Catalyst-Length Instruction Fusion Prediction

Sebastian S. Kim†
Computer Engineering Department
University of Murcia
Murcia, Spain
sebastiansumin.kim@um.es

Sawan Singh*
CSSE Cores
Advanced Micro Devices (AMD)
Cambridge, UK
sawan.singh@amd.com

Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
aros@ditec.um.es

***Abstract*—Instruction fusion is a microarchitectural technique that increases processor efficiency by treating multiple dependent instructions as a single internal operation. In commercial processors, existing fusion support is limited to short, consecutive memory instruction sequences, leaving longer-range fusion opportunities unexploited. In academia, Helios has overcome this limitation by enabling fusion of non-consecutive instructions.**

This work reveals that Helios relies on complex microarchitectural decisions, with its large tournament predictor falling far short of optimality, and models a more practical fusion architecture. Then, it proposes Catalyst-Length Instruction Fusion prediction (CLIF) for non-consecutive fusion. CLIF implements a lightweight yet accurate predictor that identifies non-consecutive memory instructions likely to benefit from a fused execution. The predictor uses multiple history lengths to discover fusible pairs of instructions but, different from branch predictors, it always trains a given fusible pair using the same context: the branch history comprehended between the two fused instructions. Our simulation results show that, compared to Helios, CLIF obtains average speedups of 2.05% and 3.16% on the SPEC CPU 2017 and MiBench benchmark suites, respectively, while reducing MPKI by 95.31% with less than half of Helios' budget.

***Index Terms*—Instruction fusion, hardware predictors, microarchitecture, instruction-level parallelism.**

I. Introduction

Computer architects continually seek to improve processor performance and energy efficiency by exploiting instruction- and memory-level parallelism (ILP/MLP) and optimizing the microarchitectural execution pipeline. Instruction fusion is a technique that contributes to these goals by combining certain sequences of instructions into a single internal operation [1]–[4]. Instruction fusion can increase the utilization of back-end structures, such as reorder buffer (ROB), load/store queue (LSQ), and register file, when a fused operation occupies a single entry. In addition, it can improve ILP by increasing the operation bandwidth in the pipeline.

Commercial processors implement instruction fusion for a variety of frequently occurring idioms [5], [6]. Intel has supported fusion since the Pentium M [6] and implements *micro-op fusion*, which combines multiple operations from the same x86 instruction into a single micro-op (μ-op), and *macro-op fusion*, which fuses micro-ops from consecutive x86 instructions into one [7], [8]. Specifically, μ-op fusion targets load-operate idioms, which consist of a load from memory that is immediately consumed by, for example, an arithmetic instruction. Address generation and data transfer (originally two separate μ-ops for memory writes) are similarly fused. Macro-op fusion can combine a compare instruction with the following conditional branch.

AMD introduced macro-op fusion for compare-and-branch idioms starting with its 15th family of processors [5]. Since the Zen 3 architecture [7] support has been extended to almost any arithmetic or logic instruction followed by a conditional jump. Zen 4 further enhanced this mechanism by enabling the fusion of a NOP with a preceding integer fast path instruction (excluding branches) [7], [9] to reduce the overhead of loop alignment in the decoder and retire control unit [9].

In the context of RISC-V, Celio et al. [10] argued for instruction fusion as a simpler alternative to extend the instruction set with complex instructions. Among the idioms proposed, the only suggested candidates involving memory-pairs are load and store pairs, mirroring ARMv8's approach [11].

The described implementations share two key concepts: (1) they are non-speculative and (2) they are restricted to consecutive instructions or μ-ops and, for memory operations, to contiguous memory accesses. Helios [12] addresses these limitations and demonstrates the benefits of non-consecutive instruction fusion by focusing on memory operations and fusing instructions accessing non-contiguous memory. To enable this kind of fusion, Helios implements a predictor and requires non-trivial micro-architecture modifications.

The predictor used in Helios is a large tournament predictor [13] adapted for fusion. Although tournament predictors historically have improved accuracy by combining simpler predictors, they are limited in adapting to complex or context-dependent patterns. Given advances in branch prediction since the tournament prediction proposal, particularly TAGE [14]–[18] that uses multiple history lengths to capture short- and long-range correlations, adopting a TAGE-based predictor appears to be the sensible choice for a fusion architecture.

However, TAGE-style predictors may not be the most suitable option for all prediction mechanisms. Recent work

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO, grant agreement No 819134), from MICIU/AEI/10.13039/501100011033 and the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00), and from Fundación Séneca (SCALE, grant 23039/GERM/25).

† This work was done while the author was a PhD student funded by the Fundación Séneca, Region of Murcia (21456/FPI/20).

∗This work was done while the author was at the Computer Engineering Department, University of Murcia.

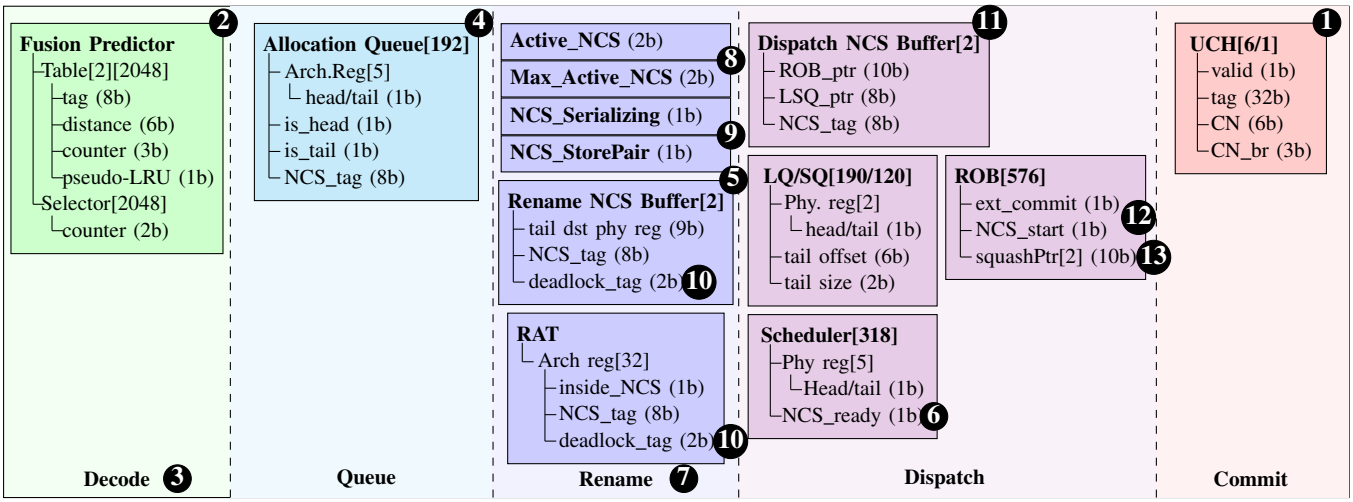


Fig. 1. Overview of Helios architecture

in memory dependence prediction has shown that identifying the appropriate history length relies on an inefficient brute-force search [19]. Instead, memory dependence prediction can benefit from a new type of context-sensitive predictor that predetermines the suitable history length for each conflicting pair. We argue in this work that such a concept is also better suited for instruction fusion prediction: if a fusible instruction pair consistently follows the same execution path, it is highly likely to form a valid fusion idiom.

Beyond the predictor design, the Helios microarchitecture makes some design choices that limit performance. Chief among these is its reliance on oversized fusion windows. This is problematic for two reasons: first, a non-consecutively fused instruction cannot issue until it is validated, introducing unnecessary delays; second, the older instruction inherits all dependencies of the younger one, further stalling execution.

This work advances the state of the art in non-consecutive fusion by first revealing the limitations of previous proposals and designing and modeling a more practical fusion architecture. Then, we propose *Catalyst-Length Instruction Fusion prediction* (CLIF), where the fusion predictor supporting non-consecutive instructions is always trained using the same context for the same speculatively fused pair. Our key contribution is to define this context as a function of the number of branches between the two fused instructions. Additionally, CLIF adapts the prediction confidence based on the type of fused pair, enabling more conservative training for pairs prone to misprediction and more aggressive training for stable pairs. Thus, CLIF increases prediction accuracy while requiring low storage. Finally, we refine the fusion microarchitecture in which fused instructions can be executed independently but still share a single ROB and LSQ entry, capturing most performance benefits while reducing complexity and drawbacks.

Our simulation results show that a cost-effective implementation of CLIF improves Helios by 2.05% on SPEC CPU 2017 and 3.16% on MiBench, while reducing mispredictions by 95.31%, using less than half the storage budget of Helios, and simplifying the microarchitectural requirements for fusion.

II. Background

Non-consecutive fusion (NCF) is a relatively recent research area with limited prior work. Aside from industry patents [20], which lack detailed descriptions and evaluations, Helios is the only published microarchitecture supporting NCF. Thus, this section reviews Helios, which serves as our baseline. In Helios, a NCF'd instruction comprises a *head nucleus* (older instruction), a *tail nucleus* (younger instruction), and the intervening instructions, called the *catalyst*.

In Helios, NCF is speculative because the two nuclei are fused before all conditions required to guarantee a valid fusion are known. In particular, the catalyst may contain dependencies or instructions that prevent the two nuclei from executing as a single instruction, and some conditions can only be checked after the nuclei have progressed further through the pipeline. Helios therefore provides three recovery mechanisms, depending on the type and time of detection of an invalid condition. A *repair* preserves the fused instruction but corrects information that became stale (such as a source register of the tail nucleus) due to renaming the tail instruction before the catalyst. An *unfusion* is required when the two nuclei cannot legally remain fused. Thus, the fused operation is converted back so that the tail nucleus can execute independently, without squashing the pipeline. Finally, when the invalid condition is detected too late, i.e., after the fused instruction initiated execution, Helios *squashes* (flushes) the affected younger instructions and re-executes them starting from the tail nucleus.

