).
- We introduce the Branch Runahead system, including dependence chain extraction, synchronization with the fetch unit, and the microarchitecture of the Dependence Chain Engine. We show that placing restrictions on chain extraction can guarantee the simplicity of the dependence chain, allowing it to be executed quickly and efficiently on the Dependence Chain Engine (DCE).
- We evaluate three methods for chain initiation, which promotes chain level parallelism and improves the timeliness of predictions.

In the remaining sections, we discuss the fundamental limitations of prior work, and how Branch Runahead addresses them. Then, we provide a detailed discussion of Branch Runahead, including implementation and important properties. Finally, we show that Branch Runahead, when configured under reasonable hardware constraints, reduces branch MPKI by 47.5% and increases IPC by an average of 16.9%.

2 LIMITATIONS OF PRIOR WORK

Branch Runahead is not the first to propose pre-computation as a substitute for branch prediction. In fact, many works have paved the way for Branch Runahead [8, 9, 21, 31, 34, 38, 39]. However, Branch Runahead is the first runtime only solution to execute light-weight dependence chains continuously. This allows Branch Runahead to execute further ahead with fewer hardware resources.

2.1 Limitations of Compiler-based Techniques

Most prior work relies on the compiler to identify candidate branches and extract helper threads [8, 9, 21, 31, 34, 38, 39]. Since compilers take a holistic view of the program, they can iterate over the control flow and data flow that lead up to a branch instruction and produce the exact minimum set of operations needed to compute the direction of the branch. Zilles et al. [38, 39] first observed, however, that building helper threads that computed branch outcome 100% accurately was not profitable, as it required too many operations to be a part of the helper thread. Since then, research has focused on using profiling techniques during compile time to more aggressively remove instructions from the helper threads [8, 21, 31]. This results in a helper thread that is significantly simpler but no longer 100% accurate. Unfortunately, the effectiveness of these techniques relies heavily on the representativeness of the profiling data. Unrepresentative data can lead to inaccurate compiler optimizations that improperly reduce the helper threads, causing them to produce

[1] More specifically, a dependence chain is the backwards dataflow slice needed to compute the branch.

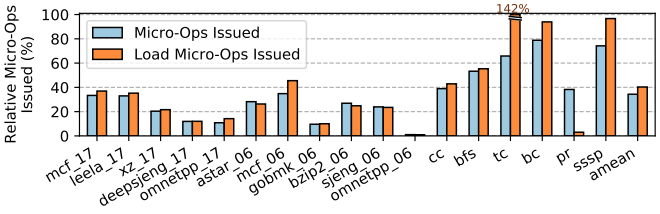

Figure 3: Increase in micro-ops due to Branch Runahead

far less accurate predictions at runtime, increasing expensive synchronizations. In addition, introducing dependence chains at the compiler level requires changes to the Instruction Set Architecture (ISA) that chip makers are usually hesitant, if not unwilling, to make. In contrast, Branch Runahead requires no modifications to the ISA or compiler to achieve its full potential.

2.2 Limitations of Prior Runtime Techniques

SlipStream and its variants [35, 36] are runtime only techniques that pre-compute the direction of branch instructions. In SlipStream, two processors are used to execute a program. The A-stream runs a filtered version of the program ahead of the R-Stream. This enables the A-stream to communicate branch directions via a hardware queue between the two cores. Slipstream reports to remove an average of 15% of all retired instruction, leaving the remaining 85% in the A-stream as overhead.

In contrast, Branch Runahead extracts only the instructions needed to predict the targeted branch. As a result, dependence chains in Branch Runahead are far simpler than the A-stream. To illustrate this, Figure 3 shows the increase in micro-ops executed due to enabling Branch Runahead. On average, Branch Runahead executes 34.3% more micro-ops, significantly less than SlipStream's 85%.[2]

Difficult-path SSMT (DP-SSMT) [9] extracts dependence chains at runtime to pre-compute hard-to-predict branches. However, DP-SSMT requires a trigger instruction to begin each instance of the dependence chains. In contrast, Branch Runahead generates dependence chains that can execute continuously, as if they were in a loop. This allows Branch Runahead to run farther ahead than DP-SSMT. Furthermore, Branch Runahead considers affector and guard branches, which enable Branch Runahead to run ahead for longer intervals with high accuracy. DP-SSMT, on the other hand, generates dependence chains that only work if control goes down a predefined path.

Dependence Chains for Prefetching. Hashemi et al. propose using dependence chains for data prefetching [15, 16]. While both papers show the predictive power of dependence chains, Branch Runahead utilizes affector/guard branches and frequent synchronizations to improve the accuracy of the dependence chains. Carlson et al. [7] propose a new way of extracting dependence chains different from the method we use. Naithani et al. [26] use a similar technique, but instead of running chains on a separate pipeline, they issue chains during cycles where the core is idle. Both works rely on branch prediction to generate the correct dependence chain, making the techniques less useful as a branch prediction alternative.

[2]The Slipstream numbers are values reported in [35], which used a slightly different set of benchmarks. However, there are several benchmarks that do overlap which confirm the large disparity.

2.3 Limitations of Heavy-weight Helper Threads.

Prior work requires helper threads to execute on another core [21, 36] or SMT context [8, 9, 31, 38] because helper threads still contain complex control flow that requires expensive out-of-order hardware to execute quickly. Requiring a separate core doubles the hardware cost for a single thread, and adds non-trivial latency for core-to-core communication. Requiring a separate SMT context requires all helper thread instructions be fetched, decoded, renamed, and access many other structures, like the Re-order Buffer (ROB), Load-Store Queue, etc.

In contrast, Branch Runahead guarantees the simplicity of dependence chains. Dependence chains are stored in the Dependence Chain Cache as a sequence of micro-ops, so they do not need to be decoded. Further, most communication is internal to the dependence chain, allowing us to break Rename into 2 phases: a one-time local rename, and a dynamic global rename. This optimization also reduces the cost of reservation stations and physical registers. These simplifications motivate the creation of the Dependence Chain Engine (DCE): a dedicated unit for executing dependence chains.

3 MOTIVATIONAL EXAMPLE

History-based branch predictors struggle with data-dependent branches because the branch outcome is not correlated to the branch history. Dependence chains, however, are a good fit for data-dependent branches because they use the application's own code to compute the direction of the branch.

Using dependence chains to predict branches. Figure 4a shows a code snippet taken from *leela*, a benchmark in the SPEC 2017 [3] benchmark suite. The code contains two hard-to-predict branches, A and B, which are two of the most frequently mispredicted branches in the benchmark. Branch A loads data from a random location on a GO board, then inspects it to see if the location is empty. The branch is hard to predict because it is a data-dependent branch with no correlated branches in the history.

