Exploring Instruction Fusion Opportunities in General Purpose Processors

Sawan Singh, Arthur Perais, Alexandra Jimborean, Alberto Ros

Exploring Instruction Fusion Opportunities in General Purpose Processors

Sawan Singh*, Arthur Perais†, Alexandra Jimborean*, Alberto Ros*

*Computer Engineering Department, University of Murcia, Murcia, Spain
†Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, Grenoble, France
Email: singh.sawan@um.es, Arthur.Perais@univ-grenoble-alpes.fr, alexandra.jimborean@um.es, aros@ditec.um.es

***Abstract*—The Complex Instruction Set Computer (CISC) paradigm has led to the introduction of instruction *cracking* in which an architectural instruction is divided into multiple microarchitectural instructions (µ-ops). However, the dual concept, instruction *fusion* is also prevalent in modern microarchitectures to maximize resource utilization. In essence, some architectural instructions are too complex to be executed as a unit, so they should be *cracked*, while others are too simple to waste resources on executing them as a unit, so they should be *fused* with others.**

In this paper, we focus on instruction *fusion* and explore opportunities for fusing additional instructions in a high-performance general purpose pipeline. We show that enabling fusion for common RISC-V idioms improves performance by 7%. Then, we determine experimentally that enabling fusion only for memory instructions achieves 86% of the potential of fusion in this particular case. Finally, we propose the *Helios* microarchitecture, able to fuse non-consecutive and non-contiguous memory instructions, and discuss microarchitectural changes required to do so efficiently while preserving correctness. Helios allows to fuse an additional 5.5% of dynamic instructions, yielding a 14.2% performance uplift over no fusion (8.2% over baseline fusion).

***Keywords*-general purpose, microarchitecture, instruction fusion**

I. Introduction

Instruction *fusion* is a well-known microarchitectural technique used in many commercially available processors [1], [3], [4], [12]. Fusion is used to better exploit available hardware resources by leveraging the fact that instructions do not always require all the resources the internal instruction format allows them to claim (e.g. physical destination registers).

A first example of fusion is architectural fusion. For instance, the *load pair* architectural instruction of Armv8 loads a chunk of contiguous memory data and writes half of it in its first destination register and the other half in its second destination register ([2], Section C3.2.3). Other ISAs such as x86 or RISC-V do not feature this instruction and must rely on regular *load* instructions (*load "single"*). In this case, performing the work done by a single Armv8 *load pair* requires two distinct architectural instructions who each occupies space in the binary and resources in the pipeline. Armv8 is therefore advantaged in this case. Generally, some ISAs already provide "fused" architectural instructions for specific operations when others rely on multiple architectural instructions to implement the operation. A typical example is loading or storing using indirect addressing, which is two instructions in RISC-V but a single one in Armv8 and x86.

Unfortunately, it is not always trivial or even desirable to add "fused" architectural instructions. Notwithstanding the fact that it introduces redundancy (e.g. two *load "single"* vs. one *load pair* achieve the same goal), it also implies that all compliant designs must support it, increasing design and validation efforts. Therefore, one could rather keep architectural instructions simple and perform any fusion necessary to optimize pipeline resource utilization at the microarchitectural level. This is the guideline followed –to the extreme– by the RISC-V ISA [7].

Moreover, microarchitectural fusion allows for more aggressive optimizations. For instance, *load pair* in Armv8 requires that the data be exactly contiguous in memory. However, one could imagine fusing two non-contiguous memory accesses as long as they fall within a certain memory region, e.g., a cacheline, or even fuse non-consecutive and/or asymmetric (different width) memory accesses [17], [29]. Consequently, this work focuses on microarchitectural fusion.

More generally, after retrieving instructions from instruction memory, many modern general purpose microarchitectures will translate architectural instructions into one or more microarchitectural operations –a.k.a. "µ-ops"– through a hardware process called *instruction cracking*. After *cracking*, all operations in flight in the pipeline are µ-ops. *Cracking* therefore re-arranges one complex architectural instruction into multiple µ-ops that are simple enough for the hardware to handle efficiently, while its dual, instruction *fusion*, re-arranges multiple µ-ops into one that is just complex enough for the hardware to handle efficiently. Fusion has the potential to improve performance by decreasing latency as well as saving pipeline resources such as Reorder Buffer (ROB), Scheduler (a.k.a. Instruction Queue, or IQ) and Load/Store Queue (LQ/SQ) entries.

This work places itself in the context of the RISC-V ISA to highlight the benefits of fusion for an ISA whose primary feature is simplicity. Indeed, many RISC-V architectural instructions do not express enough work given modern hardware capabilities, therefore, fusion is a solution to achieve higher "work per µ-op" [7]. However, the proposed microarchitectural techniques are by no means limited to the

RISC-V ISA. Specifically, this work makes the following contributions:

- Provide an overview of the different categories of fusion and highlight their limitations (Section II).
- Characterize opportunities for non consecutive and non contiguous memory fusion and highlight that focusing on memory µ-ops provides the best return on investment for the processor model we consider (Section III).
- Propose the *Helios* microarchitecture and thoroughly discuss the challenges for providing a correct and efficient execution in the presence of non-consecutive fusion (Section IV).

Our experiments with the *Helios* microarchitecture show that aggressive microarchitectural memory fusion improves performance by 14.2% over no fusion and 8.2% over consecutive and contiguous only microarchitectural memory fusion, on average (geomean).

II. Background

A. Definitions and Taxonomy

To the best of our knowledge, commercially available fusion proposals and implementations focus on *consecutive* and *contiguous* fusion [1], [3], [4], [12]. *ConSecutive Fusion (CSF)* is the operation of fusing two (or more) µ-ops that are consecutive in the dynamic execution stream of the program. *ConTiguous Fusion (CTF)* is the operation of fusing two (or more) memory µ-ops that are guaranteed to access contiguous but non-overlapping memory bytes [7].

