# Criticality Driven Fetch

Aniket Deshmukh
a.deshmukh@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Yale N. Patt
patt@ece.utexas.edu
University of Texas at Austin
Austin, Texas, USA

## ABSTRACT

**Modern OoO cores achieve high levels of performance using large instruction windows. Scaling the window size improves performance by making visible more of the parallelism present in programs. However, this leads to an exponential increase in area and power. We specify Criticality Driven Fetch (CDF), a new execution paradigm that preferentially fetches, allocates, and executes instructions on the critical path of the program. By skipping over non-critical instructions, critical instructions in the ROB can span a sequential instruction window larger than the size of the ROB. This increases the amount of parallelism that can be extracted from critical instructions, thereby improving performance.**

**In our implementation, CDF improves performance by (a) increasing the MLP for independent loads executing concurrently, (b) fetching critical path loads past hard-to-predict branches (by resolving them earlier), and (c) by initiating last level cache misses that cannot be parallelized earlier. Accelerating critical loads using CDF achieves a 6.1% IPC improvement over a baseline OoO core with prefetching. Compared to Precise Runahead, the prior state of the art work on accelerating last level cache misses on the core, we provide better performance and reduce memory traffic and energy consumption by 4.0% and 7.2% respectively.**

## KEYWORDS

OoO execution, memory level parallelism, instruction criticality

**ACM Reference Format:**
Aniket Deshmukh and Yale N. Patt. 2021. Criticality Driven Fetch . In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480115

## 1 INTRODUCTION

Single threaded performance remains an important aspect of improving program runtime on modern OoO cores. These cores use large instruction window resources - reorder buffer (ROB), reservation stations (RS), load and store queues (LQ and SQ), and physical registers that extract Instruction Level Parallelism (ILP) and Memory Level Parallelism (MLP) to drive performance. Historically, performance of OoO cores has been improved by scaling OoO window resources. Increasing the window size increases the number of instructions that can be dispatched to execution units in parallel (improving ILP) and increases the number of independent loads that can access memory concurrently (improving MLP). However, doing this is expensive since area and power scale exponentially with window size.

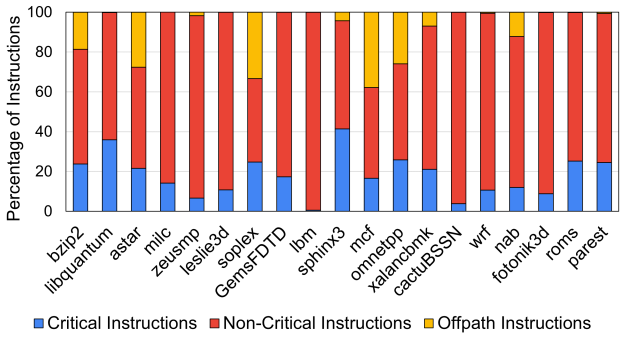


**Figure 1: Distribution of Instructions in the ROB during Full Window Stalls**

Program performance is primarily governed by instructions on the critical path of program execution, i.e., Last level cache (LLC) misses, branch mispredictions, and instructions in the dependence chains leading up to them. As these account for only 10%-40% of the dynamic instruction footprint in typical (SPEC-like) programs, during full window stalls, the processor pipeline contains more non-critical instructions than critical ones (Fig. 1). Since improving the performance of non-critical instructions does not affect the overall program runtime significantly, this leads to inefficient use of the expensive OoO structures. In other words, OoO resources are wasted on executing non-critical instructions, limiting the parallelism that could be extracted from critical path instructions.

To improve performance efficiently, we need to target instructions on the critical path. We do this by re-ordering the instructions stream so that critical instructions are prioritized while fetching. As a result, these instructions are decoded, renamed and executed earlier, and allocated more window resources. Determining an ideal fetch and schedule order at compile time is problematic as runtime events such as cache misses and branch mispredictions depend on input data and impact which dynamic instructions are critical. Prior work that targets compiler-based optimizations or uses the compiler to construct program segments to accelerate critical instructions fails to capture an optimal set of critical instructions. This includes purely compiler-based solutions [21, 30, 31], work on Speculative Multi-threading [17, 22, 34], Helper Threads [6, 10] and Decoupled Lookahead [7, 13].

Criticality Driven Fetch (CDF) identifies critical instructions and constructs their dependence chains at runtime. These chains are


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

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480115

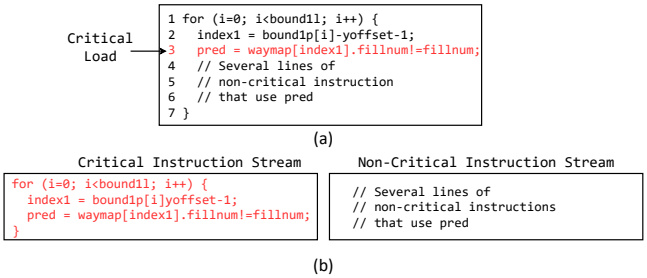


**Figure 2: Code segment from the SPEC benchmark *astar***

stored in a cache and are fetched, allocated and executed before non-critical instructions. Since non-critical instructions are initially skipped, the OoO core is filled preferentially with critical instructions, thereby spanning a sequential instruction window that is larger than the size of the ROB. This improves the performance of critical instructions at the cost of delaying the execution of non-critical ones. This tradeoff improves the overall performance since non-critical instructions affect execution time less. If critical instructions are picked appropriately, an OoO core with CDF can provide the same performance as a core with a much larger OoO window without the scaling overhead.

While CDF can accelerate all critical instructions, we focus (in this paper) on loads that miss in the LLC as these are often on the critical path and are easy to identify. CDF improves the MLP for independent critical loads by re-ordering the instruction stream. CDF initiates both independent and dependent critical loads earlier compared to a baseline OoO core which wastes cycles fetching non-critical instructions. We also mark hard-to-predict branches as critical, which enables quicker branch recovery and allows CDF to restart fetching critical instructions past branch mispredictions.

**In all, our contributions are as follows:**

- We introduce Criticality Driven Fetch, a new paradigm for OoO execution. CDF identifies critical instruction chains at runtime and prioritizes their fetch, allocation, and execution to improve performance.
- We present an implementation for CDF on an OoO core targeting critical loads and outline all the microarchitectural changes needed to support it.
- We show that with respect to accelerating cache misses on the core, CDF provides better speedup than the prior state of the art (Precise Runahead), 6.1% vs 2.6%. CDF also reduces energy consumption (by 7.2%) and extra memory traffic (by 4%) compared to Precise Runahead.

## 2 CRITICALITY DRIVEN FETCH

We discuss the key benefits that CDF provides and how they improve performance over the next few subsections. We also discuss how these benefits make CDF a better solution for accelerating critical loads compared to Runahead and Compiler based techniques.

### 2.1 Improving MLP

Fig. 2 (a) shows a code segment from the SPEC benchmark *astar*. Line 3 contains an array access whose index is a function of data in memory and is fairly random. Moreover, this array is large and does not fit in the LLC. As a result, most dynamic instances of the

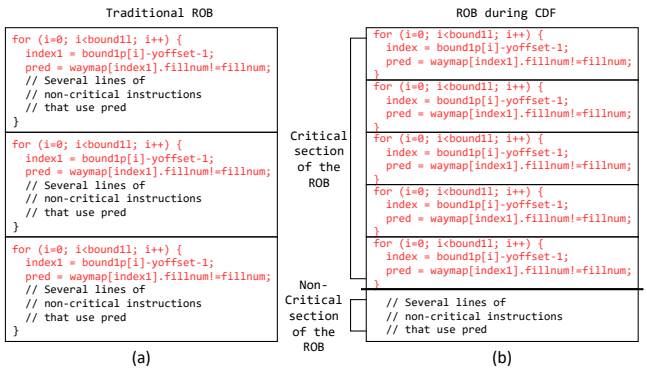


**Figure 3: Filling the code segment into the processor instruction window**

Load that reads from the array in line 3 are LLC misses and cause a full window stall. Note that the Load in line 2 sees almost no cache misses as this array is accessed sequentially and can be effectively prefetched. Fig. 3 (a) shows the processor instruction window during a full window stall while executing this code. In this example, three instances of the loop and hence three dynamic instances of the load can fit in the instruction window. Since the critical loads in subsequent loop iterations only need the loop counter and an array access that is a cache hit to compute their address, these loads can access memory in parallel (they are independent loads). The instructions in the loop after the critical load do not incur any long latency stalls and the performance of this code segment thus depends only on the critical load. Expanding the size of the ROB helps improve MLP as the processor can now fit more instances of the critical load. However, notice that the ROB also contains many non-critical instructions which prevents the instruction window from holding more instances of the critical load.