Helios supports load-load and store-store fusion through four key mechanisms: (1) fusion prediction, (2) instruction fusion, (3) validation, and (4) support for handling incorrect fusion cases, mispredictions within the catalyst, and retirement. Figure 1 summarizes the required pipeline modifications. For a Lion-Cove-like core (see Section VI), Helios requires 2.6 KiB of storage overhead for pipeline and structure modifications and 9.5 KiB for the Fusion Predictor, totaling 12.1 KiB.

A. Helios Fusion Predictor

The predictor identifies tail nuclei and provides the instruction distance to the corresponding head nuclei. It comprises the *Unfused Committed History (UCH)*, which tracks recently committed unfused instructions, and the *Fusion Predictor (FP)*, which uses this information to learn fusion opportunities.

UCH ❶ is a fully-associative cache storing a partial cache-line tag and a commit number (CN). It maintains several load entries and one store entry. A match indicates a fusion opportunity, updating the FP with the instruction distance (by substracting the CNs) and invalidating the matching entry. Otherwise, the retiring instruction is inserted. Fusion distance is limited to 64 instructions.

FP ❷ is a tournament predictor combining local and global predictors. Each entry stores a confidence counter, tag, and fusion distance. Correct predictions leave the counter unchanged, while mispredictions reset it. Matches in the UCH increment the counter until saturation.

B. Fusing the instructions

At decode ❸, the predicted distance is retrieved and fusion is attempted in the Allocation Queue (AQ) ❹. Fusion occurs if (i) the confidence counter is saturated, (ii) the pair forms a valid idiom, and (iii) the head nucleus is in the AQ or is part of the same decode group.

The head nucleus is replaced by the NCF'd instruction, while the tail remains in the AQ. The AQ is extended with (i) one bit per source and destination register to identify its nucleus, (ii) head/tail flags, and (iii) an *NCS_tag* pointing to the other nucleus in the AQ.

The head nucleus renames both nuclei. Since renaming is in order, the tail's destination register is temporarily stored in a dedicated buffer ❺ together with its *NCS_tag* and a *Deadlock_tag*. Once renamed, the tail updates the RAT.

The Issue Queue (IQ) has a *NCS_ready* bit ❻, initially cleared and set after validation. This prevents issuing a NCF'd instruction until source registers are repaired, deadlocks are checked, and, for store pairs, intervening stores are ruled out.

C. Validating the NCF'd instruction

A NCF'd instruction is validated when the tail nucleus reaches Rename ❼. First, up to two NCF levels may coexist. *Max_Active_NCF* tracks head nuclei entering Rename, while *Active_NCF* tracks nested NCF completions ❽. When *Max_Active_NCF* saturates, subsequent head nuclei are unfused. It resets once *Active_NCF* reaches zero or on a squash.

Second, the tail nucleus checks for RAW dependencies within the catalyst. Since renaming must proceed in program order, the head nucleus renames the entire NCF'd instruction before the catalyst and tail nucleus are renamed. Consequently, a source belonging to the tail may initially be renamed using a stale RAT mapping. To fix it, when the tail nucleus reaches Rename, Helios detects this dependence and *repairs* the corresponding source register in the IQ entry of the NCF'd instruction with the correct physical register. Thus, repair does not undo the fusion; it only corrects the fused instruction's operand mapping, incurring a one-cycle penalty.

Third, the tail nucleus verifies the conditions that make the predicted fusion illegal, including cyclic dependencies (i.e., the tail nucleus depends directly or indirectly on the head nucleus), serializing instructions in the catalyst, and, for NCF'd stores, intervening stores. Unlike a repair, these conditions require the nuclei to execute independently to avoid deadlocks or, in the case of intervening stores, to preserve memory ordering. In those scenarios, Helios *unfuses* the pair: the already allocated NCF'd instruction keeps the operation corresponding to the head nucleus while the tail nucleus is dispatched as a standalone instruction. Because the tail nucleus remains in the pipeline when validation occurs, unfusion can be performed at this stage without squashing younger instructions.

Helios detects cyclic dependencies using RAT deadlock tags ❿, a bit-vector encoding NCF nesting levels. Instructions propagate tags by OR-ing source tags into the destination register, while head nuclei also set their corresponding bit. A deadlock is detected when a tail nucleus reads its bit.

Serializing instructions and stores within the catalyst of a NCF'd store are tracked using the Rename-stage flags *NCF_serializing* and *NCF_storePair* ❾.

To repair or unfuse the NCF'd instruction, the tail goes to Dispatch, where a dedicated buffer ⓫ stores, per nesting level, pointers to the NCF'd instruction ROB, IQ and LSQ entries, along with its *NCS_tag*.

D. Incorrect fusion, catalyst mispredictions, and retirement

Validation acts as a point of no return for unfusion. Before validation, the tail nucleus is still present in the pipeline and can be dispatched independently if fusion is found to be illegal. After successful validation, however, the standalone tail nucleus is discarded as it has been merged with the head instruction. Consequently, an invalid condition detected after this point cannot be handled by simply unfusing the pair.

This condition is detected after execution, when the effective addresses of both nuclei become available. If the two memory accesses cannot be serviced as a single fused memory operation because their addresses span a region larger than 64 bytes (cache line size), the fusion is invalid. Since the tail nucleus is no longer present but its resources have already been allocated, Helios squashes the pipeline and re-executes the original tail nucleus as an independent instruction. We refer to this pipeline-level recovery as a *squash*, in contrast to *unfusion*, which separates the nuclei before the tail is discarded.

To preserve memory consistency and precise exceptions, Helios delays retirement until both nuclei and the catalyst are ready to commit, effectively treating them as an extended commit group. This ensures the retire of instruction in order. In other words, a NCF'd instruction is not allowed to commit until all instructions in its catalyst are ready to do so. To track this, the ROB has two additional bits, one to track if the instruction belongs to an extended commit group (*ext_commit*), and another to track the start of the group (*NCS_start* ⓬).

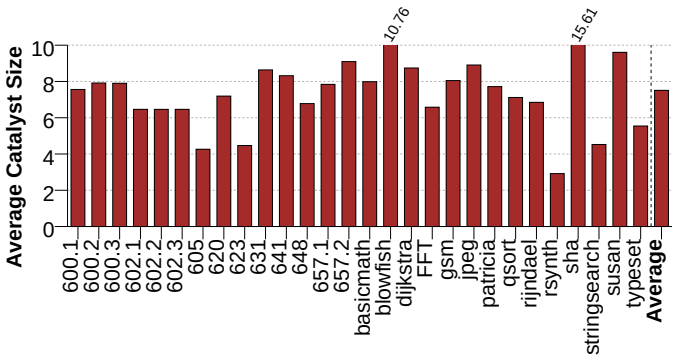


Fig. 2. Average catalyst size in an Oracle predictor

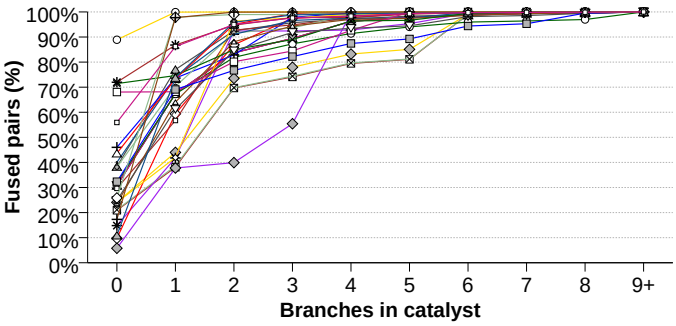


Fig. 3. Cumulative percentage of fused pairs in an Oracle predictor classified by the number of branches in the catalyst

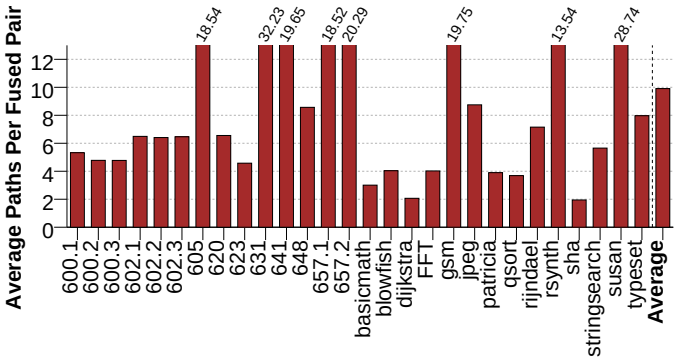


Fig. 4. Average number of paths learned per fused pair with Helios

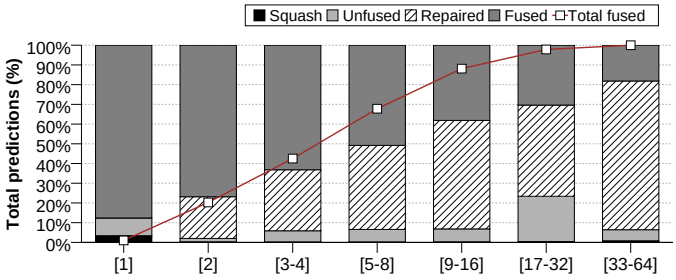


Fig. 5. Per-distance breakdown of fusion outcomes in Helios, including correct fusions, unfused opportunities due to cyclic dependencies, repaired NCF'd instructions, and squashes

Each instruction that goes through rename, and the Active_NCF is not zero, sets the ext_commit bit when dispatched. The first head nucleus of a nest also sets the NCS_start bit.

