2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)

Timely, Efficient, and Accurate Branch Precomputation

Aniket Deshmukh
University of Texas at Austin
United States
a.deshmukh@utexas.edu

Lingzhe(Chester) Cai
University of Texas at Austin
United States
chestercai@utexas.edu

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

***Abstract*—Out-of-order cores rely on high-accuracy branch predictors to supply useful instructions to the processor backend. However, there remains a large fraction of mispredictions caused by hard-to-predict (H2P) branches that modern predictors have been unable to improve. Precomputation is an alternative to prediction that speculatively executes the dependence chain of a branch earlier to override the branch predictor at Fetch time. However, prior work sacrifices H2P branch coverage and precomputation accuracy to produce timely results.**

Our work relaxes this timeliness constraint by using precomputation results to issue early misprediction flushes instead of overriding the branch predictor. This allows us to construct a highly accurate precomputation thread with good misprediction coverage, without sacrificing timeliness. The thread is efficient as it utilizes on-core execution resources and re-uses existing hardware for issuing early flushes. Using our Timely, Efficient, and Accurate thread for precomputation yields a 10.1% improvement in performance over an aggressive baseline OoO core.

***Index Terms*—Branch Prediction, Precomputation**

I. Introduction

Branch mispredictions still present a major barrier to single-thread performance in processors today. Despite years of research, there remains a portion of branches that cannot be easily predicted by modern high-accuracy branch predictors such as TAGE-SC-L [23] and Perceptron [15], even with significant hardware investment. These are commonly referred to as hard-to-predict (H2P) branches. They comprise data-dependent branches [12], [21] and branches with complex control flow patterns [29] that are hard to identify at runtime.

Prior work has looked into developing alternative solutions for these branches. Precomputation is one such approach that identifies instructions in the dependence chains leading up to H2P branches to create a standalone, speculatively executed thread. This thread begins execution early enough to override the branch predictor at Fetch time.

Early precomputation approaches used static analysis and profiling to construct "helper threads" [8], [22], [27], [30] for frequently mispredicting branches. More recent approaches [13], [21], [26] generate a lightweight instruction stream or "precomputation thread" by detecting H2P branches and analyzing their dependencies at runtime. Unfortunately, prior work faces several limitations that restrict the set of branches they can effectively precompute.

Precomputation is often late: The timeliness constraint of having the precomputation result arrive before the branch is predicted is hard to meet in many applications. Prior work therefore focuses on specific branches with fixed control flow properties to improve their timeliness. This reduces misprediction coverage as many H2P branches are left out.

Precomputation is often inaccurate, i.e. precomputation overrides correct branch predictions, causing additional mispredictions. Prior work thus only targets H2P branches with simple or specific types of dependence chains to maintain high accuracy. This further decreases misprediction coverage. Heavy-weight helper threads generated using compiler-based techniques are more accurate but perform poorly as they compromise on timeliness.

Finally, **precomputation is inefficient** as it requires a separate core or execution engine to provide timely results and to avoid wasting on-core resources when the precomputation is late or incorrect.

We argue that these limitations can be overcome if precomputation results that arrive "late", i.e. after the branch is fetched but before it is executed, can still be used to correct the instruction stream. Relaxing this constraint allows us to trace longer dependence chains across complex control flows that are initiated much earlier than prior work, thus optimizing for misprediction coverage and accuracy without hurting timeliness. Our work presents a mechanism for constructing a more **Timely, Efficient, and Accurate ("TEA")** thread that uses on-core hardware to trigger early mispredictions flushes via precomputation instead of overriding the branch predictor at fetch time. Our contributions can be summarized as follows:

- We describe a method to generate a lightweight yet highly accurate dynamic precomputation threads (99.3%) with high misprediction coverage (76%).
- Our precomputation thread contains synchronized timestamps for H2P branches that can leverage existing flush mechanisms. This greatly simplifies the hardware required for triggering early misprediction flushes and enables both direction and target precomputation.
- We show that using on-core resources with accurate precomputation threads provides performance comparable to using a dedicated core or execution engine if the processor backend prioritizes the precomputation thread.
- Finally, we show that focusing on accuracy and coverage allows the TEA thread to outperform Branch Runahead,

the current state-of-the-art in branch precomputation, which focuses mainly on timeliness.

II. Prior Work

A. *Compiler-based approaches*

Compiler-based approaches use static analysis to identify frequently mispredicting branches and instructions needed to compute these branches. This approach creates a heavy-weight helper thread that is always correct. However, many of these instructions are unnecessary as the control flow observed at runtime is very limited: accounting for correctness across all possible control flows at compile-time results in an over-inflated thread that runs too slowly to provide any benefit. Subsequent work decreased the Helper Thread size by removing instructions and branches in infrequently seen control flows using profiling [8], [17], [22], [24], [28], [30], [31]. This hurts precomputation accuracy as profiling is not always representative and cannot account for phase behavior.

CRISP [19] is a lightweight compiler solution that continuously profiles several input datasets in a data center environment to find instructions on the program's critical path. It identifies H2P branch and long latency load dependence chains and prioritizes their execution in the backend. This provides limited benefit as it only allows branch dependence chains to be scheduled to the execution units a few cycles earlier.

Hybrid approaches like Control-Flow Decoupling [24] use the compiler to hoist the control-flow computation within a loop. The hoisted code inserts computed branch directions ahead of time into a queue that is read by the rest of the instructions. This is challenging to do in the absence of loops or for loops with fewer iterations as it requires significant code duplication to account for all control flows leading to a branch.

B. *Runtime approaches*

Most runtime precomputation techniques identify H2P branches by keeping track of frequently mispredicting branches in a table but differ in how their dependence chains are constructed and executed.

Iterative Backward Dataflow Analysis (IBDA) [7] tags Register Alias Table (RAT) entries with the PC of the last instruction that writes to that register. This identifies instructions that write to registers needed by an H2P branch.

Repeating this process iteratively identifies instructions in the dependence chain of that H2P branch. The identified PCs are used to filter out non-dependence chain instructions from the fetch stream to form a precomputation thread. However, IBDA cannot trace memory dependencies, which are required for constructing long and accurate dependence chains across control flow constructs such as calls and returns. Moreover, it only captures a single level of the dependence chain every time the H2P branch is seen. This limits the overall length of the dependence chain and how quickly it is constructed. Since IBDA filters the normal fetch stream, it cannot fetch dependence chain instructions faster than the baseline OoO core (the "main thread"), limiting its performance benefits. The Branch Tracker Table in [13] functions similarly but only works for dependence chains with a single load, followed by a few arithmetic operations, leading up to a branch.

In the most recent Slipstream proposal [26], the precomputation thread is constructed by removing all control-dependent instructions for an H2P branch. Subsequent branches that are control-independent but data-dependent with respect to that H2P branch are also removed. This allows it to leverage misprediction-level parallelism [20]. However, the resulting thread is still heavy-weight and requires a separate core for execution. This also increases the communication latency of forwarding branch directions across cores, hurting timeliness.

DP-SSMT [9] used a dataflow walk to trace the dependence chain of an H2P branch. This was done by tracking the live-ins and live-outs of instructions in a post-retire buffer. The generated precomputation thread used trigger instructions to drive its control flow. A similar approach, the Backward Dataflow Walk, was introduced by Filtered Runahead [14]. It is used in Branch Runahead [21], the current state-of-the-art in branch precomputation, to identify lightweight and timely dependence chains in applications with simple control flows. It uses a post-retire buffer to trace all dependence chain instructions between two consecutive instances of an H2P branch. Branch Runahead also identifies branches whose dependence chain instructions are independent of whether previous branches were predicted correctly and initiates them early to provide more timely results. However, it performs poorly in programs with more complex control flows and requires a dedicated dependence chain engine to support the parallel execution of these chains.

The TEA thread is highly accurate and can trace long dependence chains across complex control flows while remaining lightweight. Section III discusses how the TEA thread is constructed and where its performance benefits come from. The TEA thread uses a dedicated frontend and partitions backend resources which allows it to run effectively on-core and reduces the hardware overhead associated with precomputation (Section IV).

C. *Issuing Early Misprediction Flushes*

Prior work uses a unified queue or multiple per-branch queues to forward precomputed directions to the branch predictor. These queues are expensive as they buffer hundreds of predictions per branch. They also have multiple write ports as several branches in the precomputation thread can finish executing together. Prior work therefore only precomputes branch directions, as adding targets is expensive.

