CRISP: Critical Slice Prefetching

Heiner Litz
hlitz@ucsc.edu
University of California Santa Cruz
USA

Grant Ayers
granta@google.com
Google
USA

Parthasarathy Ranganathan
parthas@google.com
Google
USA

ABSTRACT

The high access latency of DRAM continues to be a performance challenge for contemporary microprocessor systems. Prefetching is a well-established technique to address this problem, however, existing implemented designs fail to provide any performance benefits in the presence of irregular memory access patterns. The hardware complexity of prior techniques that can predict irregular memory accesses such as runahead execution has proven untenable for implementation in real hardware. We propose a lightweight mechanism to hide the high latency of irregular memory access patterns by leveraging criticality-based scheduling. In particular, our technique executes delinquent loads and their load slices as early as possible, hiding a significant fraction of their latency. Furthermore, we observe that the latency induced by branch mispredictions and other high latency instructions can be hidden with a similar approach. Our proposal only requires minimal hardware modifications by performing memory access classification, load and branch slice extraction, as well as priority analysis exclusively in software. As a result, our technique is feasible to implement, introducing only a simple new instruction prefix while requiring minimal modifications of the instruction scheduler. Our technique increases the IPC of memory-latency-bound applications by up to 38% and by 8.4% on average.

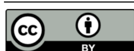
CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**.

KEYWORDS

prefetching, criticality, instruction scheduling, branch prediction, out-of-order execution

ACM Reference Format:
Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: Critical Slice Prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507745



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507745

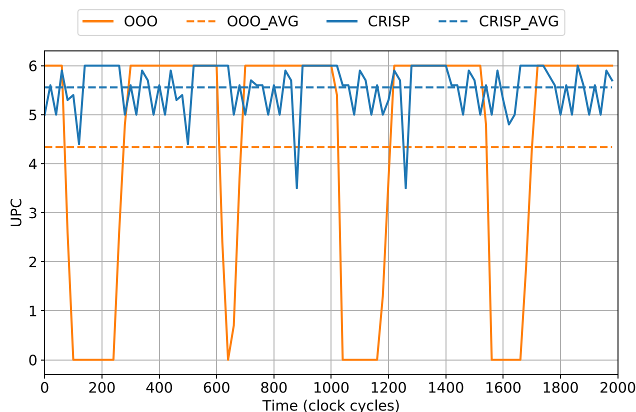


Figure 1: UPC improvement provided by *CRISP* over OOO execution for a pointer-chasing microbenchmark

1 INTRODUCTION

The processor-memory performance gap continues to be one of the most significant performance challenges of contemporary microprocessors. Memory loads missing the cache hierarchy suffer from long DRAM latency, leading to pipeline stalls with a detrimental effect on instructions per cycle (IPC). Two main approaches have been developed in the past to address this challenge. *Latency-avoiding* techniques utilize prefetching of regular [12, 28, 52, 53, 55, 61, 75, 76, 86, 94, 99, 108, 110, 111] or irregular [16, 48, 82, 84, 95] patterns as well as helper threads [24, 33, 33, 70, 73, 123] to read data speculatively into caches before the demand load of the data occurs, effectively hiding the high memory access latency. *Latency-tolerating* techniques such as out-of-order (OOO) execution [116] continue to execute independent instructions behind long-latency loads in the sequential instruction stream, avoiding pipeline stalls. The opportunities for reordering instructions have been further extended by systems that consider instruction criticality [3, 20, 66, 81, 90, 91, 102] and increase the number of instructions that can be reordered with delinquent loads.

While these prior approaches have shown promising performance benefits under favorable conditions, they suffer from two major shortcomings: *high hardware complexity* and *low flexibility*. For instance, runahead prefetchers [84, 85, 95] introduce significant overheads for learning and storing promising prefetch candidates and their instruction slices. They may introduce power overheads by executing redundant instructions and, if the runahead interval is too short, are ineffective. Continuous runahead engines [48] also introduce additional compute cores for executing (redundant) instructions to generate prefetches. OOO processors have exploited criticality mainly for improving energy efficiency [3, 20, 66, 81, 90, 91, 102]

and for optimizing the cache hierarchy [10, 87, 115] but not for prefetching. These techniques also introduce complex hardware and storage requirements for classifying critical instructions at runtime. As a result, there has been limited adoption of these proposals in modern microprocessors.

Furthermore, previously proposed hardware techniques suffer from limited flexibility while our proposed software technique allows us to adjust the criteria for defining instruction criticality and enable application-specific optimizations. Instruction scheduling is NP-Hard [13, 14] for all but non-trivial pipelines and, for performance reasons, needs to be performed with minimal latency. As a result, deploying complex scheduling heuristics in hardware is infeasible, and hence most CPU schedulers utilize an oldest-instruction-first policy at best, ignoring criticality. In contrast, software techniques enable sophisticated policies for classifying delinquent loads as critical instructions based on the execution frequency, miss ratio, and memory-level parallelism (MLP) properties of each individual load. Furthermore, the number of address-generating instructions referred to as *load slices* and the critical-path length of these load slices can be considered for determining criticality. In contrast to hardware techniques that can only process load slices of a limited size such as 32 instructions [48], software techniques are not limited by meta-data storage overheads. Finally, while hardware techniques for extracting load slices based on iterative backwards dependency analysis (IBDA) [20] only consider register dependencies, our proposed software technique observes dependencies through memory, a critical capability enabling precise and comprehensive load slices.

To address the shortcomings of prior work, we propose *CRISP*, a lightweight yet flexible mechanism for accelerating hard-to-predict, delinquent load instructions that frequently miss the cache. Our approach leverages the performance monitoring unit (PMU) hardware counters and instruction tracing capabilities of modern microprocessors to determine critical loads and their load-address-generating instruction slice profiles at compile time. We then utilize post-link-time optimization [22, 88] to tag the critical instructions of a *load slice* as prioritized. Our technique then "prefetches" these critical instructions by extending the processor's instruction scheduler to prioritize critical instructions. To further improve performance, *CRISP* introduces *branch slices* which reduce branch misprediction penalties by executing hard-to-predict branch instructions as early as possible. In general, *CRISP* can analyze and prioritize any high-latency instruction (such as division or FBSTP), although delinquent loads generally have the highest performance impact.

The only hardware requirement of our proposal is a mechanism to mark and prioritize the execution of critical instructions inside the CPU. Without loss of generality, on x86 we propose a new instruction prefix for tagging critical instructions, and we minimally modify the scheduler to prioritize their execution. By performing most of the work in software as part of a feedback-driven optimization (FDO) [22] flow, our technique provides high flexibility while avoiding complex and costly hardware modifications.

To illustrate the operation of *CRISP*, we analyze a synthetic workload that interleaves a linked list traversal exhibiting hard-to-predict loads with an embarrassingly parallel vector multiplication as part of a loop. In each loop iteration, a scalar number is read from the next linked list element and multiplied with a vector. Figure 1 shows the μops retired per cycle (UPC) for a traditional OOO processor and *CRISP*, for four loop iterations. The OOO processor operates at the maximum UPC of 6 until it experiences a last-level cache miss due to reading the next element from the linked list. At this point, the processor stalls since all further instructions depend on the load. *CRISP* addresses this challenge by tagging the delinquent load and its load slice as critical which promotes its execution to *before* the vector multiplication of the *previous* loop iteration. By executing critical instructions as early as possible, *CRISP* improves the average UPC of this workload by over 30%. Existing OOO schedulers generally do not pick younger loads for scheduling as long as older ready instructions are available in the reservation station. This issue also cannot be easily addressed with the compiler as it requires re-ordering of instructions across loop-iterations.

In summary, *CRISP* contributes over prior work as follows:

- *CRISP* avoids the execution of redundant instructions inherent to runahead prefetching
- *CRISP* avoids the complexity and storage overheads for extracting load slices in hardware
- *CRISP* increases flexibility by performing criticality extraction in software
- *CRISP* allows for higher miss coverage and accuracy by observing memory dependencies
- *CRISP* extracts only the critical instructions of a load-slice
- *CRISP* introduces branch slices for reducing the impact of branch mispredictions
- *CRISP* introduces a low-complexity, criticality observing, matrix-based instruction scheduler

CRISP leverages the combined techniques above to improve the performance of applications exhibiting hard-to-prefetch loads by up to 38% and by 8.4% IPC on average while introducing only minimal hardware modifications. *CRISP* improves performance over IBDA by up to 4.3× while avoiding most of its hardware overheads.