Fig. 2 (b) shows the same code segment, but split into critical and non-critical instructions. The load and all the instructions in its dependence chain are classified as critical. The rest are marked non-critical. If the critical instructions are fetched and allocated preferentially into the ROB (which is **partitioned into two sections** with a larger section assigned to critical instructions) as shown in Fig. 3 (b), the processor is able to hold and initiate more instances of the critical load. This improves MLP significantly. Non-critical instructions are relegated to a much smaller section of the ROB. The non-critical instructions corresponding to subsequent iterations of the loop are fetched when this smaller section is freed up which reduces their throughput since we keep the non-critical section of the ROB small. However, this does not affect the overall execution time much since the performance of this code segment depends only on the critical load. CDF can thus achieve the same amount of parallelism as a processor with a much larger ROB without the scaling overhead.

### 2.2 Fetching Critical Loads past Hard-to-Predict Branches

Branch mispredictions limit the number of correct-path critical instructions that CDF can fetch ahead. This can eliminate the benefit of improved MLP as off-path critical loads may not have correct addresses. CDF can fetch useful critical loads past hard-to-predict

branches if these branches are marked critical. When marked critical, hard-to-predict branches are fetched, executed, and resolved earlier along with other critical instructions. This allows CDF to continue fetching critical instructions on the correct path after mispredicted branches are resolved without having to wait for intermediate non-critical instructions.

### 2.3 Initiating Critical Loads Earlier

When fetching critical instructions, CDF skips over non-critical ones, increasing the effective frontend bandwidth for critical instructions. As a result, both dependent and independent critical loads are fetched and initiated earlier compared to a baseline OoO core. This effect is prominent in applications where loads that cause full window stalls are far apart (more than 1000 instructions away). The performance of these programs is still limited by these loads as the intermediate instructions between the critical loads can be executed quickly. However, it takes many cycles to reach and initiate and next critical load since all the intermediate non-critical instructions need to be fetched. Even though CDF is unable to extract MLP from these loads, it is able to significantly reduce the number of intermediate instruction fetch thus improving the performance of these applications.

### 2.4 Comparison Against Runahead and Compiler Based Techniques

Runahead [8, 9, 19] executes dependence chains corresponding to multiple cache misses either on a full window stall on the core, or in parallel on a separate Runahead engine. It is similar to CDF in that it executes dependence chains for critical loads in order to increase MLP. The work most similar to CDF is Precise Runahead [20].

Precise Runahead runs dependence chains corresponding to multiple cache misses on a full window stall and allows branches to be predicted when fetching these chains. It uses empty Reservation Stations and Physical Registers to reduce the penalty for entering and exiting Runahead mode, enabling smaller Runahead intervals compared to prior work. **CDF has the following key advantages over any Runahead scheme on the core:**

(a) Runahead execution is limited by the full window stall duration. This can significantly reduce the benefits of Runahead on applications that have shorter and fewer full window stalls and gets worse with better memory systems. CDF is unaffected by this. (b) If the branches encountered while fetching Runahead instructions are mispredicted, subsequent off-path loads may execute incorrectly. This reduces the performance of Runahead on applications that have a high branch MPKI. Marking hard-to-predict branches critical allows CDF to fetch correct-path critical loads beyond these branches by resolving them early. (c) Applications with full window stalls that are spaced far apart do not benefit from Runahead as it is unable to fetch instructions far enough to reach the next critical load during the full window stall. While CDF is unable to extract parallelism from these loads, it initiates the next critical load more quickly thus improving performance. (d) Runahead chains can be incorrect and generate a lot of additional memory traffic. Moreover, Runahead instructions are duplicates that are executed twice on the core. CDF does not have this overhead as the critical instructions are part of the main instruction stream.

Criticality driven fetch is not fundamentally limited to loads and can be expanded to any instructions in the program that are critical. It is therefore a more general solution.

Compilers can identify critical loads through profiling and rearrange code by unrolling loops and hoisting critical loads to achieve the same effect as CDF. They can avoid the hardware overhead of constructing critical load dependence chains at runtime. However, they have a few major flaws.

Profiling can find the set of static loads likely to miss in the LLC at runtime but cannot accurately predict which dynamic instances of the loads miss [21]. Input data and runtime behavior usually determine whether a load is critical. For example (Figure 2), the index of the array access on line 3 depends on the contents of another array (*bound1p*). Depending on the nature of the data in this input-dependent array (*bound1p*), the load in question could be critical or hit in the caches frequently. This can only be determined at runtime. The contents of *bound1p* can also be modified by the program, which can result in the load on line 3 changing its behavior across program phases. This occurs in many workloads and compilers either fail to capture these critical loads or end up reordering code around non-critical loads which does not improve performance. Moreover, load hoisting is limited by architectural register pressure [30]; CDF does not have this problem since critical instructions communicate their values through physical registers.

## 3 IMPLEMENTATION

### 3.1 Overview

Implementing CDF on an OoO core requires additional structures in various stages of the pipeline and changes to how instructions are allocated in the Reorder Buffer, Reservation Stations, and Load and Store Queues. An overview of the changes is shown in Fig. 4.

As instructions retire, the criticality of retired loads and branches is predicted using the Critical Count Tables. The Fill Buffer records the last 1024 retired uops. When full, it is walked starting from the youngest uop and all the uops in the dependence chains of loads and branches that were predicted critical are also marked critical. Following this, the critical uops are collected into (decoded) uop traces and added to the Critical Uop Cache.

The processor has two modes of operation: CDF mode and regular mode. On a hit in the Critical Uop Cache, the processor enters CDF mode. In CDF mode all fetch address are computed and branches predicted when fetching critical uops from the Critical Uop Cache. This forms the critical instruction stream. Uops in the critical instruction stream are renamed in the Critical Rename stage which uses a separate critical Register Alias Table (RAT) and are subsequently Issued to the processor backend.

In parallel, instructions are fetched from the I-cache using the branch predictions and targets generated while fetching critical uops (to ensure that they follow the same control flow path). This forms the non-critical instruction stream which contains both critical and non-critical uops. These uops are Decoded and sent to the Regular Rename stage. All uops in the non-critical instruction stream update the RAT. Non-critical uops are renamed normally. For critical uops, the corresponding renaming operations that were performed in the Critical Rename stage are replayed. This ensures that the state of the RAT is updated in-order and non-critical uops

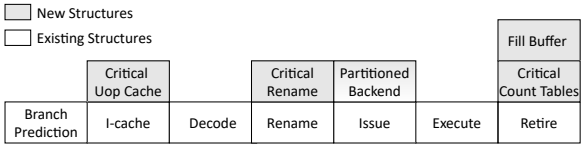


**Figure 4: Overview of changes to the OoO pipeline**

that depend on critical uops read the correct physical registers. The critical uops are then discarded and the non-critical uops are Issued to the processor backend.

Structures in the backend are partitioned into two sections: one for critical instructions and the other for non-critical instructions. The partitioning is dynamic and changes over the course of execution but is generally skewed towards a larger critical section. Dynamic partitioning allows CDF to adjust the throughput of the critical and non-critical instruction streams. Instructions in each section of the ROB are present in program order, and the oldest instructions in each section are looked up to ensure retirement occurs in-order. We discuss all these changes with examples and go over their implementation details in the next few subsections.