Implementing early resolution with these queues (in addition to overriding the branch predictor) requires support for simultaneous reads from multiple pipeline stages in the frontend. Alternatively, the in-flight branch queue and the precomputed branch queues can be scanned in parallel to match precomputation thread branches to their corresponding main thread counterpart. This enables early resolution for branches in the backend as well. However, both solutions increase the hardware cost and complexity of these queues. [11] uses the scanning approach but does not discuss its implementation overhead. [13] on the other hand only has a few fixed flush points in the frontend.

In contrast, the TEA thread issues early misprediction flushes using synchronized timestamps (Section III-B). This allows TEA thread branches to correctly flush all instructions younger than the corresponding main thread branch utilizing existing flush mechanisms. This enables both direction and target mispredictions to be precomputed without any additional overhead. A detailed description of this mechanism is provided in Section IV-F.

III. The TEA Thread

The TEA thread is an independent, speculatively executed thread that precomputes H2P branch directions and targets and issues early mispredictions flushes for both direct and indirect branches. It has a dedicated frontend that only fetches instructions in the dependence chains of H2P branches and shares the processor backend with the main thread. The TEA thread runs ahead of the main thread as it skips the fetch and execution of non-dependence chain instruction and is not limited by in-order retirement. Main thread instructions do not affect TEA thread performance as it has a dedicated frontend, preferential access to Issue ports (going from Rename to Reservation Stations), and a dedicated partition of execution resources in the backend (Section IV-A).

The TEA thread uses the main branch predictor to drive its control flow. This allows the TEA thread to trace dependence chains across complex control flows. It also enables accurate prediction of intermediate non-H2P branches, thereby reducing the size of the TEA thread as non-H2P branch dependence chains need not be included for accurate precomputation (Section III-B).

Using the main branch predictor allows the same timestamp (or branch IDs) to be assigned to a TEA thread branch and its main thread counterpart. The TEA thread uses this synchronized timestamp to issue early misprediction flushes irrespective of where in the pipeline the corresponding main thread branch is. Since the TEA thread uses existing flush mechanisms (Section IV-F), it supports out-of-order and nested branch resolutions. This significantly reduces the timeliness constraint as the precomputation result only needs to arrive before the corresponding main thread branch finishes execution to provide some benefit.

Our chain construction mechanism can trace longer and more accurate dependence chains that are valid across multiple control flows. Precomputation for longer chains can be initiated much earlier in the instruction stream, which improves timeliness. The higher accuracy helps improve misprediction coverage significantly (Section III-C,D,E).

The TEA thread and main thread share Physical Registers, Reservation Stations, and Execution Units. Since the TEA thread is highly accurate and timely, it does not waste resources when executing on-core and can be prioritized (at Issue and Scheduling) over the main thread. This allows the on-core implementation to approach the performance of a dedicated execution engine (Section V-D).

The rest of this Section provides an overview of the TEA thread's construction and fetch mechanisms and discusses the key features that allow it to maximize misprediction coverage and accuracy over prior work.

A. *Identifying Dependence Chain Instructions*

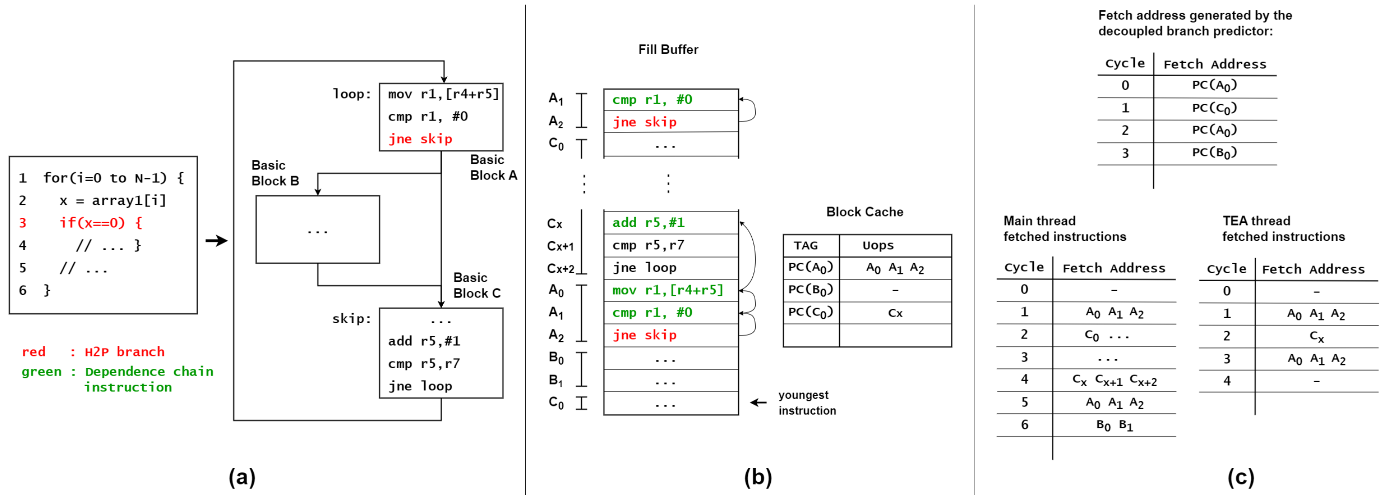
We use a variant of the Backward Dataflow Walk proposed in Criticality Driven Fetch [10] for identifying dependence chain instructions. This requires identifying H2P branches and is done by training a table of counters called the H2P Table (Section IV-A). The chain construction process begins by saving retired instructions and their operands (in program order) in a ”Fill Buffer”. Retired branches that are likely to benefit from precomputation are marked using the H2P Table when entering the Fill Buffer. The example in Fig.1-(a) shows a code snippet containing an H2P branch with its assembly and control flow diagram. Fig.1-(b) (left) shows how the Fill Buffer is populated after several iterations of the code snippet have retired. The H2P branch is marked in red.

Performing the Backward Dataflow Walk When the Fill Buffer is full, a Backward Dataflow Walk is initiated starting at the youngest instruction: C_0 in the above example. The Fill Buffer is traversed instruction by instruction until an H2P branch is encountered. At this point, the registers and memory addresses needed to compute that branch are added to a ”Source List”. In the example (Fig.1-(b)), A_2 is the first H2P branch encountered during the Backward Dataflow walk and the condition code register (RFLAGS for x86) is added to the Source List.

An instruction that writes to a register or memory location in the Source List is part of the H2P branch dependence chains. Thus A_1 is marked as a dependence chain instruction (green). At the same time, the $A_1's$ destination register (RFLAGS) is removed from the Source List, and $A_1's$ source (R1) is added to the Source List instead. This ensures that the Source List tracks the minimum set of live-ins needed to compute the marked H2P branches.

Continuing upwards, A_0 is marked as a dependence chain instruction and the Source List is modified to contain register R4, register R5, and memory location [R4+R5]. The Backward Dataflow Walk continues until it reaches the oldest instruction in the Fill Buffer (B_0). Note that chains for all branches marked as H2P, including multiple dynamic instances of the same H2P branches, are traced simultaneously via this mechanism.

Storing dependence chain instructions: The next step collects all the marked instructions (including H2P branches) in the Fill Buffer into basic block-sized segments. These segments are stored as a single entry in a ”Block Cache” that records all identified dependence chain instructions. Block Cache entries are tagged with the PC of the first instruction in that basic block, as shown in Fig.1-(b) (right). In the example, all instructions in basic block A are marked, therefore its entry contains A_0, A_1, A_2 and is tagged with the PC of A_0. Basic block B's entry is empty as none of its instructions are part



```
1  for(i=0 to N-1) {
2     x = array1[i]
3     if(x==0) {
4       // ... }
5    // ...
6  }
```

```
loop: mov r1,[r4+r5]
      cmp r1, #0
      jne skip
      ...
skip: ...
      add r5,#1
      cmp r5,r7
      jne loop
```

Fill Buffer

A_1	cmp r1, #0
A_2	jne skip
C_0	...
⋮	⋮
Cx	add r5,#1
Cx+1	cmp r5,r7
Cx+2	jne loop
A_0	mov r1,[r4+r5]
A_1	cmp r1, #0
A_2	jne skip
B_0	...
B_1	...
C_0	...

Block Cache

TAG	Uops
PC(A_0)	A_0 A_1 A_2
PC(B_0)	-
PC(C_0)	Cx

Fetch address generated by the decoupled branch predictor:

Cycle	Fetch Address
0	PC(A_0)
1	PC(C_0)
2	PC(A_0)
3	PC(B_0)

Main thread fetched instructions

Cycle	Fetch Address
0	-
1	A_0 A_1 A_2
2	C_0 ...
3	...
4	C_x C_{x+1} C_{x+2}
5	A_0 A_1 A_2
6	B_0 B_1

TEA thread fetched instructions

Cycle	Fetch Address
0	-
1	A_0 A_1 A_2
2	C_x
3	A_0 A_1 A_2
4	-

Fig. 1. Constructing dependence chains for an H2P Branch. This example uses a code snippet that models a commonly seen control flow pattern leading up to an H2P branch in many applications (the GAP benchmarks *bfs, bc, tc, cc, sssp* have several loops with this control flow pattern)

of the dependence chain. The entry for basic block C only contains C_X.