2 BACKGROUND

Modern microprocessors leverage out-of-order (OOO) execution for improving IPC by executing instructions when their operands are available rather than by program order. As part of this, processors implement a *decoupled-frontend* architecture [97, 114] which can fetch up to hundreds of instructions ahead in the sequential execution stream and place them into an instruction queue. To fetch future instructions across basic block boundaries of sequential code, processors employ branch prediction [71, 109, 121] mechanisms such as the state-of-the-art TAGE predictor [103]. With a decoupled frontend, a processor can exploit instruction-level parallelism by searching the instruction queue for instructions that are ready to execute, even if earlier instructions have unresolved register or memory dependencies. Before instructions are selected and issued by the scheduler, they are placed in the reorder buffer (ROB) which tracks in-flight instructions and commits them back in program order. To avoid stalls due to a full ROB, schedulers generally prefer to issue the oldest instructions as early as possible to increase the likelihood that the instruction at the head of the ROB can retire. However, in many cases, executing latency-critical instructions early can improve performance far more than selecting the oldest instructions, but information about criticality is generally unavailable to the scheduler. As we will show in Section 3.1, loads that miss

the CPU cache hierarchy needing to be served by main memory frequently induce pipeline stalls by remaining at the head of the ROB until the data arrives from main memory, thus prohibiting all other instructions from retiring.

Prior work that considered criticality-based instruction scheduling including Long Term Parking [102] and Delay and Bypass [3] have focused on in-order pipelines for improving energy efficiency. These approaches sacrifice 10% and 5% performance, respectively, over an OOO core by simplifying the scheduling of non-critical instructions to improve energy efficiency in the frontend by 67% and 47%, respectively. Our technique leverages criticality for *improving IPC* by reducing memory stall cycles induced by hard-to-prefetch irregular memory access patterns. Both prior works also introduce significant hardware complexity for implementing the IBDA [20] required to extract load slices. Furthermore, these load slices are often incomplete as IBDA struggles to observe dependencies through memory, only supports a limited number of small slices, and suffers from limited flexibility as the algorithm is hard-coded in hardware. *CRISP* addresses these challenges by introducing a new software-based approach.

3 INSTRUCTION CRITICALITY

CRISP identifies high-latency load instructions that frequently induce pipeline stalls due to cache misses and tracks their load-address-generating instructions (slices). These load slices can be as simple as a single constant or might involve hundreds of prior instructions that combine to form an address. By tagging these instructions as critical and prioritizing their execution, the instruction scheduler can hide a large fraction of the memory access latency, improve memory-level parallelism (MLP) and overall performance. In the following, we discuss a code example to show the operating principle of *CRISP*.

```
1  #define VEC_SIZE 32
2  struct Node {
3    Node *next;
4    int val;
5  };
6  std::vector<int>vec(VEC_SIZE);
7  Node *current; // set elsewhere
8  while (current) {
9    //__builtin_prefetch(current->next);
10   for (int i = 0; i < VEC_SIZE; i++) {
11     vec[i] *= current->val;
12   }
13   current = current->next;
14 }
```

Figure 2: Linked List Pseudocode

3.1 Motivating Example

Consider the code snippet in Figure 2 which shows a linked list traversal in an outer loop as well as a vector-scalar multiplication in the inner loop. The vector multiplication can execute at high IPC due to abundant instruction-level parallelism (ILP). However, the memory access in line 11 is likely to miss the cache resulting in high latency. The load operation of `current->next` stalls the CPU pipeline as the vector-multiply instructions of the next outer loop iteration depend on the value returned by the missing load (`current->val`). Out-of-order execution cannot hide the load latency in this case as the processor would have to execute the delinquent load instruction *before* executing the inner loop, as this would allow overlapping the delinquent load of the next node with the current node's vector multiplication. Existing OOO processors fail to do this as the scheduler generally prioritizes older instructions such as the loads reading the vector elements. We confirmed this performance pathology on an Intel Xeon Gold 5117 by compiling the kernel shown in Figure 2 with GCC version 9.3. The kernel executes at an IPC of 1.89, whereas if we manually move the pointer-chasing memory operation of the next iteration to *before* the vector multiplication by inserting the prefetch in line 12, IPC increases to 2.71. In the next section, we propose an automatic software-based approach that determines delinquent loads as well as their load-address-generating instructions to determine critical instructions that should be executed as early as possible. By providing this information to the CPU scheduler it can prioritize delinquent loads and their instruction slices, avoiding the need for manual prefetch insertions while effectively prefetching even irregular memory accesses such as in pointer chasing applications.

3.2 Determining Delinquent Loads

Due to the complexity of modern OOO cores, determining the latency criticality of instructions is challenging. We define a load to be critical if its last level cache (LLC) miss rate is higher than a particular threshold, for instance, 20% (Section 5.5 explores this threshold), its memory address cannot be easily predicted by the hardware prefetcher (not a constant or stride), and if the number of independent instructions behind the load in the sequential instruction stream is small. Furthermore, criticality is application-specific. For instance, for a memory-bound application, a large fraction of the program instructions may be part of at least one load slice. In the case where most instructions of a program are flagged as critical, the scheduler will have no opportunity to prioritize any instruction. We empirically determined that the prioritization of critical instructions performs best if the ratio of critical instructions among all instructions is 5%-40%. In other words, there must be a sufficient mix of non-critical instructions for the scheduler to deprioritize, in order to hide the latency of the critical loads. By performing multiple profiling passes *CRISP* can empirically test different injection thresholds to determine the application-specific optimized ratio of critical instructions. There exist several other factors that determine the criticality of a load and its load slice including,

- the load's execution ratio over other loads in the program
- the LLC miss rate of the load
- the pipeline stalls induced by the load
- the baseline IPC and instruction mix of a program
- the MLP of the program at the time where the load occurs
- the time a load becomes ready to be scheduled, determined by its dependency on other high latency instructions

While modern CPUs generally lack the observability for directly measuring all of the above metrics, Intel Xeon[1] processors provide other measurement capabilities such as performance monitoring unit (PMU) counters [25], precise event-based sampling (PEBS) [118], last branch record (LBR) [31], and processor trace (PT) [64]. For instance, IPC can be measured directly while the instruction mix can be obtained via VTune or PMU counters. The execution frequency of a specific load compared to all others can be measured with PEBS or PT. A load's average memory access time (AMAT) can be approximated by measuring its cache misses via PEBS. The pipeline stalls induced by a load and MLP can be approximated by observing precise back-end stalls and load queue occupancy. Based on measurements of these metrics, we derived a heuristic for determining criticality performance gains for our analyzed applications. In particular, we only flag loads as critical if they represent more than 5% of all executed loads of a program, if their LLC miss ratio is above 20%, and if the average MLP is below 5 for phases that include said load. Based on the instruction mix (number of loads over other instructions) and baseline IPC we scale these percentages linearly to account for application-specific behavior.

3.3 Load Slice Extraction

In order to increase the opportunities for hiding memory access latency, it is necessary to reorder critical loads before non-critical instructions as early as possible. Loads frequently depend on other instructions (e.g., to compute a non-trivial memory load address), and these instructions, which we refer to as a *load slice*, need to be identified and prioritized as well. For our study, we trace the execution of a program using DynamoRIO's [17, 18] Memtrace tool, alternatively, Intel's PT can be used[2]. After obtaining a trace, we perform load slice extraction by iterating through the trace until one of the delinquent load instructions (see Section 3.2) is found. We then traverse the trace in the reverse program order direction, following the data dependencies between instructions to determine all *ancestor* instructions. Therefore, the algorithm maintains a queue called ***frontier*** containing all instructions for which ancestors have not been explored yet. The algorithm starts by inserting the delinquent load into the frontier and by analyzing its source registers. The current instruction is removed from the frontier, and ancestors are inserted into the frontier, to be analyzed in the next round, except for the following reasons: (1) The ancestor instruction is already contained in the load-slice, (2) The instruction source operand is a constant and does not have an ancestor (3) The ancestor instruction is a system call return instruction, (4) The end of the trace has been reached. The algorithm terminates for a given delinquent load whenever the frontier is empty.