## 3.2 Identifying and Storing Critical Instructions

**Identifying Critical Loads** To track critical loads, we use a 64-entry table (Critical Count Table) that contains two saturating counters for each load in the table. The counters are updated at retire time and incremented or decremented based on whether the loads miss in the last level cache.

A load is considered critical if it has an entry in the Critical Count Table and its counter value exceeds a set threshold. For most benchmarks, having a stricter threshold is better as it reduces the density of critical instructions and allows CDF to expand the effective instruction window further. However, some benchmarks benefit from greater coverage. To account for these two sets of behaviors, we have two counters per load in the Critical Count Table: one with a stricter threshold and the other with a more permissive threshold (the counters have different lengths). At runtime, we measure the percentage of instructions marked critical by CDF and dynamically pick the more permissive counters for prediction if too few loads are marked critical. Hard to predict branches are tracked similarly in a separate table and have different thresholds.

**Adding Instructions to the Fill Buffer** As instructions retire, they update the Critical Count Tables and are marked critical based on the counters. The retired instructions are then added to a FIFO called the Fill Buffer. Each Fill Buffer entry contains a decoded uop, a bit vector for the registers written to and read by the uop, a tag for memory locations written to and read by the uop, and a bit to indicate whether the instruction is critical. This is shown in Fig. 6.

When the Fill Buffer is full, we walk the Fill Buffer from the youngest to the oldest instruction, constructing the dependence chains for critical loads similar to the scheme used in Filtered Runahead [9]. In our case however, there can be multiple critical loads in the Fill Buffer and the uops marked critical are thus a combination of the uops in the dependence chains of multiple critical loads. This process is shows for an example in Fig. 5.

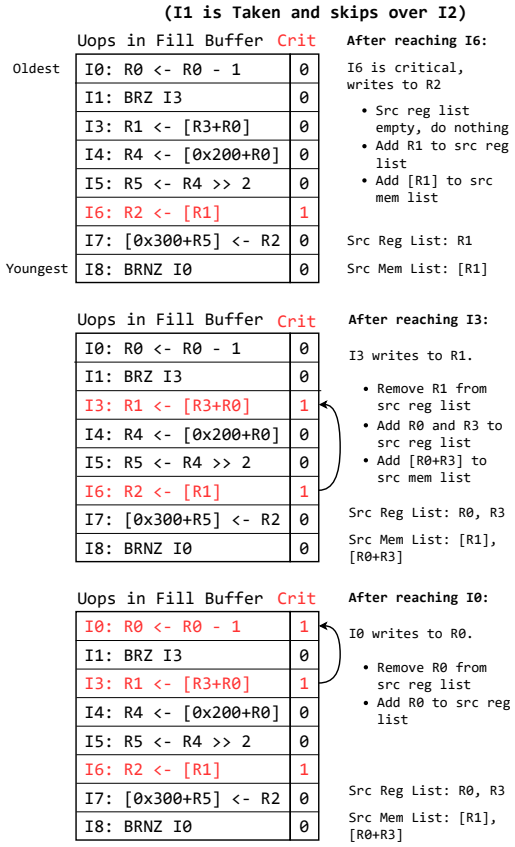


**Figure 5: Backwards dataflow walk in the Fill Buffer**

**Adding instructions to the Critical Uop Cache** When performing the backwards dataflow walk, we keep track of basic block boundaries by marking branches. When a complete basic block has been marked in the fill buffer, the critical uops corresponding to the basic block are collected into a trace. These uops are added to the Critical Uop Cache and are tagged with the first instruction in the basic block. When filling the next basic block, the tag from the prior basic block is saved and added to the critical uop trace. This allows the frontend to compute the fetch address of the next critical trace by either predicting the branch at the end of the basic block (a bit is added to the trace if it ends in a branch), or by using the address saved at the end of the trace. This is shown in Fig. 7. Note that the backwards dataflow walk does not end at basic block boundaries. We use these boundaries to break the critical uops in the Fill Buffer into basic block sized traces for ease of storage. If a basic block exceeds the line limit (8 uops), it is broken into multiple Critical Uop Cache entries. This process is repeated and uops are added to the Fill Buffer every 10k instructions. The latency for marking the dependent uops critical and filling them into the Critical Uop Cache is around 1200 cycles, as iterating over individual instructions and inserting a trace into the Critical Uop Cache each take a single cycle. These are accounted for in our simulations.

**Mask Cache** The uops in the dependence chain of a critical load may be different on different control flow paths. Thus different sets of uops may be marked critical for the same basic block. To ensure

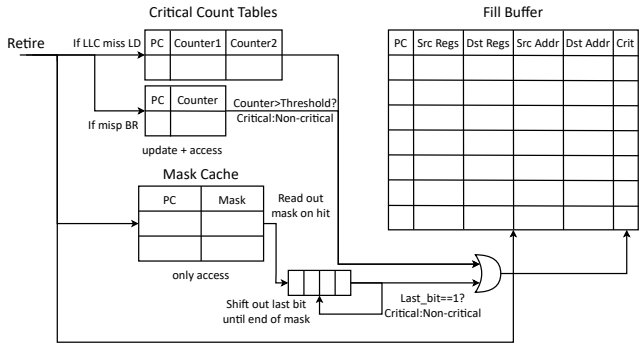


**Figure 6: Adding Uops to the Fill Buffer**

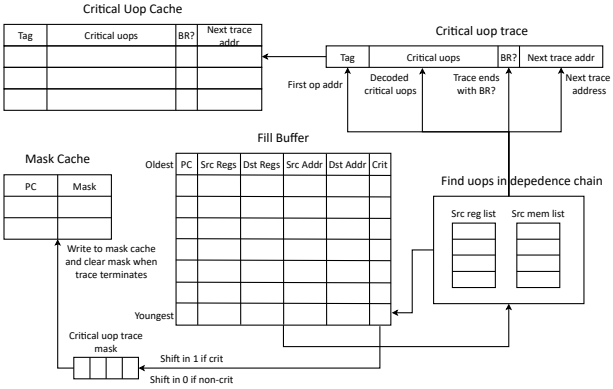


**Figure 7: Marking dependent instructions critical and adding critical uop traces to the Critical Uop Cache**

that the same set of critical uops can be used on multiple control flow paths we combine the set of uops marked critical across all previously encountered instances of that basic block.

To do this, we record a bit-mask for each basic block in the Fill Buffer. The masks are created when performing the backwards dataflow walk and contain a 1 for every critical uop in the basic block. The masks for all basic blocks encountered when walking the Fill Buffer are stored in the Mask Cache. If a basic block whose mask is present in the Mask Cache is seen the next time instructions are added to the Fill Buffer, its mask is read out and placed in a shift register. As uops are added, they can be either marked critical by the Critical Count Table or if the basic block mask being currently read contains a 1 for that uop. Thus, the mask accumulates critical uops for the same basic block seen on different control flow paths.

Storing the contents of the Fill Buffer as a single long trace for every encountered control flow path also prevents this problem. However, this results in a large number of duplicate instructions in the Critical Uop Cache. Storing basic blocks of critical uops reduces duplicates and allows the CDF frontend to fetch control flow paths that include basic blocks sequences that have not been seen before. Each entry in the mask cache is a 64-bit mask that is tagged with the address of the first instruction in the basic block. It is periodically reset (every 200k instructions) to clear uops accumulated in the masks from control flow paths that are no longer active.

When marking instructions in the Fill Buffer, if too few (<2%) or too many (>50%) instructions are marked critical, they are not

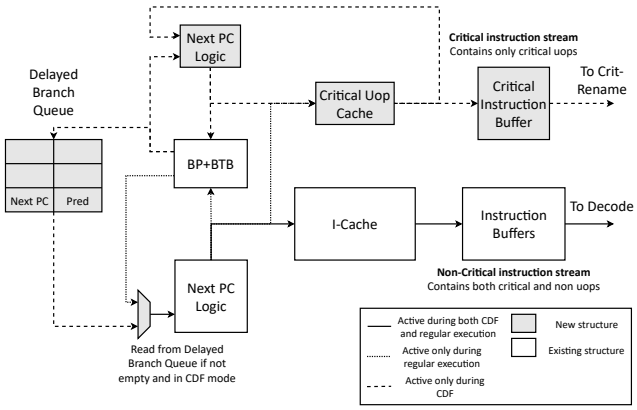


**Figure 8: CDF Fetch Stage**

added to the Critical Uop Cache or the Mask Cache as CDF does not improve performance in these situations. Instead, the corresponding basic blocks are removed from these structures to prevent the processor from entering CDF mode.