B. Constructing the TEA Thread at Fetch

Our baseline OoO core has a decoupled Branch Predictor (BP) that generates fetch addresses for the main thread and pushes them into a Fetch Queue. Fig.1-(c) shows how this works for the previous example. [1] The first address, PC(A_0) is pushed into the queue in cycle 0. The BP predicts taken for branch A_2, and the next fetch address PC(C_0) is added in cycle 1. The stream of fetch addresses generated by the decoupled BP is shown in the upper half of Fig.1-(c).

These fetch addresses are used by the main thread to fetch instructions from the I-cache. In the example, the main thread uses PC(A_0) in cycle 1 to read instructions in basic block A. However, it takes three cycles (cycles 2, 3, and 4) for all the instructions in basic block C to be fetched (Fig.1-(c), bottom-left).

The TEA thread has a dedicated Fetch stage (as part of the TEA thread frontend) that fetches TEA thread instructions in parallel with the main thread. It uses the same fetch addresses generated by the decoupled BP to stitch together basic block segments in the Block Cache. In the example, the first address, PC(A_0) is used to read the Block Cache entry for basic block A in cycle 1. In cycle 2, the TEA thread reads the Block Cache entry for PC(C_0). Since only C_X is part of the dependence chain, the TEA thread fetches the basic block segment for C in a single cycle. Subsequently, it can fetch and initiate the precomputation for the next instance of H2P branch A_2 faster than the main thread.

The TEA thread runs faster because the throughput of decoupled branch predictors is much higher than the frontend bandwidth. Aggressive processor designs today have an 8-wide frontend, but the decoupled BP can predict up to 32 instructions (or 128B) per cycle. This allows the decoupled BP and the TEA thread to run several cycles ahead of the main thread. The TEA thread runs further ahead every time the main thread is stuck fetching non-dependence chain instructions or stalled due to backend pressure. This adds up over time (until the first misprediction is detected) even if there are cycles when the TEA thread fetches nothing. In the example, every time basic block C is seen, the TEA thread gains two additional cycles over the main thread. Thus even though the TEA thread fetches nothing in cycle 4, it can initiate the third instance of branch A_2 in cycle 6 which is four cycles ahead of the main thread. The run-ahead distance is limited by the size of the Fetch Queue that buffers fetch addresses for the main thread, which is 128 addresses in our design.

The main advantage of this model is the ability to trace any possible control flow by composing different basic block segments using the fetch addresses generated by the branch predictor. This provides much better misprediction coverage compared to other solutions that only work on specific control flow patterns [7], [13], [21].

Dealing with intermediate branches: Non-H2P intermediate branches can be encountered while fetching instructions from the TEA thread. The direction of these branches often determines which instructions are needed to precompute an H2P branch even though they are not H2P themselves. Prior work attempts to deal with these branches by either using simple but inaccurate mechanisms to predict them or by marking them as H2P so that their directions can also be precomputed [9], [21].

The TEA thread predicts intermediate branches with high accuracy as its fetch addresses are generated by the main branch predictor (TAGE). For instance, in the example in Fig.3, the branch on line 4 (end of basic block A) can be accurately predicted by TAGE. Its chain is not included in the TEA thread even if it affects the dependence chain of the H2P branch, thus keeping the TEA thread lightweight. TAGE also predicts many H2P branches with a reasonably high accuracy (~80%) which allows the TEA thread to correctly fetch instructions past them and begin precomputation for the next H2P branch earlier. This improves both the timeliness and accuracy of the TEA thread.

[1] Assuming the branch predictor has a throughput of one taken branch per cycle and capped by the BTB line size if a taken branch is not found, which can be as high as 128B or ~32 instructions in aggressive out-of-order cores.

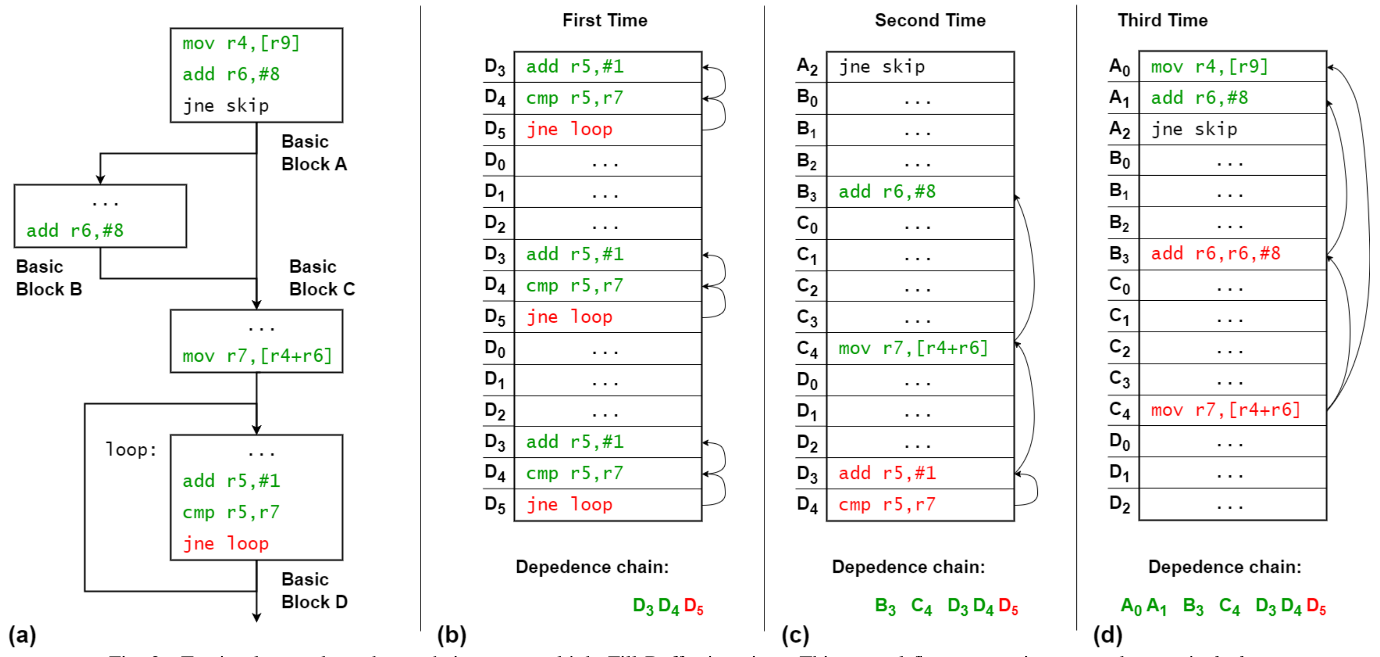


Fig. 2. Tracing longer dependence chains over multiple Fill Buffer iterations. This control flow pattern is commonly seen in *leela*.

Some prior work can fetch instructions faster than the TEA thread as they either ignore or statically predict intermediate branch directions, whereas the TEA thread has to wait for the main branch predictor to generate fetch addresses. This only works if the intermediate branches are biased. Prior work can also deal with intermediate branches if H2P branch dependence chains are independent, i.e., the dependence chain is unaffected by the direction of these intermediate branches [20], [21]. This works well in benchmarks where simple control flows are common, but provides very little performance in applications with more complex control flows as the opportunities for exploiting control independence are limited. The TEA thread provides good performance for both classes of applications as it can capture some of the benefits associated with control independence if TAGE predicts intermediate branches accurately. We evaluated these effects in Sec. V-C.

C. *Tracing Longer Dependence Chains*

Longer dependence chains improve timeliness as they allow precomputation to be initiated earlier. Adding more instructions to a precomputation thread often improves timeliness rather than hurting it if the added instructions increase the length of the dependence chains in the precomputation thread.

The length of the dependence chains traced in the Fill Buffer is limited by the size of the Fill Buffer and the position of the H2P branch in the Fill Buffer. This is seen in the example in Fig.2-(a) which contains an H2P loop branch. Only a few instructions in the dependence chain of this H2P branch can be traced the first time it is seen in the Fill Buffer (Fig.2-(b)). This limitation can be overcome if TEA thread instructions are used as initiation points for the Backward Dataflow Walk the next time the corresponding main thread instructions are seen in the Fill Buffer. Fig.2-(c) depicts a future Fill Buffer iteration. The Backward Dataflow Walk in this case is initiated at dependence chain instructions (red) instead of an H2P branch. Additional instances of the Backward Dataflow Walk (Fig.2-(d)) trace more and more of the dependence chain. The TEA thread thus contains dependence chains that span thousands of instructions even though the Fill Buffer can only hold 512 instructions.