Figure 3 shows an example of the load slice extraction process based on assembly code obtained by compiling the source code of Figure 2. Using the techniques described in Section 3.2 the load with PC `0x15e9` on line 30 is flagged as a critical load instruction. We traverse its data dependencies in the backwards direction until we reach line 25 with PC `0x15da`. As `0x15da` depends on the previous

[1] AMD and ARM-based microprocessors provide similar capabilities.
[2] To observe dependencies through memory the `PTWrite` instruction [26] available on Kaby Lake is required.

```
1   for(int i = 0; i < LIST_SIZE - 1; i++) {
2   1594:      movl   $0x0,-0x18(%rbp)
3   159b:      mov    -0x18(%rbp),%eax
4   159e:      cltq
5   15a0:      cmp    $0x1fffe,%rax
6   15a6:      ja     15f7 <_Z8traversev+0x86>
7     for(int e = 0; e < VEC_SIZE; e += 1) {
8   15a8:      movl   $0x0,-0x14(%rbp)
9   15af:      cmpl   $0x3f,-0x14(%rbp)
10  15b3:      jg     15da <_Z8traversev+0x69>
11      vector[e] *= val;
12  15b5:      mov    -0x14(%rbp),%eax
13  15b8:      cltq
14  15ba:      mov    %rax,%rsi
15  15bd:      lea    0x5bac(%rip),%rdi
16  15c4:      callq  1b66 <_ZNSt6vectorImSaIm>
17  15c9:      mov    (%rax),%rdx
18  15cc:      imul   -0x8(%rbp),%rdx
19  15d1:      mov    %rdx,(%rax)
20      for(int e = 0; e < VEC_SIZE; e += 1) {
21  15d4:      addl   $0x1,-0x14(%rbp)
22  15d8:      jmp    15af <_Z8traversev+0x3e>
23      }
24      cur = cur->next;
25  15da:      mov    -0x10(%rbp),%rax
26  15de:      mov    (%rax),%rax
27  15e1:      mov    %rax,-0x10(%rbp)
28      val = cur->val;
29  15e5:      mov    -0x10(%rbp),%rax
30  15e9:      mov    0x8(%rax),%rax
31  15ed:      mov    %rax,-0x8(%rbp)
32      for(int i = 0; i < LIST_SIZE - 1; i++) {
33  15f1:      addl   $0x1,-0x18(%rbp)
34  15f5:      jmp    159b <_Z8traversev+0x2a>
```

Figure 3: Pointer Chase Assembly

loop execution of `0x15e1`, the load slice extraction terminates due to a recursive dependency as `0x15e1` is already contained in the load slice. Note that line 18 with PC `0x15cc` is not part of the load slice as there only exists a forward dependency between this instruction and the load slice. In particular, the instruction instance of `0x15cc` of the *next* loop iteration only depends on the value produced by `0x15e9`. As a result, the processor can reorder the execution of the critical load slice before line 2, hiding the memory latency of the memory access in line 30.

After extracting the load slice, we flag all load slice instructions as critical by prepending the new 'critical' instruction prefix during the feedback-driven optimization pass. Our implementation is similar to prior work that leverages dataflow analysis for classifying memory access patterns [6].

3.4 Branch Slice Extraction

Initial experiments with our system showed that the benefit of prioritizing loads with irregular memory access patterns and their load slices is significantly higher on a system with a perfect branch predictor. For loops that contain hard-to-predict branches (misprediction rate > 15%), we observed that the execution time of a loop

iteration is determined to a large degree by the time required to resolve the branch outcome. Hence, prioritizing load slices within those loops does not automatically lead to significant performance gains. This effect was particularly relevant for SPEC's `lbm` as discussed in Section 5.3. We address this challenge by extending our classification methodology to include hard-to-predict branches and their *branch slices* as critical, forcing the scheduler to also compute these branches as early as possible, thus hiding a large fraction of the original branch misprediction latency. Similarly to load slices we define a branch slice as the set of instructions required to compute the branch outcome. While branch slicing improves performance for branch-bound applications in isolation, it notably synergizes well with load slicing. In Section 5.3 we show that the benefit of combining load and branch slicing can be greater than their individual contributions. Branch slicing highlights the flexibility of *CRISP*, since we are adapting its software-based criticality extraction for new branch policies and optimizations. We discuss additional opportunities for how our technique can be leveraged in Section 6.

3.5 Why Hardware-Only Techniques Are Insufficient

Hardware proposals such as the load slice architecture [20], long term parking (LTP) [102], and Delay and Bypass [3] prioritize loads and their slices to hide the memory access latency in in-order processors. When applied to an OOO machine, these prior techniques suffer from the following shortcomings: a) a lack of detailed profiling information, b) an inability to observe dependencies through memory, c) an inability to perform critical path analysis, and d) limited on-chip storage capacity for slices. Determining performance-critical loads requires detailed information about the execution frequency of each load, its cache-level miss rate, and the dependencies of other instructions on this load. This information is generally unavailable in hardware and hence prior techniques have resorted to simple, less accurate techniques such as treating all loads as critical. Hardware methods that utilize IBDA, capture incomplete instruction slices as they can only observe dependencies through registers, but not memory which is crucial for x86 due to register spilling. For instance, in Figure 3, line 31 the value in the `rax` register is passed through the stack (`rbp`). For complex applications, load slices can contain thousands of instructions exceeding the ROB and reservation station size of modern processors. Figure 4 shows the average size of a load slice for Spec and datacenter applications (we provide application details in Section 5). The challenge is that if the instructions of a load slice fill all slots of the reservation station, there exist no opportunities for the scheduler to prioritize critical over non-critical instructions. *CRISP*, therefore, only promotes the instructions of a slice that are on the critical path. Therefore, *CRISP* treats the instruction slice as a directed acyclic graph (DAG) and it then computes the aggregated path latency between each leaf instruction and the root (the delinquent load). For most instructions, we assign a fixed latency according to the processor implementation [1, 41] where for loads we utilize the AMAT in cycles as determined in Section 3.2. The DAG required for this analysis can be computed from an instruction trace but is generally unavailable

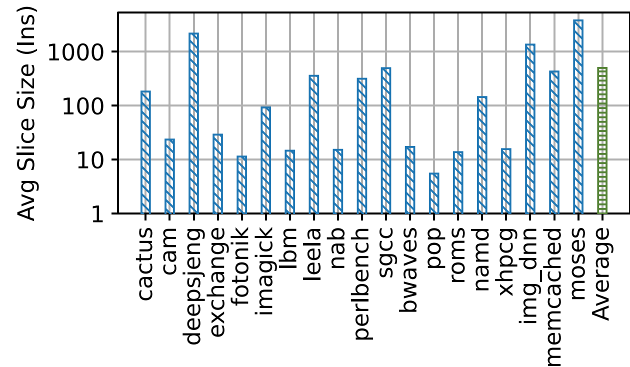


Figure 4: Average load slice size

in hardware. Lastly, storing criticality information in hardware introduces significant storage overheads whereas *CRISP* introduces only minimal hardware modifications.

3.6 Why Software-Only Techniques Are Insufficient

Instructions can be reordered by the compiler to prioritize critical loads. In fact, compilers today implement code-hoisting techniques to execute loads and their slices as early as possible. For instance, LLVM's `Hoist()` function moves loop-invariant loads into the loop's preheader block by storing the loaded value into a register. This optimization is performed by most modern compilers and partially hides the load latency while guaranteeing that loads are executed only once instead of at each loop iteration. Most performance-critical loads, however, are loop-dependent and as such, they cannot be reordered across basic blocks resulting in limited optimization opportunities. CRISP, on the other hand, leverages hardware speculation to execute prioritized load slices as early as possible, even across basic blocks. In particular, as the frontend performs branch prediction to provide future instructions, the CRISP scheduler selects critical instructions to be executed first while skipping older non-critical instructions of earlier loop iterations. These early-executed, critical instructions do not affect correctness if misspeculated, as they are squashed by the hardware after resolving the misprediction. Another issue of static code hoisting techniques is that loads can be potentially be moved from a cold into a hot basic block, reducing performance. For this reason, LLVM had to introduce PGO-based instruction hoisting which only moves instructions if the recipient basic block is cold, highlighting the need for a more dynamic mechanism. More generally, static code scheduling such as embraced by VLIW [35, 39, 104] has shown to be inferior to dynamic instruction scheduling mechanisms.

4 IMPLEMENTATION

CRISP is implemented predominantly in software to achieve high flexibility and performance while introducing only minimal hardware modifications.