## 3.3 Fetching Critical Instructions OoO

To fetch critical instructions preferentially, we augment the frontend of the pipeline with extra fetch logic. We add a new PC register, replicate the next-PC logic, and add the Delayed Branch Queue: a 256-entry FIFO that stores the directions and targets of all branches encountered in CDF mode. These changes are shown in Fig. 8.

**Entering CDF mode** On a hit in the Critical Uop Cache, the processor copies the contents of the PC to the critical next-PC logic and begins CDF mode. During CDF mode, all fetch addresses are computed and branches predicted by the critical fetch logic. These addresses are sent to the Critical Uop Cache and the uops read out are added to the Critical Instruction Buffers. Since we store decoded uops, they are directly sent to the Critical Rename stage. The predictions and targets of all the branches are added to the Delayed Branch Queue. If no branch is encountered (or the branch is predicted not-taken), the next fetch address is obtained from the critical uop trace that was previously read out.

**Fetching Non-Critical Instructions** In CDF mode, the regular fetch logic continues to function as usual, except the predictions for all branches and their targets are read out from the Delayed Branch Queue. Thus, branch prediction is performed in order and the predictors are only accessed once when fetching critical uops. Reading from the Delayed Branch Queue ensures that the non-critical instruction stream follows the same control flow path that the critical fetch logic took. We fetch all instructions (critical and non-critical) from the I-cache and add them to the Instruction Buffers. The critical uops are discarded at the Rename stage.

It is possible to avoid the overhead of re-fetching and decoding critical uops from the I-cache by having a separate Non-Critical Uop Cache. This also improves the fetch bandwidth for non-critical instructions. However, having a Non-Critical Uop Cache adds additional area and complexity to the design of the Fetch Unit and complicates renaming. Moreover, we observed that non-critical instructions are generally less sensitive to fetch bandwidth and decided not to implement a separate Non-Critical Uop Cache.

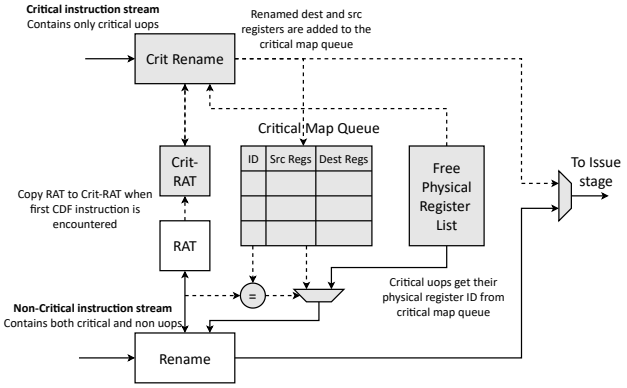


**Figure 9: CDF Rename Stage**

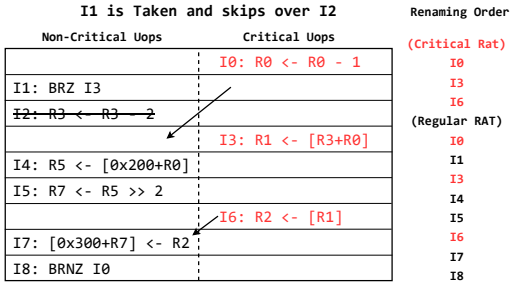


**Figure 10: Simple example to demonstrate renaming. There are no dependencies from non-critical to critical instructions. However non-critical instructions may use data written by critical instructions.**

**Assigning Timestamps** OoO cores assign timestamps to instructions to specify their relative ordering. Since we know exactly how many intermediate non-critical uops are present in a basic block while constructing critical uop traces, we can assign timestamps to critical uops appropriately when they are fetched by skipping over the timestamp values that would have been assigned to non-critical uops. The critical uop traces contains an additional field per uop that enables this. When instructions are fetched as part of the non-critical instruction stream, timestamps are assigned as usual starting from the same value as when CDF began. This ensures that the timestamps correctly identify the program order of critical and non-critical uops.

## 3.4 Renaming Instructions OoO

**Renaming the Critical Instruction Stream** When critical uops reach the rename stage, they create a copy of the Register Alias Table (RAT) after the last regular mode instruction has been renamed. The critical uops then proceed to get renamed as they would in a normal OoO core using the critical RAT. This is shown in Fig. 9

In the example in Fig. 10, there are no critical uops that depend on a non-critical uop. This is because any non-critical uop that writes to a source register read by a critical uop is also marked critical during Trace Construction. As a result, renaming critical uops first correctly outlines the dependencies between them. The destination physical registers written to by critical uops are recorded in the Critical Map Queue, which is a 256-entry FIFO.

**Renaming the Non-critical Instruction Stream** Non-critical uops can depend on critical uops. Looking at the example in Fig. 10, we see that I4 and I7 depend on critical uops. To correctly execute them, the destination physical registers used by critical uops (I0 and I6 in the example) need to be communicated to the subsequent non-critical uops, which is done by the Critical Map Queue. When the non-critical instruction stream (which contain all uops) is renamed, non-critical uops are renamed normally using the Free Physical Register List and update the regular RAT. Critical uops read the head of the Critical Map Queue instead to update the RAT. The entry at the head of the Critical Map Queue contains the physical register that was assigned to the next critical uop when it was renamed as part of the critical instruction stream. This allows the regular RAT to be updated in-order as we essentially replay the renaming operation of the intermediate critical uops. In the example, reading out the destination physical registers for the intermediate critical uops (I0, I6) from the Critical Map Queue ensures that I4 and I7 have source physical register IDs that correspond to the actual data produced by I0 and I6.

Thus, the state of the regular RAT is always updated in program order. The Critical Map Queue also filters out critical uops after updating the RAT as they are already issued to the backend as part of the critical instruction stream. Note that the performance of non-critical instructions is reduced here because they are not renamed at peak bandwidth due to the intermediate renaming operations that are replayed for the critical uops. However, this does not degrade the overall performance much as non-critical instructions can generally tolerate this reduced bandwidth. The non-critical instruction stream stalls if the corresponding critical uops in the critical instruction stream have not been renamed.

## 3.5 Allocating Instructions and Partitioning Window Resources

When the first critical uop reaches the ROB, a critical partition is created for critical instructions. This is done by assigning a new set of fill and retire pointers. The pointers to the start and end of the critical section are stored in separate registers. The increment logic for the critical fill and retire pointers is modified to skip over non-critical instructions in the ROB and vice-versa. The Load and Store Queues are also partitioned with similarly modified fill and retire pointer logic. The Reservation Stations and Physical Registers are partitioned by imposing a limit on the number of critical uops in both the structures.

**Issue and Dispatch** Both the critical and non-critical Rename stages contain uops that need to be issued to the ROB and the Reservation Stations. The Issue logic always picks uops from the critical Rename stage if it is not empty and the critical section of the ROB, LQ or SQ are not stalled. Otherwise, it issues non-critical uops. When dispatching instructions from the Reservation Stations to the Execution Units, we use oldest-first scheduling with preference given to critical uops. The total issue and dispatch bandwidth in CDF is the same as that of the underlying OoO core.

**Dynamically Changing the Partition Size** Reducing the partition size for non-critical instructions reduces their throughput but does not generally affect performance. However, assigning too

small a partition for non-critical instructions will eventually lead to them bottlenecking execution as they are unable to make sufficient forward progress. Moreover, not all cache misses are marked critical as the Critical Count Table can be wrong in its prediction of whether a load (or branch) is critical. Therefore, instead of keeping the partitions static, we allow the partition size to change by modifying the fill and retire pointer boundaries. When the critical partition in the ROB needs to be expanded, the first non-critical instruction slot after the end of the critical partition is marked. If it empty, the register containing the pointer to the end of the critical section is incremented to include the next slot and the size of the critical section increases by one. If it is not empty, we wait until the instruction in the non-critical ROB entry retires and then add the slot to the critical section. Reducing the size of the critical section works similarly: we mark the slot at the beginning of the critical section, and when empty, increment the pointer to the start of the critical section to exclude that slot.