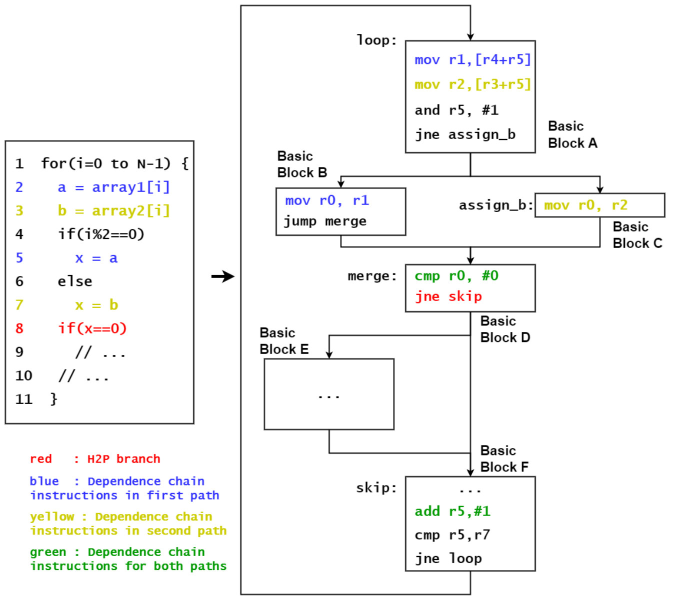
D. *Incorporating Memory Dependencies*

The TEA thread tracks memory dependencies during the Backward Dataflow Walk using the Source List as correlated loads and stores can impact precomputation accuracy significantly. This is common when an H2P branch is within a function body. Pre-computing its branch direction early requires tracing dependence chain instructions across the function call. The input variables to the function are often part of the dependence chain of the H2P branch in such cases; memory dependencies need to be tracked so that the push and pop operations that communicate input variables are included in the precomputation thread.

Unlike register dependencies, memory addresses corresponding to correlated load-store pairs can change over time. Thus, incorporating memory dependencies sometimes increases the precomputation thread size without improving its accuracy. However, the overall improvement in the length and accuracy of the dependence chains makes up for this in most benchmarks. A quantitative analysis of how memory dependencies affect precomputation accuracy and timeliness is presented in Section V-E.

E. *Combining chains across multiple control flows*

An H2P branch can have different dependence chains for each control flow leading up to it. Fig.3 shows such an example. Under the control flow A-B-D, the first instruction in basic block A is part of the dependence chain (in blue) for the H2P branch (in red). For A-C-D, the second instruction in A is part of the dependence chain (in yellow). Saving the longest or the most recent dependence chain produces



```
for(i=0 to N-1) {
  a = array1[i]
  b = array2[i]
  if(i%2==0)
    x = a
  else
    x = b
  if(x==0)
    // ...
  // ...
}
```

```
loop:
    mov r1,[r4+r5]
    mov r2,[r3+r5]
    and r5, #1
    jne assign_b
    mov r0, r1
    jump merge
assign_b:
    mov r0, r2
merge:
    cmp r0, #0
    jne skip
    ...
skip:
    ...
    add r5,#1
    cmp r5,r7
    jne loop
```

Basic Block A
Basic Block B
Basic Block C
Basic Block D
Basic Block E
Basic Block F
red : H2P branch
blue : Dependence chain instructions in first path
yellow : Dependence chain instructions in second path
green : Dependence chain instructions for both paths

Fig. 3. Multiple control flows leading up to an H2P branch. *mcf* has these types of control flows

incorrect results when the saved dependence chain does not match the actual control flow during execution. Saving all possible dependence chain versions is not viable in terms of storage as each additional branch added to the control flow (branch at the end of block A in Fig. 3) exponentially increases the number of possible chains.

Combining the chains from multiple paths ensures that the precomputation is correct across all these paths. The TEA thread does this by storing a bit-mask along with each basic block that indicates which instructions in the basic block are part of H2P branch dependence chains. Different versions of each basic block can be combined by bit-wise ORing the masks. In Fig.3, two masks are generated for basic block A: 1000 for control flow A-B-D and 0100 for control flow A-C-D. The "1" in the bit mask indicates that the instruction is part of the H2P branch dependence chain. After bit-wise ORing these masks, both the first and second instructions in basic block A are saved in the Block Cache. This ensures the TEA thread is correct under both control flows. However, this also adds an additional instruction to the TEA thread on either path that is not needed to precompute the H2P branch on that path.

The additional instructions negatively impact timeliness. However, this is compensated for by the increased chain accuracy and length. Section V-E provides a quantitative analysis of how the bit-masks impact precomputation accuracy and timeliness, and shows that maximizing accuracy and coverage is almost always better for performance.

IV. Implementation Details

A. *Overview*

The Baseline OoO Core has an 8-wide frontend with a decoupled branch predictor that can predict up to one taken branch or sequential instructions spanning 128B per cycle. Fetch addresses generated by the decoupled BP are filled into a Fetch Queue. For each fetch address, up to two sequential cache lines can be read every cycle. Decode, Rename, and Issue process up to 8 uops per cycle. The frontend is 12 cycles deep and the minimum fetch to resolution (end of execution) latency for a branch is 15 cycles (Table-I).

TEA thread structures An overview of the additional hardware required to construct and fetch the TEA threads is shown in Fig.4. The H2P Branch Table and Fill Buffer located in the backend are responsible for identifying H2P branches and tracing dependence chains. The TEA thread has dedicated 8-wide Fetch and Rename Stages. The Block Cache and Fetch unit construct the TEA thread using fetch addresses inserted into a shadow Fetch Queue by the decoupled BP. The overall frontend latency for the TEA thread is 9 cycles. A shadow RAT manages TEA thread dependencies. The two threads converge at Issue, with the TEA thread getting preferential access to the Issue ports. Backend execution resources (Physical Registers and Reservations Stations) are partitioned statically when the TEA thread is active. TEA thread instructions do not enter the ROB and their Physical Registers are freed using a separate physical register map table. The TEA thread has a small store data cache for buffering store values (Table-II).

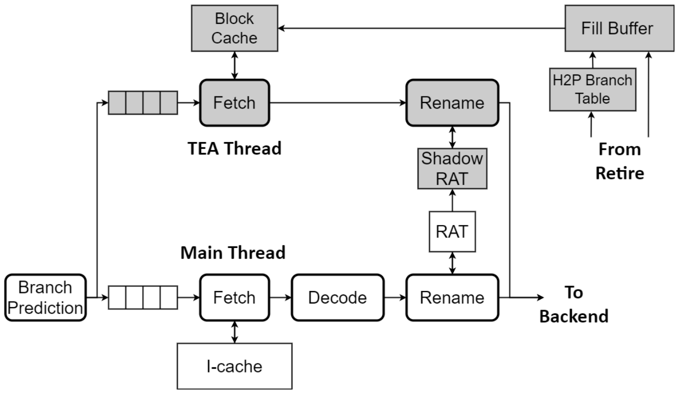


Fig. 4. Hardware required to construct and fetch TEA Threads

B. *Finding H2P Branches*

Similar to previous work, H2P branches are identified using a table of per-branch counters to record the number of mispredictions. It tracks direction and target mispredictions for both direct and indirect branches. The H2P Table is an 8-way set associative structure with 256 entries, indexed with the branch PC. Each entry contains a 3-bit saturating counter. An entry in the H2P Table is created for a branch when it mispredicts with its counter value initialized to 1. If the same branch is mispredicted again, its counter value is incremented. A branch is considered H2P if it has an entry in the H2P Table and its counter value is greater than 1. All counters in the H2P Table are decremented by 1 every 50k instructions so that counter values for branches that mispredict less often than once every 50k instructions (less than 0.02 MPKI) tend towards 0. Branches with a counter value of 0 are prioritized for replacement. The periodic decrement allows branches above a certain MPKI threshold to be identified as H2P. We experimented with various decrement periods to find the best-performing parameters. Our general observation was that marking more branches as H2P improves misprediction coverage and provides better performance. This begins to drop off only when highly accurate branches are marked as H2P as it starts to hurt timeliness significantly.