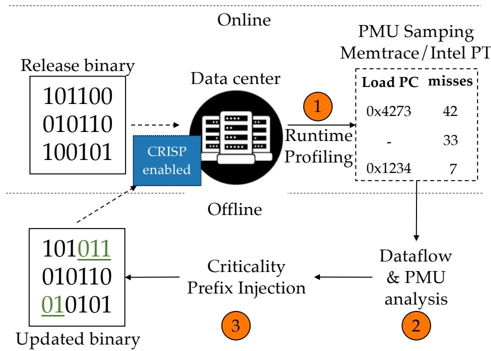


Figure 5: *CRISP* Software Infrastructure

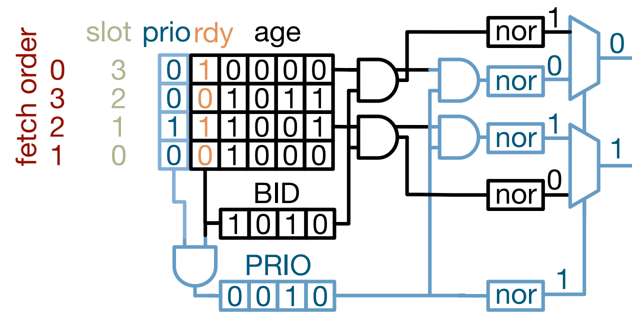


Figure 6: *CRISP* Hardware Implementation

4.1 Software Support

Datacenter operators such as Google and Facebook rely on feedback-driven optimization [22] and post-link-time optimization [68, 88] to improve the performance of their workloads. These techniques are particularly successful in datacenters, as the cost and overhead introduced by post-compile-time optimizations can be easily amortized in large-scale systems. To facilitate such an approach, datacenters automate the sequence of compilation, profiling, tracing, post-compile optimization, and deployment. We integrate *CRISP* into this flow as steps (2) and (3) as shown in Figure 5. *CRISP* leverages system-wide profiling techniques (1) such as GWP [56, 98] or ODS [15] and AsmDB as a system-wide instruction tracing method [7]. The profiling and tracing data as well as the unmodified binaries are then fed into a tool described in Section 3.3 to extract load slices and branch slices. Here we also perform optimizations such as filtering out uncommon code paths and merging code slices that refer to the same delinquent load instruction. We then rewrite the compiled assembly code adding the new instruction prefix to every critical instruction using post-link-time compilation [44, 68, 88]. The analyzed traces contain 100M instructions and have a size of 5GB (1.6GB compressed). The analysis and slice extraction step takes on the order of 100 seconds. Previous works including AutoFDO [22], BOLT [88], AsmDB [7], I-SPY [59], Ripple [60], and Twig [58] have shown that profile-guided optimization techniques are practical and that they are deployed in data centers today.

4.2 Hardware Support

CRISP requires minimal hardware to recognize and prioritize flagged latency-critical instructions as well as a method to mark critical instructions, either directly in their encodings or through a sideband method. Without loss of generality, our implementation targets an x86-based system and introduces two modifications to achieve this. First, we extend the instruction decoder to interpret the new latency-critical instruction prefix and tag all critical instructions as they progress through the CPU pipeline. Second, we extend the scheduler to prioritize these critical instructions. Our approach can be implemented in both unified reservation station (RS) architectures as well as in systems that leverage a separate scheduler for each functional unit. The scheduler needs to observe the criticality of every instruction and select critical-tagged instructions over non-critical instructions.

We implement *CRISP* as part of an *age matrix*-based RAND scheduler which represents the state-of-the-art in contemporary processors [5, 100]. RAND schedulers insert newly-fetched instructions into random slots in the instruction queue (IQ), providing space efficiency while significantly reducing the circuit complexity over traditional self-compacting queue-based schedulers (SHIFT) [36]. While SHIFT provides the advantage of perfectly ordering instructions according to their fetch cycle, it is no longer used [5] as compaction is too expensive to be feasible at high clock frequencies. The IPC performance of RAND schedulers can be improved by observing instruction fetch order, leveraging an *age matrix* [92, 100], as deployed by AMD Bulldozer [43] and IBM POWER8 [107] processors. The age matrix for an IQ with N entries works as follows: every entry in the IQ maintains an N-bit age vector initialized to all ones. When an instruction is enqueued, the bit corresponding to its slot in the IQ is cleared in the age vector. When subsequent instructions enter the IQ, they clear their corresponding bit in all age vectors of prior instructions. When an instruction is ready to schedule (source operands available) it sets the bit according to its slot in the *BID vector*. Each ready instruction then performs a bitwise AND of its age mask and the BID vector. Only for the oldest instruction, the result will be all zeros. Therefore, to pick a particular instruction, the result vector is reduced to a single-bit signal using an n-bit NOR operation. The overall propagation delay is determined mainly by the reduction-NOR which consists of n transistors and hence scales with the size of the IQ. All bitwise operators can be implemented with a single logic level. To enable *CRISP*, we extend the age matrix circuit as follows: in addition to the BID vector, we also generate a PRIO vector for all instructions that are both *ready* and *prioritized*. Each instruction performs a bitwise AND of its age mask and the PRIO vector. The PRIO vector is reduced via n-bit NOR to generate the select signal. Furthermore, *CRISP* adds a multiplexer to select between the oldest prioritized instruction and, if such does not exist, the oldest instruction. The additional gates introduced by *CRISP* are shown in Figure 6 in blue.

4.3 Storage and Timing Overhead

To support *CRISP*, each slot in the IQ is extended with a single bit to identify its priority, resulting in a space overhead of $1/n$. The impact of *CRISP* on the critical path delay is limited as most of the additional logic, including the costly NOR-reduction, is processed in

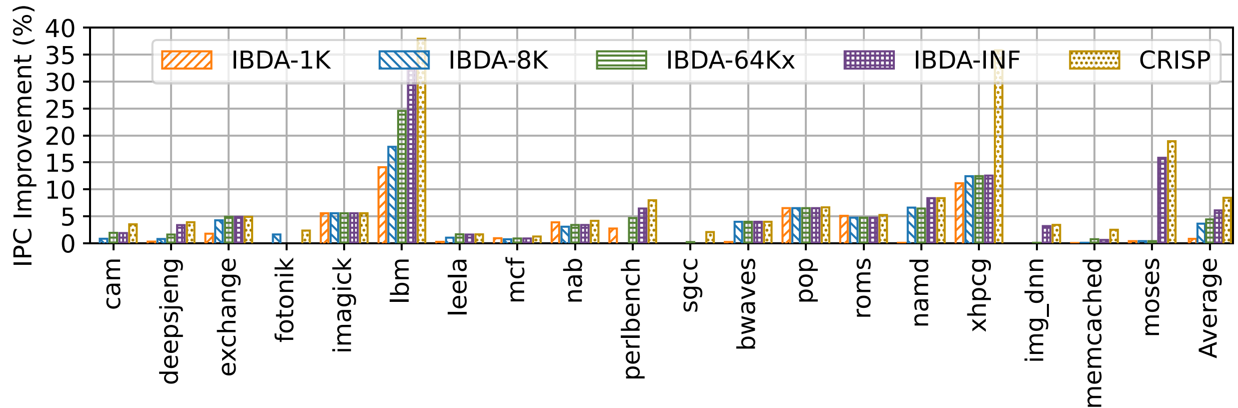


Figure 7: IPC Improvement of *CRISP* over the OOO and IBDA baselines

Table 1: Simulated System

Parameter	Value
CPU	Intel Xeon Skylake
Number of cores per socket	20
All-core turbo frequency	3.0 GHz
Frontend width and retirement	6-way
Functional Units	4 ALU, 2 Load, 1 Store
Branch Predictor	TAGE [103]
Branch Target Buffer (BTB)	8K entries
ROB	224 entries
Reservation Station	96 entries (unified)
Baseline Scheduler	6-oldest-ready-instructions-first
Data Prefetcher	BOP [76] and Stream
Instruction Prefetcher	FDIP [96], 128 FTQ entries
Load Buffer	64 entries
Store Buffer	128 entries
Uncore	
L1 instruction cache	32 KiB, 8-way
L1 data cache	32 KiB, 8-way
LLC unified cache	Shared 1 MiB/core, 20-way
L1 D-cache latency	4 cycles
L1 I-cache latency	3 cycles
L3 cache latency	36 cycles
Memory	DDR4-2400 (1 channel)

parallel to the existing design. The critical path length is increased by one additional logic level (AND to generate PRIO) and one multiplexer. The multiplexer can be implemented with a transmission gate without introducing an additional logic level. If the picker logic still represents the most timing critical stage, the AND gate can be moved into the previous pipeline stage so that the PRIO and RDY signals are generated in the same cycle (this implementation increases the storage overhead to $2/n$). As a result, *CRISP* can be implemented with minimal overheads to the critical path delay of the scheduler. In the evaluation in Section 5 we assume the same scheduling latency for *CRISP* as for the baseline scheduler. *CRISP* also introduces additional runtime overheads in the instruction cache by adding priority prefixes to instructions. We evaluate the impact of this modification in Section 5.7.

5 EVALUATION

We evaluate our approach via cycle-accurate simulation. We first evaluate the IPC performance gains provided by *CRISP* over an OOO baseline. We then evaluate branch slicing and its synergy with load slicing. Next, we perform sensitivity studies to evaluate the impact of the ROB size on *CRISP*'s performance and we analyze different thresholds for determining critical instructions in software. Finally, we evaluate the code footprint overhead of introducing our new instruction prefix.