Finally, to allow precise exceptions and mispredictions, each ROB entry also holds two pointers (13) (one per allowed nested level) to unfuse any NCF'd instruction whose catalyst holds the faulting instruction, and flush from the faulting instruction.

III. Motivation

Helios is a pioneering approach that demonstrates the potential of non-consecutive instruction fusion, but its microarchitectural design exposes several important limitations. In particular, we identify three key issues: (i) the rigidity and inefficiency of its tournament-based predictor with many mispredictions, (ii) limited effectiveness when spanning larger fusion windows, and (iii) overall microarchitectural complexity. These shortcomings motivate CLIF, a more compact, adaptive, and highly-accurate non-consecutive fusion design.

A. Limitations of a tournament fusion predictor

Helios relies on a tournament predictor composed of a meta-predictor and two sub-predictors: a PC-indexed local predictor and a gshare-style global predictor [21] indexed by the PC and using a fixed global history length. We observed that using 11 branches for the global history provides the best results for Helios (see Section VII-A).

Instruction fusion is primarily driven by short-range correlations between nearby dynamic instructions. Our analysis shows that the average catalyst length in an Oracle fusion predictor is only 8 instructions (Figure 2), with just 1.37 branches on average in the catalyst (Figure 3). Consequently, long global histories may offer limited additional benefit, while potentially increasing aliasing and exacerbating path explosion in the predictor global table.

As a result, Helios must redundantly learn multiple histories for the same fused pair. Indeed, Figure 4 shows that on average 10 distinct histories are observed per pair, with up to 32.2 in the worst case (631.deepsjeng), spreading the same prediction across multiple entries in the predictor.

The tournament structure further exacerbates these limitations. The meta-predictor requires training to select between sub-predictors, increasing warm-up time and delaying adaptation. In addition, because Helios invalidates entries on misprediction, previously learned fused pairs must be relearned.

B. Locality of fused instructions

The most profitable fusion opportunities arise when the head and tail nuclei are close in the dynamic instruction stream. However, Helios supports fusion across distances of up to 64 instructions. In practice, such large fusion windows provide limited benefit and frequently introduce pathological cases that are either unfused or stall the head nucleus, reducing the effectiveness of fusion.

Figure 5 characterizes fused instruction pairs as a function of their distance. We classify the outcome of these pairs into four categories: *squash* represent incorrect fusions causing

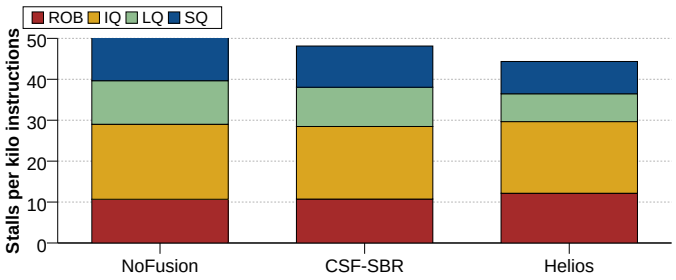


Fig. 6. Stalls of Helios against not fusing, and fusion consecutive instructions

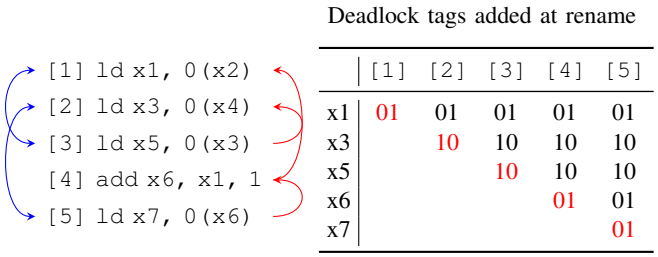


Fig. 7. Undetected cyclic dependence between interleaved NCF'd instructions

pipeline flushes; *unfused* opportunities are reverted fusion pairs due to cyclic dependencies or deadlocks; *repaired* NCF'd instructions require correction due to RAW dependencies in the catalyst); and *fused* pairs that execute correctly. The overlaid red line shows the cumulative contribution of each distance interval to all correct fusion events.

Results highlight a strong locality trend. For distances of one to two instructions, most fusion attempts are correct and rarely require repair. In contrast, fusion reliability degrades rapidly with distance. Fusions beyond 16 instructions account for only 12% of correct predictions and are highly error-prone, as up to 80% require repair or are undone due to cyclic dependencies.

Overall, these results show that short-range fusion captures the majority of useful opportunities while remaining significantly more reliable. Long-distance fusion is not only rare and difficult to predict but also incurs a disproportionate overhead.

C. Complexity of Helios

A final limitation of Helios lies in the complexity of its fusion mechanism. Beyond the cost of computing multiple memory addresses and snooping the LSQ for memory-pair operations, Helios increases pressure on both register renaming and instruction scheduling.

During register renaming, the head nucleus of the NCF'd instruction renames the destination registers of both nuclei and allocates physical registers for all source operands, but only updates the RAT with the head nucleus destination register. The tail nucleus then detects dependencies between the head and tail and repairs the instruction in the presence of RAW dependencies within the catalyst. This process introduces redundancy: the NCF'd instruction renames the tail's source registers, while the tail nucleus also performs renaming to detect conflicts and cyclic dependencies. This duplication increases both complexity and energy.

Instruction scheduling is similarly impacted. The head nucleus must wait not only for its own dependencies but also for those of the tail nucleus, which may be younger instructions, or at least wait for the tail to validate the fusion. This added latency often outweighs the benefits of reduced issue queue occupancy. Figure 6 illustrates these effects. While Helios reduces stalls due to full LQ and SQ occupancy, it fails to reduce—and in fact slightly increases—stalls in the ROB and IQ. This stems from two factors: (1) extended commit groups must wait until every instruction in the group is ready to retire, and (2) the added issue latency further increases these stalls.

Furthermore, Helios does not specify how tail nucleus memory dependencies are handled after fusion. Two options exist: (1) discard the tail nucleus's memory dependence prediction, or (2) propagate it to the NCF'd instruction. The former becomes increasingly costly as processors grow more aggressive due to higher memory order violation rates and squash costs, whereas the latter can lead to deadlocks. Consider the following example, where loads `[1]` and `[3]` are fused:

```
[1] ld x1, 0(x2)
[2] st x1, 4(x2)
[3] ld x4, 4(x2)
```

This indirect dependence goes undetected because the RAT-based poisoning mechanism ends at the store, which cannot propagate deadlock tags, an aspect not explicitly accounted for in Helios and needed to capture such transitive dependencies.

Deadlocks can also arise with interleaved NCF'd instructions. Figure 7 illustrates this scenario. Instructions [1] and [3], as well as [2] and [5], are speculatively fused, as indicated by the blue arrows, while red arrows denote register dependencies. These dependencies form an indirect chain across the fused pairs, which can lead to cyclic interactions. The table alongside the figure reports the deadlock tags associated with each register, illustrating how these dependencies propagate through the register file. However, because tail nuclei only propagate the OR of their source-register deadlock tags, these cyclic dependencies remain undetected, and similarly such scenarios are not explicitly addressed in Helios.

To address these issues, we assume two extensions for Helios. First, loads also consider the stores they are predicted to depend on when checking cyclic dependencies. Second, each NCF'd instruction carries a bit indicating whether it depends on another interleaved pair; upon detecting a cyclic dependence, the younger pair is unfused to avoid deadlock.

IV. Catalyst-Length Instruction Fusion Prediction

This section introduces our catalyst-length instruction fusion prediction, CLIF. Although our design takes as a reference the RISC-V instruction set architecture, it can be adapted to other instruction sets. CLIF includes a predictor that detects load-load and store-store pairs that should be fused, and takes advantage of (i) the strong spatial locality of fusion opportunities, (ii) a per-pair catalyst-length history, and (iii) an improved training policy.

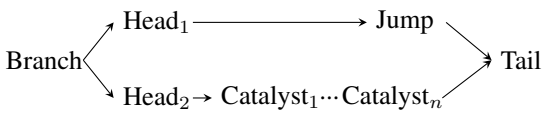


Fig. 8. Examples of path ambiguity

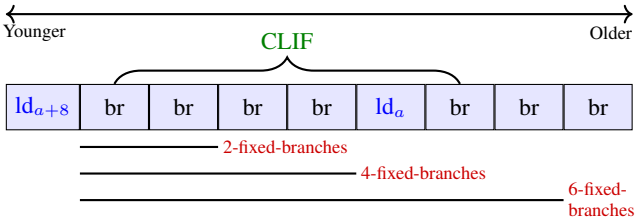


Fig. 9. Comparison of global history selection strategy

A. Catalyst-Length History

Over-fitted histories are unnecessary and can degrade predictor efficiency by introducing multiple execution contexts per fusion pair. The key challenge is therefore to capture sufficient context while avoiding path explosion. Our key claim is that instruction fusion is characterized by branches in the catalyst between the nuclei of a given fused pair.

Therefore, CLIF associates each fusion pair with the context-specific history length corresponding to the dynamic region between the head and tail nuclei: the catalyst. We restrict the prediction history to divergent branches, namely conditional and indirect control-flow instructions, following the insights of previous work in memory dependence prediction [19]. Conditional branches contribute to the history with their direction, whereas indirect branches contribute with their target. To further disambiguate paths that converge in the same fusion region, CLIF also includes the target of the most recent divergent branch preceding the head nucleus. This captures the entry path into the fusion region and prevents aliasing between distinct control-flow paths.