In this paper, we propose and study techniques to increase the number of fused memory instructions, notably *non-consecutive* and *non-contiguous* fusion. *Non-ConSecutive Fusion (NCSF)* is the operation of fusing two (or more) µ-ops that are *not* consecutive in the dynamic execution stream of the program. *Non-ConTiguous Fusion (NCTF)* is the operation of fusing two (or more) memory µ-ops that access *non*-contiguous memory bytes.

In addition to the different fusion categories, this paper borrows from nuclear fusion taxonomy to differentiate between a fused µ-op and a simple µ-op that is fused with another one to create a fused µ-op. The *head nucleus* is the oldest µ-op (in program order) used to create a fused µ-op. The *tail nucleus* is the youngest µ-op used to create a fused µ-op. As we consider only 2-µop fusion in this paper, a fused µ-op is therefore always created from the *head nucleus* and the *tail nucleus*. In the context of non-consecutive fusion, we refer to the µ-ops that are "in between" (in program order) the *head nucleus* and the *tail nucleus* as the *catalyst*. This taxonomy is exemplified for a non-consecutive but contiguous *load pair* fused µ-op (*ldp*) in Figure 1. In this case, the *head* and *tail nucleii* each access 8 bytes from $[x2]$ and $[x2+8]$ respectively, and the *tail nucleus* does not depend on the *head nucleus*. Hence, they can be fused in a single *load pair* µ-op that accesses 16 bytes from $[x2]$. Note that in this particular case, the *catalyst* facilitates the fusion process since there exists no register or memory dependencies between the *nucleii* and the *catalyst*. However, as we will show in Section IV-B, this is not always the case, which is likely why Kim and Lipasti [17] considered fusing non consecutive memory accesses but only for contiguous memory accesses and only with specific *catalysts*.

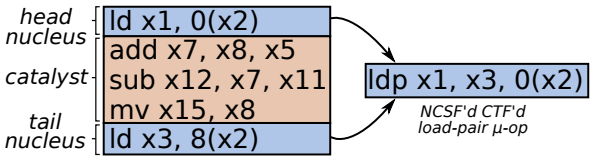

Figure 1. Non-consecutive fusion of the *head nucleus* and *tail nucleus* over the *catalyst* region.

B. Baseline Microarchitecture for Fusion

Implementing µ-op fusion requires feeding the relevant decoded fields of pairs of instructions to combinatorial logic. In most cases, the opcode and architectural source and destination registers are sufficient. For instance, given the example of Figure 1, fusing the two *ld* instructions can be determined through the following formula:

$$\begin{aligned} fuse(op_0, op_1) = (op_0 == ld) \wedge (op_1 == ld) \wedge \\ (breg_0 == breg_1) \wedge (mem\_size_0 == mem\_size_1) \wedge \\ (|imm_0 - imm_1| == mem\_size_0) \end{aligned}$$

The substitution of regular µ-ops by their fused equivalent, and any instruction collapsing within the decode group caused by the disappearance of one or more µ-ops has to take place before Rename. In effect, fusion is the replacement of the older µ-op (*head nucleus*) by a fused µ-op and the disappearance of the younger µ-op (*tail nucleus*) from the pipeline. This process takes place within a fusion *window*, which can for instance be a decode group in a superscalar processor. In this case, it is not possible for two back-to-back µ-ops that are not in the same decode group to be fused. To do so, a queue may be added between Decode and Rename.

Two of the proposed fusion idioms for RISC-V (although not limited to RISC-V) are *load pair* and *store pair*, where two *consecutive* loads (resp. stores) to contiguous memory are fused into a single *load pair* (resp. *store pair*) µ-op [7]. Assuming the microarchitecture can handle µ-ops with two destination registers in the load execution pipeline, a *load pair* µ-op can then perform a single cache access to retrieve two registers worth of data, which reduces latency. Interestingly, at the microarchitectural level, we can even fuse non-contiguous memory operations if the cache circuit allows it. That is, two loads may be fused even if they access non-contiguous data as long as that data fits within a specific memory region (e.g., cacheline).

Cacheline Crossers: A key limitation of baseline fusion is that it can only inspect static information. As a result, while a load pair idiom can easily be identified, it cannot be guaranteed that both accesses fall within the same cache line. Indeed, consider *ld x4, 0(x1)* immediately followed by *ld x5, 8(x1)*. These two instructions form a load pair idiom, however, there is no guarantee that a single cacheline can provide all the requested data. As a result, one has to consider that the fusion of memory µ-ops may not necessarily improve latency, as one *load pair* µ-op may translate to two serialized cache accesses.[1] Fortunately, hardware to handle this case is already present in modern pipelines since even a single memory µ-op may access two consecutive cache lines. The penalty of crossing a cache line is generally small in modern microarchitectures, e.g., a single cycle in Amd processors ([1], Section 2.6.2), suggesting two serialized accesses to the cache).

This further entails that for *load pair* fusion to behave optimally, the two destination registers should be provided to dependents independently.

Dependent loads: Consider instruction *ld x1, 0(x1)* immediately followed by *ld x5, 0(x1)*. At first glance, the two loads appear to qualify for fusion since they use the same base register. However, the second load actually depends on the first through *x*1. Thus, the two loads cannot compute their effective address and access the cache concurrently and they cannot be fused.

Store-to-load forwarding (STLDF): This technique is implemented in modern microarchitectures to i) allow a load to obtain data produced by a store that is still in flight and ii) detect when a younger load was issued before an older store to the same address. STLDF already has to handle accesses that cross cachelines or even pages. This work assumes that all LQ/SQ entries contain the address of the first byte they access as well as a *max_access_size* bit-vector informing which bytes are being accessed. Given this design, finding a match requires subtracting the two base addresses, shifting one of the bit-vector by the relevant amount, and finally OR'ing and AND'ing the bit-vectors to determine overlap and full match respectively. We note that several LQ/SQ designs are possible, some of which would remove the need for shifting at the cost of larger bitvectors. However, the particular details of the LQ/SQ design are out of the scope of this paper.

III. Motivation

Exploring additional microarchitectural fusion opportunities is motivated by several observations. First, the rise of the RISC-V ISA which features very simple, and therefore very *fuseable* instructions. Second, the fact that architectural fusion (e.g. the load pair instruction in Armv8 [2]) is limited in what it can fuse. Third, the fact that currently implemented microarchitectural fusion remains, to the best of our knowledge, conservative, by only considering consecutive µ-ops.