5.1 Methodology

We evaluate *CRISP* on Scarab [51], a cycle-accurate simulator that models modern out-of-order cores with high fidelity. Scarab models a detailed decoupled frontend, functional units (of different types) contention, branch prediction, BTB, RAS, a multi-level cache hierarchy, and a detailed memory system leveraging Ramulator [63]. The most important system parameters resemble a Skylake-like Intel processor [27] and are summarized in Table 1. The *CRISP* implementation as well as all baselines share the same parameters apart from the scheduler modifications introduced by *CRISP*.

We evaluate memory-intensive workloads from SPEC2017 [50], Xhpcg [30], as well as data center applications including Moses, Memcached, and Img-dnn from Tailbench [57]. For each application, we execute 200M representative instructions. For all experiments, we enable a best-offset data prefetcher (BOP) [76] which prefetches periodic access patterns and regular strides such as the vector loads in listing 2. We also experimented with a regular stride and GHB [86] prefetcher, however, we omit these results for brevity as the performance improvement of *CRISP* over these baselines was similar in comparison to BOP.

To perform our experiments we execute the application once, obtaining PMU counter measurements and instruction traces as described in Section 3.3. We then determine critical instructions and annotate them in the code. We then re-execute the applications on Scarab, observing criticality in the scheduler. We perform profiling and performance evaluation on two separate executions, utilizing different inputs. In particular, for profiling and slice-extraction we leverage SPEC's *train* inputs, while for evaluation we utilize the *ref* inputs. We also varied the inputs for the other applications such as using different input dimension parameters for xhpcg.

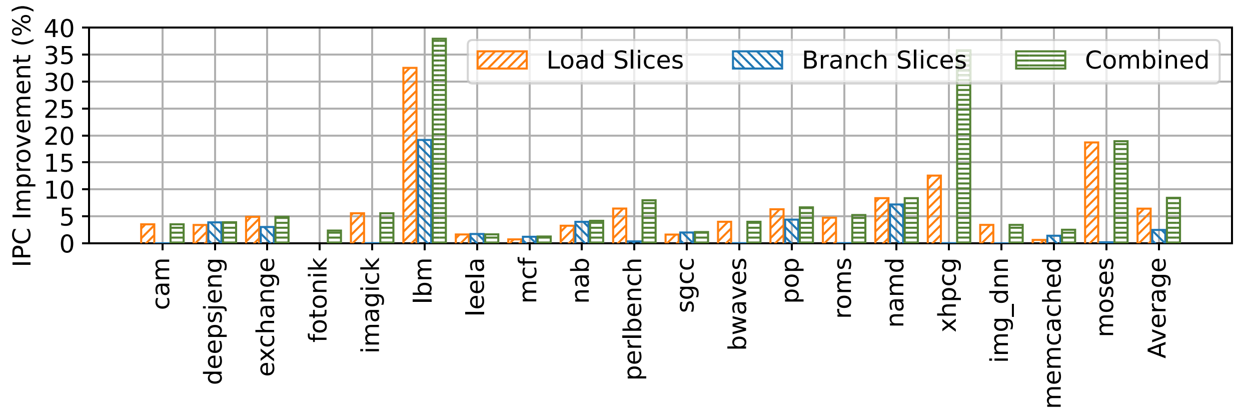


Figure 8: Performance impact of branch slices, load slices, and their combination

5.2 IPC Improvement of *CRISP*

We evaluate the IPC improvement provided by *CRISP* over the *OOO* baseline in Figure 7. We also compare *CRISP* to a hardware-only design referred to as *IBDA* which performs load slice extraction via iterative backwards dependency analysis. IBDA utilizes the same OOO core as *CRISP*, while adding a 4-way set associative instruction slice table (IST) with 1024 entries as proposed in the load-slice architecture [20] for iterative backwards dependency analysis. We also evaluate an 8K entry (8-way), 64K entry (16-way), and infinitely sized IST. IBDA also maintains a 32 entry delinquent load table to capture the most frequently executed loads missing the LLC. We compare *CRISP* to IBDA to evaluate the advantages of software-based load and branch slice extraction.

Figure 7 shows that *CRISP* provides an average IPC speedup of 8.4% and a maximum speedup of 38% over the OOO baseline. *CRISP* also significantly outperforms IBDA which achieves an average improvement of only 4.3× % over the baseline. There exist various reasons why IBDA cannot provide the same performance level as *CRISP*. In `moses`, load slices are too long and too large to be captured by the IST. For `lbm`, IBDA suffers by lacking the capability of extracting branch slices as even for an infinite IST, IBDA underperforms *CRISP*. In `namd` and `Xhpcg`, IBDA misses important load slices due to its inability of following dependencies through memory. In `bwaves`, IBDA captures the wrong delinquent loads which, although exhibiting high LLC MPKI, are not performance-critical as they are executed in phases of high MLP. For `fotonik, perlbench and moses`, IBDA selects too many (non-critical) instructions from the load slice (as it lacks critical path analysis) inducing a performance reduction over the baseline. Furthermore, note that *CRISP* introduces virtually no hardware overheads whereas IBDA introduces significant processor modifications and metadata storage overheads. We do not show MPKI numbers as *CRISP* only reorders memory accesses without reducing cache misses. A helpful metric to confirm the IPC gains is to count the cycles that instructions reside at the head of the ROB without retiring. We can observe that *CRISP* indeed reduces these stall cycles as reflected by the IPC gains.

5.3 Branch Slicing

CRISP leverages load slices and branch slices to execute long latency instructions as early as possible. We developed branch slicing after observing that in the case of `lbm`, the benefits from load slicing were significantly increased when enabling a perfect branch predictor. The reason is that frequent branch mispredictions prevent the decoupled frontend from running ahead and fetching sufficient instructions into the reservation stations, which prohibits the scheduler to exploit criticality efficiently. Figure 8 shows the IPC performance gain of *CRISP* when utilizing load slices, branch slices, and both combined. We can see that `cactus`, `lbm`, `perlbench`, and `memcached` synergistically utilize branch and load slices to improve performance as the combined performance is both higher compared to only prioritizing branch- or load slices. `deepsjeng`, `lbm`, `nab`, `namd` provide over 3% IPC gains just from prioritizing branch slices, showing that critical branch prioritization is a useful technique on its own.

5.4 Reservation Station Size Sensitivity Study

In *CRISP*, the reservation station (RS)–and to a lesser degree, the ROB size–determine the opportunities for the scheduler to reorder instructions. As future microprocessors are likely to increase the size of these structures (e.g., Intel's Sunny Cove architecture increased the RS size from 96 to 128 entries over Intel Skylake), we analyze the performance improvements of *CRISP* for an RS and ROB increased by 50% and 100%, respectively. As shown in Figure 9, *CRISP* provides significant performance improvement across the different RS/ROB configurations. `Xhpcg` benefits significantly from larger structures improving its IPC gain from 12.5% (Skylake) to over 25% for a Sunny-Cove-like core. On the other hand `moses` exhibits the largest performance gains for the smaller 64RS/180ROB configuration. Analyzing the results we found that for `moses` a large ROB improves performance significantly and hence the relative improvement provided by *CRISP* is smaller. `Xhpcg` can realize additional prioritization opportunities from *CRISP* with a large ROB enabling significant performance gains.