Instead of trying to predict branch A, Branch Runahead extracts all instructions required to compute the outcome of the branch, forming the dependence chain for branch A. Figure 4b shows the assembly code generated by the compiler. Branch Runahead performs a backwards dataflow walk on branch A to find all instructions required to produce its outcome. These instructions (marked with the letter A) are included in the dependence chain for branch A. The resulting dependence chain is shown in Figure 4c.

Once extracted, we speculate the dependence chain can be used to compute future predictions. The dependence chain is shipped to the Dependence Chain Engine where it is executed continuously, as if it were in a loop. This process generates accurate predictions for the duration of the **for** loop (until i reaches 8). Once that happens, the dependence chain diverges from the main thread, because it continues to assume that it is in a loop. This will cause future predictions for branch A to be inaccurate. Once this is detected,

```
pos = position on GO board
for (i = 0; i < 8; i++) {
  sq = pos + neighbor_offset[i]
  if ( board[sq] == EMPTY ) // Branch A
    if( not board[sq].self_atari() ) // Branch B
      do_work()
}
```
(a)

```
A LOOP  mov    (%rbx), %ebp
A       add    %r14d,  %ebp
A       movslq %ebp,   %rax
A       cmpl   $0x2,0x6f0(%r12,%rax,4)
A       jne    SKIP        ; Branch A
B       movzwl 0x1ba4(%r12,%rax,2),%edx
B       mov    %edx,   %eax
B       sar    $0x8,   %eax
B       and    $0x7,   %eax
B       cmp    $0x1,   %eax
B       jle    SKIP        ; Branch B
        call   do_work
A SKIP  add    $0x4,  %rbx
        cmp    %rbx,  %r13
        jne    LOOP
```
(b)

Tag: $< A, * >$

```
add    $0x4,   %rbx
mov    (%rbx), %ebp
add    %r14d,  %ebp
movslq %ebp,   %rax
cmpl   $0x2,0x6f0(%r12,%rax,4)
jne    SKIP        ; Branch A
```
(c)

Tag: $< A, NT >$

```
movzwl 0x1ba4(%r12,%rax,2),%edx
mov    %edx,   %eax
sar    $0x8,   %eax
and    $0x7,   %eax
cmp    $0x1,   %eax
jle    SKIP        ; Branch B
```
(d)

Figure 4: Code snippet from *leela*, a SPEC 2017 benchmark [3].

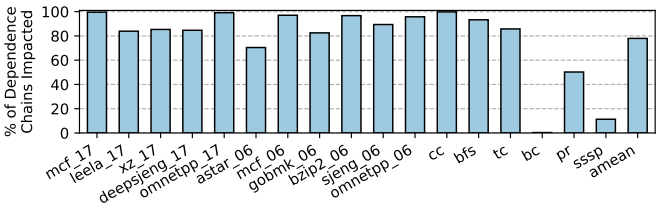

Figure 5: Dependence Chains with Affectors or Guards

the core will synchronize with branch A's dependence chain and resume its execution.

Affector and Guard Branches. Frequent synchronizations inhibit Branch Runahead's ability to stay ahead of the main thread. Therefore, we need the dependence chains to remain accurate long enough for the Dependence Chain Engine to run ahead of the main thread. This requires Branch Runahead to be aware of frequently changing branches that may affect the instructions in the dependence chain.

We classify such branches into two groups. *Guard branches* are branches that control the execution of another branch. For example, in Figure 4a, branch B is *guarded* by branch A and only occurs when branch A is not-taken. *Affector branches* are branches that can affect the source data of other branches; i.e., the direction of one branch could have an impact on the data-dependencies of another branch. Both affector and guard branches can have a serious impact on the accuracy of dependence chains if ignored. Figure 5 shows the percentage of dependence chains in SPEC 2017 [3] that are impacted by affectors and guards.

As affector and guard branches impact such a large fraction of dependence chains, it is important that dependence chains be aware of affector and guard relationships. Branch Runahead uses our new dynamic merge point predictor to identify the merge point[3] of a branch at runtime. Once the merge point is known, it can be used to detect affector and guard relationships. This process will be discussed in detail in section 4.4.

What does the dependence chain for branch B look like? Similar to branch A, the chain extraction process starts by doing a backward dataflow walk starting at branch B. However, because branch A guards branch B, the chain extraction process terminates once branch A has been reached. Additionally, the dependence chain is tagged $< A, NT >$, representing the Program Counter (PC) of branch A, as well as the branch outcome (of branch A) required to execute branch B. The resulting dependence chain is shown in Figure 4d.

Tags are used to identify the action which initiates the execution of the dependence chain. For example, any time branch A is not-taken, that will match with the tag $< A, NT >$ and the dependence chain for branch B can begin executing. Additionally, the dependence chain for branch A is tagged as $< A, * >$. The '*' denotes a wildcard, meaning that any outcome of branch A will match this tag and initiate branch A's dependence chain.

Biased branches and memory address aliasing. Branch Runahead assumes that highly biased branches will remain biased and ignores them during chain extraction, even if a biased branch is also an affector or guard branch. Additionally, Branch Runahead assumes that memory address aliasing (i.e., overlapping memory addresses) between store-load pairs will also persist. These assumptions are, of course, not always true and can cause the dependence chains to diverge from the main thread. For example, once the **for** loop in Figure 4a terminates, the dependence chain for branch A will no longer be valid and any predictions it generates

[3]The merge point of a branch is the instruction where control converges regardless of the true direction of the branch.

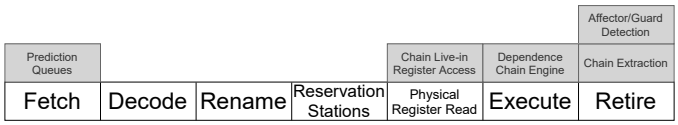

Figure 6: Pipeline Modifications

will likely be incorrect. Dependence chains will be deactivated when a misprediction is detected.

Putting it all together. Once the chains for branch A and B have been extracted, they are installed in a chain cache located in the Dependence Chain Engine. Newly installed chains cannot start executing until the core synchronizes the chain by initializing the chain's local register file with the correct input data. This does not happen until the next time either branch A or branch B mispredicts. When a misprediction occurs, the correct direction of the branch is broadcast to the Instruction Fetch Unit. Any chains whose tag matches the branch address and outcome are activated. At this point, newly activated chains copy their live-ins from the physical register file and begin execution. Finally, when the dependence chain finishes execution, the branch address and outcome produced by the chain are used to identify the next set of chains. The dependence chains continue to execute, completely asynchronously from the core, until a misprediction from the dependence chains is detected. At that point, the mispredicting chain is synchronized by copying the correct values from the physical register file and chain execution resumes.