Figure 8 illustrates the effect of including the target of the divergent branch preceding the head nucleus. The upper path contains the head nucleus and a non-divergent jump, whereas the lower path contains a different head nucleus and multiple non-control instructions. Without encoding the divergent branch preceding the head nucleus, both paths become indistinguishable, increasing mispredictions. Including this information resolves such ambiguities and improves accuracy.

Figure 9 highlights the benefits of this approach for a representative fusion pair ($\texttt{ld}_a$, $\texttt{ld}_{a+8}$). By selecting only the divergent branches in the catalyst along with the preceding branch target, CLIF captures the full control-flow context. In contrast, fixed-length histories are either insufficient or excessive: shorter histories (e.g., 2 or 4 branches) fail to capture the full control context, leading to aliasing, while longer histories (e.g., 6 branches) cause path explosion and reduce predictor efficiency.

We extend each UCH, i.e., the small cache used to train the fusion predictor, entry with a *divergent branch commit number* (CN_{br}), analogous to Helios' commit number (CN). Unlike CN, CN_{br} increments only when a divergent branch commits, and loads and stores copy its value. When snooping the UCH for fusion opportunities, the number of divergent branches between two nuclei is computed as the difference between their CN_{br} values, analogous to computing the fusion distance with CN. Given the small UCH size (six load and one store entries), this extension incurs negligible storage overhead, requiring only $\log_2(\textit{max_branch_tracked}) + 1$ additional bits per entry.

Overall, tailoring history length to each fusion instance and encoding only relevant control-flow information improves accuracy and training efficiency while mitigating path explosion.

B. Safe and Fast Training

The training policy can be improved by differenciating fusion pairs based on their risk profile. We differentiate the training according to the base register of the memory operations. Pairs accessing different base registers are more prone to mispredictions that trigger pipeline flushes, and are therefore trained conservatively in CLIF. In contrast, pairs that share the same base register are typically more stable and can be trained aggressively. Our results indicate that, on average, 30% of mispredictions originate from same-base-register pairs.

To enable this distinction, each UCH entry stores the architectural base register of the unfused instruction using 5 additional bits. When a fusion opportunity is encountered, the predictor compares the stored registers: if the base registers match, the confidence counter is saturated; otherwise, it is initialized to one and incremented on subsequent encounters up to its maximum value.

In addition, to prevent repeated learning of error-prone fusion pairs that lead to misprediction, invalidation, and relearning, we extend the confidence mechanism to explicitly encode negative outcomes. Specifically, negative counter values are used to predict that a pair should not be fused.

When a tail nucleus is predicted not to fuse, it retains the instruction distance to the head nucleus. At commit, the tail nucleus probes the UCH: if a valid entry exists at the same distance and the pair is determined to be non-fusable, the confidence counter is set to strongly not to fuse. Otherwise, the counter is incremented.

C. A cost-effective implementation

CLIF is implemented using four prediction tables, each accessed in parallel on every prediction. Each table is organized as a cache and corresponds to a different control-flow history length (number of divergent branches), selected from {0, 1, 2, 3}, as instruction fusion exhibits strong spatial locality and most fused pairs occur within a small dynamic window. Since the divergent branch immediately preceding the head nucleus is always included, the effective history lengths range from 1 to 4. Longer histories are truncated to limit complexity and storage overhead. For example, if a fusion pair spans five branches, only the four closest to the tail nucleus are retained.

The history is formed using the taken bit of conditional branches and the five least significant bits of the target of

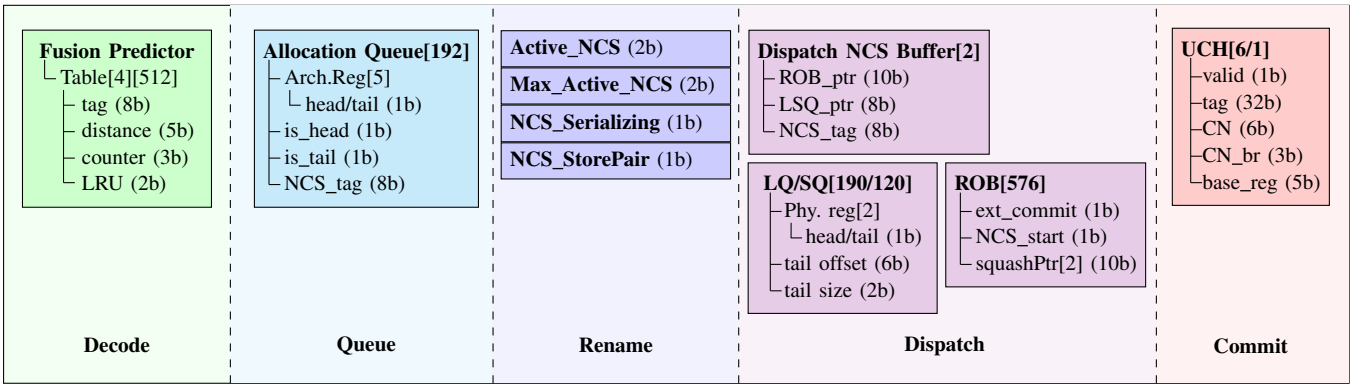


Fig. 10. Overview of CLIF architecture

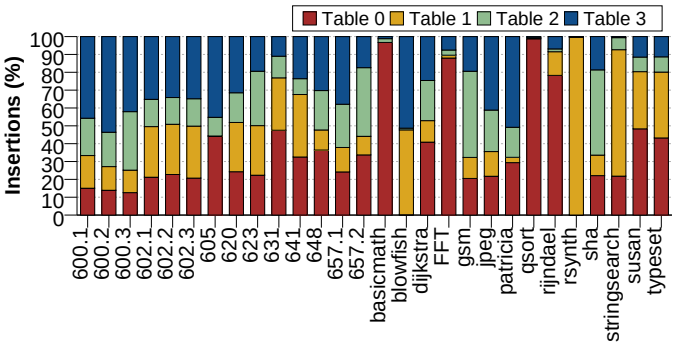


Fig. 11. Per-benchmark distribution of insertion percentages across tables

indirect branches. The branch immediately preceding the head nucleus always uses the five least significant bits of its target, regardless of the type of divergent branch. Extending the previous example, the three branches closest to the tail nucleus contribute either their taken bit or target, while the fourth branch always contributes its target. To compute the set index and tag to access the tables, the PC (with the least significant bit discarded) is hashed with the history.

An important observation that we make in this work is that the distribution of fusion pairs across history lengths is not uniform, as shown in Figure 11: each benchmark has a unique pattern. To balance table utilization we implement a bank interleaving technique in our predictor, inspired by the ISL-TAGE and TAGE-SC-L branch predictors [17], [22]. With bank interleaving, tables are not statically assigned to history lengths, but are the two least significant PC bits the ones that select the starting table (length 1), rotating across the remaining tables in a round-robin manner.

Each table is 4-way associative with 128 sets, for a total of 512 entries per table and 2,048 entries overall. Each entry contains an 8-bit tag, a 5-bit distance field encoding the distance between fused instructions, a 3-bit signed confidence counter, and a 2-bit LRU field, for a total of 18 bits per entry. The 5-bit distance field limits fusion to 32 instructions, capturing most fusion opportunities in Helios (see Figure 5), thereby achieving high coverage while avoiding highly error-prone long-distance pairs. The overall storage cost is 4.5 KiB.

Figure 10 summarizes the key structure changes to implement CLIF. Overall, in addition to the 4.5 KiB, CLIF requires 2.32 KiB for a Lion-cove-like processor (See Section VI), totaling 6.82 KiB of storage overhead.

V. Unconstraining non-consecutive fusion

The main advantage of speculative non-consecutive instruction fusion is the reduction of pressure on backend structures (e.g., IQ, ROB, LSQ). However, as discussed in Section III-C, the increase in the issue latency of NCF'd instructions often offsets the benefit of sharing a single IQ entry.

To mitigate this limitation, we propose a partial fusion approach that restricts fusion to the ROB and LSQ, allowing instructions to share entries as in Helios, while keeping the head and tail nuclei unfused at the IQ and execution stages. Consequently, the nuclei are issued and executed independently, similar to Intel's micro-op fusion technique [7].

This decoupling simplifies register renaming for NCF'd instructions. By issuing fused instructions independently, CLIF avoids dependency cycles and RAW hazards between the tail nucleus and the catalyst. Thus, unfusion is only required when the catalyst contains serializing instructions (e.g., fences or atomic operations) or when additional stores appear in the catalyst of a fused store pair, to preserve memory consistency.

Independent issue also enables each nucleus to rename its source and destination operands separately, removing the redundant renaming overhead present in Helios. Similarly, memory address generation is simplified: each nucleus independently computes its address, performs LSQ snooping, and initiates memory accesses, eliminating the need for coordinated dual-address handling within a fused operation. However, similarly to Helios, if the memory accesses of the two nuclei span more than a single cache line, a misprediction is triggered and the pipeline is flushed, as the LSQ entry of the fuse pair cannot encode both memory addresses.