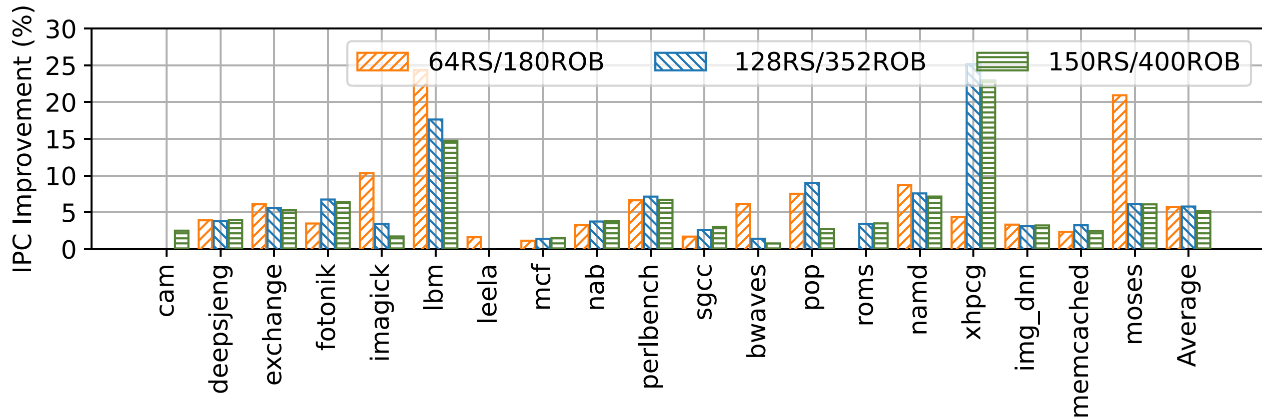


Figure 9: Impact of large RS and ROB on *CRISP* performance

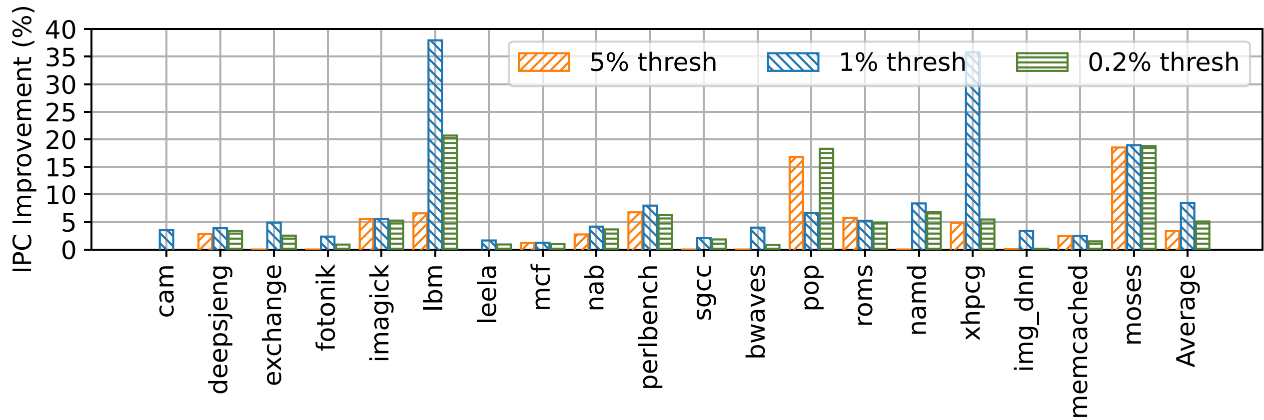


Figure 10: Sensitivity study for miss rate criticality threshold

5.5 Load Slice Criticality Threshold Study

The flexible software-based design of *CRISP* enables sophisticated heuristics to classify between critical and non-critical instructions. Here we evaluate one important control knob; the miss ratio of delinquent loads. Tagging delinquent loads with a high execution frequency but low miss rate as critical can be problematic, as even loads that are likely to be served by the on-chip caches will be prioritized by the scheduler. This reduces the opportunity for prioritizing other critical instructions that are actually going to miss the LLC. In Figure 10 we compare *CRISP* performance for three configurations varying the miss threshold T from 5% to 1% to 0.2%. In particular, *CRISP* prioritizes a load if it contributes greater than T misses of the total misses experienced by the application. As can be seen, a miss threshold of 1% provides the best overall performance, while, for instance, moses performs best with a miss threshold of 2%. For Most other applications this miss threshold provides reduced performance gains. For future work, we envision an iterative mechanism that profiles applications with different miss ratio thresholds to enable additional application-specific optimizations.

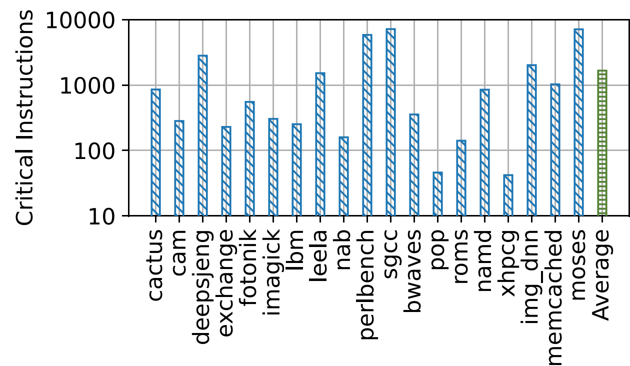


Figure 11: Total number of critical instructions

5.6 Total Number of Critical Instructions

In Figure 11 we show the total number of critical instructions of all load and branch slices for the evaluated applications. For

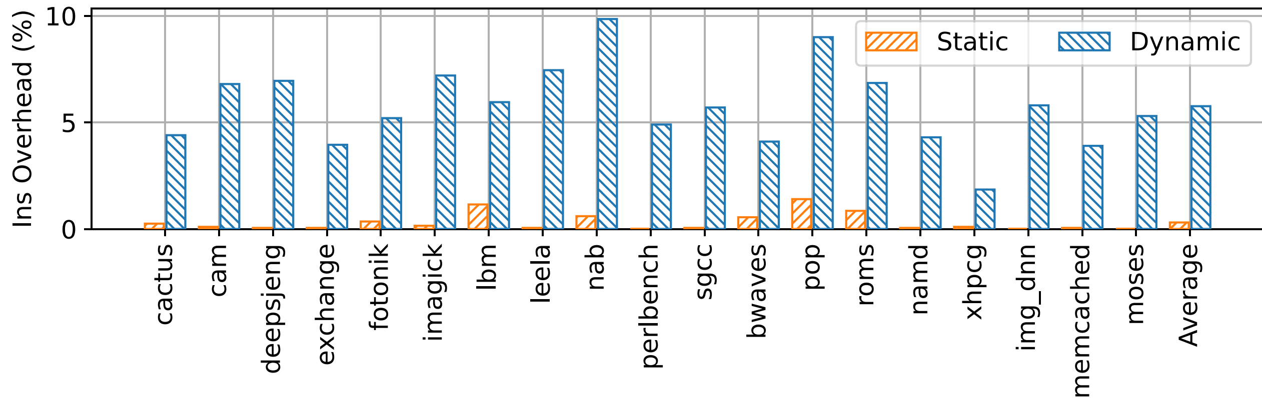


Figure 12: Static and dynamic instruction footprint overhead introduced by the *CRISP* prefix

perlbench, gcc, and moses, *CRISP* classifies over 10,000 unique instructions as critical. To perfectly capture those instructions, hardware techniques such as IBDA would have to deploy 100's of KB of meta-data storage. *CRISP*, in contrast, only adds a single byte prefix per critical instruction and by storing prefixes as part of the instruction footprint additional storage structures are avoided.

5.7 Instruction Prefix Overhead

CRISP introduces a new instruction prefix to identify load and branch slices, increasing the static and dynamic (considering the execution frequency of instructions) code footprint of applications by one byte per critical instruction as shown in Figure 12. While the static code size increases only minimally, the dynamic code footprint increases more significantly, by 5.2% in average, as critical instructions are common in hot loops. While this increases the pressure on the icache, we observed a worst-case increase in instruction cache MPKI of only 2.6% for our evaluated applications.

6 DISCUSSION

We explore additional techniques to exploit criticality and discuss the impact of *CRISP* on security.

6.1 Further Exploiting Criticality

We envision additional opportunities for leveraging the newly-proposed instruction prefix. First, other high-latency instructions such as division can be accelerated with *CRISP*. Here, the challenge is to determine the exact performance impact of a specific instruction as, in the case of division, the latency can depend on the input operands. Therefore, we envision adding new events to the PMU for determining the PC of arbitrary instructions that induce significant stall cycles. Another class of instructions that may benefit from criticality support are vector instructions such as AVX-512, which are becoming increasingly important in the machine learning domain. Prior works [32, 45, 101] have observed a significant transition latency leading to CPU stalls when the first AVX-512 instruction is detected after a phase of AVX-512 inactivity. By prioritizing the first AVX-512 instruction, the vector unit can be enabled before additional AVX-512 instructions clog the pipeline. Second, criticality information can be leveraged by other components in the system aside from the scheduler. For instance, depending on the memory consistency model, the latency of delinquent loads may be further reduced by re-ordering them in the load queues, on-chip interconnection network [77], and DRAM scheduler [42]. Third, criticality can be leveraged to increase performance predictability for enforcing service level objectives (SLO) such as tail latency requirements. In an SMT [34]-enabled processor, the instructions of a latency-sensitive thread can be prioritized over instructions of a latency-insensitive thread enabling both high CPU utilization while enforcing SLOs.

6.2 Security Implications

Speculative execution attacks such as Spectre [65] and Meltdown [72] have shown how to leak information through covert side channels. We discuss the potential effects of *CRISP* on security, focusing on its hardware components as the software analysis and tracing techniques leveraged by *CRISP* are already available today. One possible attack enabled by *CRISP* is to execute an application with all critical or all non-critical instructions on a hardware thread to leak the number of critical instructions of a separate thread running on the same SMT core. By observing the criticality of instructions of a neighboring thread in certain application phases, it may be possible to extract signatures identifying the other application running on the same core. We claim that leaking criticality is less severe than leaking data through SMT as previously demonstrated by PORTSMASH [2], TLBleed [46], CacheBleed [120], and MemJam [78]. The previously-proposed mitigation techniques such as disabling SMT and leveraging port-independent code are equally applicable to *CRISP*.