4 BRANCH RUNAHEAD

Figure 6 shows a summary of the changes Branch Runahead requires on top of a typical Out-of-Order pipeline. We break these changes up into three categories: 1) dependence chain extraction, the process of identifying hard-to-predict branches and the uops that belong to their dependence chains. Once extracted, dependence chains are stored in the dependence chain cache. 2) Dependence Chain Control synchronizes the dependence chains with the core and allows them to execute continuously, producing near perfect branch predictions, which are used instead of predictions from the TAGE-SC-L predictor. Finally, the chains are executed on 3) the Dependence Chain Engine (DCE), which is a specialized unit designed to execute dependence chains more efficiently than is possible on the core. Figure 7 shows a block diagram of the DCE. This section discusses Dependence Chain Control, followed by the microarchitecture of DCE, then concludes with dependence chain extraction and affector/guard detection.

4.1 Dependence Chain Control

Entering Runahead Mode. Once the dependence chains have been extracted, they are copied to the dependence chain cache where they wait to be *initiated*. Dependence chains cannot be initiated until their live-in data is synchronized with the core. Branch mispredictions present a convenient time to perform this synchronization, as the core backend and frontend are synchronized. When the core detects a branch misprediction, the branch address and outcome, which together form a tag, are used to look up an entry in the chain cache. If there is a hit, then the matching dependence chain is *initiated*; i.e., its uops are written into the reservation stations and its live-ins are copied from the core's physical register file. Additionally, the corresponding prediction queue is synchronized with instruction fetch. Once complete, the chain may begin execution.

Continuous Execution. Once initiated by the core, dependence chains execute continuously by using the dependence chain outcome to initiate future dependence chains. The aggressiveness of chain initiation impacts the level of chain level parallelism, which impacts the timeliness of the computed branch outcomes. In this paper, we evaluate three initiation techniques.

Non-speculative Initiation. A dependence chain must finish execution entirely before initiating the next dependence chain. Once the dependence chain has finished execution, the pre-computed branch outcome and the branch address are used to index the chain cache and initiate all matching chains. Chain level parallelism is minimized in this mode as dependence chains must wait for their predecessors to finish execution before they can begin processing.

Independent-early Initiation. Chains with a wildcard tag are initiated when a triggering branch is issued into the reservation station. Recall from section 3 that some chains are marked with a wildcard tag, indicating that the direction of the triggering branch does not matter. In this case there is no need to wait for the predecessor chain to finish execution as the result of the branch will not affect whether or not the wildcard chain is initiated. Instead, wildcard chains are initiated as soon as their predecessor chains finish initiation. This mode increases chain level parallelism as chains with wildcard tags can now execute in parallel[4]. Non-wildcard chains, however, must wait for the predecessor chain to finish execution.

Predictive Initiation. Before, chains with non-wildcard tags required the direction of the triggering branch to be known in order to initiate them; however, in this mode the outcome of each branch is predicted at initiation time. This allows the branch address and the predicted outcome to index the chain cache and initiate non-wildcard matching chains early. If the prediction turns out to be incorrect, the speculatively initiated chains are simply flushed and the correct chains are initiated in their place. This mode maximizes chain level parallelism when the predicted branch outcomes are correct, or, in the case of mispredictions, results in chains being initiated no later than they would have in the prior two modes. In this mode, wildcard chains are handled exactly as they were in Independent-early Initiation. It is important to note that the prediction is only used to increase chain level parallelism; thus, any level of accuracy will likely improve the timeliness of branch outcomes. We use a simple per-branch 3-bit counter as the prediction mechanism.

In all three modes, chain execution continues as long as the produced tags continue to hit in the chain cache. If ever the produced tag misses, then there are no more chains to initiate and runahead mode is exited.

Detecting Dependence Chain Divergence. Unfortunately, dependence chains will eventually diverge from the main thread. When this happens, the dependence chains will begin to produce incorrect predictions. Branch Runahead monitors all prediction

[4]Note that initiation simply affects when the chain is added to the reservation stations. The individual uops within the dependence chain are still required to wait until their data dependencies have been satisfied before they are scheduled to the functional unit.

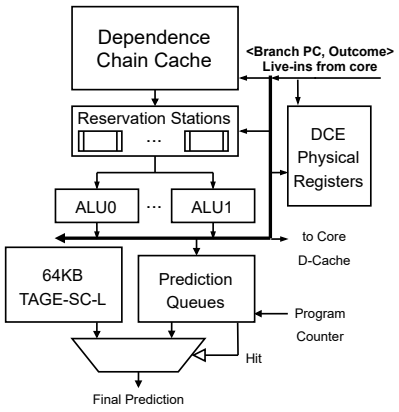

Figure 7: DCE Microarchitecture

produced by dependence chains. Once a misprediction is detected, the chain is synchronized by once again copying the live-in values from the core to the DCE.

4.2 DCE Microarchitecture

The microarchitecture of the DCE is specialized to execute dependence chains more efficiently than the core. This is accomplished by observing that most of the communication between uops occurs within the dependence chain. Tailoring the microarchitecture of the DCE around this allows us to reduce the number of ports required for physical registers and reservation stations, in turn reducing the cost-per-entry of those structures. Alternatively, we also evaluate a *Core-Only* implementation of the DCE, which shares physical registers, reservation stations, and functional units with the core. This section covers each component of the DCE microarchitecture, how it works, and what utility it provides towards improving branch prediction.

Dependence Chain Cache. Once the dependence chains have been extracted, they are copied to the dependence chain cache where they wait to be executed. The dependence chain cache holds up to 32 dependence chains and uses LRU as a replacement policy.

Rename and Instruction Scheduling. Instruction scheduling happens at two levels of granularity: global (i.e., initiating the dependence chains, discussed in section 4.1) and local (i.e., scheduling the uops within a chain). To accomplish this, Branch Runahead uses two levels of rename. Local rename happens once, during chain extraction, where communication within the chain is assigned a local physical register. Global rename happens dynamically, when the dependence chain is initiated, and results in assigning the dependence chain to a local register file and reservation station.

Physical Registers. The physical register file is divided into several local register files (Figure 8). Each local register file is treated as an independent, single-ported bank. Dependence chains read and write to their own local register file, except for live-in values [5], which are read from the local register file of the producer chain, creating the possibility of a bank conflict.

Reservation Stations. The reservation stations are divided into several local reservation stations, each with a capacity of 16 uops

[5]Values produced by another dependence chain or the core.

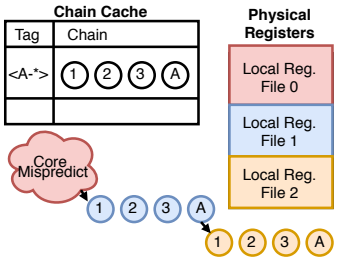

Figure 8: Global Rename Example

(1 chain). As uops are executed, their results are broadcast to the physical register file and to the reservation stations. Once all sources of a uop are ready, then the uop may be scheduled for execution. Uops are scheduled to execute out-of-order. We experimented with in-order instruction scheduling; however, we found that in-order execution was not able to expose enough Memory Level Parallelism (MLP) to significantly benefit the dependence chains.