C. *Constructing and storing H2P branch dependence chains*

As described in Section III-B, the Fill Buffer is responsible for marking instructions in the dependence chains of H2P branches. The Fill Buffer operates on micro-ops (uops) instead of instructions[2]. Each entry in the Fill Buffer contains a valid bit, the decoded bytes for a uop (which includes register and memory locations written to and read by that uop), and its PC. In addition, a chain-bit indicates whether that uop is an H2P branch or part of an H2P branch dependence chain. This amounts to a total of 16B per entry. The chain bit is initially set both for H2P branches and for instructions that were executed as part of the TEA thread as it helps trace longer dependence chains (Section III-C). This is done by marking main thread instructions that were also fetched as part of the TEA thread using the bit masks in the Block Cache entries.

The **Backward Dataflow Walk** is performed by a state machine and takes $\sim$500 cycles. Register dependencies are tracked using a bit vector with a length equal to the number of architectural registers. Memory dependencies are tracked using a small 16-entry buffer that stores memory addresses. Together, these structures form the ”Source List” mentioned in Section III-B. Instructions retired during the backward Dataflow Walk are discarded. Thus the Fill Buffer only samples a portion of the retired instruction stream. We found that performance is not very sensitive to the duration of the Backward Dataflow Walk, and the associated structures thus do not need multiple access ports to perform the walk faster. The Fill Buffer size does not affect performance significantly ($\sim$1% change) as the bit-mask allows longer chains to be traced over multiple Backward Dataflow Walks.

Block Cache Uops with their chain-bit set after the Backward Dataflow are divided into basic block segments and stored in the Block Cache. The Block Cache has 512 entries and is divided into a tag store and a data store. The tag store holds a 40-bit tag (PC of the first instruction in the basic block). The data store contains decoded dependence chain uops (on average 4B per uop) and the bit-mask (32-bits long). A bigger Block Cache improves the misprediction coverage of the TEA thread. To improve coverage without increasing storage significantly, we added a smaller 256-entry Block Cache tag store. This is reserved exclusively for Block Cache entries with no dependence chain uops and therefore does not need associated data store entries (as their bit-masks are zero as well). The tag store for these entries is needed as it indicates there are potentially more dependence chain uops past the empty basic block segment and that the TEA thread should not terminate.

Basic block segments belonging to a cache line are stored in separate ways with the same cache-line index. Segments longer than 8 sequential uops are divided into multiple entries. Entries from consecutive cache lines are distributed across two banks (similar to the I-cache). Fetch addresses generated by the decoupled BP read out all Block Cache entries for two consecutive cache lines. This ensures the Block Cache can deliver up to 8 uops per cycle distributed across multiple sequential segments until the first taken branch.

Periodically Resetting the Bit-masks: Some control flow patterns are only observed during specific execution phases. Dependence chains captured with older control flows may not be valid after a phase change and therefore need to be removed to keep the TEA thread lightweight. This is done by periodically resetting the bit-masks in the Block Cache and ensures that future Backward Dataflow Walks do not use these instructions are initiation points. We found that a reset period of 500K instructions works best for performance.

D. *TEA thread frontend*

Fetch addresses generated by the branch predictor are sent to both the Block Cache and the I-cache as shown in Fig.4. The main thread consumes fetch addresses at a lower rate compared to the TEA thread. To account for this mismatch, the Fetch Queue for the main thread is much larger than a traditional Fetch Queue (can hold up to 128 fetch addresses). This allows the Branch Predictor to function at peak throughput and generate fetch addresses faster to match the TEA thread's throughput. The TEA thread is initiated on a hit in the Block Cache. The uops read out are rotated and sent directly to the shadow Rename stage (as they are decoded). The corresponding bit masks are added to a small queue that feeds the main thread. The bit-mask is used to mark instructions in the main thread that are part of the TEA thread so they can be used as initiation points for the Backward Dataflow Walk in the Fill Buffer.

TEA thread instructions are renamed using a shadow RAT. The contents of the main RAT are copied into the shadow RAT before the first TEA thread instruction is renamed to synchronize the state of both threads. After renaming, TEA thread instructions are sent to the Issue logic which picks between the two 8-wide Rename output stages. The Issue logic is 8-wide but prioritizes TEA thread instructions and uses the leftover Issue slots for the main thread. The dedicated frontend, preferential access to Issue slots, and partitioned backend ensure that the TEA thread is not blocked due to back-pressure caused by main thread instructions.

E. *TEA thread backend*

Prior work on precomputation often uses a dedicated execution engine or core. This is because their precomputation threads are often inaccurate or late which wastes execution bandwidth and slows down the main thread. Schemes that perform precomputation on-core do so by either using a small section of the on-core resources or by scavenging Issue slots, Execution ports, and RS entries that are unused.

The TEA thread is extremely accurate (less than 0.7% of the TEA thread branches are incorrectly computed) and timely

[2]This is because we use the x86 ISA in our simulations. Operating at the instruction granularity works fine for fixed-length ISAs.

($\sim$76% of TEA thread branches save at least one cycle of misprediction penalty). Executing its instructions using on-core resources leads to very little wastage. We found that prioritizing the Issue of TEA thread instructions and allocating a significant proportion of Reservation Station entries and Physical Registers to it is better for performance, even though it delays the execution of some main thread instructions. This is because H2P branch mispredictions are almost always on the critical path of execution, even for applications with relatively lower branch MPKI. Prioritizing their execution, even at the cost of other instructions, provides better performance. A dedicated execution engine or separate core for the TEA thread shows marginal improvement but is much more expensive.

192 Reservation Stations and Physical Registers are reserved for the TEA thread when it is active. Instructions from both threads share Execution units, cache ports, and MSHRs. TEA thread instructions are identified by an additional bit in the Reservation Station entries and are discarded when they finish execution.

Freeing Physical Registers The TEA thread only maintains a speculative RAT. Its instructions free up backend resources as soon as possible to avoid the in-order retirement bottleneck. TEA thread instructions do not enter the ROB. They use a separate table containing a Valid bit and a 5-bit Reference Counter per Physical Register (PR) for identifying PRs that can be freed. These are initialized to 1 and 0 respectively when the TEA thread is initiated. When a TEA thread instruction is renamed, it sets the Valid bit for the previous PR mapped to its destination Architectural Register (AR) to 0. If this previous PR is not currently being used (Valid=0, Reference Counter=0), it is freed.

Every instruction that wants to read from a PR increments its Reference Counter (at Rename). The counter is decremented when the instruction reads the data value for that PR (just before it enters the Execution Units). After decrementing the counter, if it is also found to be invalid (Valid=0, Reference Count=0) because a younger instruction overwrote its mapping, the PR is freed. This works because there are no more instructions that need this PR in the Reservation Stations, and instructions still in the frontend use the new mapping for that AR. Note that this can result in incorrect precomputation if the counter overflows (this does not affect the main thread). However, this is quite rare since the TEA thread is frequently flushed and does not affect performance significantly.

Dealing with Stores Most instructions can be executed speculatively without any impact on the architectural state. This includes TEA thread loads as it is similar to a prefetch that speculatively brings data into the D-cache. No ordering needs to be enforced on the loads and they are not allocated Load Queue entries. Loads only leave the Reservations Stations when they have a D-cache port and TEA thread loads that miss in the D-cache carry their destination PR in the MSHR entry. However, TEA thread stores cannot write to the cache as they update the architectural state of the machine. Instead, they write to a small cache that buffers the last 16 half-lines written to by TEA thread stores.

F. *Branch misprediction flushes*

When a mispredicted TEA thread branch finishes execution, a flush operation is triggered. TEA thread branches inherit the timestamps generated by the branch predictor and thus have the same timestamps as their main thread counterparts. Similar to a regular flush operation, all instructions younger than the mispredicted branch are discarded. This applies to both the main thread and TEA thread instructions. The processor backend is partially flushed exactly as it would be in a normal OoO core and the checkpointed or recovered state of the RAT is copied over to both the main RAT and the shadow RAT. This, along with fixing the Branch Predictor history and PC, synchronizes the state of both threads.

Partially Flushing the Frontend: Normally, when a branch misprediction is detected in the backend, all the frontend pipeline stages are flushed. However, we support partial flushes in the processor frontend, i.e. instructions older than the mispredicting branch in the frontend pipeline stages are also flushed. This situation arises when the TEA thread runs so far ahead that the main thread branch being flushed is in the frontend. In this case, some instructions in the frontend are older than the mispredicting branch. Partial flushes in the frontend are supported by adding a comparator before the flush signal for each pipeline stage to compare the timestamp of the instructions in that stage and the mispredicting branch. The comparator latency overlaps with other tasks performed during the flush operation such as fixing the branch predictor history and is unlikely to impact the overall misprediction flush latency. The Fetch Queue is also partially flushed; when this happens, the full misprediction penalty for that branch is saved. Note that when the frontend is partially flushed, the state of the main RAT does not need to be recovered. The shadow RAT does need to be fixed, and this is achieved by checkpointing the contents of the shadow RAT instead of the main RAT when the TEA thread is running far ahead of the main thread.