[1]This is also the case for "architectural" fusion when the compiler transforms two regular Armv8 *ldr* instructions into one Armv8 *ldp* instruction.

Table I
Several RISC-V fusion idioms envisioned in [7].

add rd, rs1, rs2 *ld rd, 0(rd)*	*lui rd, imm[31:12]* *addi rd, rd, imm[11:0]*
ld rd, imm(rs1) *add rs1, rs1, 8*	*auipc t, imm20* *jalr ra, imm12(t)*
slli rd, rs1, {1,2,3} *add rd, rd, rs2*	*mulh[[S]U] rdh, rs1, rs2* *mul rdl, rs1, rs2*
slli rd, rs1, 32 *srli rd, rd, 29/30/31/32*	*div[U] rdq, rs1, rs2* *rem[U] rdr, rs1, rs*
lui rd, imm[31:12] *ld rd, imm[11:0](rd)*	*auipc rd, symbol[31:12]* *ld rd, symbol[11:0](rd)*
ld rd1, imm(rs1) ***ld rd2, imm+8(rs1)***	***st rs2, imm(rs1)*** ***st rs3, imm+8(rs1)***

A. The Case of RISC-V

RISC-V favors simple and uniform instructions that adhere strictly to the RISC "Two Sources One Destination" (2S1D) paradigm [30]. This philosophy explains the absence of indirect addressing in RISC-V, even for loads. Other typical missing idioms include pre- and post-increment addressing which are natively present in Armv8 ([2], Section C1.3.3) but require multiple instructions in RISC-V. As a result, RISC-V is an interesting vessel to revisit instruction fusion since it is bound to have noticeable impact on the performance of a RISC-V microarchitecture.

B. A focus on Memory Pairing Fusion Idioms

The fusion idioms introduced by Celio et al. [7] for RISC-V are summarized in Table I. One observation we make in this work is that memory pairing idioms (in bold in Table I) are more common than other idioms, and furthermore, they also provide larger performance benefits as they not only reduce IQ/ROB pressure but also LQ and SQ pressure.

This first observation is illustrated in Figure 2. The Figure shows the percentage of fused pairs divided into *Memory*, i.e., those pairs in **bold** in Table I and *Others*, i.e., the other pairs in Table I, for the applications evaluated in this work (see Section V-A for details). On average 5.6% of the dynamic µ-ops belong to the *Memory* category while 1.1% belong to *Others*. Exceptions are *657.xz_2*, *bitcount*, and *susan* where non memory fusion is prevalent.

Figure 3 reports the corresponding IPC normalized to a baseline without fusion. We can observe that differences between fusing all µ-ops and just fusing memory µ-ops are minimal (1 percentage point on average). Indeed, only *susan* shows a significant performance degradation (6.5 percentage points) when only memory pairs are considered compared to all idioms. Our initial findings motivate the need to explore additional fusion opportunities of memory µ-ops.

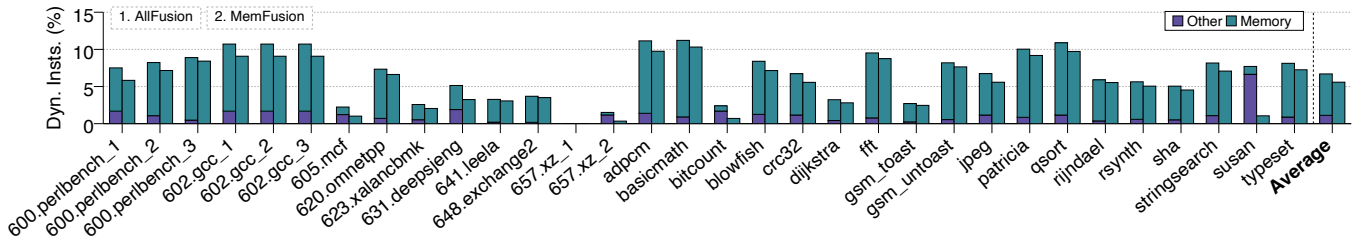

Figure 2. Percentage of fused µ-ops considering all or just memory fusion idioms, relative to total dynamic µ-ops.

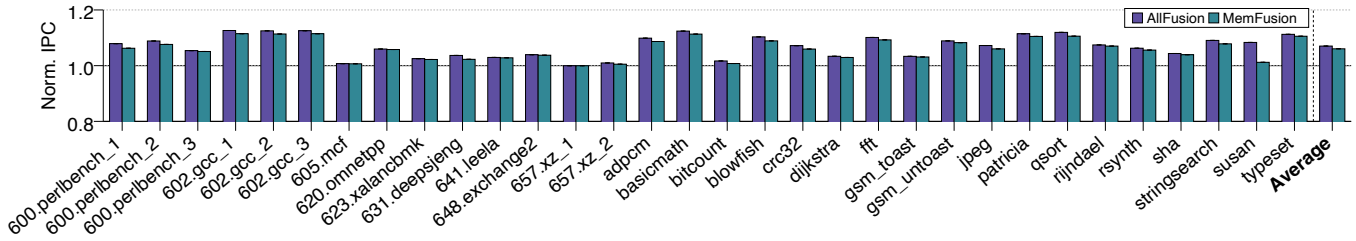

Figure 3. Normalized IPC improvement of fusion pairs presented in Figure 2 with respect to no fusion as baseline.

C. Architectural vs. Microarchitectural Fusion of Memory Operations

Armv8 features the load pair (ldp) and store pair (stp) architectural instructions ([2], Section C3.2.3). One could therefore argue that those instructions should be added to RISC-V (or x86) as extensions to let the compiler perform fusion and simplify the microarchitecture. However, architectural pairing is limited in that i) it requires both accesses to be exactly contiguous in memory and ii) it requires both accesses to have the same size. Hence, architectural fusion misses on accesses that have overlapping bytes, are asymmetric, or are close accesses but do not have overlapping bytes.