Instruction Window. The physical register file and reservation stations together form the instruction window. The number of local register files and reservation stations directly affects the number of dynamic chain instances that are actively executing at once. Increasing the window size allows for more chain level parallelism; however, it can come at a significant cost. Alternatively, the entire instruction window can be shared with the core, minimizing cost, but also minimizing chain level parallelism. See section 5.2 for a detailed analysis.

Figure 8 shows an example. Here, a branch misprediction from the core triggers the tag $< A, * >$. The misprediction also triggers a synchronization between the DCE and the core. The chain's live-ins are read from the core physical registers and copied to a new local register file (red). While the live-ins are being copied, the matching dependence chain is issued into a reservation station and allocated a new local register file (blue). The chain's source register file is set to red. Once the branch is issued, its address is broadcast to the chain cache to initiate any matching wildcard tags. In this example, the same chain matches, which triggers another dynamic instance of the chain to be issued (orange) and the chain's source registers are set to blue. Once the core has copied all live-ins to the red register file, the ready-bits in the physical register file will be set and the reservation station will be notified. At this point, the chain may begin executing. As uops execute, their results are written back to the appropriate local register file and the reservation station is notified.

Memory Accesses. The DCE shares the D-Cache and D-TLB with the core. The main thread is given priority to the D-Cache and D-TLB ports, and the DCE may only use these structures when available. Dependence chains do not contain any store instructions (see section 4.3), so the main thread does not have to worry about data corruption by the dependence chains.

The Prediction Queues. The prediction queues ensure that predictions between the DCE and core are synchronized. The size of each prediction queue also limits how far ahead (or behind) the DCE can be, which can affect performance. The DCE contains 16 per-branch prediction queues. When a dependence chain is

initiated, a slot in the corresponding prediction queue is allocated. [6] Finally, when the dependence chain finishes execution, it pushes the outcome of the branch into the prediction queue. When the branch is fetched, it will see that its queue contains a prediction and will use that result instead of TAGE-SC-L. However, if the core fetches the branch before the DCE has computed the next prediction, then the slot is marked as consumed, even though it has not yet been filled. Later, when the DCE finishes computing the prediction, the already consumed slot in the prediction queue will be filled in case there is a recovery. To maintain the state of the queue, we use three pointers: *DCE push* for inserting new predictions, *core fetch* for consuming predictions at fetch, and *core retire* for removing retired predictions.

Recovery. During a branch recovery, the core fetch pointer is restored to its state before the misprediction, effectively reinserting previously consumed predictions into their original positions in the queue. This is accomplished by checkpointing the state of the core fetch pointer at each branch.

Prediction Throttling. Each prediction queue has a 2-bit throttle counter, which is incremented when the DCE is correct and TAGE is incorrect and decremented when the DCE is incorrect and TAGE is correct. When the counter is negative, predictions produced by the DCE are ignored.

4.3 Chain Extraction Hardware

Detecting Hard to Predict Branches. A new structure, the *Hard Branch Table (HBT)* (Figure 9), detects hard-to-predict branches. New entries are allocated when a conditional branch retires (if space available). Each entry consists of a 5-bit saturating misprediction counter that is incremented upon retiring a mispredicted branch. A branch is considered hard-to-predict when its misprediction counter saturates. Misprediction counters are periodically decremented by 15 every 1000 retired branches.[7], and old entries can be overwritten when their counter is 0.

Tracking Affector and Guard Branches. In addition to identifying hard-to-predict branches, the HBT also keeps track of affectors and guards. Once detected (section 4.4), affector/guard branches are allocated in the HBT. The *AG* field is set to indicate the branch is an affector/guard, which allows the branch to remain in the table even if it is not hard-to-predict. Affector/guard branches can only be replaced when the hard-to-predict branch they are associated with is removed. In addition, the affector/guard branch is added to the *affector/guard list (AGL)* field of the hard-to-predict branch.[8] If this branch was not previously in the affector/guard list, then the *affector/guard changed (AGC)* field is set, indicating that a new affector/guard branch was found.

Branch Runahead ignores highly biased affector/guard branches. Therefore, the HBT tracks the bias of each affector/guard branch using a 7-bit bias counter, which is incremented when the direction of a retired branch matches the direction stored in the *Biased Direction (BD)* field and is decremented by 9 periodically[9]. If an affector/guard branch is found to be biased, then it is removed from the hard-to-predict branch's affector/guard list, and the AGC field is set if appropriate.

Extracting the Dependence Chain. The chain extraction algorithm is adapted from Hashemi et al. [16] where the authors use dependence chains to create prefetches for load instructions. Chain extraction begins when a hard-to-predict branch is retired.[10] Chain extraction terminates when either 1) a second instance of the same branch is found, or 2) an affector/guard branch is found. Upon termination, the dependence chain is tagged with the PC and outcome of the terminating branch and installed into the dependence chain cache.

Chain extraction performs a backwards dataflow walk starting at the most recently retired hard-to-predict branch instruction. To facilitate this, we add a circular buffer, called the chain extraction buffer (CEB), that holds the last 512 micro-operations (uops) retired. Uops in the CEB are searched cycle-by-cycle to see if they belong to the dependence chain. Figure 9 shows an example. In cycle 0, the second, younger dynamic instance of the mispredicting branch (shaded in grey) is added to the dependence chain and the search list (LIV) is initialized with the live-in table entry associated with this branch. Additionally, all of the branch's source registers (i.e., the condition code register) are added to the search list. Then, the CEB is iteratively scanned for uops whose destination register(s) match in the search list. On a match, we 1) add the matching uop to the dependence chain, 2) remove the matching register(s) from the search list, and 3) add the source registers of the matching uop to the search list. In cycle 1, the CEB is scanned for uops that write the condition codes, resulting in the addition of the CMP uop to the dependence chain and its source registers added to the search list. In cycle 2, the CEB is scanned for uops that write to R0. This results in the LD uop being added to the dependence chain, P0 being removed from the search list, and the sources of the LD (P2) added to the search list. When a load op is found, its address is compared to other addresses in the CEB store buffer. If a corresponding store op is found, then the store is also added to the dependence chain. In cycle 3, the MOV uop is added to the dependence chain because of its write to P2. Finally, in cycle N, we reached the initial instance of the branch, ending chain extraction. The shaded uops make up the final resulting dependence chain. Several uops in the CEB were omitted because they are not added to the dependence chain.

Chain extraction takes place one chain at a time, off the critical path, and is not latency sensitive. The process takes multiple cycles, as shown in Figure 9. In all of our experiments, we model this latency accurately [11]; however, we experimented with much longer latency (1000s of cycles) and found no sensitivity. This is because chain extraction very rarely produces a chain that is not currently in the chain cache.

Live-in and Live-out Tables. The live-in and live-out tables hold the architectural live-ins/outs for each chain detected during

[6] Slots must be allocated at initiation to ensure they appear in the prediction queue in program order.