We vary the partition sizes of the ROB, LQ and SQ independently (the number of critical uops in the RS and PRF change with the ROB partition size). The partitioning mechanism is controlled by counters that measure the relative number of full window stalls caused due to these structures in both the critical and non-critical sections. If the number of full window stalls in a section exceeds the other by a set threshold, it is expanded. In our simulations, we use a full window stall cycle threshold of 4 cycles. The partition sizes of the ROB/RS are incremented or decremented by 8 entries when this threshold is reached. The partition sizes of the LQ/SQ are updated by 2. In general, we observed that finer granularities for both the stall threshold and partition size changes work better.

Dynamic partitioning allows CDF to adjust the throughput of the critical and non-critical instruction streams. This prevents non-critical instructions from bottlenecking performance while maximizing the amount of parallelism that can be extracted from critical instructions. Since the optimal partition sizes are program/execution phase dependent, **the ability to dynamically pick a partition size significantly improves the performance of CDF.**

**Memory Disambiguation** In OoO cores with Total Store Ordering (TSO) or more relaxed forms of memory ordering, associative lookups are performed on instruction timestamps in the Load and Store Queues to make sure memory ordering is not violated. Since CDF assigns timestamps for both critical and non-critical uops in program order, the memory disambiguation logic does not change significantly. While CDF partitions the Load and Store Queues, the memory operations within the two sections of the queues are always present in program order. The memory disambiguation logic thus needs look up two sets of (smaller) ordered queues to ensure correctness.

During CDF, there may be non-critical loads and stores (which are older than critical memory operations already in the queues) that have not been issued to the processor backend. However, critical memory operations are not committed until retire time. Since retirement happens in order, the missing non-critical loads and stores (that are in program order before the critical ones) will be issued to the LQ and SQ before the critical loads and stores are committed. The memory operations can then be checked for dependence violations and do not require any additional logic beyond what is already present in a regular OoO core. If a memory dependence violation is detected, the pipeline is flushed and the processor restarts fetching instructions in regular mode from the instruction at which the violation was detected.

**In-Order Retirement** In both the critical and non-critical sections of the ROB, instructions are present in program order. Only the oldest instructions in each section (as indicated by the two retire pointers) need to be checked for retirement and a simple comparison of their timestamps gives us the next instruction to be retired. While this increases the complexity of the retirement logic, it does not affect performance much since very few programs are limited by the Retire stage latency.

## 3.6 Overall Pipeline Changes

**Branch Mispredictions** CDF deals with branch mispredictions in the same way as a regular OoO core: on a detected misprediction, all instructions in the processor younger than the mispredicted branch in program order are flushed. This includes critical and non-critical instructions. Both the Critical Map Queue and Delayed Branch Queue are partially flushed. This does not add any additional overhead to the misprediction latency as entries in these structures are arranged in program order and flushing them is thus straightforward.

If a mispredicted branch is a marked critical, recovering to the mispredicted branch does not end CDF mode. This allows the CDF frontend to continue fetching critical instructions after the branch is resolved. The state of the critical RAT is checkpointed and recovered using the same mechanism as the regular RAT. A mispredicted branch in the non-critical instruction stream does not end CDF mode either if it was fetched when CDF mode was active. It only sees a higher misprediction latency since all branches are predicted when fetching critical instructions. Recovering to a mispredicted branch that was fetched as part of regular execution does end CDF mode. Subsequent instructions can re-start CDF mode again on a hit in the Critical Uop Cache.

**Dependence Violations in the Critical Instruction Stream** The dependence chains constructed as part of the backward dataflow walk can be incorrect in rare cases. This primarily happens when critical uops are fetched on a control flow path that has not been seen before, or the critical load dependence chain goes beyond 1024 uops (the capacity of the Fill Buffer). Memory dependence violations are detected and resolved by the memory disambiguation logic. When a register dependence violation occurs, a critical uop that depends on an incorrectly marked non-critical uop can be fetched and executed before the non-critical uop leading to incorrect execution. An example of this is shown in Fig. 12 where only the taken path has been recorded in the Fill Buffer. When the not-taken path is seen, I2 is not fetched as part of the critical instruction stream even though it writes to a critical instruction (I3). I0, I3 and I6 are fetched and executed first, which leads to I3 and subsequently I6 being incorrectly executed.

The mask cache helps reduce the number of dependence violations significantly since it combines instructions marked critical across multiple control flow paths for a basic block. However, the rare cases in which instructions are incorrectly executed need to be detected and resolved to ensure correctness.

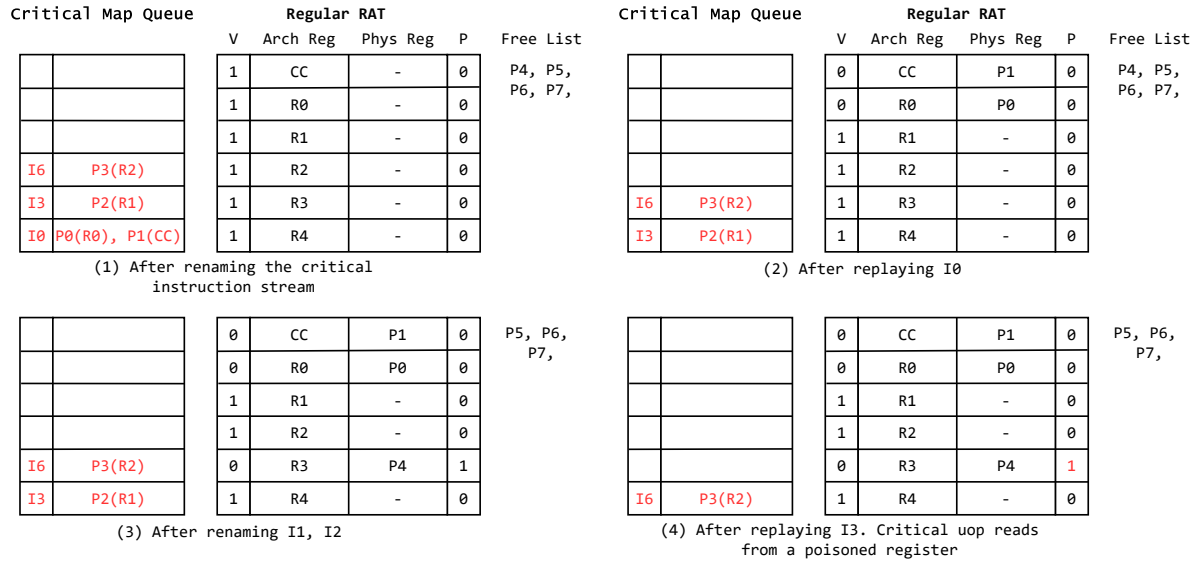


**Figure 11: I2 poisons register R3. I3 is a critical uop that reads from R3. When the renaming operation of I3 is replayed, a dependence violation is detected.**

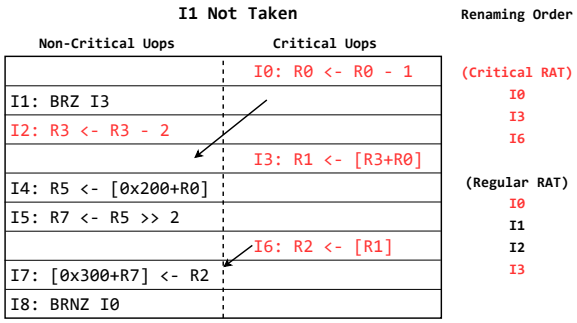


**Figure 12: If I1 is not-taken, there is a non-critical instruction that writes to a critical instruction as this control flow behavior has not been captured in the Fill Buffer**

To detect whether any non-critical uop writes to a critical uop, we use a poison bit in the regular RAT for each architectural register. When the non-critical uops in the non-critical instruction stream are renamed, the poison bit in the regular RAT corresponding to its destination register is set. When performed for all the non-critical uops, this poisons the destination register set for all non-critical uops. If a subsequent critical uop (whose renaming operation is replayed) reads a poisoned register, it has executed incorrectly since it should have used the value computed by the non-critical uop which poisoned that register. This process is shown for the previous example in Fig. 11. This constitutes a dependence violation. Subsequent critical uops may have executed incorrectly; therefore, all critical instructions in program order after the instruction at which the violation was detected need to be flushed from the pipeline. The frontend then restarts regular execution at the instruction at which the violation was detected. When replaying the renaming operations of critical uops in the non-critical instruction stream, the poison bit for their destination registers is cleared since it is not in the set of registers written to by non-critical instructions.