Conversely, at the microarchitectural level, the cache circuit access granularity may be greater than 8B and may be as high as the whole cache line (e.g., to support full width AVX512 [13]). Performing wider than strictly necessary accesses is a known technique to improve cache bandwidth [6], [14], [31], but previous work has so far considered it to avoid accessing the data cache multiple times, rather than to reduce pressure on pipeline structures.

Thus, memory µ-ops may be fused in the microarchitecture as long as they access data that falls within a *cache access granularity* region. This requirement is satisfied by accesses with overlapping bytes, asymmetric accesses, and accesses that do not have overlapping bytes but are close enough.

Figure 4 reports the contribution of different consecutive memory fusion categories: contiguous, overlapping, same cacheline and two contiguous cachelines, assuming the cache access granularity is 64 bytes. Interestingly, very few pairs access overlapping bytes, hinting that architectural fusion would capture most consecutive and contiguous load pair idioms, at the cost of increasing ISA complexity. Nonetheless, architectural fusion would still leave potential on the table since 1% additional memory µ-ops could fuse with the adjacent memory µ-op if non-contiguous fusion within a 64B region were supported (SameLine + NextLine).

D. Limitations of Microarchitectural Fusion

Consecutivity: Fusion is usually thought of as a technique that will fuse two (or more) µ-ops that are consecutive in the dynamic instruction stream. However, we find that if this constraint were to be relaxed, a non-negligible amount of additional µ-ops could be fused. Figure 5 reports the additional fusion potential brought by non-consecutive fusion (NCSF) for memory µ-ops. Note that in NCSF, the number of asymetric accesses is quite high, at 12.1% of the NCSF pairs. Conversely, the vast majority of CSF pairs are symmetric accesses.

Static Information Only: Fusion hardware relies on static information to decide whether to fuse two memory µ-ops or not. In the context of RISC-V, memory µ-ops may be fused if : i) They are either all loads, or all stores and ii) They share the same *physical* base register and iii) They access data that resides within a *cacheline_size* region.

However, not all fuseable µ-op pairs meeting condition iii) also validate condition ii). Indeed, we find there exist pairs of both consecutive and non-consecutive memory µ-ops that do not share a *physical* base register and yet access the same cacheline sized memory region, meaning that they could be fused. Yet, it is not easy or even possible to identify those pairs using only static information, as effective addresses are needed to confirm fuseability. Therefore, fusion based on static information is leaving potential on the table. Figure 5 also reports the fusion potential brought by fusing instructions that do not share the same physical base register (DBR suffix in the Figure), and amount to 1.5% of dynamic µ-ops on average, including CSF and NCSF.

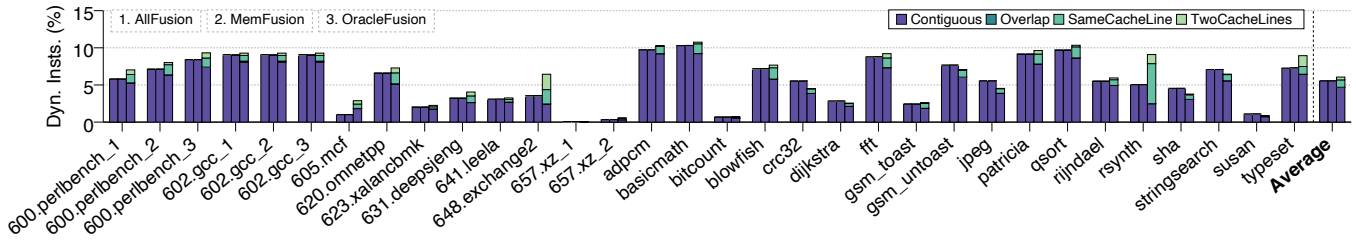

Figure 4. Paired consecutive memory µ-ops relative to total dynamic µ-ops.

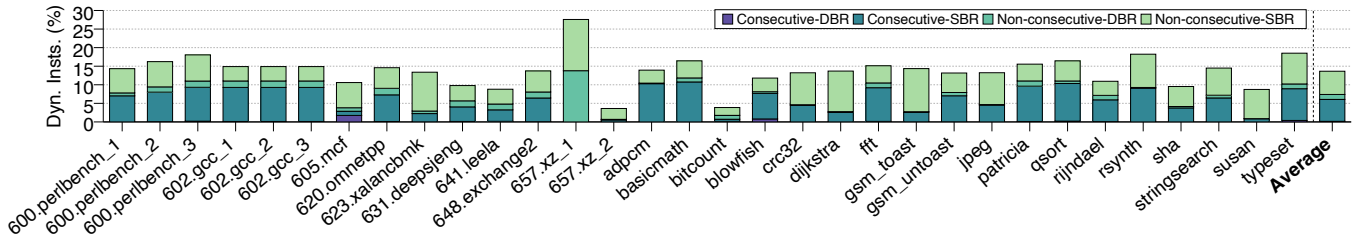

Figure 5. Paired memory µ-ops with consecutive, non-consecutive, and different-base register relative to total dynamic µ-ops.

IV. The Helios Microarchitecture

To support non-consecutive (NCS) and non-contiguous (NCT) fusion of memory µ-ops that potentially use a Different Base Register (DBR), *Helios* relies on multiple changes along the pipeline, which are summarized in Figure 6.

At a high level, *Helios* relies on three key mechanisms. First, one way to perform NCSF is to have each µ-op inspect all older µ-ops in the Allocation Queue, which sits between Decode and Rename. Such an exhaustive approach to NCSF is costly. Therefore, *Helios* uses a predictive approach to identify not only NCS but also NCT and DBR candidates. Second, NCS fusion may suffer from the presence of dependencies between *nucleii* and their *catalyst*. *Helios* identifies dependencies and, when possible, addresses them so that fusion can still proceed. Third and last, *Helios* handles incorrectly fused instructions as well as other mispredictions (e.g., branch) within a *catalyst*.

A. A Unified Predictor

Helios implements a unified hardware predictor for NCSF, NCTF, and DBR that, given a µ-op PC,[2] provides the *distance*, in µ-ops, to the *head nucleus* to fuse with.