[7] The decrement amount and counter width was calculated using a binomial distribution that theoretically detects branches that make up at least 1.5% of the total misprediction rate with a 1% false positive rate.

[8] The AGL is stored as a bit-vector, 1 bit per entry in the HBT.

[9] The decrement amount and counter width was calculated using an arithmetic model that theoretically detects a bias of 90% or more with a false positive rate of 1%.

[10] The branch must be contained in the HBT and have either saturated its misprediction counter or be randomly selected. Branches are randomly selected with a 1% probability.

[11] uops in CEB / retire width

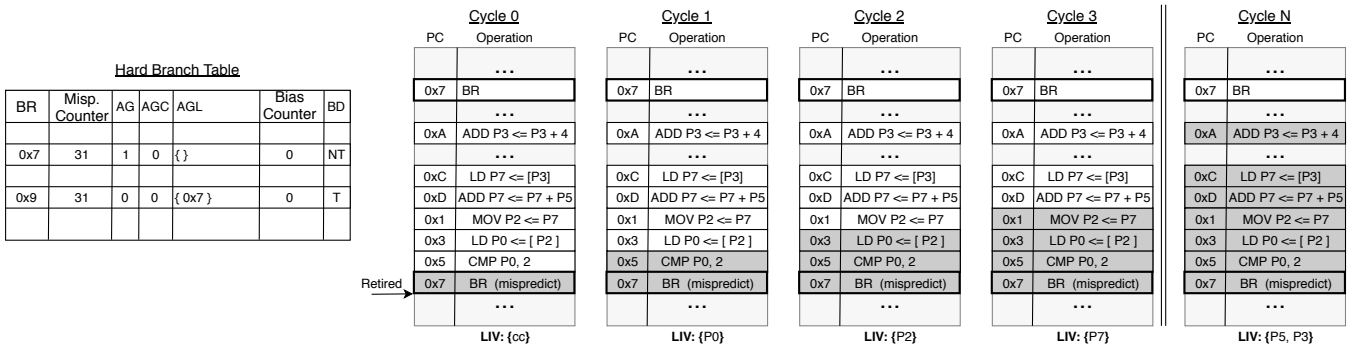

Figure 9: The Chain Extraction Buffer (CEB)

Table 1: Baseline Configuration

Core	4-Wide Issue, 256-Entry ROB, 92-Entry Reservation Station, 3.2 GHz, 64KB TAGE-SC-L Branch Predictor [32]. Modeled by Scarab [2].
WPB	128-entry, 4-way, max merge point distance 256 uops.
L1 Caches	32 KB I-Cache, 32 KB D-Cache, 64 Byte Lines, 2 Ports, 3-Cycle Hit Latency, 8-Way, Write-Back.
L2 Cache	2 MB 12-Way, 18-Cycle Latency, Write-Back.
Memory Controller	64-Entry Memory Queue.
Prefetchers	Stream: 64 Streams, Distance 16. Prefetch into LLC.
DRAM	DDR4, 8Gb, x8, 2400R, Modeled by Ramulator [20].

Table 2: Branch Runahead Configuration

	Core-Only (9KB)	Mini (17KB)	Big (Unlimited)
uOps	Integer: add/multiply/subtract/mov/load. Logical:and/or/xor/not/shift/sign-extend.		
Chain Cache	32-entry (2KB) 1 uop per entry, 4B per uop.		1024-entry
PRF	0 (0KB)	64x 8-entry (4KB)	1024x 8-entry
RSV	0 (0KB)	64x 32-entry (4KB)	1024x 32-entry
MSHRs	48-entry	48-entry	64-entry
Prediction Queue	16x 256-entry (4KB)		1024-entry
HBT	64-entry (1KB)		
CEB	512-entry (2KB)		2048-entry

chain extraction. In Figure 9, the architectural registers corresponding to the live-in vector (LIV) are stored in the live-in table at the end of chain extraction. Further, the Live-out vector, i.e., architectural registers corresponding to P0, P2, P3, P7, is written into the live-out table. This information is used during local rename and during synchronizations with the core.

Local Rename. Local rename happens once during chain extraction. During this process local registers (i.e., communication within a chain) are renamed and global registers (communication between chains) are identified and partially renamed to prepare for global rename. Local registers are renamed to minimize physical register footprint. Global registers are identified by comparing the live-in/live-out registers of the chain with the live-out/live-in registers of the producer/consumer branches, respectively. Global registers are renamed in-order to guarantee the same name is used between different chain extractions.

Dependence Chain Optimizations. All move uops are move-eliminated. Further, because store-load pairs detected during chain extraction are logically equivalent to a move, all store-load pairs are move eliminated as well. This optimization guarantees that dependence chains will not contain any store instructions.

4.4 Detecting Affector and Guard Branches

Branch Runahead uses a new merge point prediction algorithm to detect control and data dependencies between dependence chains. Prior work in merge point prediction makes assumptions about code layout [10, 11], which results in a lower accuracy and coverage [29]. Our algorithm leverages branch mispredictions, comparing the wrong path and correct path of the branch to see where they intersect.

Merge Point Prediction. Due to high fetch rates and the large instruction windows, it is common for modern processors to fetch hundreds of wrong path instructions before detecting a misprediction[12]. This suggests the merge point is likely fetched and allocated while going down the wrong path. Thus, to detect the merge point, we can save wrong path instructions, and compare them to instructions retired on the correct path. The first instruction that appears both in the wrong path and in the correct path is the predicted merge point.

To find the merge point, we add a new structure to retirement called the Wrong Path Buffer (WPB). The WPB is a 128 entry, 4-way set associative cache that stores wrong path program counters (PC). When a flush is triggered, the Reorder Buffer (ROB) contains wrong path instructions. Those instructions, starting with the first instruction after the mispredicted branch, are copied from the ROB to the WPB by conducting a forward ROB-walk[13]. This process takes multiple cycles as instructions are copied over in batches. [14] The ROB walk typically completes when a second dynamic instance of the mispredicting branch is found, indicating that we are in a loop. If the second dynamic instance of the branch is not found, the ROB-walk terminates when the tail of the ROB has been reached, or when the maximum merge point distance has been exceeded (100 uops in our experiments). When the copy is complete, the WPB is

[12]We measured an average of 100 wrong path uops fetched on workloads in SPEC CPU2006

[13]We do not expect the latency of the ROB-walk to be an issue. ROB-walks are commonly used during a flush to restore the state of the speculative register alias table. Their latency is typically hidden by the front-end as it refills the pipeline.

[14]In our experiments, we assume that we can copy instructions to the WPB at the same rate instructions can be retired.

marked as valid, indicating that correct path instructions should be compared to its contents. After recovery, retired correct path instructions use their PC to index the WPB. If there is a hit, then we have found a PC that is common to both the wrong path and the correct path, i.e., a merge point. If the second instance of the branch is retired on the correct path or the maximum merge point distance is exceeded before finding a merge point, then the process terminates and the WPB is invalidated.