A dependence violation can potentially flush a large number of critical instructions from the pipeline. However, since they are rare, we found that this overhead was under 2% (in terms of execution cycles) for most benchmarks. The branch flush logic can be reused for flushing the pipeline in the event of a dependence violation.

**Exiting CDF mode** The fetch unit stops fetching critical uops on any of the following events: (a) the fetch unit encounters a miss in the Critical Uop Cache, (b) a dependence violation is detected, (c) the branch being recovered to was not fetched in CDF mode.

Any remaining non-critical instructions (as indicated by the Delayed Branch Queue being not empty) are then fetched. Once the last non-critical instruction is fetched, the processor exits CDF mode and resumes regular execution. Since the non-critical instruction stream updates the state of the regular RAT in program order, no additional work needs to be done to transition back to regular execution. Note that it is possible to have unretired critical instructions in the pipeline when the frontend exits CDF mode and starts fetching instructions normally. To deal with this, all instructions that are fetched regularly are treated as non-critical and the size of the critical section of all the backend structures is gradually decreased till the pending critical instructions retire.

# 4 EVALUATION

## 4.1 Methodology

To evaluate how CDF affects processor performance, we simulate the micro-architecture of an aggressive OoO core augmented with the structures and logic needed to support CDF. We use Scarab [1], an execution-driven cycle-accurate x86-64 simulator, to implement CDF, and Ramulator [11] to model main memory. CACTI [18] and McPAT [15] are used to provide energy and area measurements. The system details for the baseline OoO core and additional CDF structures are listed in Table 1. The baseline core parameters are modelled after the Intel Sunny Cove microarchitecture.

| | |
|---|---|
| Core | 3.2 GHz, 6-wide issue, TAGE-SC-L Predictor [25]<br>352 Entry ROB, 160 Entry Reservation Station<br>128 Entry Load & 72 Entry Store Queues |
| Caches | 32KB 8-way L1 I-cache & D-cache, 2-cycle access<br>1MB 16-way LLC cache, 18-cycle access, 64B lines |
| Prefetcher | Stream Prefetcher, 64 Streams (always on),<br>Feedback Directed Prefetching to throttle prefetcher |
| Memory | DDR4_2400R: 1 rank, 2 channels<br>4 bank groups and 4 banks per channel<br>tRP-tCL-tRCD: 16-16-16 |
| CDF Caches | 64B 2-way Critical Count Tables, 1-cycle access<br>4KB 4-way Mask Cache, 1-cycle access<br>18KB 4-way Critical Uop Cache, 1-cycle access<br>8 uops (8B each) per entry |
| CDF FIFOs | 1024-entry, 16KB Fill Buffer<br>256-entry, 1KB Delayed Branch Queue<br>256-entry, 512B Critical Map Queue |

**Table 1: Simulation Parameters**

**Benchmarks** We use the set of memory intensive applications from SPEC CPU2006 and SPEC CPU2017 (speed). We use SimPoints [26] to get representative regions for these benchmarks and generate up to 5 SimPoints per benchmark, with 200 million instructions per SimPoint. In all our simulations we warm up the caches and branch predictor for 200 million instructions before the SimPoint.

**Precise Runahead** For a fair comparison, we use the same mechanism as CDF for marking and fetching critical instructions in Precise Runahead (PRE), except we only mark loads that cause full window stalls as critical. The Critical Uop Cache can hold more critical instructions compared to PRE's Stalling Slice Table (SST) and hence provides better performance. Our implementation of PRE also does not include the Extended uop Queue (EMQ) since fetching from the Critical Uop Cache means non-critical instructions need not be buffered.

## 4.2 Performance

Fig. 13 shows the speedup of CDF and PRE over the baseline OoO core (with prefetching). CDF provides a geomean speedup of 6.1% over the baseline, whereas PRE provides a speedup of 2.6%.

**Application Level Analysis** CDF performs better than PRE on most benchmarks. This can be attributed to the three reasons we discussed in Section 2. On benchmarks such as *lbm*, the full window stall duration is too short to enable any useful Runahead prefetches. CDF does not have this limitation. CDF does well on *bzip*, *astar*, *mcf* and *soplex* as we mark hard-to-predict branches critical. As a result, mispredicted branches are resolved earlier which allows CDF to continue fetching critical instructions. Not marking these branches critical eliminates the benefits of CDF in these applications and reduces the geomean speedup to 3.8%. All benchmarks benefit from CDF's improved critical instruction fetch bandwidth. *bzip* and *nab* in particular perform better than the baseline due to faster initiation of critical loads and do not benefit much from improved parallelism.

CDF provides a large performance improvement when the instructions marked critical are sparse. This allows CDF to skip more non-critical instructions and fill up the processor instruction window with critical loads, which is the case in our best performing

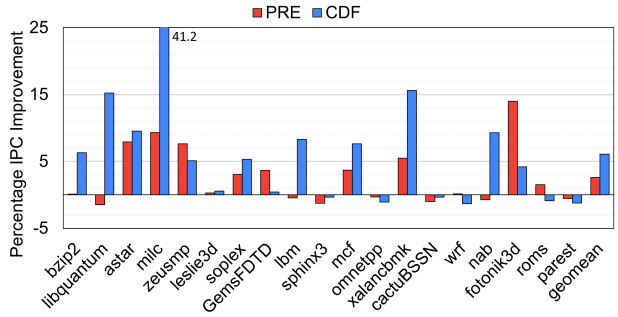


**Figure 13: Percentage IPC improvement for CDF and PRE over the baseline**

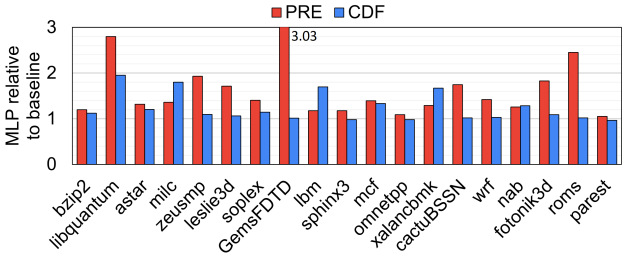


**Figure 14: MLP for CDF and PRE relative to baseline**

benchmarks (*milc, xalanbmk, libquantum*). PRE performs well on some benchmarks as its prefetch distance is not limited by the ROB or LQ. CDF cannot fetch as far ahead as PRE in these benchmarks as the critical instructions are not sparse enough. These include *zeusmp*, *GemsFDTD*, *fotonik3d* and *roms*.

*leslie3d*, *sphinx*, *wrf*, *parest* and *omnetpp* do not do well with either CDF or PRE. The critical instruction densities in these benchmarks does not fit into the two broad categories that we use in Section 3.2. **Accelerating cache misses in benchmarks that CDF does poorly on is possible but requires a more complex mechanism for manipulating critical instruction and load densities in a fine-grained manner.** We leave this optimization to future work.