A. Design comparison with Helios

This section compares CLIF and Helios to highlight their key architectural differences. Both mechanisms target non-consecutive memory instruction fusion, but they adopt fundamentally different design choices across pipeline stages,

TABLE I
SYSTEM CONFIGURATION

8-core Processor	
Front-end width	10-wide fetch/decode/rename
Allocation queue	192 entries
Branch predictor	TAGE-SC-L [17]
Back-end width	18 execution ports, 12-wide commit
ROB/IQ/LQ/SB	576/318/190/120 entries
Memory hierarchy	
L1I (private)	32KB, 8 ways, 4-cycle hit latency, pipelined, 32 MSHRs
L1D (private)	64KB 8 ways, 5-cycle hit latency, pipelined, 32 MSHRs
L1D prefetcher	IP-stride with a prefetch degree of 3
L2 (private)	3MB, 24 ways, 14-cycle hit latency, 32 MSHRs
L3 (shared)	3MB/bank (8 banks), 24 ways, 36-cycle hit latency, 32 MSHRs
Memory	4GB, 100-cycle access latency

resulting in distinct trade-offs in complexity, latency, and robustness. Rather than focusing on performance, we examine each pipeline stage to explain how CLIF departs from Helios and why these differences impact correctness and efficiency.

1) Decode and Commit Stages: The Decode and Commit stages are identical to Helios'. CLIF's fusion predictor is queried at decode by loads and stores. Upon a prediction, the head nucleus in the AQ is replaced by the NCF'd instruction. At commit, unfused memory operations query the UCH for fusion opportunities or are inserted into the UCH otherwise.

2) Rename and Dispatch Stages: In Helios, a NCF'd instruction simultaneous renames both head and tail nuclei. This entails renaming all source and destination registers at the head nucleus, followed by re-renaming at the tail nucleus to detect direct and indirect dependencies. If a RAW conflict with the catalyst is detected, the fused instruction must be repaired.

CLIF simplifies this process by limiting the fusion advantages to the reduction in occupancy in the ROB and LSQ structures. The head and tail nuclei are renamed independently, each processing its own source and destination operands. This eliminates the need for dependency cycle detection and RAW conflict checks between the tail nucleus and the catalyst, removing redundant renaming work and complexity. Upon dispatch, the tail nucleus updates the destination physical register fields in the ROB and LSQ entries.

In Helios, to prevent deadlocks, a NCF'd instruction is unfused whenever the tail nucleus depends on the head nucleus. Unfusion is also required when the catalyst contains serializing instructions. Furthermore, to preserve memory consistency, a NCF'd store pair cannot include additional stores in the catalyst. Additionally, RAW conflicts between the tail nucleus and the catalyst require repairing the NCF'd instruction.

In contrast, CLIF decouples the execution of nuclei, eliminating the need to check for dependency cycles and RAW hazards between the tail nucleus and the catalyst. Consequently, unfusion is only required when the catalyst contains serializing instructions or additional stores in a fused store pair.

3) Issue Stage: In Helios, the head nucleus of a fused pair cannot issue until the dependencies of both nuclei are satisfied. As the catalyst size increases, the head nucleus is increasingly delayed by younger dependencies from the tail nucleus, resulting in higher issue latency. Furthermore, the fused instruction must stall until the tail nucleus reaches the Issue stage to validate the fusion.

In contrast, CLIF issues nuclei in an unfused manner, by allowing the head and tail nuclei to be issued and executed independently. Each nucleus occupies its own IQ entry and waits only on its own dependencies, enabling forward progress even when the other nucleus is delayed. This approach retains most backend capacity benefits of fusion while avoiding the issue-stage stalls that frequently diminish Helios' effectiveness.

VI. METHODOLOGY

Our simulation infrastructure consists of a cycle-level in-house simulator modeling in detail an out-of-order processor with RISC-V instruction set architecture. The core is fed with an instruction flow generated by a modified version of the RISC-V Spike Simulator [23] that runs in full-system mode with a Linux kernel. The memory hierarchy is modeled with Ruby/SLICC [24], using its embedded GARNET interconnect network model [25]. The simulated core is configured with parameters similar to an Intel Lion Cove P-Core microarchitecture [26]. System parameters are summarized in Table I.

The pipeline depth of our model is 12 stages, although load and store instructions experience an effective depth of 16 stages due to the additional latency of the L1D cache access and LSQ snoop (searched parallel and incurring the same latency [7]). The minimum penalty for branch misprediction is 9 cycles. The pipeline has three ports for load execution, three ports for generating store address, and two ports for store data. Each cycle, up to 3 memory instructions can initiate the cache access: either 1 store and 2 loads, 2 stores and 1 load, or 3 loads [7], [27], [28]. Our model implements a Total-Store-Order consistency model, being compliant with the RISC-V TSO extension (Ztso).

Each instruction in a rename bundle is provisioned with two Rename Map Table (RMT) read ports and one RMT write port. Similarly, each issued instruction may consume up to two Physical Register File (PRF) read ports and one PRF write port. To keep the rename logic and physical register file unchanged across all configurations, fused instructions occupy two rename slots/execution lanes and consume the resources of two conventional instructions. Thus, a fused instruction may use up to four RMT reads, two RMT writes, four PRF reads, and two PRF writes, without requiring additional port provisioning beyond the baseline design.*

The evaluation is carried out with the SPEC CPU 2017 (Speed suite, reference inputs) [29]† and MiBench [30] suites.

*When CLIF splits the execution of a NCF'd instruction, the head and tail is renamed/executed independently and each occupies a single slot/lane.

†Due to a Spike limitation (https://github.com/riscv-collab/riscv-gcc/issues/175), we were unable to run application 625.x264_s in full system mode.

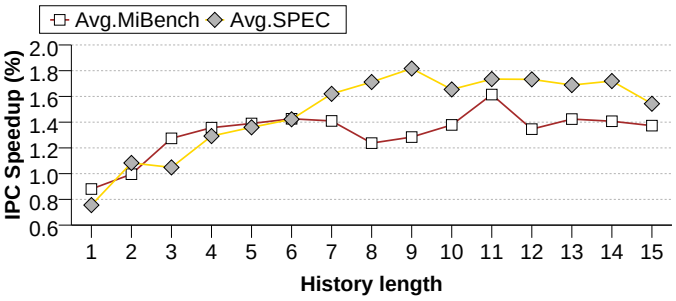


Fig. 12. Average speedup for Helios with different history lengths over CSF-SBR as a baseline

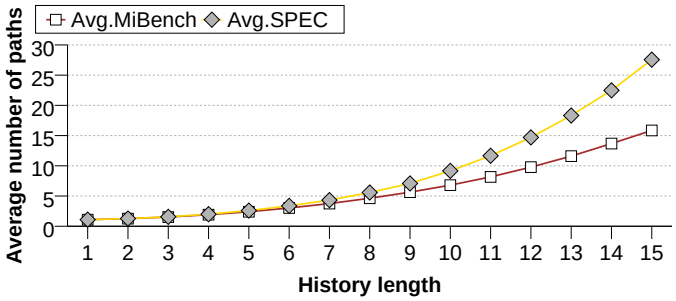


Fig. 13. Average number of paths per pair detected with different history lengths of Helios

We skip the Linux kernel boot and setup for all applications. For MiBench, the full application is run. For SPEC, we fast-forward to the start of the ROI and proceed to report results for the next 500M instructions. Benchmarks with multiple inputs are labeled with increasing suffixes for each combination of app-input. All binaries are compiled with GCC 13.2 targeting the *RV64GC* instruction set [31] with *-O3 -static*.

The fusion mechanisms evaluated in this work include: *CSF-SBR* for ConSecutive Fusion with Same Base Register, which fuses consecutive instructions that access contiguous data through the same base register, where the accesses may overlap or be asymmetric (different access sizes); a 9.5KiB *Helios*; and our 4.5KiB implementation of *CLIF*. In addition, we evaluate *OracleCLIF*, an oracle predictor based on CLIF that knows in advance the effective addresses of all memory operations and whether fusion-preventing elements are present in the catalyst. However, this Oracle is not ideal: it lacks awareness of dynamic constraints such as congestion in backend structures.

VII. Experimental Results

This section is organized into four parts. First, we evaluate Helios under varying history lengths. While the original design uses a 9-branch history, we explore a range of histories to assess their impact on performance. Next, we evaluate different fusion mechanisms relative to a baseline without fusion. We then compare CLIF against Helios. Finally, we present a sensitivity analysis of CLIF.

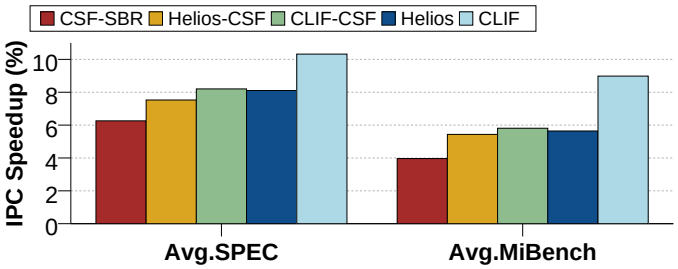


Fig. 14. Average speedup for different fusion policies over no fusion

A. Impact of History Length on Helios

Since we advocate for shorter history lengths, we evaluate their impact on Helios by sweeping history sizes from 1 to 15 branches with all other predictor parameters fixed. Figure 12 reports IPC speedup relative to CSF-SBR, and Figure 13 shows the average number of paths detected per fused pair.

For SPEC CPU 2017, performance improves steadily up to a history length of 9 branches, after which it gradually degrades for longer histories. In contrast, MiBench exhibits limited sensitivity to history length, with most configurations showing minimal variation beyond 4 branches; the best configuration uses 11 branches. Across both suites, performance saturates, while the number of detected paths per pair grows exponentially, particularly for SPEC.

Overall, these results suggest that a fixed history length is suboptimal across workloads, motivating adaptive mechanisms that dynamically adjust the effective history length. Nevertheless, for the remainder of this evaluation, we use an 11-branch history configuration for Helios, as it provides the best overall performance across both benchmark suites.