In addition to the merge point, the WPB also supplies a bit vector indicating all architectural destination registers that are written on either side of the branch. We refer to this vector as the dest set. To create this, we store a dest set at each entry in the WPB that indicates the destination registers seen up to that point on the wrong path. A bloom filter is used to track destination addresses of memory writes. We also use a single correct-path dest set to accumulate this information as correct path instructions retire. Once the merge point has been found, we take the logical OR of the hitting wrong-path dest set and the correct-path dest set. This results in a single dest set that indicates registers written on either side of the branch. We refer to this as the both-path dest set.

Detecting Guard Branches By definition, a branch guards any branches that are observed between itself and the merge point (excluding biased branches). Therefore, any branches observed on the wrong-path or correct-path during merge point prediction are marked as being guarded by the merge predicted branch. If no merge point is found, then no branches are marked.

Detecting Affector Branches The merge predicted branch is an affector for any branch that sources data that is affected by the direction of the merge predicted branch. Fortunately, the merge point predictor provides us with the both-path dest set, which marks all registers/memory addresses affected by the merge predicted branch. To detect affectee branches, we use the both-path dest set and the poison algorithm adapted from Runahead Execution [25]. Registers and memory addresses marked in the both-path dest set are initialized as poisoned. As correct path instructions after the merge point retire, they propagate any poison they source to the destination register. If an instruction outputs to a poisoned register without sourcing any poisoned registers, then the destination register poison is removed. Any branch, including the merge predicted branch, that sources poison is considered to be an affectee branch [15] (i.e., the merge predicted branch is an affector of the poison sourcing branch). The process terminates when the second instance of the merge predicted branch is seen or when the maximum merge point distance has been reached.

5 RESULTS

5.1 Evaluation Methodology

To simulate our proposal, we use Scarab [2]— an open source simulator commissioned by Intel in their Intel/NSF FoMR initiative [1]. Scarab is an execution-driven, cycle-accurate x86 simulator whose front-end is based on PIN [23]. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed non-uniform access latency DDR4 memory system, modeled by Ramulator [20]. We model the 64KB TAGE-SC-L [32] branch predictor with the configuration submitted to CBP-2016. The 64KB TAGE-SC-L is the best known realistic branch predictor. Table 1 describes our system.

[15] Excluding biased branches.

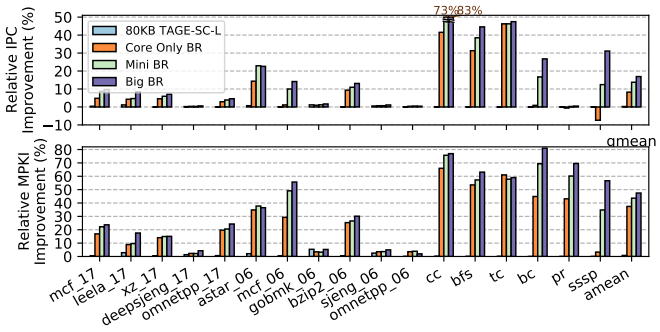

Figure 10: IPC and MPKI Improvement of Branch Runahead compared to 64KB TAGE-SC-L

Branch Runahead Configuration. Branch Runahead is evaluated on three configurations— Core-Only (9KB), Mini (17KB), and Big (unlimited). Table 2 contains the details of each configuration.

Benchmarks. We evaluate Branch Runahead on SPEC CPU2017 Integer Speed, SPEC CPU2006 Integer [3] and GAP Benchmark Suites [6]. From that set, we select the branch misprediction intensive benchmarks with an average MPKI greater than 2. We use the SimPoints [27] methodology to identify anywhere between one to five representative regions per benchmark. We run each region for 200 million instructions, then compute the weighted average of all the regions. We run SPEC benchmarks on the ref input set, and use *-g 19 -n 300* inputs for GAP. If there is more than one ref input, then the benchmark is run on each input, and a weighted average, weighing each input by the total dynamic instruction count, is used to compute a single metric for the entire benchmark.

Energy and Area. We model chip energy and area using McPAT [22]. The DCE is modeled as a stripped down core, removing structures like decode, register rename, floating point pipeline, prefetchers, and others required for maintaining precise state, such as the ROB.

Metrics. We use Instructions Per Cycle (IPC) as the performance metric and Branch Mispredictions Per Kilo Instruction (MPKI) to evaluate improvements in prediction accuracy. MPKI Improvement is computed as the difference between Branch Runahead MPKI and TAGE-SC-L MPKI, normalized to TAGE-SC-L MPKI.

5.2 Branch Runahead Results

Figure 10 shows the performance results for the Core-Only, Mini, and Big implementations of Branch Runahead. The results show that Branch Runahead reduces MPKI by an average of 37.5%, 43.6%, and 47.5% and increases IPC by an average of 8.2%, 13.7%, and 16.9%, respectively. The trade-off comes down to cost vs chain level parallelism. Big-Branch Runahead maximizes chain level parallelism by providing the most physical registers and reservation station entries, while the Core-only model minimizes these same qualities.

Figure 10 also shows the performance of an 80KB TAGE-SC-L predictor (left most bar), which requires roughly the same storage overhead as Mini Branch Runahead (16KB). However, the 80KB

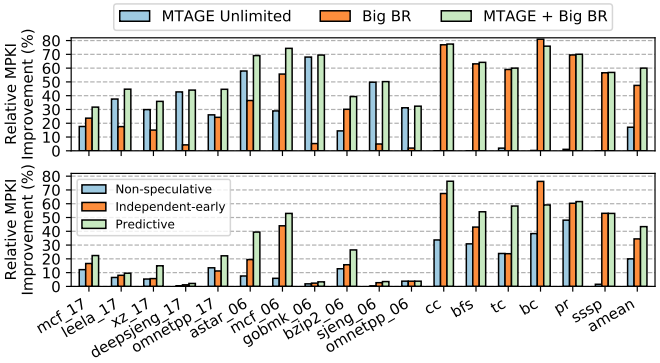

Figure 11: MPKI Improvement of MTAGE and Branch Runahead (top) and MPKI Improvement of Chain Initiation Methods (bottom)

TAGE-SC-L only improves MPKI by 0.8%, resulting in only 0.3% improvement in IPC. Mini Branch Runahead improves the misprediction rates of targeted branches by an average of 55%, while 80KB TAGE-SC-L has a negligible effect on these same branches. This result supports the claim that history-based predictors are fundamentally unable to predict these data-dependent branches.

Limits of Branch Runahead. Big Branch Runahead uses unlimited storage to demonstrate the maximum potential of Branch Runahead. In this model, parameters of the microarchitecture are increased far beyond their reasonable limits. As Figure 10 shows, Big Branch Runahead improves MPKI over Mini Branch Runahead by only 3.8%, suggesting that Mini Branch Runahead is very close to its peak potential.