When a TEA thread branch resolves, the in-flight branch queue entry (which tracks information for all main thread in-flight branches in the pipeline) for the corresponding main thread branch is modified to reflect the precomputed branch direction and target. When the main thread branch finishes execution, it reads the in-flight branch queue to check whether the misprediction has been resolved correctly. If the TEA thread branch was incorrectly computed, another misprediction flush is issued to correct the control flow. However, this is rare and its impact on performance is negligible.

G. *TEA thread termination*

The TEA thread is terminated on a miss in the Block Cache or when a TEA thread branch is incorrectly computed. On a miss, the remaining TEA thread instructions continue to precompute directions for any remaining branches. If an incorrect precomputation is detected, all remaining TEA thread instructions are drained out of the processor.

Incorrect precomputations can be detected in two ways. The first examines the precomputed branch direction and target saved in the modified in-flight branch queue entry

Core	3.2GHz, 8-wide issue, 12 cycle FE latency 512 Entry ROB, 352 Entry Reservation Station, 16-wide retire 12 Execution Ports (6-ALU, 2-LD, 2-LD/ST, 2-FP) 400 Physical Regs, 256 entry load queue, 192 entry store queue
Predictors	64KB TAGE-SC-L [23], 128-entry Fetch Queue History-based indirect branch predictor, RAS 1 taken per cycle, 4k entry BTB
Caches	32KB 8-way L1 I-cache (4-cycle access) 2R, 1W ports (4 banks) 48KB 12-way D-cache (4-cycle access) 1MB 16-way LLC cache (18-cycle access), 64B lines
Memory	DDR4_2400R: 1 rank, 2 channels 4 bank groups and 4 banks per channel tRP-tCL-tRCD: 16-16-16

TABLE I
CORE PARAMETERS

Core	8-wide Fetch and Rename Stages, 8-cycle latency Issue-ports shared with the main thread 192 PRs, 192 RS reserved for TEA threads when active
Caches	H2P Branch Table: 256-entry cache, 0.2KB, 1-cycle access Block Cache: 512-entry cache, 256 zero-tags, 8-way, 19KB Store data cache for TEA thread: 16 entry cache (32B per entry)
Other structures	Fill Buffer: 512-entry queue, 8KB, single access port PR map queue, 2400 bits, Shadow RAT

TABLE II
TEA THREAD STRUCTURES

(as explained above) and serves as a fail-safe. The second uses bit-masks stored in the Block Cache entries to examine instructions in the main thread. When the TEA thread is initiated, a poison bit in the main RAT is initialized to 0 for all ARs. Main thread instructions that are not part of the H2P branch dependence chains (have a 0 in the bit-mask) set the poison bit for the AR they write to. Main thread instructions that are part of the dependence chains (have a 1 in the bit mask) clear the poison bit for their destination AR. If a main thread instruction that is part of the H2P branch dependence chains (has a 1 in the bit-mask) reads from a poisoned register, the TEA thread has an incorrect dependence chain and precomputation is preemptively terminated. This is because reading from a poisoned register means that the dependence chain instruction needed a result produced by a non-dependence chain instruction (after the TEA thread started), which is incorrect by definition. When this is detected, TEA thread branches that have not been executed yet and are younger than the instruction that caused this dependence violation are blocked from triggering misprediction flushes. The rest of the TEA thread instructions are gradually drained out of the backend. This helps filter out most incorrect precomputations without requiring a second misprediction flush, reducing the number of additional flushes to below 0.001 PKI.

H. *Discussion on hardware overhead*

The TEA thread configuration increases the total dynamic instruction footprint by 31.9% over the baseline OoO core. This is much lower that Slipstream [26] (85%), and comparable to Branch Runahead [21] (34%). The TEA thread executes instructions on-core rather than using a separate execution engine or core, making it much more efficient compared to prior work. We use McPAT [18] to estimate area, power, and energy consumption.

Area The main area overhead associated with the TEA thread structures comes from the Fill Buffer and the Block Cache. The duplicated pipeline stages and shadow RAT also add to this overhead. This is approximately 3.5% of the total core area with over 2% coming from just the caches and Fill Buffer. Comparatively, a true 16-wide OoO core is much more challenging to implement: it costs ∼10% more area while only providing 2.8% performance. This is because scaling the frontend width without increasing predictor bandwidth provides very little benefit, and there are no efficient ways to consistently fetch more than one taken branch per cycle.

Power The additional TEA thread structures increase peak power by 8.5% according to McPAT. The additional frontend for the TEA thread is responsible for over 6% of the additional power: particularly the Block Cache, Rename stage, and the shadow RAT.

Energy The TEA thread increases the overall dynamic instruction footprint, leading to increased Fetch, Rename, and backend energy in addition to the energy consumed by the additional structures. However, this is mitigated by the reduction in overall execution time and the reduction in wrong path instruction fetches in the main thread. According to McPAT, the overall energy consumption over our set of evaluated benchmarks reduces by 2%. A per-benchmark breakdown for the increase in dynamic instructions fetched is shown in Table-III. *nab* and *pr* stand out because the reduction in wrong path fetches in the main thread for these benchmarks is very large.

V. EVALUATION

A. *Methodology*

We use Scarab [4], an execution-driven cycle-accurate x86-64 simulator, to simulate the micro-architecture of an aggressive OoO core augmented with the structures and logic needed to construct and execute the TEA thread. Ramulator [16] is used to model main memory. Parameters of the baseline OoO core and added structures are listed in Tables I and II. The baseline core parameters model an aggressive 8-wide OoO core similar to many industry products [1]–[3]. All results are relative to this baseline core configuration.

Benchmarks We use SPEC CPU2017 benchmarks [5] with the ref input set and the GAP benchmarks suite [6] (with inputs g=19 and n=300) in our evaluation. Benchmarks with MPKI less than 0.5 are not included (5 floating point benchmarks). We use the SimPoint [25] methodology to find representative regions and generate up to 5 Simpoints per benchmark, with 200 million instructions per Simpoint. Each run is preceded by a warmup period of 200 million instructions.

B. *Performance*

Fig.5 shows the performance improvement provided by the TEA thread when used to precompute branch directions and targets using on-core resources. The geomean improvement is 10.1% and is relatively evenly distributed across most benchmarks. The branch MPKI for these applications is shown in Fig.6. Fig.7 contains a breakdown of the percentage of mispredictions A covered by the TEA thread.

The best-performing benchmarks: *mcf*, *bfs*, *cc* and *tc* all have many mispredictions, most of which are on the critical path of the program. They also have many loads that are guarded

Benchmarks	perlbench	gcc	mcf	omnettpp	xalanbmk	x264	deepsjeng	leela	exchange2
Dynamic I-count	17.63%	25.51%	40.01%	23.07%	22.21%	14.39%	32.15%	44.62%	27.67%
Benchmarks	xz	pop2	nab	bfs	sssp	pr	cc	bc	tc
Dynamic I-count	51.03%	16.59%	10.25%	43.49%	56.79%	-1.27%	45.96%	48.76%	40.72%

TABLE III
PERCENTAGE OF EXTRA DYNAMIC INSTRUCTIONS FETCHED RELATIVE TO BASELINE OOO CORE

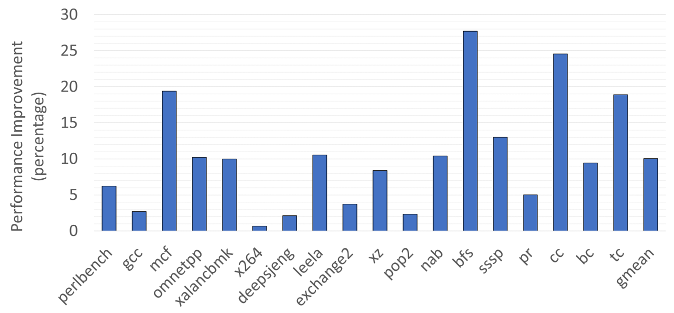


Fig. 5. Performance benefit of pre-computing branches using the TEA thread

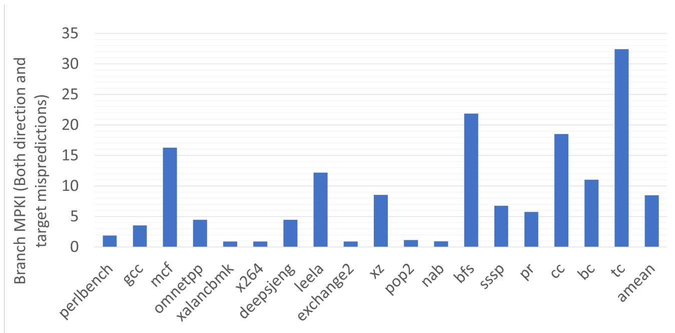


Fig. 6. Total number of direction and target mispredictions per 1000 instructions

by H2P branches and hit in the Last Level Cache (LLC) or go out to main memory. Resolving these branches early allows correct path loads to begin execution sooner and improves the amount of memory level parallelism. These effects add up to provide substantial performance improvements. *perlbench* and *nab* do not have a high branch MPKI but show substantial improvement as they have many long latency loads in the shadow of a few H2P branches.