B. Impact of Fusion Scope on Performance

Instruction fusion can be implemented with varying degrees of aggressiveness, trading off complexity and risk for potential performance gains. In this subsection, we evaluate a spectrum of fusion strategies relative to a no-fusion baseline. We begin with a non-speculative consecutive fusion policy with contiguous accesses (CSF-SBR), followed by consecutive variants of Helios (Helios-CSF) and CLIF (CLIF-CSF), which additionally support fusion across cache-line boundaries and across different base registers. Finally, we evaluate fully non-consecutive fusion using Helios and CLIF. The results of this study are shown in Figure 14.

Overall, CSF-SBR achieves speedups of 6.3% and 4.0% for SPEC and MiBench, respectively. The consecutive variant of Helios improves performance to 7.5% and 5.4%, while CLIF-CSF further increases speedups to 8.2% and 5.8%.

When enabling non-consecutive fusion, Helios fails to effectively exploit these opportunities due to the limitations discussed in Section III, particularly for the MiBench suite. In contrast, CLIF achieves speedups of 10.3% and 9.0% over the no-fusion baseline for SPEC and MiBench, respectively. The next section presents a per-application comparison of Helios and CLIF, using CSF-SBR as the baseline.

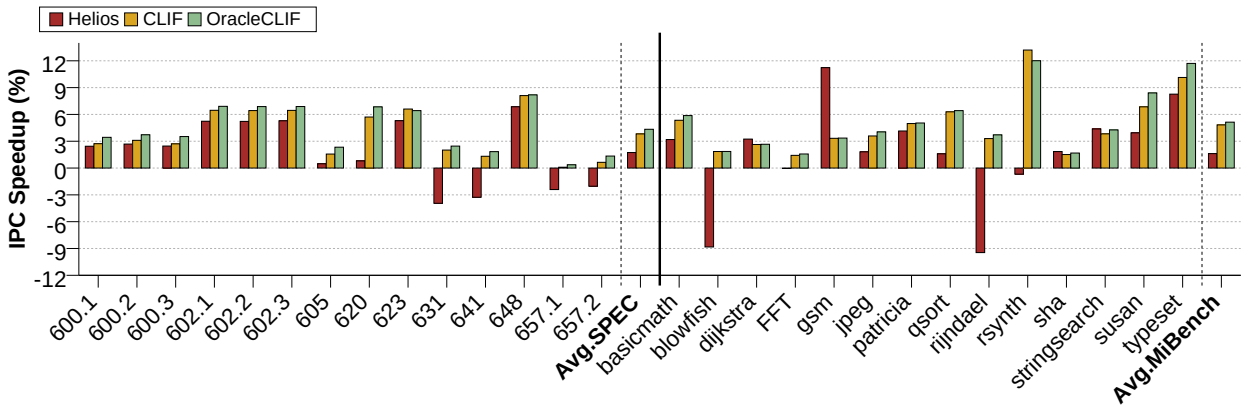


Fig. 15. Speedup over CSF-SBR for Helios, CLIF, and the Oracle version of CLIF

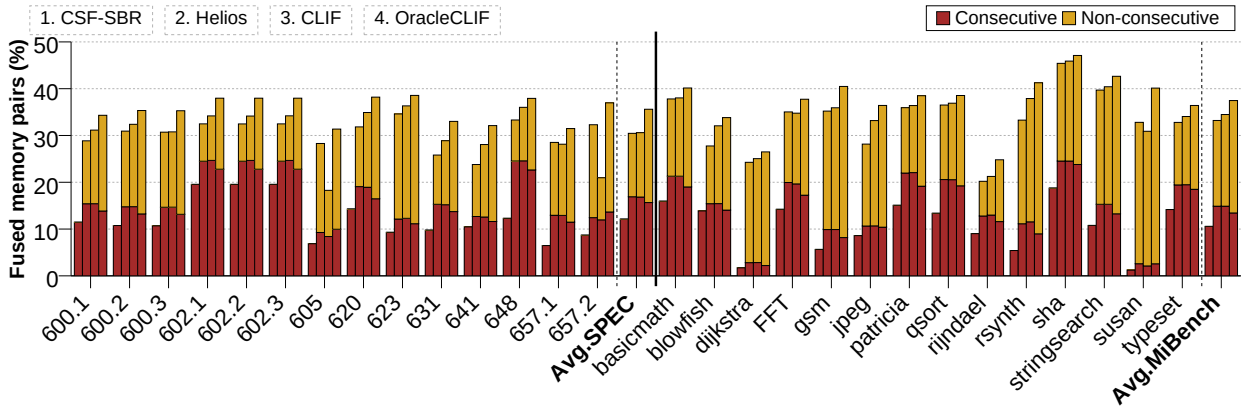


Fig. 16. Percentage total dynamic memory instructions with CSF and NCF for CSF-SBR, Helios, CLIF, and its Oracle version

C. Non-Consecutive Fusion Performance Analysis

Figure 15 reports speedup over CSF-SBR for Helios, CLIF, and an Oracle predictor based on CLIF. On average, CLIF outperforms both CSF-SBR and Helios, achieving mean speedups of 3.82% and 2.05% on SPEC, and 4.83% and 3.16% on MiBench, respectively. Despite these gains, Helios still outperforms CLIF in selected benchmarks, primarily in MiBench, most notably *gsm*.

In *631.deepsjeng* and *641.leela*, Helios suffers from high MPKI, leading to frequent pipeline flushes. In *657.xz*, although MPKI is moderate, 74% of correctly fused cases require repair, introducing issue delays for the head nucleus and increasing backend congestion. In all four benchmarks, the limited benefit from non-consecutive fusion exposes these overheads, resulting in net performance degradation.

641.leela exhibits numerous hard-to-predict memory accesses due to pointer-chasing patterns. These addresses are both difficult to predict and often insufficiently proximate to form a valid fusion idiom, causing frequent mispredictions and pipeline flushes when the nuclei are incorrectly fused.

A similar behavior is observed in *rijndael*, where Helios exhibits the highest number of incorrectly fused NCF'd instructions, leading to frequent pipeline flushes. This arises from the frequent use of function parameters as base addresses, which can form valid non-consecutive fusion pairs using different base registers. When such addresses appear close during training, the predictor is misled into learning incorrect fusion patterns, resulting in wrong fusion decisions in subsequent iterations where the addresses no longer constitute a valid fusion idiom. In contrast, CLIF mitigates this issue by learning to avoid fusing such pairs in both benchmarks.

gsm is the only benchmark where Helios greatly outperforms CLIF under non-consecutive fusion. In this workload, fewer than 12% of fused instructions require repair, minimizing dependency-induced stalls for the head nucleus. Under these conditions, sharing a single IQ entry substantially improves performance. However, *gsm* is an outlier. In most benchmarks, the head nucleus frequently waits for tail dependencies, negating the benefits of IQ entry sharing.

This effect is most pronounced in *blowfish*, which features a tight loop with loop-carried and periodic memory dependencies. The compiled code relies heavily on stack operations, repeatedly loading and storing base addresses. In Helios, aggressive fusion causes the head nucleus to inherit the tail dependencies, delaying issue and creating backpressure in backend structures. As extended commit groups cannot retire until all instructions are ready, ROB, IQ, and LSQ entries remain occupied longer, ultimately stalling the front-end.

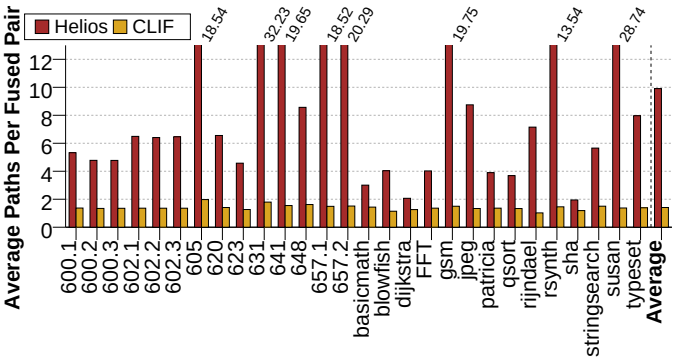


Fig. 17. Average number of paths seen per fused pair in Helios and CLIF

The following code fragment from *blowfish* illustrates this behavior, where instructions [1] and [6] form a valid NCF pair:

```
[1] sb a5,-57(s0)
[2] mv a4,s2
[3] ld a5,-56(s0)
[4] add a5,a5,a4
[5] lbu a5,0(a5)
[6] sb a5,-58(s0)
```

Fusing [1] and [6] forces [1] to inherit all dependencies of [6], delaying its execution until the catalyst completes. This delays dependent loads and creates a cascading effect that saturates backend structures, demonstrating that aggressive fusion can be detrimental.

CLIF avoids this behavior by decoupling issue and execution: the head and tail nuclei proceed independently, allowing the head nucleus and its dependents to execute without waiting for the tail nucleus's dependencies.

Figure 16 shows the fraction of memory instructions fused. As observed by Singh et al. [12], Helios favors consecutive fusion due to training on unfused instructions, and CLIF exhibits the same behavior, as it follows a similar training methodology. Non-consecutive fusion approximately doubles the number of fused memory operations; however, the associated overheads limit the resulting performance gains.

CLIF achieves slightly higher fusion coverage than Helios by (1) reducing the number of paths tracked per fused pair and (2) avoiding unfusion when cyclic dependencies are detected, thereby improving ROB and LSQ utilization. However, in certain benchmarks, such as *605.mcf* and *657.xz.2*, learning to not fuse pairs that caused mispredictions can reduce non-consecutive fusion coverage, as pairs with low misprediction frequency are still filtered out.