**Note on PRE Results** The speedup for PRE in our implementation is much lower than shown in prior work. In our evaluation, we used up to five SimPoints per benchmark, whereas all prior work on Runahead (including PRE) uses only a single SimPoint. Some SimPoints are not memory intensive and can provide neutral or even negative benefits: for example some SimPoints in *libquantum* and *CactuBSSN* show poor performance due to corruption of the cache state and excess memory traffic respectively. Also, we use a more aggressive memory configuration and baseline OoO core in our evaluation which reduces the duration of full window stalls in many benchmarks and limits the benefits of PRE.

**MLP** Fig. 14 shows the MLP for CDF and PRE relative to the baseline OoO core. A large percentage of the increased MLP for PRE is due wrongpath loads or loads with incorrect dependence chains which do not contribute to improved performance. In contrast, almost all the extra parallelism exposed by CDF contains critical loads with correct addresses that lead to performance improvements which is reflected in the low overhead of CDF.

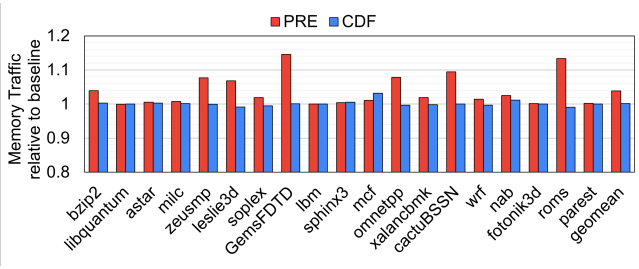


**Figure 15: Memory traffic relative to baseline**

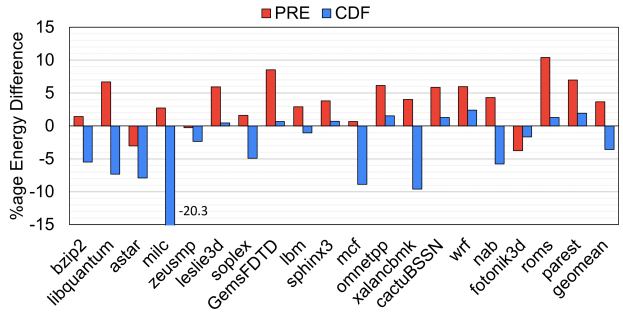


**Figure 16: Percentage change in energy consumption relative to baseline**

## 4.3 Overheads

**Memory Traffic** Fig. 15 shows the memory traffic for CDF and PRE relative to the baseline. In addition to the extra incorrect memory accesses, PRE also triggers additional Stream Prefetcher accesses which further adds to the memory traffic. CDF issues useful loads that are part of the main instruction stream and thus increases memory traffic only by 0.1%, whereas PRE has a 4.1% overhead.

**Energy** CDF reduces the energy consumed for most applications as the overall runtime decreases as seen in Fig. 16. The energy overhead of all the additional structures adds up to 2% of the baseline. The Critical Uop Cache, Mask Cache and critical RAT contribute to most of this. The added FIFOs and pipeline logic do not have significant overhead since read and write operation energy and static energy for these structures are much lower due to their lower complexity. Benchmarks that do not do well in CDF mode default to regular execution which reduces the dynamic energy due to the CDF structures in these benchmarks.

PRE performs worse than the baseline in terms of energy due to the large increase in memory traffic and the large number of duplicate instructions fetched and executed in the pipeline. Even though CDF fetches and renames critical uops twice (which we model), this overhead is much lower since it does not lead to additional Branch predictor or backend activity (PRF, LQ, RS) in the pipeline. Overall, CDF reduce the energy consumption by 3.5%, while Precise Runahead increases energy consumption by 3.7%.

**Area and Cycle Time** CDF has a total area overhead of 3.2%. Majority of this overhead is due to the Critical Uop Cache, Mask Cache and critical RAT. The FIFOs and additional pipeline logic (Fetch logic, Instruction Buffers, Critical Rename logic) have fewer access ports and add much lower overhead.

The operations involving trace construction are not on the critical path of the processor pipeline and do not affect the cycle time. The Critical Uop Cache can be accessed in a single cycle. Accesses to the Delayed Branch Queue and Critical Map Queue take very little time since they are FIFOs. The only significant logic added directly on the critical path of the pipeline is the logic in the Allocation Stage which can pick between both critical and non-critical uops and prioritizes critical uops. To model the worst-case scenario, we added an addition pipeline stage at the end of Rename during CDF.

## 4.4 Scaling Studies

With CDF, a larger OoO core provides increased parallelism since more critical loads can be packed together in the ROB and LQ. If the application benefits from increased MLP, CDF provides significant performance benefits with even larger OoO cores. Fig. 17 shows how the IPC and energy consumption of a CDF OoO core scale in comparison to a regular OoO core. Applications like *roms* and *fotonik* perform better with a larger baseline since the larger windows allow CDF to fetch further ahead. A scaled OoO core with area comparable to our CDF implementation provides only 3.7% IPC improvement (a major part of this comes from improved ILP which CDF cannot currently extract) and consumes 2.5% more energy.

# 5 RELATED WORK

There has been a lot of prior work on improving single-threaded parallelism using separate threads or through pre-computation. Speculative multi-threading [17, 22, 34] splits the program into speculative threads at compile time and executes these threads on a different core to improve parallelism. In Lookahead execution [7, 13], a 'skeleton' or reduced version of the main program precomputes branch directions and load addresses before the program reaches the corresponding instructions. Helper threads [6, 10] are similar and are run as a separate context on the same core. [4] uses a similar "future thread" that executes (but does not commit) on a partitioned section of the core and forwards registers values to the main program.

Speculative multi-threading leverages the compiler to find good points to parallelize code. These threads are not optimal since instruction criticality cannot be accurately computed at compile time. Also, speculative multi-threading is expensive as it requires an additional core. Lookahead execution and helper threads also face similar problems as the skeleton code or the helper threads cannot determine which instructions will be critical at runtime. As a result, they contain a lot of instructions and require a significant amount of resources to run ahead of the main program. Moreover, these instructions are duplicates that need to be executed twice, leading to additional overhead. This code can also be inaccurate, causing extra memory traffic. While the most recent work on Slipstream [29] computes instruction criticality at runtime, it still requires a separate core and re-executes a large number of instructions. CDF takes the more direct approach of changing the fetch and schedule order of the instruction stream to trigger loads in a more timely manner and focuses on extracting more parallelism from the available window resources.

Continuous Runahead [8] runs simple dependence chains on a separate Runahead engine to prefetch load misses. Since the Runahead engine is lightweight, it does not have a complex branch

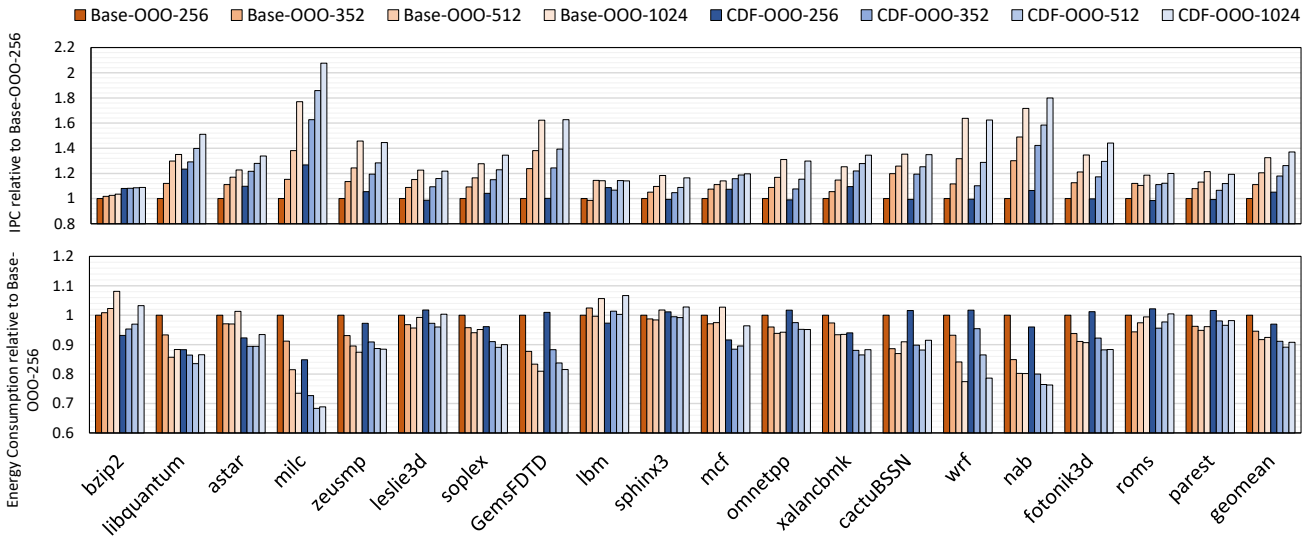


**Figure 17: CDF and Baseline OoO cores with different ROB configurations (other core structures are scaled proportionately)**

predictor or support execution of floating point operations. This leads to many memory accesses at incorrect addresses which results in additional memory traffic. This overhead is shared by most aggressive prefetching algorithms as well.