Prioritizing instructions of one hyperthread over another hyperthread on the same core introduces a potential denial of service attack (DoS) by simply tagging all instructions of a program as critical. Here, a mitigation technique that guarantees forward progress of the victim thread is to reserve some execution resources ensuring that every SMT thread is guaranteed to execute a certain number of instructions per time period. This problem can also be addressed by policies guaranteeing the scheduling of some non-critical instructions in the presence of abundant ready, critical instructions, or by simple round-robin arbitration between threads.

7 RELATED WORK

Prior work has addressed the performance challenges introduced by long-latency DRAM accesses via latency-avoiding and latency-toleration techniques. There exists a large body of work on prefetching regular memory accesses. Stream prefetchers [52, 99] and pattern-based prefetchers [28, 53, 61, 62, 76, 86, 94, 105, 108] learn the delta between the effective addresses of subsequent cache misses to predict future accesses. These designs introduce moderate hardware complexity and can prefetch simple stride and periodic patterns.

Spatial prefetchers [9, 12, 37, 111] have higher metadata storage overheads than delta prefetchers, allowing them to memorize arbitrary lines within a page for prefetching. As a result, these types of prefetchers can increase coverage over delta prefetchers. However, they are still limited to prefetching recurrent patterns and hence fail to prefetch, for instance, linked list traversals and other irregular memory access patterns. *CRISP* can be combined with these prior approaches to increase coverage by reducing the miss penalty of irregular memory accesses. Temporal prefetchers [8, 49, 54, 119, 122] track the temporal order of cache line accesses based on Markov prefetching [55] introducing significant storage overheads in the order of megabytes in contrast to *CRISP*. Runahead prefetchers [6, 33, 48, 82–84, 89, 95] and helper threads [21, 23, 24, 70, 73, 74, 110, 117, 123, 124] prefetch irregular memory accesses as in linked-list traversals, however, they introduce significant hardware complexity or consume separate SMT-threads [34] whereas *CRISP* requires only minimal hardware modifications. Branch runahead [93] addresses hard-to-predict branches by prioritizing their execution minimizing the missprediction penalty. As other runahead techniques, this approach introduces significant hardware complexity and pipeline modifications.

Latency toleration techniques such as OOO execution [19, 116] hide high memory access latency by executing independent instructions instead of stalling the CPU pipeline. In contrast to *CRISP*, OOO techniques, however, fail to improve performance if there do not exist sufficient independent instructions after the delinquent load (see Figure 1). Instruction criticality has been leveraged to improve scheduling in prior works including Fiforder [4], Long-term parking [102], and Delay-and-Bypass [3]. These works partition the instruction queue into smaller sub-queues holding ready, non-ready, critical, and non-critical instructions to improve the scheduling energy-efficiency. The Load-Slice (LC) [20], Forward Slice Core [67], Freeway [66], and Front-end Execution Architecture [106] works propose architectures in which non-critical instructions are executed by an in-order pipeline while loads are allowed to bypass. In contrast to *CRISP*, all of these prior works do not improve delinquent load latency and, in fact, often reduce performance by 5% [102] to 9% [3] over an OOO baseline. Criticality Driven Fetch [29] proposes to determine critical instruction chains and prioritizes their fetch, allocation, and execution. This work requires considerable modifications of the entire processor pipeline. NOREBA [47] proposes a hardware-software co-designed technique to early-retire non-speculative instructions for freeing up slots in the ROB. While this technique enables issuing additional instruction after the load it does not enable issuing instructions before the load. Balasubramonian [10], Subramaniam [115], and Nori [87] have leveraged load-criticality to optimize data allocation in caches to reduce the latency and power consumption of caches requiring significant cache modifications in contrast to *CRISP*. All these works determine criticality solely in hardware introducing complexity and overheads while limiting accuracy. For instance, LC [20] treats all loads as critical instructions because measuring the cache miss rate for each load for a more precise measure would introduce substantial metadata storage overheads. Prior works perform load slice extraction via IBDA [20, 125] leading to incomplete slices as they lack the capability of observing instruction dependencies through memory. *CRISP*, in contrast, enables sophisticated, flexible, and application-specific metrics for determining instruction criticality while enabling the extraction of comprehensive load slices. Furthermore, *CRISP* introduces branch slices over prior work to reduce the performance impact of branch mispredictions.

Further techniques optimizing load scheduling include the Recovery Buffer [79, 80] and the Waiting Instruction Buffer [69] which move the dependent instructions of a cache miss into a separate queue to reduce the size of the main instruction queue. In contrast to *CRISP*, these techniques target instructions *after* a delinquent load for improving energy-efficiency. Srinivasan [112, 113], Fisk [40], Fields [38] and Muthler [81] have explored deferring instructions based on their slack, defined as the time an instruction can be delayed without affecting performance. These approaches serve loads with high slack from slower hardware resources providing additional resources to latency-critical loads. In contrast to *CRISP*, the approaches introduce significant hardware overheads and complexity due to redesigning the processor frontend. Furthermore, estimating the slack requires microarchitectural simulation suffering from inconsistencies between the simulator and the real hardware. Focused value prediction [11] tries to predict the results feeding into long-latency loads to reduce the critical path length of dependent instructions. Value prediction is complementary to *CRISP* as breaking dependency chains exposes more ILP, generating additional opportunities for criticality-based scheduling mechanisms.

8 CONCLUSION

Memory accesses missing the last-level cache suffer from high DRAM latency which introduces pipeline stalls that greatly reduce performance. Out-of-order execution fails to hide the memory access latency if most of the subsequent instructions depend on the missed memory access. Prefetching techniques are either limited to handling easy-to-predict regular memory access patterns or they introduce unacceptable hardware complexity or cost. We propose *CRISP*, a technique to prefetch critical memory accesses and their load-address-generating instructions by prioritizing them in the instruction scheduler. *CRISP* is predominantly implemented in software to increase flexibility and performance by leveraging load-slice filtering and by observing dependencies through memory. *CRISP* also introduces branch slicing, a technique that reduces the impact of branch mispredictions, especially when combined with critical load prefetching. Our techniques improve the IPC performance of modern microprocessors for memory-latency-bound applications by up to 38% and by 8.4% on average.

ACKNOWLEDGEMENTS

This work was supported by Google and the Intel Corporation.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2019. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 673–686.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[3] Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer, and Rakesh Kumar. 2020. Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 424–434.

[4] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. 2019. Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 716–721.

[5] Hideki Ando. 2019. SWQUE: A Mode Switching Issue Queue with Priority-Correcting Circular Queue. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 506–518.

[6] 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*. 513–526.

[7] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. 462–473.

[8] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[9] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 399–411.

[10] Rajeev Balasubramonian, Viji Srinivasan, Sandhya Dwarkadas, and Alper Buyuktosunoglu. 2003. Hot-and-cold: Using criticality in the design of energy-efficient caches. In *International Workshop on Power-Aware Computer Systems*. Springer, 180–195.

[11] Sumeet Bandishte, Jayesh Gaur, Zeev Sperber, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. 2020. Focused Value Prediction*: Concepts, techniques and implementations presented in this paper are subject matter of pending patent applications, which have been filed by Intel Corporation. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 79–91.

[12] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. Dspatch: Dual spatial pattern prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 531–544.

[13] David Bernstein and Izidor Gertner. 1989. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 1 (1989), 57–66.

[14] David Bernstein, Michael Rodeh, and Izidor Gertner. 1989. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Transactions on computers* 38, 9 (1989), 1308–1313.

[15] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. 2011. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1071–1080.

[16] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*.

[17] Derek Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[18] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*.

[19] Michael Butler and Yale Patt. 1992. An investigation of the performance of various dynamic scheduling techniques. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 1–9.

[20] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 272–284.

[21] 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 International Symposium on Computer Architecture (Cat. No. 99CB36367)*. IEEE, 186–195.

[22] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 12–23.

[23] Jamison D Collins, Dean M Tullsen, Hong Wang, and John Paul Shen. 2001. Dynamic speculative precomputation. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 306–317.

[24] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 14–25.

[25] Intel Corparation. 2016. Intel (R) 64 and IA-32 Architectures Software Developer's Manual. *Combined Volumes, Dec* (2016).

[26] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse Debugging of Failures in Deployed Software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 17–32.

[27] Ian Cutress. 2019. Examining Intel's Ice Lake Processors: Taking a Bite of the Sunny Cove Microarchitecture.

[28] Fredrik Dahlgren and Per Stenstrom. 1995. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors.. In *hpca*. 68–77.

[29] Aniket Deshmukh and Yale N Patt. 2021. Criticality Driven Fetch. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 380–391.

[30] Jack Dongarra, Piotr Luszczek, and M Heroux. 2013. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752* (2013).