Figure 17 shows that the average number of paths detected per fused pair is below 2 across all benchmarks, contributing to CLIF's low storage overhead and stable performance.

Figure 18 shows the performance of different Helios and CLIF budgets relative to the CSF-SBR baseline. Helios is relatively insensitive to budget reductions, whereas CLIF degrades more noticeably—especially on SPEC—due to increased aliasing. Increasing Helios' budget benefits only SPEC, while CLIF yields modest gains in both benchmark suites.

TABLE II
FUSION PREDICTOR COVERAGE (%) VS. ORACLE AND MPKI FROM FLUSH-INDUCING MISPREDICTIONS

Benchmark	Helios	CLIF
600.perlbench.1	65.61 / 0.137	76.76 / 0.059
600.perlbench.2	73.04 / 0.094	79.59 / 0.029
600.perlbench.3	72.40 / 0.095	72.65 / 0.033
602.gcc.1	51.82 / 0.296	62.78 / 0.057
602.gcc.2	51.73 / 0.299	62.57 / 0.057
602.gcc.3	51.80 / 0.294	62.89 / 0.058
605.mcf	85.37 / 0.699	44.42 / 0.012
620.omnetpp	58.49 / 0.222	73.04 / 0.019
623.xalancbmk	81.37 / 0.196	87.61 / 0.008
631.deepsjeng	54.26 / 0.666	69.66 / 0.010
641.leela	54.69 / 0.646	74.87 / 0.005
648.nab	57.86 / 0.125	74.47 / 0.015
657.xz.1	76.33 / 0.353	74.05 / 0.001
657.xz.2	77.67 / 0.436	34.68 / 0.001
basicmath	78.21 / 0.006	79.25 / 0.000
blowfish	62.39 / 0.086	84.11 / 0.000
dijkstra	88.19 / 0.001	91.41 / 0.001
FFT	74.17 / 0.086	72.87 / 0.000
gsm	78.40 / 0.050	80.57 / 0.000
jpeg	67.27 / 0.125	86.51 / 0.001
patricia	72.88 / 0.017	74.95 / 0.000
qsort	83.98 / 0.285	85.82 / 0.000
rijndael	53.21 / 1.715	59.17 / 0.001
rsynth	68.61 / 0.021	81.61 / 0.000
sha	89.61 / 0.000	91.67 / 0.000
stringsearch	83.07 / 0.100	85.50 / 0.001
susan	80.10 / 0.321	75.20 / 0.000
typeset	75.56 / 0.379	81.65 / 0.005
Average	70.29 / 0.277	74.30 / 0.013

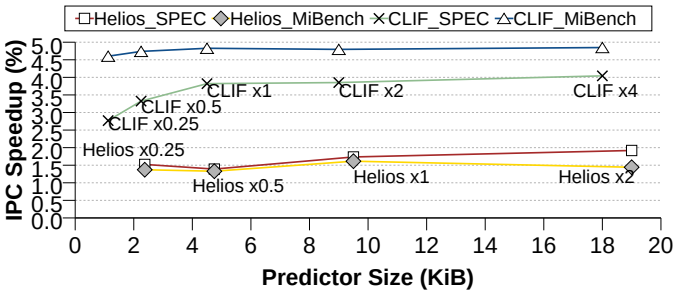


Fig. 18. Speedup over CSF-SBR versus storage for Helios and CLIF at different budgets

Finally, Table II reports coverage (for consecutive fusion with different base register and non-consecutive fusion) relative to the Oracle CLIF predictor and the MPKI of flush-inducing mispredictions. On average, CLIF reduces MPKI by 95.31% and improves fusion coverage by 4.0% over Helios.

D. Decomposing CLIF

In this section, we analyze the contribution of each CLIF component through an incremental ablation study, where each feature builds upon the previous one. We evaluate: (1) the CLIF predictor integrated into Helios with the original training policy, (2) interleaving, (3) hysteresis through the negative counter to avoid fusing unstable memory pairs, (4) the train-

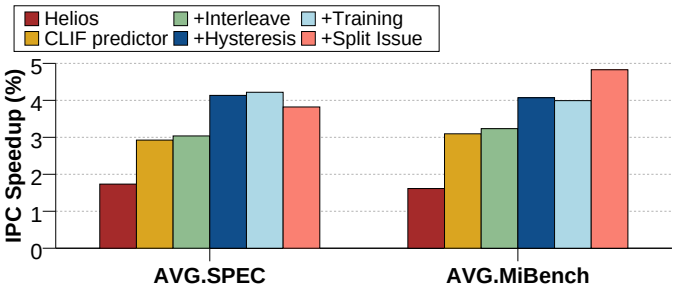


Fig. 19. Speedup over CSF-SBR for Helios and the different components of CLIF

ing policy that saturates same-base-register pairs upon first encounter, and (5) issuing nuclei as independent instructions. Figure 19 presents the speedups of Helios and the different versions of CLIF over CSF-SBR.

The CLIF predictor alone already outperforms Helios, achieving a 3% speedup over CSF-SBR in both benchmark suites. This improvement primarily stems from tailoring the history length to each particular fused pair, which leads to (1) increased coverage and (2) fewer mispredictions that trigger pipeline flushes. Interleaving further improves performance in both suites by distributing pressure more evenly across predictor tables. Adding hysteresis yields an additional 1% speedup by preventing the fusion of unstable pairs. In SPEC, an average of 5.3% of tail nuclei were marked as non-fusible, whereas only 1.4% were marked as such in MiBench. Consequently, CLIF's MPKI reduction over Helios increased from 51.22% to 95.31%.

Finally, augmenting hysteresis with the proposed training policy improves performance on SPEC but not on MiBench. Conversely, issuing nuclei independently slightly degrades SPEC performance while increasing MiBench speedup by 1%. This behavior stems from MiBench containing many small loops with intra-loop dependencies that delay the issue of the NCF'd instruction, whereas in SPEC the increased IQ pressure outweighs the benefits of issuing nuclei earlier.

E. Impact of Queue Size on CLIF Performance

Figure 20 reports the speedup of Helios and CLIF over CSF-SBR while scaling the size of the IQ, ROB, LQ, and SQ from 50% to 120% of the baseline.

Reducing the IQ capacity slightly degrades CLIF's speedup, while Helios remains largely unaffected. This behavior is expected, as in CLIF, fused nuclei do not share IQ entries. Consequently, under constrained configurations, the IQ becomes a bottleneck that limits the benefits of CLIF. Increasing the IQ size, however, has no observable impact on performance.

A similar but less pronounced trend is observed with CLIF for the ROB. Reducing ROB capacity slightly decreases speedup, although the impact is smaller than for the IQ. While fused nuclei share a ROB entry, this entry (along with those corresponding to instructions in the catalyst) cannot be freed until all constituent operations are ready to commit. Under tighter ROB pressure, this delayed commit becomes more pronounced, partially limiting the gains of CLIF. Increasing ROB size yields only marginal improvements. However, MiBench performance in Helios is significantly affected by ROB reductions due to delayed instruction issue and retirement.

In contrast, scaling the load and store queues exhibits the opposite trend for both designs. Reducing either the LQ or SQ capacity increases the observed speedup of Helios and CLIF, indicating that Non-consecutive fusion is particularly effective under LSQ-constrained conditions. When the LQ and SQ are enlarged, the relative advantage of Helios and CLIF is marginally reduced.

VIII. Related Work

Predictors using Increasing History Lengths. Many predictors in the literature leverage histories of varying lengths to improve accuracy in speculative execution. A prominent example is TAGE [14], [15], [17], [22], [32], [33], which indexes multiple tagged prediction tables using exponentially increasing history lengths. This organization allows TAGE to capture both short- and long-range control-flow correlations, and to blend predictions based on availability and confidence. TAGE has been adapted for other domains, including speculative memory bypass [34], [35], memory dependence prediction [36], and value prediction [37]. Recently, Kim and Ros [19] proposed a memory dependence predictor that avoids brute-force history tuning. Instead, it derives the relevant history length directly from the dynamic path between the conflicting store and load, reducing the amount of entries allocated for the same pair.

Coalescing Memory Operations. Prior work has also explored mechanisms to coalesce memory operations and reduce cache pressure. Olukotun et al. [38] propose to take advantage of FIFO access in front of the L1D to match multiple load addresses against an incoming cache line, serving them in parallel. Rivers et al. [39] take a similar approach to multi-porting the L1D by attaching line buffers to each cache bank, enabling multiple loads per cycle from recently accessed lines. Along the same line, Baoni et al. develop *Fat Loads* [40], which prefetch an entire cache line into a dedicated buffer to accelerate subsequent accesses to the same region. Additional strategies include always reading double words regardless of the requested size [41], extending load/store lifetimes post-retirement to expand forwarding opportunities [34], [42], [43], and coalescing committed non-consecutive stores while preserving total store order [44].

These techniques operate independently of instruction order, data dependencies, or register reuse. Furthermore, they do not require changing μ-op formats, thus they do not reduce pressure in the pipeline structures. In contrast, instruction fusion collapses two μ-ops into a single pipeline entity, directly reducing pressure on structures like ROB, LQ, and SQ.