The predictor infrastructure consist of 2 structures: the *Unfused Committed History* (UCH) and the *Fusion Predictor* (FP). UCH lives in Commit stage. It is used to find potential fusion pairs, i.e., memory operations that access the same cache line, to train FP. FP is placed in the frontend (Decode), and predicts which µ-ops should be speculatively fused.

[2]In this paper, RISC-V memory instructions always translate to a single µ-op, hence the PC to µ-op equivalence for memory µ-ops.

1) Unfused Committed History (UCH): UCH keeps the cache lines accessed by the last committed memory µ-ops eligible for fusion, i.e., the non-already fused memory µ-ops. It is organized as a cache where each entry contains a valid bit, a 32-bit tag (partial cache line address), and a 7-bit commit number (CN), for a total of 5 bytes per entry. Each entry may also feature replacement information depending on the actual design (LRU is done through the CN in this work). Distinct histories are implemented for loads and for stores.

For loads, the UCH is organized as a fully-associative cache. In our experiments, we find that the *head nucleus* can generally be found a few loads ahead, and therefore we implement a 6-entry UCH for loads with LRU replacement. For stores, a single entry holding the last unfused committed store is kept in the UCH as in Helios, stores cannot be fused across other stores to prevent memory consistency issues. At Commit, loads search the UCH for loads (Ld-UCH) and stores search the UCH for stores (St-UCH). Overall, the UCH structure requires just 280 bits.

When a retiring µ-op matches a tag in the UCH, a potential fuseable pair has been found. The distance between the two µ-ops is computed by subtracting the CN of the committing µ-op to the matching entry's CN, and the matching entry is invalidated, as µ-ops can only fuse with a single other µ-op. The maximum distance that we allow for fusion is 64 µ-ops, so the CN field requires 7 bits (the last bit controls the CN overflow). The FP is then updated using the computed distance as explained in the next Section.

On a miss in the UCH, the µ-op is inserted into the UCH. Invalid entries resulting from a previous match are preferred victims for replacement, followed by the LRU entry. It should

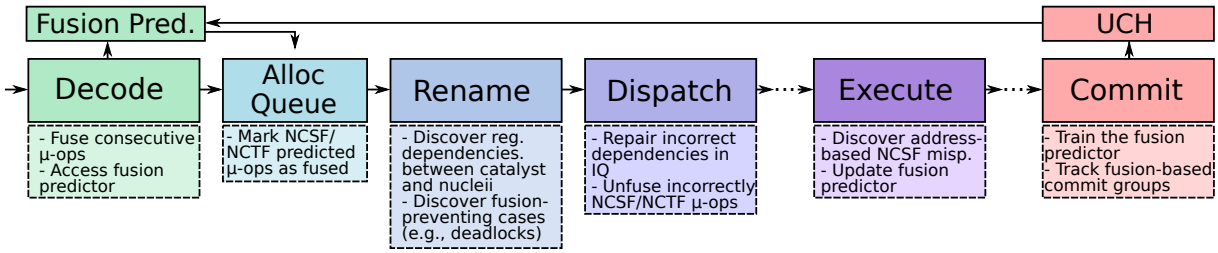

Figure 6. Overview of fusion-related responsibilities in Helios.

be noted that this process is out of the execution critical path and can be done post-commit. That is, a post-commit decoupling queue in which at most n committing loads (resp. store) µ-ops are inserted each cycle can be implemented. If the queue is full, µ-ops are simply dropped and will get a chance to train at a later time. The queue is drained at a rate of m µ-op each cycle, with m the number of UCH ports. In our experiments, on average 0.23 loads update and 0.28 loads search the UCH per cycle at commit (0.13 and 0.16 per cycle for stores). Experiments further suggest that implementing an 8-entry queue in front of the load UCH and allowing a single UCH search and update per cycle has no impact on the performance of Helios.

2) Fusion Predictor (FP): FP contains information about potential *tail nucleii*. FP is organized similarly to a cache, each entry containing an 8-bit tag, a 6-bit µ-op distance to the head *nucleus* to fuse with, a 2-bit saturating counter, and a pseudo-LRU bit. Each entry therefore requires 17 bits.

The processor attempts to allocate an FP entry for a committing µ-op when a match is found with an older UCH entry. If a match is found, FP is searched and if the tag is already present in FP and the distance matches, the confidence counter of the entry is increased. If the distance does not match, the new distance is inserted and the confidence is set to 1. If the tag is not found, an entry is selected for eviction following a pseudo-LRU replacement policy.

In this work, we chose a tournament predictor [15], which selects from a "local" PC-based predictor and a "global" *gshare*-like predictor, to implement FP. It includes a 512-set, 4-way structure indexed by the PC and another 512-set, 4-way structure indexed by a XOR of the PC and the global branch direction history. Each structure therefore features 2048 entries, amounting to 34Kbits each. A 2048-set direct-mapped and untagged selection table containing 2-bit saturating counters (4Kbits) is used to select which prediction is used. The total predictor bitcount is therefore 72Kbits (9KB). Alternatively, other predictors, such as TAGE-based [27] or local history based [32], can be employed. In the context of RISC-V, which features aligned instructions, the predictor structures may be implemented as multiple single-ported banks interleaved on PC. In practice, a number of banks greater than the decode width is preferable to handle cases where µ-ops belonging to different basic blocks are at Decode. A high number of banks also permits to perform both predictions and updates in the same cycle if they go to different banks, as described by Seznec et al. [26].

Once a *distance* is retrieved from the FP at Decode, fusion is attempted in the Allocation Queue, and is successful only if the following conditions are met:

1) The saturating counter has the maximum value (3).
2) The two µ-ops form a valid fusion idiom, that is:
   - Both µ-ops are loads or both are stores.
   - The *head nucleus* is not already a fused µ-op.
3) The *head nucleus* still resides in the Allocation Queue or is in the same Decode Group as the *tail nucleus*.