Limits of History-based Predictors. Figure 11 (top) compares Big-Branch Runahead to MTAGE-SC, winner of the unlimited storage category in CPB-2016 [33]. While MTAGE-SC improves MPKI significantly, particularly in the SPEC workloads, it is still outperformed on average by Big Branch Runahead (middle bar). This is because MTAGE-SC performs poorly on GAP workloads, which are dominated by data-dependent branches. Combining MTAGE-SC and Big Branch Runahead (right bar) further improves MPKI on every benchmark, demonstrating that Branch Runahead is capable of predicting branches that TAGE fundamentally cannot predict.

Chain Initiation Method. Chain Initiation is the most important factor towards improving timeliness. As discussed in section 4.1, Chain Initiation affects chain level parallelism. Figure 11 (bottom) shows the MPKI improvement of each initiation method. Unsurprisingly, the Predictive Initiation method, which maximizes chain level parallelism, has the highest impact on MPKI. While this method does produce the highest degree of performance, it comes at the cost of flushing the DCE on a misprediction, which wastes energy.

Timeliness of predictions. DCE predictions need to be timely in order to be useful. The stacked bars in Figure 12 show the fraction of predictions supplied by the DCE that are inactive or late. The inactive category means that, at the time the core needed the prediction, no dependence chains had been activated to produce that prediction. This generally happens once the branch is fetched, but before the first synchronizing misprediction occurs. Branch

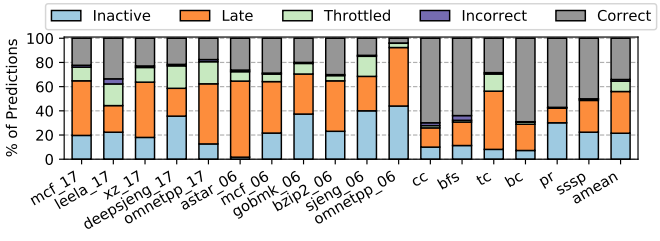

Figure 12: Prediction Breakdown

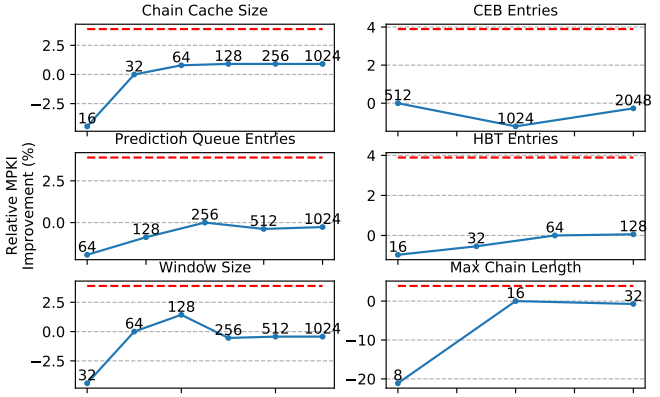

Figure 13: MPKI Improvement relative to Mini Branch Runahead. Parameters are swept individually up to the level of Big Branch Runahead (Red dotted line) to show each parameters contribution.

Runahead requires a mispredicted branch to synchronize, which unfortunately has the effect of activating chains late. The late category refers to predictions which have active chains, but are generated too late to be useful for the core. This generally happens when the dependence chain contains too many long latency operations. The throttle category refers to predictions that are throttled (as described in section 4.2). The remaining two categories, correct and incorrect, refer to predictions that are used by the core. This figure demonstrates two things. First, predictions generated by Branch Runahead are very accurate, with nearly all of the used predictions being correct. Second, nearly 40% of predictions are generated on time. Timeliness is the most difficult issue Branch Runahead faces, with late predictions making up the largest category outside of correct predictions.

Sweeps. Figure 13 shows MPKI improvement for a DCE with various configurations [16]. Parameters are swept up to the values used for the Big Branch Runahead configuration. On average, Big Branch Runahead improves the MPKI of Mini Branch Runahead by 3.89%. The figure suggests that this improvement is primarily due to the increased window size and chain cache size. Furthermore, the graph suggests that optimal values for window size and chain cache would be 128-entry and 64-entry respectively, meaning that

[16]Due to the large number of simulations, sweeps ran for 10 Million instructions, as opposed to the 200 Million used for all other experiments.

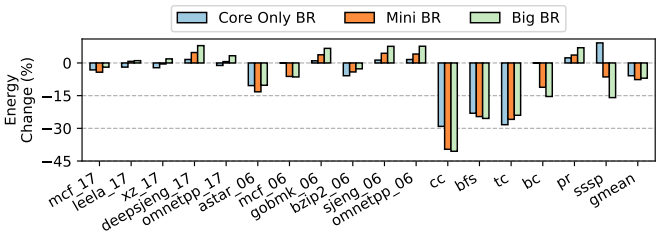

Figure 14: Energy Impact (Lower is better)

Big Branch Runahead could be implemented using 27KB of total storage.

Energy. Figure 14 shows the change in energy due to Branch Ruanhead, as estimated by McPat. Branch Runahead decreases energy on average, primarily due to faster run times. However, Branch Runahead does increase energy usage in two ways. First, there is the increase in static and dynamic power that is generated by new structures. Second, Branch Runahead increases the total number of instructions executed and memory accesses. Figure 3 shows the total increase in both ops executed and total memory accesses.

Area. McPat estimates the DCE engine area to be $0.38mm^2$, or about 2.2% of a baseline out-of-order core ($16.96mm^2$ at a 22nm process). Of this total, $0.09mm^2$ is dedicated to the dependence chain cache, $0.15mm^2$ is dedicated to functional units, reservation stations, and physical registers, and $0.14mm^2$ is dedicated to chain extraction and the HBT. [17]

Impact on clock frequency. Branch Runahead minimally affects clock frequency, as almost all units are off the critical-path and are not sensitive to latency. The only component on the critical path is a MUX, which selects between TAGE-SC-L and the DCE engine prediction queues. The DCE executes the dependence chain across many cycles, off the critical path, and inserts the result into a prediction queue. Therefore, processor throughput is minimally impacted.

6 RELATED WORK

Gupta et al. [14] target dependence chains that contain one load instruction with a predictable address. While this technique is effective for a subset of branches, Branch Runahead is a more general technique that is able to capture more benefit. Their targeted approach does simplify some hardware (no affector/guards, simplifies chain scheduling), however much of the same hardware is needed to execute the dependence chains.

Farooq el al. [12] propose Store-Load-Branch (SLB) predictor, which predicts data-dependent branches by identifying dependent store-load-branch chains in a program using the compiler. Gao et al. [13] propose a new predictor that targets data-dependent branches by correlating a load's memory address with the result of an upcoming branch. The EXACT predictor [4] also targets data-dependent branches by distinguishing branch instances based on their feeder load's address.

Premillieu et al. [28] save branch results computed on the wrong-path and replay them as predictions later on the correct-path. However, this technique is limited to *control-independent/data-independent* branches that are executed in the shadow of a branch misprediction.