The TEA thread also has the side-effect of prefetching loads that are part of H2P branch dependence chains. We turned off early resolution in the TEA thread to measure how much this skews performance, and it only provides an overall 1.2% performance gain. This makes sense as the TEA thread only focuses on H2P branch chains. *xalanbmk* is the only exception as most of its benefit comes from data prefetching.

Deepsjeng and *omnetpp* are limited primarily by the Block Cache size. This is reflected in their lower misprediction coverage (due to no dependence chain being available). Increasing the number of basic block segments tracked by the Block Cache significantly improves performance on these benchmarks (by 5%). However, Block Cache storage is not easy to scale because it is not as dense as the I-cache. An entry within the Block Cache can hold up to 8 uops, but the number of dependence chains uops in a basic block can vary significantly. To reduce the storage inefficiency, we added additional tag store entries for basic blocks with no

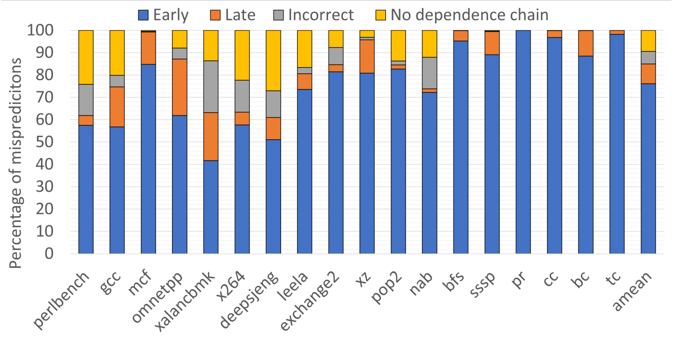


Fig. 7. Breakdown of branch mispredictions covered by the TEA thread

dependence chain uops in them (Section IV-B). This helps improve performance in *perlbench*, *gcc*, *omnetpp*, *deepsjeng*, and *leela* by optimizing the Block Cache capacity.

gcc, mcf, omnetpp, and xz have some ”late” precomputations. This predominantly occurs when the TEA thread and main thread finish execution in the same cycle. There are a few cases ($< 0.1\%$) where the TEA thread finishes executing an H2P branch after the main thread. If this occurs more than 4 times, the TEA thread is terminated. However, this is very infrequent and does not affect the overall performance. Some of the SPEC17 benchmarks have a non-negligible portion of incorrect precomputions as seen in Fig.7, but these are almost always detected by the RAT-poisoning scheme which prevents them from issuing additional misprediction flushes ($< 0.05\%$).

C. Comparison against Branch Runahead

We implemented a scaled-up version of Branch Runahead using a dedicated execution engine on the same baseline OoO core and measured its performance improvement. This is shown in Fig.8. For comparison, we split the applications into two categories: those with simple control flows and those with more complex control flow patterns[3]. Benchmarks with simple control flows have many independent branches. These are branches whose dependence chains are unaffected by the directions of any other branches around them (including other dynamic instances of the same branch). The H2P branch in Fig.1 is one such example. Most H2P branches in these benchmarks are also confined to simple loops without complex loop exit conditions. All the GAP benchmarks are classified as simple control flow applications, with *xz* being the only SPEC benchmark in this category.

Precomputation using the TEA thread overall does much better than Branch Runahead (10.1% vs 7.3% geomean improvement) even though it uses on-core execution resources compared to Branch Runahead's dedicated execution engine

[3]Note that a simple control flow leading up to a branch does not make it easy to predict.

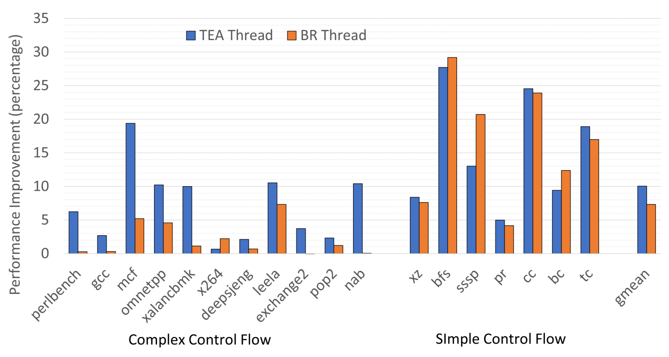


Fig. 8. Comparison again Branch Runahead

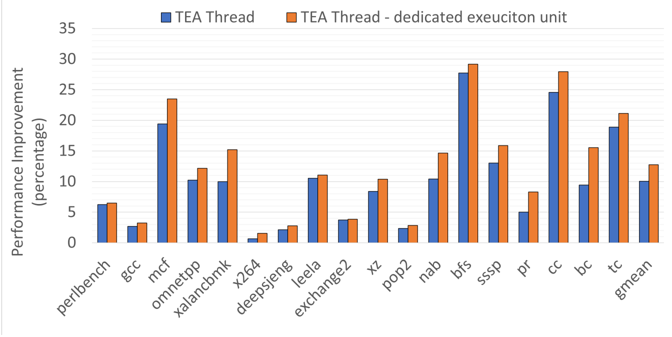


Fig. 9. Performance benefit of pre-computing branches using the TEA thread with a separate execution engine

(which has a large backend optimized to execute branch dependence chains). However, Branch Runahead performs comparably on benchmarks with simple control flows. This is because Branch Runahead explicitly identifies independent branches within a loop using merge point prediction. This allows multiple invocations of the independent branch to be issued and executed in parallel even if a previous branch is mispredicted. The TEA thread cannot leverage this as it uses the main branch predictor to drive its control flow, but the higher prediction accuracy of TAGE on intermediate branches extracts part of this benefit (as explained in Section III-B). As a result, the TEA thread has slightly smaller per-branch savings in simple control flow applications as it is less timely, but is made up for by the higher misprediction coverage. Branch Runahead only does much better than the TEA thread on *sssp* and *bc*, and this is primarily because it has a lot more backend execution resources available to it.

Benchmarks with more complex control flows have fewer independent branches and more complicated dependence chains. It is harder to extract misprediction level parallelism or leverage control independence in these benchmarks. Branch Runahead only performs optimally for specific types of control flows confined to loop boundaries and thus does not do well on such benchmarks.

D. *On-core vs dedicated execution engine*

Executing the TEA thread on a separate execution engine eliminates any interference associated with using on-core resources. Fig.9 shows the performance of the TEA thread when executed on an independent execution engine with the same amount of resources as the on-core implementation (192 Reservation Stations and Physical Registers) and 16 dedicated execution units. The size of the baseline OoO core remains the same. This configuration pushes the overall performance up to 12.3%. This is not a significant increase given the execution engine has a much larger area and power overhead associated with it compared to using on-core resources. Most of the improvement comes from *mcf*, *xalanbmk*, *nab*, *pr*, and *bc*, and these mainly benefit from the increase in the number of execution units. *sssp* in particular benefits from the minimized interference between the two threads. We also experimented with a much larger execution engine (the same backend as the main OoO core), but that provided very little additional benefit (12.8% geomean improvement).

E. *TEA thread features*

Fig.10-(a) shows the impact of the thread construction features discussed in Section III on precomputation accuracy. It consists of 5 configurations that include the TEA thread as-is, and the TEA thread without certain features. The ”only loops” category records the dependence chain instructions only between two consecutive instances of an H2P branch, thus limiting the chains to loops. The ”no masks” configuration does not combine Block Cache entries and allows Backward Dataflow Walks to start only at H2P branches, limiting the dependence chain length. The ”no mem” configuration does not incorporate memory dependencies during the Backward Dataflow Walk. Finally, the Branch Runahead dependence chains are also added for comparison.