LTP [24] and Shelf OoO Execution [27] leverage instruction criticality to improve the efficiency of the Reservation Stations but cannot extract MLP beyond the capacity of the ROB since it allocates instructions to the ROB in-order. CDF could be implemented such that it performs in-order fetch, buffers instructions after fetch, and allocates critical instructions preferentially (similar to LTP, but with a partitioned backend). However, the parallelism that can be extracted from such a design would be limited by the size of the intermediate buffer. Moreover, we would lose the benefit of improved critical instruction fetch bandwidth by fetching in-order, which reduces the performance of this design significantly.

Slice OoO execution [5, 14] tries to extract MLP from InO cores by executing load slices out-of-order. CDF instead maximizes the parallelism that can be extracted from an OoO core. [2] uses instruction criticality and control independence to reduce in-order fetch bottlenecks. Continual Flow Pipelines (CFP) [28] finds independent critical load chains in a tight loop and creates a self-contained execution loop in hardware by releasing resources early and allowing subsequent iterations of the chains to acquire them. The CFP chains however have significantly lower coverage compared to CDF which can have multiple critical loads and diverse control flow paths. Unlike CDF, CFP uses a ROB-less architecture [3] to enable a larger instruction window for critical loads.

Since CDF expands the speculative window of the OoO core, it affects side-channel attacks that exploit speculative execution [12, 16]. For instance, an attacker can artificially trigger cache misses to have longer speculative windows which may make some attacks easier to execute. This however is the same problem that a processor with an actual larger OoO window would face, and existing defenses against speculative side channels [23, 32, 33] can be made to work on a CDF OoO core.

## 6 CONCLUSION AND FUTURE WORK

In this paper, we present CDF - a new execution paradigm for accelerating critical instructions on OoO cores. CDF expands the effective size of the instruction window seen by critical instructions by preferentially fetching, allocating and executing them at the expense of reducing the throughput of non-critical instructions. We implement CDF on an OoO core and show that it improves the performance of loads that miss in the LLC by marking all such loads and the instructions in their dependence chains critical. CDF does this by exposing more MLP for independent loads and by fetching critical loads earlier. It can also fetch critical loads past hard-to-predict branches by resolving the branches early. CDF improves performance by 6.1% over a baseline OoO core.

CDF and techniques such as Runahead provide different benefits and can potentially be combined. CDF is better suited to the task of extracting parallelism from loads that lie a few 1000 instructions away, whereas memory prefetchers and variants of Runahead are better suited to prefetching loads far into the future. CDF is not limited to loads and can improve the performance of most programs that show better performance with a larger OoO window. To this end, accurately predicting instruction criticality and finding the optimal critical instruction density for an application's execution phases are essential. While compilers cannot identify critical instructions and find the optimal level of loop unrolling statically, they can be used to augment CDF by statically generating a set of possible chains that CDF can then choose to fetch and execute at runtime. This can help reduce the hardware overhead and complexity of CDF significantly.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers and members of HPS for their feedback and help with improving this paper. We thank Intel, Cockrell Foundation, Arm and NSF (Award 2011145) for their financial support. We acknowledge the Intel Labs Academic Computing Environment (ACE) for providing compute resources.

# REFERENCES


[1] 2021. Scarab. https://github.com/hpsresearchgroup/scarab.

[2] M. Agarwal, N. Navale, K. Malik, and M. I. Frank. 2008. Fetch-Criticality Reduction through Control Independence. In *2008 International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA.2008.39

[3] H. Akkary, R. Rajwar, and S.T. Srinivasan. 2003. Checkpoint processing and recovery: towards scalable large instruction window processors. In *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2003.1253246

[4] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. 2001. Dynamically allocating processor resources between nearby and distant ILP. In *28th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2001.937428

[5] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. 2015. The Load Slice Core microarchitecture. In *42nd Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2749469.2750407

[6] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.1999.765950

[7] A. Garg and M. C. Huang. 2008. A performance-correctness explicitly-decoupled architecture. In *41st International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2008.4771800

[8] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *49th International Symposium on Microarchitecture (MICRO)*.

[9] M. Hashemi and Y. N. Patt. 2015. Filtered runahead execution with a runahead buffer. In *48th International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830812

[10] Jiwei Lu, A. Das, Wei-Chung Hsu, Khoa Nguyen, and S. G. Abraham. 2005. Dynamic helper threaded prefetching on the Sun UltraSPARC/spl reg/ CMP processor. In *38th International Symposium on Microarchitecture (MICRO'05)*. https://doi.org/10.1109/MICRO.2005.18

[11] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016). https://doi.org/10.1109/LCA.2015.2414456

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2019.00002

[13] Sushant Kondguli and Michael Huang. 2019. Bootstrapping: Using SMT Hardware to Improve Single-Thread Performance. In *ASPLOS '19*. 14. https://doi.org/10.1145/3297858.3304052

[14] R. Kumar, M. Alipour, and D. Black-Schaffer. 2019. Freeway: Maximizing MLP for Slice-Out-of-Order Execution. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2019.00009

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

[16] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[17] Carlos Madriles, Pedro López, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raúl Martinez, and Antonio Gonzalez. 2009. Boosting Single-Thread Performance in Multi-Core Systems through Fine-Grain Multi-Threading. In *36th Annual International Symposium on Computer Architecture (ISCA)*. 10. https://doi.org/10.1145/1555754.1555813

[18] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *40th International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2007.30

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *9th International Symposium on High-Performance Computer Architecture, (HPCA)*. https://doi.org/10.1109/HPCA.2003.1183532

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

[21] Vlad-Mihai Panait, Amit Sasturkar, and Weng-Fai Wong. 2004. Static Identification of Delinquent Loads. In *International Symposium on Code Generation and Optimization (CGO) (CGO '04)*. USA.

[22] A. Roth and G. S. Sohi. 2001. Speculative data-driven multithreading. In *7th International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2001.903250

[23] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO '52)*. New York, NY, USA. https://doi.org/10.1145/3352460.3358314

[24] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. 2015. Long term parking (LTP): Criticality-aware resource allocation in OOO processors. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830815

[25] A. Seznec. 2014. TAGE-SC-L Branch Predictors. In *JILP - Championship Branch Prediction*. https://doi.org/10.1145/3192366.3192393

[26] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/605397.605403

[27] F. M. Sleiman and T. F. Wenisch. 2016. Efficiently Scaling Out-of-Order Cores for Simultaneous Multithreading. In *43rd Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2016.45

[28] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. 2004. Continual Flow Pipelines: Achieving Resource-Efficient Latency Tolerance. *IEEE Micro* (2004). https://doi.org/10.1109/MM.2004.71

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

[30] K. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. 2017. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2017.7863738

[31] Kim-Anh Tran, Alexandra Jimborean, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras. 2018. SWOOP: Software-Hardware Co-Design for Non-Speculative, Execute-Ahead, in-Order Cores. In *PLDI*. https://doi.org/10.1145/3192366.3192393

[32] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2018.00042

[33] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO '52)*. https://doi.org/10.1145/3352460.3358274

[34] C. Zilles and G. Sohi. 2001. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2001.937426