Ayers et al. [5] propose a new methodology to classify the memory access patterns of applications. This technique effectively categorizes dependence chains for load instructions, enabling reasoning about prefetcher timeliness and criticality.

Zangeneh et al. propose BranchNet [37] offline training of a CNN to improve prediction accuracy for hard-to-predict branches. However, this technique requires correlation between branch outcomes and history, making the technique less effective for data-dependent branches.

7 CONCLUSION

Branch Runahead represents a new opportunity to achieve high prediction accuracy on hard-to-predict, data-dependent branches. This paper demonstrates that history based predictors, like TAGE-SC-L and MTAGE-SC are not capable of predicting this category of branches. Branch Runahead, however, uses the application's own code to pre-compute the result of these branches, leading to an accuracy improvement of 55% (TAGE-SC-L) and 44% (MTAGE-SC) for branches which Branch Runahead targets.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and the members of the HPS Research Group for their feedback and help improving this paper. We would like to thank Intel, the Cockrell Foundation, Arm, and NSF Award #2011145 for their financial support.

REFERENCES

[1] [n. d.]. NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR). ([n. d.]). https://www.nsf.gov/pubs/2019/nsf19598/nsf19598.htm
[2] [n. d.]. Scarab. ([n. d.]). https://github.com/hpsresearchgroup/scarab
[3] [n. d.]. The Standard Performance Evaluation Corporation (SPEC). The SPEC Benchmark Suite. ([n. d.]). http://www.spec.org
[4] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. 2010. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/1787275.1787321
[5] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. https://doi.org/10.1145/3373376.3378498
[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619
[7] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The Load Slice Core Microarchitecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 272–284. https://doi.org/10.1145/2749469.2750407
[8] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA '99)*. IEEE Computer Society, Washington, DC, USA, 186–195. https://doi.org/10.1145/300979.300995

[17] For reference, McPat estimates the 64KB TAGE-SC-L predictor to be $0.73mm^2$. This estimate is a lower bound on the total area, as McPat does not faithfully model the interconnect, muxes, and adders contained within TAGE-SC-L predictor, which consume a non-negligible area.

[9] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. 2002. Difficult-path branch prediction using subordinate microthreads. In *Proceedings 29th Annual International Symposium on Computer Architecture.* 307–317. https://doi.org/10.1109/ISCA.2002.1003588

[10] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney. 2020. Auto-Predication of Critical Branches*. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 92–104. https://doi.org/10.1109/ISCA45697.2020.00019

[11] J. D. Collins, D. M. Tullsen, and Hong Wang. 2004. Control Flow Optimization Via Dynamic Reconvergence Prediction. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on.* 129–140. https://doi.org/10.1109/MICRO.2004.13

[12] M. U. Farooq, Khubaib, and L. K. John. 2013. Store-Load-Branch (SLB) predictor: A compiler assisted branch prediction for data dependent branches. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).* 59–70.

[13] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. 2008. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture.* 74–85. https://doi.org/10.1109/HPCA.2008.4658629

[14] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. 2020. Opportunistic Early Pipeline Re-Steering for Data-Dependent Branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20).* Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3410463.3414628

[15] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49).* IEEE Press, Piscataway, NJ, USA, Article 61, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195712

[16] Milad Hashemi and Yale N. Patt. 2015. Filtered Runahead Execution with a Runahead Buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48).* ACM, New York, NY, USA, 358–369. https://doi.org/10.1145/2830772.2830812

[17] D Jiménez. 2016. Multiperspective Perceptron Predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).*

[18] D Jiménez. 2016. Multiperspective Perceptron Predictor with TAGE. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).*

[19] D. A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture.* 197–206. https://doi.org/10.1109/HPCA.2001.903263

[20] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49.

[21] S. Kondguli and M. Huang. 2019. R3-DLA (Reduce, Reuse, Recycle): A More Efficient Approach to Decoupled Look-Ahead Architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 533–544. https://doi.org/10.1109/HPCA.2019.00064

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 469–480.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[24] Pierre Michaud. 2018. An Alternative TAGE-like Conditional Branch Predictor. *ACM Trans. Archit. Code Optim.* 15, 3, Article 30 (Aug. 2018), 23 pages. https://doi.org/10.1145/3226098

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro* 23, 6 (Nov 2003), 20–25. https://doi.org/10.1109/MM.2003.1261383

[26] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. 2020. Precise Runahead Execution. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 397–410. https://doi.org/10.1109/HPCA47549.2020.00040

[27] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '03).* ACM, New York, NY, USA, 318–319. https://doi.org/10.1145/781027.781076

[28] Nathanael Premillieu and Andre Seznec. 2012. SYRANT: SYmmetric Resource Allocation on Not-taken and Taken Paths. *ACM Trans. Archit. Code Optim.* 8, 4, Article 43 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086722

[29] Stephen Pruett and Yale Patt. 2020. Dynamic Merge Point Prediction. arXiv:cs.AR/2005.14691

[30] Stephen Pruett, Siavash Zangeneh, Ali Fakhrzadehgan, Ben Lin, and Yale Patt. 2016. Dynamically Sizing the TAGE Branch Predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).*

[31] A. Roth and G. S. Sohi. 2001. Speculative data-driven multithreading. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture.* 37–48. https://doi.org/10.1109/HPCA.2001.903250

[32] André Seznec. 2014. TAGE-SC-L Branch Predictors. In *JILP - Championship Branch Prediction.* Minneapolis, United States. https://hal.inria.fr/hal-01086920

[33] Andre Seznec. 2016. Exploring branch predictability limits with the MTAGE+SC predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).*

[34] R. Sheikh, J. Tuck, and E. Rotenberg. 2015. Control-Flow Decoupling: An Approach for Timely, Non-Speculative Branching. *IEEE Trans. Comput.* 64, 8 (Aug 2015), 2182–2203. https://doi.org/10.1109/TC.2014.2361526

[35] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg. 2020. Slipstream Processors Revisited: Exploiting Branch Sets. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 105–117. https://doi.org/10.1109/ISCA45697.2020.00020

[36] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. 2000. Slipstream Processors: Improving Both Performance and Fault Tolerance. *SIGPLAN Not.* 35, 11 (Nov. 2000), 257–268. https://doi.org/10.1145/356989.357013

[37] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt. 2020. BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 118–130. https://doi.org/10.1109/MICRO50266.2020.00022

[38] Craig Zilles and Gurindar Sohi. 2001. Execution-based Prediction Using Speculative Slices. *SIGARCH Comput. Archit. News* 29, 2 (May 2001), 2–13. https://doi.org/10.1145/384285.379246

[39] C. B. Zilles and G. S. Sohi. 2000. Understanding the backward slices of performance degrading instructions. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201).* 172–181. https://doi.org/10.1145/339647.339676