The 2-bit saturating counter is updated when the fused µ-op executes by computing the target addresses, and a misprediction is uncovered. On a correct prediction, the entry is not updated since the confidence counter has already saturated from the UCH-based training process. Updates are achieved through a dedicated structure that contains relevant prediction information (e.g., index of tables used for prediction, predicted distance, confidence) for µ-ops that flow down the pipeline, similarly to how branch or value prediction update may be handled [24]. While its exact size depends on implementation details (e.g. how many entries are sufficient to prevent stalling), each entry requires 29-bit of storage given the predictor we consider (assuming selector and PC-based set indexes can be regenerated from the PC at update time). In our experiments, we consider an unlimited queue. The confidence counter is reset to 0 on a fusion misprediction.

FP can be integrated in a microarchitecture featuring a µ-op cache by having FP and the µ-uop cache searched in parallel. Further integration of FP in the µ-op cache appears wasteful because not all µ-ops are eligible for non-consecutive fusion. However, directly caching consecutively fused µ-ops in µ-op cache entries is a possibility, as long as consecutively fused µ-ops contain enough information to be unfused at the output of the cache if a branch jumps to the *tail-nucleus*. Caching NCSF'd µ-ops appears less synergistic because NCS fusion is inherently dynamic. For instance, depending on control flow, a load may fuse with younger load A or younger load B (e.g. if A is on the taken path and B is on the fallthrough of the same conditional branch). Statically caching one of the two possible NCSF'd µ-ops in the µ-op cache would be

unable to capture this behavior. It may however be adapted to constrained NCS fusion schemes that do not allow any control-flow change within the *catalyst*.

B. Preserving ISA Semantics

Helios builds on consecutive memory fusion. Thus, µ-ops already feature three source and two destination registers, as this is required even for consecutive and contiguous fusion of memory operations [7]. To support DBR, *store pair* µ-ops may actually need four source registers. Fortunately, we find that DBR *store pair* fusion represents a negligible fraction of the fused stores (0.54%), as a result, we only support SBR *store pair* fusion. In the remainder of this paper, the reported storage cost are compared to a baseline with consecutive and contiguous (for memory µ-ops) fusion.

To correctly fuse non-consecutive µ-ops, NCSF does not remove the *tail nucleus* from the Allocation Queue (AQ). At fusion time, the *head nucleus* is replaced by the NCSF'd µ-op, and the *tail nucleus* is left in the queue. This contrasts with consecutive fusion where the *tail nucleus* can disappear after fusion. Furthermore, we introduce the following definitions:

- A *validated* NCSF'd µ-op is known to i) Possess its correct source physical register identifiers, ii) Not produce a deadlock through a register dependency or a serializing instruction and iii) Not have a store in its *catalyst* if it is a *store pair* µ-op.
- A *pending* NCSF'd µ-op is an NCSF'd µ-op that is not yet *validated*.

1) Allocation Queue: To track which µ-ops are *head* or *tail nucleii*, each AQ entry is augmented with the *Is_Head_Nucleus* and *Is_Tail_ Nucleus* bits. Each entry is also augmented with a tag field, *NCS_Tag*. The tag is actually a pointer to the other nucleus µ-op in the AQ, which is 140 entries in our model, yielding an 8-bit tag. The tags are managed implicitly as µ-ops enter and leave the AQ. *head nucleii* carry their own AQ entry number until they are dispatched to the ROB/IQ/LQ/SQ, while *tail nucleii* leaving the AQ carry their respective *NCS_Tag* until they are dispatched. Those changes to the AQ are depicted in ❶ of Figure 7 and amount to 1.37Kbits of storage in the AQ.

2) Register Renaming & NCSF: For generality, we describe a scheme that can handle nested[3] NCSF. We first augment the Rename stage with two counters: *Max_Active_NCS* and *Active_NCS*. The first tracks the number of *head nucleii* of the current NCSF nest that have entered Rename so far. The second is used to determine when Rename finishes processing the NCSF nest. Both counters are incremented when a *head nucleus* enters Rename. Only *Active_NCS* can be decremented, when a *tail nucleus* leaves Rename. However, when *Active_NCS* is decremented to 0, Rename is not processing an NCSF nest anymore, and *Max_Active_NCS* is reset. Both counters are also reset on a pipeline flush. We found that supporting only two nested NCSF'd µ-ops at any given time is sufficient to achieve most of the benefits. Any *head nucleus* entering Rename while *Max_Active_NCS* is saturated behaves as unfused and the *tail nucleus* is marked as not fused in the AQ through the *NCS_Tag*. The additions to the Rename stage are depicted in ❷ of Figure 7 and amount to 4 bits of storage. Moreover, one bit is added to each source and destination physical register identifier flowing in the pipeline to inform whether that physical register belongs to the *head nucleus* or to the *tail nucleus*, as shown in ❸ of Figure 7. This amounts to 700 bits in the AQ, 800 bits in the IQ and 256 bits in the LQ.

Nevertheless, non-consecutive fusion is problematic if there exists register dependencies between i) The *catalyst* and the *tail nucleus* and ii) The *head* and *tail nucleii*, either direct or indirect. To illustrate the first class of dependencies, consider the following example where µ-op `3` is fused with µ-op `1`:

```
[1] ld x1, 0(x2)
[2] add x2, x4, 1
[3] ld x4, 8(x2)
```

Instruction `3` has a Write-after-Read (WaR) dependency with `2` through `x4` as well as a Read-after-Write (RaW) dependency through `x2`. Both dependencies can be circumvented through register renaming.

Write-after-Read (`x4`): All the destination registers of the fused µ-op are renamed together, when the fused µ-op (replacing `1`) enters rename. This is incorrect because it means that `2` will be able to see the version of `x4` produced by `3` in the Register Alias Table (RAT) and use it as a source.