Instruction Fusion has also been explored to simplify scheduling and improve efficiency. Kim and Lipasti [45] propose macro-op scheduling, where pairs of μ-ops are treated as a single unit in the IQ to reduce select logic complexity. Although this improves IQ throughput, the fused μ-ops still execute as separate instructions. Thakker et al. [20] are more aggressive, yet still avoid fusing when RAW or WAR hazards

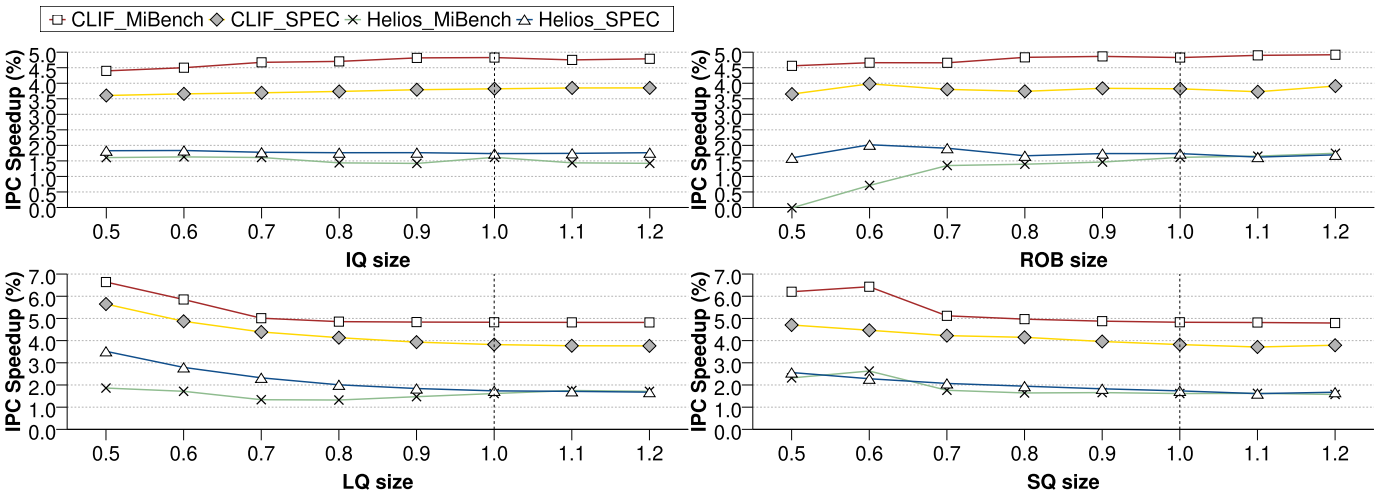


Fig. 20. Helios and CLIF sensitivity over varying sizes of backend queues

exist in the catalyst. Celio et al. [10] advocate for fusion in RISC-V designs to increase *work per μ-op* instead of extending the instruction set. Their approach fuses load and stores pairs, but is limited to consecutive and contiguous access. Reddy et al. propose CAIF [46], a compiler technique that identifies fusible but non-consecutive memory operations and reorders them to become consecutive, enabling instruction fusion without hardware prediction. CAIF improves Helios by 2.9% on SPEC, largely driven by 657.xz, whereas CLIF improves Helios across most remaining benchmarks, suggesting that CAIF and CLIF exploit complementary opportunities.

IX. Conclusion

This work presents CLIF, a lightweight, path-based predictor for non-consecutive memory fusion that significantly improves upon the state of the art both in performance and simplicity. CLIF uses control-flow information within the catalyst, defines a different confidence threshold for different fusible pairs, and treats the head and tail nuclei as independent instructions for issuing and execution.

Across SPEC CPU 2017 and MiBench, CLIF improves performance over the state of the art, Helios, by 2.05% and 3.16%, respectively, while reducing MPKI by 95.31% and consuming less than half of Helios' predictor storage. These results pave the way for simpler, more practical fusion mechanisms for future processors.

Acknowledgment

We thank Ricardo Fernández-Pascual (University of Murcia) for his valuable assistance with the setup and configuration of the RISC-V toolchain and Spike simulator.

References

[1] Intel, "Intel® 64 and IA-32 Architectures Optimization Reference Manual, Pub 248966-045," pp. 3–12, 2021, [Online; accessed Apr.-2022; Section 3.4.2.2].

[2] AMD, "Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3," 2020, [Online; accessed Apr.-2022].

[3] ARM, "Arm® Cortex™-A77 Core Software Optimization Guide, Issue 3," 2019, [Online; accessed Apr.-2022; page 68, Section 4.13].

[4] ——, "Arm® Neoverse™-N2 Core Software Optimization Guide, Issue 3," 2021, [Online; accessed Apr.-2022; page 88, Section 4.13].

[5] J. D. Gelas, "The bulldozer aftermath delving-even-deeper," https://www.anandtech.com/show/5057/the-bulldozer-aftermath-delving-even-deeper, May 2012.

[6] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine, "The intel® pentium® m processor: Microarchitecture and performance." *Intel technology journal*, vol. 7, no. 2, pp. 21–36, 2003.

[7] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," https://www.agner.org/optimize/microarchitecture.pdf, Technical University of Denmark, Nov. 2022.

[8] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Pub 325462-075," 2021, [Online; accessed Apr.-2022; Section 2.2.3].

[9] AMD, "Software optimization guide for the amd Zen4 microarchitecture," Apr. 2023.

[10] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v," *arXiv preprint arXiv:1607.02318*, 2016.

[11] ARM, "Arm architecture reference manual armv8-a," 2015.

[12] S. Singh, A. Perais, A. Ros, and A. Jimborean, "Exploring instruction fusion opportunities in general purpose processors," in *55th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2022, pp. 199–212.

[13] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The alpha 21264 microprocessor architecture," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No. 98CB36273)*, Oct. 1998, pp. 90–95.

[14] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, pp. 1–23, 2006.

[15] A. Seznec, "The L-TAGE branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 9, pp. 1–13, May 2007.

[16] ——, "A new case for the TAGE branch predictor," in *44th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2011, pp. 117–127.

[17] ——, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[18] T. Koizumi, T. Maekawa, M. Mizuno, M. Kuroki, T. Tsumura, and R. Shioya, "Runlts: Register-value-aware predictor utilizing nested large tables," *The 6th Championship Branch Prediction*, 2025.

[19] S. S. Kim and A. Ros, "Effective context-sensitive memory dependence prediction," in *30th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2024, pp. 515–527.

[20] H. Thakker, T. P. Speier, R. W. Smith, K. Jaget, J. N. Dieffenderfer, M. Morrow, P. Ghoshal, Y. C. Tekmen, B. Stempel, S. H. Lee, and M. Garg, "Combining load or store instructions," Feb. 2020, patent No. 20200004550A1, Filed June 29th.,2018, Issued February 1st., 2020.

[21] S. McFarling, "Combining branch predictors," Digital Western Research Laboratory, Technical report TN-36, Jun. 1993.

[22] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction (CBP-2)*, 2011.

[23] RISC-V Software, "Spike riscv-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim, Dec. 2021.

[24] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[25] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[26] clamchowder, "Lion cove: Intel's p-core roars," https://old.chipsandcheese.com/2024/09/27/lion-cove-intels-p-core-roars/, Sep. 2024.

[27] Intel, *Intel™ 64 and IA-32 Architectures Optimization Reference Manual*, 355308th ed., Intel Corp., Aug. 2023, section 2.3.1.

[28] Cardyak, "Cardyak's microarchitecture cheat sheet," https://t.co/u17qKZbeRw, 2020.

[29] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: http://www.spec.org/cpu2017

[30] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *4th Int'l Workshop on Workload Characterization (WWC)*, Dec. 2001, pp. 3–14.

[31] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[32] A. Seznec and P. Michaud, "De-aliased hybrid branch predictors," INRIA, Research report RR-3618, 1999.

[33] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: A new dimension in branch history," in *48th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2015, pp. 347–357.

[34] A. Perais and A. Seznec, "Cost-effective physical register sharing," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 694–706.

[35] K. H. Mose, S. S. Kim, A. Ros, T. M. Jones, and R. D. Mullins, "Mascot: Predicting memory dependencies and opportunities for speculative memory bypassing," in *31st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, mar 2025, pp. 59–71.

[36] A. Perais and A. Seznec, "Storage-free memory dependency prediction," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 149–152, Jul. 2017.

[37] ——, "Practical data value speculation for future high-end processors," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, mar 2014, pp. 428–439.

[38] K. Olukotun, M. Rosenblum, and K. Wilson, "Increasing cache port efficiency for dynamic superscalar microprocessors," in *23rd Int'l Symp. on Computer Architecture (ISCA)*, 1996, pp. 147–147.

[39] J. Rivers, G. Tyson, E. Davidson, and T. Austin, "On high-bandwidth data cache design for multi-issue processors," in *30th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1997, pp. 46–56.

[40] V. Baoni, A. Mittal, and G. S. Sohi, "Fat loads: Exploiting locality amongst contemporaneous load operations to optimize cache accesses," in *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, pp. 366–379.

[41] L. Jin, H. Lee, and S. Cho, "A flexible data to l2 cache mapping approach for future multicore processors," in *2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, Oct. 2006, pp. 92–101.

[42] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store buffer," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 436–448.

[43] D. Nicolaescu, A. Veidenbaum, and A. Nicolau, "Reducing data cache energy consumption via cached load/store queue," in *2003 Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2003, pp. 252–257.

[44] A. Ros and S. Kaxiras, "Non-speculative store coalescing in total store order," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 221–234.

[45] C. Kim, D. Burger, and S. W. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99–107, Nov. 2003.

[46] R. R. Reddy, S. Singh, A. Perais, A. Ros, and A. Jimborean, "Compiler-assisted instruction fusion," in *24th International Symposium on Code Generation and Optimization (CGO)*. Sydney (Australia): IEEE Computer Society, Feb. 2026, pp. 726–739. [Online]. Available: http://webs.um.es/aros/papers/pdfs/rrreddy-cgo26.pdf