[31] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* (2017).

[32] Travis Downs. [n.d.]. Gathering Intel on Intel AVX-512 Transitions. https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html.

[33] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing*. Citeseer, 68–75.

[34] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. 1997. Simultaneous multithreading: A platform for next-generation processors. *IEEE micro* 17, 5 (1997), 12–19.

[35] John R Ellis. 1985. *Bulldog: A compiler for VLIW architectures.* Technical Report. Yale Univ., New Haven, CT (USA).

[36] James A Farrell and Timothy C Fischer. 1998. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits* 33, 5 (1998), 707–712.

[37] Michael Ferdman, Stephen Somogyi, and Babak Falsafi. 2009. Spatial memory streaming with rotated patterns. *1st JILP Data Prefetching Championship* 29 (2009).

[38] Brian Fields, Rastislav Bodík, and Mark D Hill. 2002. Slack: Maximizing performance under technological constraints. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 47–58.

[39] Joseph A Fisher. 1983. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th annual international symposium on Computer architecture*. 140–150.

[40] Brian R Fisk and R Iris Bahar. 1999. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*. IEEE, 538–545.

[41] Agner Fog et al. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[42] Saugata Ghose, Hyodong Lee, and José F Martínez. 2013. Improving memory scheduling via processor-side load criticality information. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 84–95.

[43] Michael Golden, Srikanth Arekapudi, and James Vinh. 2011. 40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86–64 core. In *2011 IEEE International Solid-State Circuits Conference*. IEEE, 80–82.

[44] Google. [n.d.]. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller.

[45] Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. 2020. AVX overhead profiling: how much does your fast code slow you down?. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 59–66.

[46] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 955–972.

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

[48] 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*. IEEE Press, 61.

[49] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329* (2018).

[50] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[51] HPS. [n.d.]. Scarab. https://github.com/hpsresearchgroup/scarab.

[52] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 397–408.

[53] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13 (2011), 1–24.

[54] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 247–259.

[55] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *ACM SIGARCH Computer Architecture News*, Vol. 25. ACM, 252–263.

[56] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[57] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC)*.

[58] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[59] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 146–159. https://doi.org/10.1109/MICRO50266.2020.00024

[60] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA), ISCA*.

[61] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[62] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices* 52, 4 (2017), 737–749.

[63] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[64] Andi Kleen and Beeman Strong. 2015. Intel processor trace on linux. *Tracing Summit* 2015 (2015).

[65] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[66] Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2019. Freeway: Maximizing mlp for slice-out-of-order execution. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 558–569.

[67] Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2020. The Forward Slice Core Microarchitecture. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 361–372.

[68] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*. 65–75.

[69] Alvin R Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. *ACM SIGARCH Computer Architecture News* 30, 2 (2002), 59–70.

[70] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. 2008. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 9 (2008), 1309–1324.

[71] Johnny KF Lee and Alan Jay Smith. 1984. Branch prediction strategies and branch target buffer design. *Computer* 1 (1984), 6–22.

[72] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.

[73] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G Abraham. 2005. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 93–104.

[74] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 40–51.

[75] R Manikantan, R Govindarajan, and Kaushik Rajan. 2011. Extended histories: improving regularity and performance in correlation prefetchers. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. 67–76.

[76] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.

[77] Joshua San Miguel and Natalie Enright Jerger. 2015. Data criticality in network-on-chip design. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. 1–8.

[78] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* 47, 4 (2019), 538–570.

[79] Enric Morancho, José María Llabería, and Àngel Olivé. 2001. Recovery mechanism for latency misprediction. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 118–128.

[80] Enric Morancho, José María Llabería, and Àngel Olivé. 2007. On reducing energy-consumption by late-inserting instructions into the issue queue. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED'07)*. IEEE, 371–374.

[81] Gregory A Muthler, David Crowe, Sanjay J Patel, and Steven S Lumetta. 2002. Instruction fetch deferral using static slack. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings*. IEEE, 51–61.

[82] Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2005. Techniques for efficient processing in runahead execution engines. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 370–381.

[83] Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2006. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro* 26, 1 (2006), 10–20.

[84] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro* (2003).

[85] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. 2020. Precise runahead execution. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 397–410.

[86] Kyle J Nesbit and James E Smith. 2004. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*. IEEE, 96–96.

[87] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 96–109.

[88] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[89] Reena Panda, Paul V Gratz, and Daniel A Jiménez. 2011. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters* 11, 2 (2011), 41–44.

[90] Miquel Pericas, Adrian Cristal, Francisco J Cazorla, Ruben González, Alex Veidenbaum, Daniel A Jiménez, and Mateo Valero. 2008. A two-level load/store queue based on execution locality. In *2008 International Symposium on Computer Architecture*. IEEE, 25–36.

[91] Miquel Pericas, Adrian Cristal, Ruben González, Daniel A Jiménez, and Mateo Valero. 2006. A decoupled kilo-instruction processor. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006*. IEEE, 53–64.

[92] Ronald P Preston, Roy W Badeau, Daniel W Bailey, Shane L Bell, Larry L Biro, William J Bowhill, Daniel E Dever, Stephen Felix, Richard Gammack, Valeria Germini, et al. 2002. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 02CH37315)*, Vol. 1. IEEE, 334–472.

[93] Stephen Pruett and Yale Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 804–815.

[94] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 626–637.

[95] Tanausu Ramirez, Alex Pajuelo, Oliverio J Santana, and Mateo Valero. 2008. Runahead threads to improve SMT performance. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 149–158.
[96] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.
[97] Glenn Reinman, Brad Calder, and Todd Austin. 2001. Optimizations enabled by a decoupled front-end architecture. *IEEE Trans. Comput.* 50, 4 (2001), 338–355.
[98] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro* (2010).
[99] Suleyman Sair, Timothy Sherwood, and Brad Calder. 2003. A decoupled predictor-directed stream prefetching architecture. *IEEE Trans. Comput.* 52, 3 (2003), 260–276.
[100] Peter G Sassone, Jeff Rupley, Edward Brekelbaum, Gabriel H Loh, and Bryan Black. 2007. Matrix scheduler reloaded. *ACM SIGARCH Computer Architecture News* 35, 2 (2007), 335–346.
[101] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. 2019. Energy efficiency features of the intel skylake-sp processor and their impact on performance. *arXiv preprint arXiv:1905.12468* (2019).
[102] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. 2015. Long term parking (LTP) criticality-aware resource allocation in OOO processors. In *Proceedings of the 48th International Symposium on Microarchitecture*. 334–346.
[103] André Seznec. 2006. A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism* (2006).
[104] Harsh Sharangpani. 1999. Intel® Itanium™ processor microarchitecture overview. In *Microprocessor Forum*, Vol. 10.
[105] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 141–152.
[106] Ryota Shioya, Masahiro Goshima, and Hideki Ando. 2014. A front-end execution architecture for high energy efficiency. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 419–431.
[107] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.
[108] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 12 (1978), 7–21.
[109] James E Smith. 1998. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*. 202–215.
[110] Yan Solihin, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 171–182.
[111] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 252–263.
[112] Srikanth T Srinivasan, R Dz-Ching Ju, Alvin R Lebeck, and Chris Wilkerson. 2001. Locality vs. criticality. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 132–143.
[113] Srikanth T Srinivasan and Alvin R Lebeck. 1998. Load latency tolerance in dynamically scheduled processors. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 148–159.
[114] Jared Stark, Paul Racunas, and Yale N Patt. 1997. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 34–43.
[115] Samantika Subramaniam, Anne Bracy, Hong Wang, and Gabriel H Loh. 2009. Criticality-based optimizations for efficient load processing. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 419–430.
[116] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.
[117] Perry H Wang, Jamison D Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B Yunus, Terry Sych, Stephen F Moore, and John P Shen. 2004. Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform. *ACM SIGPLAN Notices* 39, 11 (2004), 144–155.
[118] Vincent M Weaver. 2016. *Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface*. Technical Report. Tech. rep. UMAINEVMW-TR-PEBS-IBS-SAMPLING-2016-08. http://web. eece. maine ....
[119] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 79–90.
[120] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.
[121] Tse-Yu Yeh and Yale N Patt. 1992. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News* 20, 2 (1992), 124–134.
[122] Chao Zhang, Yuan Zeng, John Shalf, and Xiaochen Guo. 2020. RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[123] Weifeng Zhang, Dean M Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for effcient prefetching. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 85–95.
[124] Craig Zilles and Gurindar Sohi. 2001. Execution-based prediction using speculative slices. In *International Symposium on Computer Architecture*.
[125] Craig B Zilles and Gurindar S Sohi. 2000. Understanding the backward slices of performance degrading instructions. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 172–181.