To prevent `2` from observing the new name of `x4`, *Helios* prevents the NCSF'd µ-op from updating the RAT for the destination register(s) of the *tail nucleus*. Rather, those physical registers are stored in a dedicated buffer. When the corresponding *tail nucleus* goes through Rename, destination renaming is performed by reading the physical register identifier(s) from the buffer to update the RAT. In this work, a 2-entry buffer (one per NCSF nesting level) is sufficient, each entry storing a physical register identifier. This buffer is tagged using the entry number of the *head nucleus*, which is the *NCS_Tag* of the *tail nucleus*. It is shown in ❹ of Figure 7 and amount to 34 bits of storage (additional information is needed to handle deadlocks and discussed later). This scheme enforces in-order destination register renaming, allowing the Active List (which contains the in-flight register mappings) to remain consistent with program order. The buffer is entirely invalidated and physical registers names in valid entries are put back on the Free List on a pipeline flush.

Read-after-Write (`x2`): When the NCSF'd µ-op renames its sources, `2` has neither been allocated a new physical register for `x2` nor has it updated the RAT. Consequently, the fused µ-op will incorrectly rename the source operand of `3`.

[3]In this paper, "nested" includes interleaved pairs, e.g., µ-ops ld_0 ld_1 ld_2 ld_3 where ld_0 fuses with ld_2 and ld_1 fuses with ld_3.

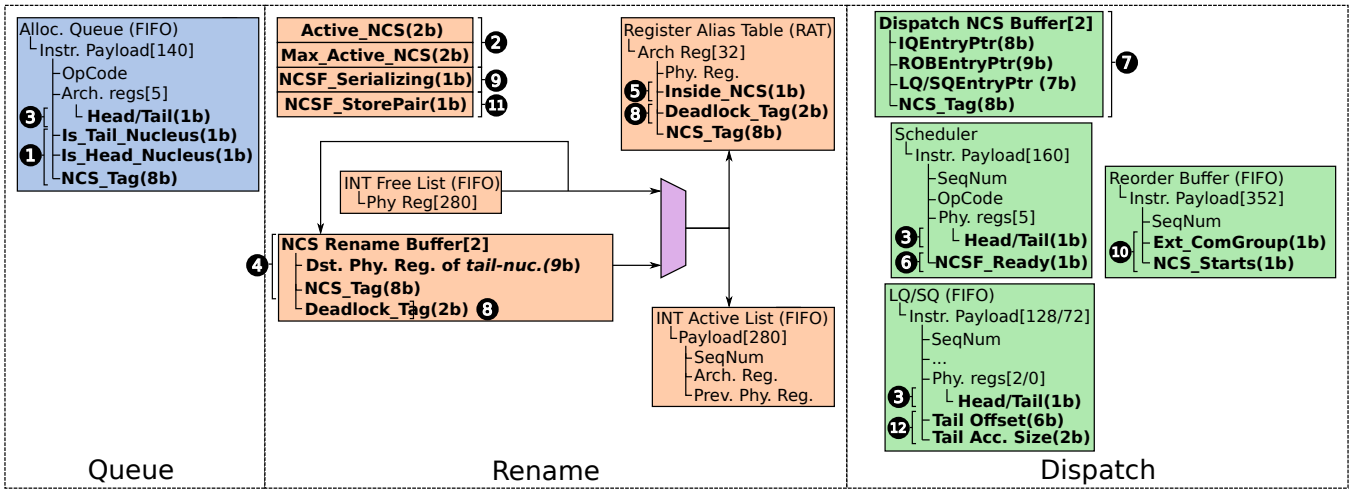

Figure 7. Key pipeline structure changes (bold) in AQ (blue), Rename (orange) and Dispatch (green). Pipeline structures are sized similarly to Intel Icelake.

When the existence of a RaW dependency is eventually determined, it may not be trivial to re-execute the NCSF'd µ-op as it may already have left the IQ cycles ago. To alleviate this concern, *Helios* prevents *pending* NCSF'd µ-op from issuing until the existence or absence of a RaW dependency is determined. This allows the correct source register name to be provided to the IQ entry of the NCSF'd µ-op as it becomes *validated*.

To that extent, each IQ entry is first augmented with a *NCS_Ready* bit, which encodes whether an NCSF'd µ-op is *pending* or *validated*. A µ-op may issue only if *NCS_Ready* is set. All µ-ops in flight between Rename and the IQ also carry this bit. When an NCSF'd µ-op leaves Rename, its associated *NCS_Ready* bit is reset, while all other µ-ops leave Rename with the bit set. Those additional bits are depicted in ❻ of Figure 7, and amount to 160 bits of IQ storage.

Second, each entry of the RAT is augmented with an *Inside_NCS* bit. *Inside_NCS* is set in the RAT entry when a µ-op renames its destination register and Rename is currently processing an NCSF'd µ-op (*Active_NCS* is not 0). All *Inside_NCS* bits are reset when Rename stops processing NCSF'd µ-ops (*Active_NCS* is decremented to 0), or when the pipeline flushes. Those additional bits are depicted in ❺ of Figure 7, and amount to 32 bits of storage in the RAT. When a *tail nucleus* enters Rename, it still looks up its source registers in the RAT for the purpose of validating the NCSF'd µ-op. If one of the source has the *Inside_NCS* bit set, there is a RaW dependency between the *tail nucleus* and the *catalyst*. This design may yield false positives, however, this is inconsequential in *Helios* since absence or presence of a RaW incurs the same validation latency.

To summarize, when an NCSF'd µ-op dispatches to the IQ, it cannot issue because the *NCSF_Ready* bit is not set yet. Eventually, the corresponding *tail nucleus* leaves Rename having potentially detected a RaW dependency, hence an incorrect NCSF'd µ-op residing in the IQ.

Regardless, the *tail nucleus* flows to Dispatch to set the *NCSF_Ready* bit in the corresponding NCSF'd µ-op IQ entry, and, if needed, to correct the source register name(s). To do so, it uses a Dispatch slot to write the IQ entry. The ROB and LQ/SQ do not need to be amended.

The precise IQ entry to amend is tracked by the Dispatch stage using a small fully associative buffer that has as many entries as the supported nesting NCSF depth (2 in our case), much like the Rename buffer. Entries are tagged with the AQ entry number of the *head nucleus* (which is the *NCS_Tag* of the *tail nucleus*) and are allocated when a *pending* NCSF'd µ-op dispatches. Entries are reclaimed when the corresponding *tail nucleus* validates the *head nucleus* IQ entry, or there is a pipeline flush. The Dispatch buffer is shown in ❼ of Figure 7 and amounts to 64 bits of storage (the buffer also stores pointers to the ROB and LQ/SQ for the purpose of dynamically unfusing a *pending* NCSF'd µ-op).