Chain Accuracy Overall, no single feature provides most of the benefit. Incorporating memory dependencies is the least important for accurate precomputation for most benchmarks. Among the graph benchmarks, *sssp*, *pr*, and *bc* are relatively unaffected by any of the features. This is because most of the H2P branches in these benchmarks match the control flow modeled by the example in Fig.1, which can easily be captured by any dynamic chain construction scheme. Thus, even Branch Runahead has close to a 100% precomputation accuracy on these benchmarks. Any complexity over this simple case requires a more comprehensive thread construction mechanism to ensure accurate precomputation. The TEA thread provides good accuracy across all the applications (with an average precomputation accuracy of 99.3%). Performance-wise, all other configurations are 3% to 5% worse than the TEA, with masks having the greatest impact on improving the TEA thread’s performance.

Misprediction Coverage Branch Runahead has a high precomputation accuracy for the H2P branches it covers as it actively removes chains corresponding to H2P branches that provide incorrect precomputations. This is reflected in its low misprediction coverage, shown in Fig.10-(b). In contrast, The TEA thread has a much higher misprediction coverage (76%). Removing all the features drops the overall coverage down to 39% (much lower than removing any one feature),

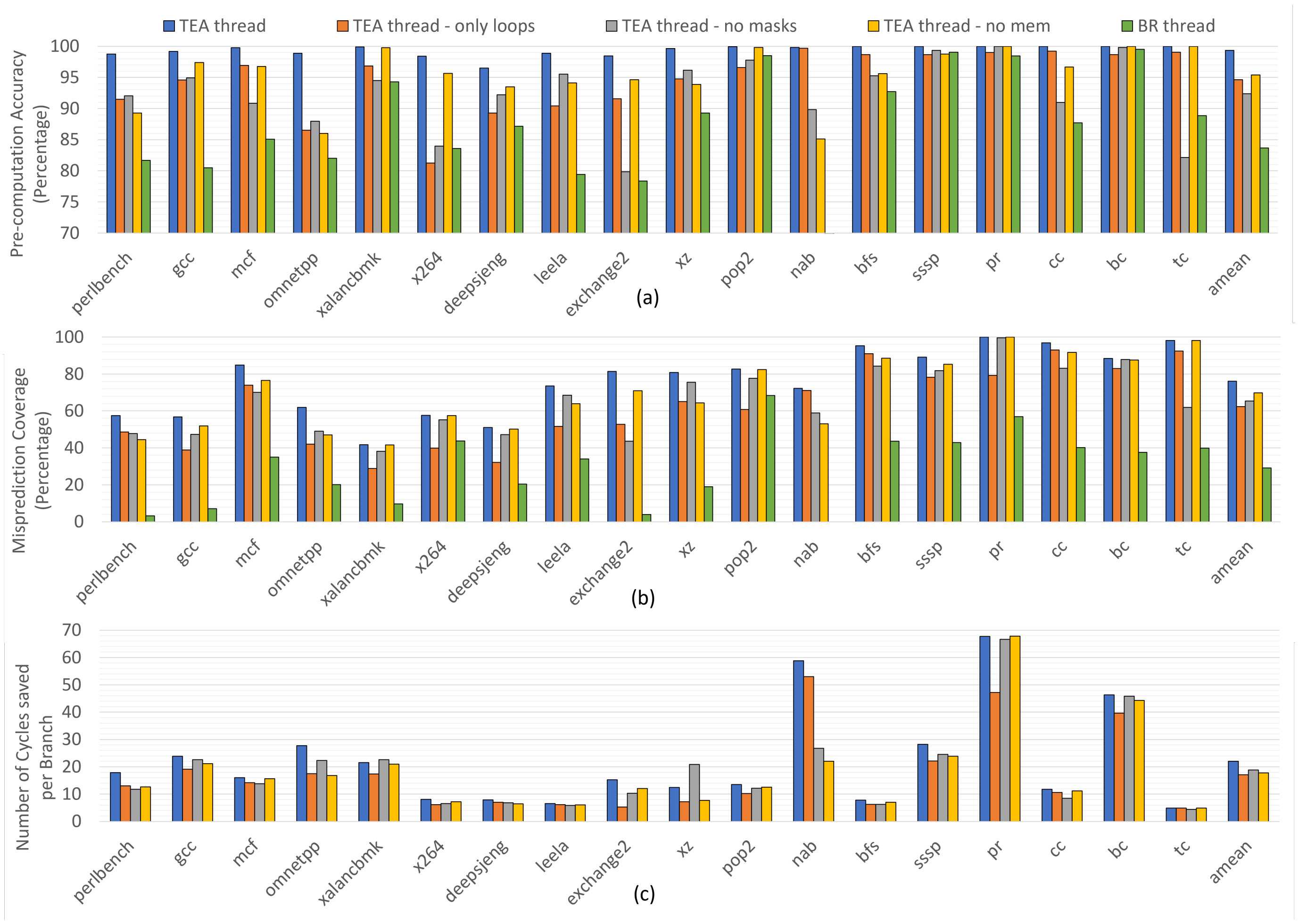


Fig. 10. Precomputation thread accuracy, coverage and timeliness

thus reinforcing the fact that all the proposed techniques for augmenting the TEA thread are important.

Timeliness Fig.10-(c) provides a measure of timeliness by counting the average number of misprediction cycles saved per branch (measured with respect to the main thread's branch resolution cycle). Across most applications, the per-branch savings are the highest for the configuration with all the features even though it is the heaviest precomputation thread in terms of the number of dynamic instructions executed (31.19% of the total dynamic instructions as compared to 21% for no features). The only two exceptions to this are *xalanbmk* and *xz*. Combining dependence chains across multiple control flows (using the Block Cache masks) hurts timeliness in these benchmarks.

VI. Conclusion

Improvements in branch prediction accuracy have slowed down in recent years as increasingly complex algorithms are needed to cover the remaining branch mispredictions. This has made alternative solutions to branch prediction more attractive. Precomputation was one of the first alternative solutions proposed to deal with branches that are fundamentally hard to predict for history-based predictors. While it provides good benefits, recent work has been limited to specific types of branches and control flows as a more general solution cannot produce precomputation results in time to override the branch prediction. We show that relaxing the timeliness constraint on precomputation by allowing early misprediction flushes enables broader precomputation schemes that provide high misprediction coverage. The TEA thread we propose performs well due to its high accuracy and misprediction coverage without compromising on timeliness. It also provides a simple hardware solution for issuing early misprediction flushes, making it viable to implement. The TEA thread beats the prior state-of-the-art in precomputation, showing that focusing purely on timeliness does not provide better performance. This invites a wider array of future solutions to precomputation-based problems that give more importance to accuracy and coverage rather than timeliness.

Acknowledgment

We thank the anonymous reviewers and the members of the HPS Research Group for their feedback and help in improving this paper. We also thank Intel, Arm, and NSF grant #2011145 for their financial support.

REFERENCES

[1] "AMD Zen4 microarchitecture," https://www.anandtech.com/show/17585/amd-zen-4-ryzen-9-7950x-and-ryzen-5-7600x-review-retaking-the-high-end/8.

[2] "Apple M1 microarchitecture," https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2.

[3] "Intel Goldencove microarchitecture," https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3.

[4] "Scarab," https://github.com/hpsresearchgroup/scarab.

[5] "The standard performance evaluation corporation (spec)," 1997. [Online]. Available: https://www.spec.org/

[6] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[7] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 272–284.

[8] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 186–195.

[9] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 307–317.

[10] A. Deshmukh and Y. N. Patt, "Criticality driven fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 380–391. [Online]. Available: https://doi.org/10.1145/3466752.3480115

[11] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998, pp. 59–68.

[12] M. U. Farooq, Khubaib, and L. K. John, "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)*, 2013, pp. 59–70.

[13] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney, "Opportunistic early pipeline re-steering for data-dependent branches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3410463.3414628

[14] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 358–369.

[15] D. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[16] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, 2016.

[17] S. Kondguli and M. Huang, "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)*, 2019, pp. 533–544.

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[19] H. Litz, G. Ayers, and P. Ranganathan, "Crisp: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

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

[21] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

[22] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 37–48.

[23] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, p. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[24] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling: An approach for timely, non-speculative branching," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2182–2203, 2015.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," ser. ASPLOS X, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[26] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg, "Slipstream processors revisited: Exploiting branch sets," ser. ISCA '20, 2020, p. 105–117. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00020

[27] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *SIGPLAN Not.*, vol. 35, no. 11, p. 257–268, nov 2000. [Online]. Available: https://doi.org/10.1145/356989.357013

[28] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform," ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 144–155. [Online]. Available: https://doi.org/10.1145/1024393.1024411

[29] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.

[30] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 2–13. [Online]. Available: https://doi.org/10.1145/379240.379246

[31] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 172–181. [Online]. Available: https://doi.org/10.1145/339647.339676