Note that if the two *nucleii* are in the same rename group, we assume that the Rename stage can take care of any RaW dependency in-place, and the NCSF'd µ-op therefore leaves Rename *validated*.

Deadlocks: In Section II-B, we highlighted that standard fusion does not consider consecutive *load pair* patterns if the *tail nucleus* **directly** depends on the *head nucleus*. This also applies in the context of NCSF, although given the predictive nature of *Helios*, there is no easy way to determine that two non-consecutive µ-ops are directly dependent in the AQ. Moreover, indirect dependencies also have to be considered. Let us consider the example below where `1` and `3` are assumed to be fused despite having different architectural base registers:

```
[1] ld x1, 0(x2)
[2] add x3, x1, 1
[3] ld x4, 0(x3)
```

We cannot issue the fused *load pair* comprised of `1` and `3` until `x2` and `x3` are available. However, `x3` cannot be produced until `x1` is produced by the *load pair* µ-op, which cannot happen until `x3` is produced by `2`. In other words, the *load pair* cannot issue until it executes and execution is therefore deadlocked.

To detect if the *tail nucleus* directly or indirectly depends on the *head nucleus*, we rely on a dedicated detection mechanism that lives in the Renamer: When an NCSF'd µ-op is renamed, it writes a *Deadlock_Tag* field in the RAT entries of its destination register(s). This field is a one-hot vector with as many bits as nesting levels (2 in our case) that encodes the current value of *Maximum_Active_NCS*. The *Deadlock_Tag* is propagated from source(s) to destination(s) as younger µ-ops are renamed. If there are multiple source registers, their *Deadlock_Tag* fields are OR'd before being written in the RAT entries of the destination register(s). Similarly, a younger (i.e., nested) *head nucleus* sets its corresponding bit in the *Deadlock_Tag* of its destination register(s) but also propagates the *Deadlock_Tag* from its sources by OR'ing the two tags. All *Deadlock_Tag* bits are reset when the last *tail nucleus* of an NCSF nest leaves Rename or when the pipeline flushes. Those additional bits are illustrated in ❽ of Figure 7 and amount to 64 bits of storage in the RAT as well as 4 bits in the Rename Buffer.

If any of the source registers of a *tail nucleus* has a *Deadlock_Tag* with the relevant bit set, then there is a dependency-based deadlock and NCSF should not have taken place. The relevant bit to check can be retrieved from the 2-entry Rename buffer used to handle WaR dependencies which is tagged with the *NCS_Tag* of the *tail nucleus* (hence the 4 additional bits of storage in this buffer). Recovery is done by letting the *tail nucleus* flow through Dispatch and unfuse the corresponding NCSF'd µ-op by amending the relevant structures (pointer to ROB/IQ/LQ/SQ entries of *pending* NCSF'd µ-op are kept in the dedicated Dispatch buffer to enable this). In particular, the source and destination registers that belong to the *tail nucleus* will be marked as invalid in the IQ and LQ, and the tail access offset and size will be amended to 0 in the LQ. The *tail nucleus* further occupies a second dispatch slot to get its own entries.

Deadlocks stemming from the presence of serializing instructions (including fences) within the *catalyst* can be identified by adding a single bit, *NCSF_Serializing*, in the Rename stage which is set if *Max_Active_NCS* is at least one and a serializing instruction enters Rename. When a *tail nucleus* enters Rename and this bit is set, it performs the same actions as when a dependency-based deadlock is identified to "unfuse" the corresponding *pending* NCSF'd µ-op. This bit is depicted in ❾ of Figure 7.

3) Instruction Commit, Exception and Interrupts: To preserve in-order semantics, both *nucleii* and *catalyst* must be ready to retire for the NCSF'd µ-op to commit. This ensures that if a misprediction or an exception (also known as fault) is detected in the *catalyst*, the *head-nucleus* has not retired yet and it can be unfused or flushed, thereby guaranteeing precise exceptions. This does not mean that the *nucleii* and *catalyst* have to commit in a single cycle, however. As a consequence, if an extended commit group has started committing, the group must finish committing before any pending interrupt is processed. In our experiments, the *catalyst* size is often limited (10.5 µ-ops on average), meaning that the latency increase in processing interrupts will be minor.

This is achieved through the *Active_NCS* Rename counter. Specifically, all µ-ops that went through Rename when *Active_NCS* was not 0 are part of an "extended" commit group, which they indicate by setting the *Ext_ComGroup* bit in their ROB entry. To avoid deadlocks, a second bit is added in each ROB entry. This bit set to 1 by the first *head nucleus* of an NCSF nest, and serves to delineate "extended" commit groups. The Commit stage then leverages those bits to determine if the ROB head can retire. We note that Commit may establish the boundaries of such groups off the critical path by scanning the ROB as µ-ops dispatch. Those bits are depicted in ❿ of Figure 7 and amounts to 704 bits of storage in the ROB.

4) Memory Consistency & Sequential Semantics: In *Helios*, NCSF is speculative, therefore, it cannot guarantee that i) There is no store µ-op in the *catalyst* of a *store pair* NCSF'd µ-op and ii) If there is such a store µ-op, it cannot guarantee that is does not overlap with the *tail nucleus*. As a result an NCSF'd *store pair* µ-op risks violating store-store ordering in models enforcing it and same-address sequential-consistency for all memory models.

Helios prevents NCS *store pair* fusion when finding a store µ-op in the *catalyst*. This is achieved by adding a *NCSF_StorePair* bit to Rename, which is set when any store µ-op other than the first *head nucleus* of the NCSF nest is renamed. Any store *tail nucleus* seeing this bit set will proceed to unfuse the corresponding *pending* NCSF'd *store pair* µ-op waiting in the IQ, similarly to the deadlock case. The *NCSF_StorePair* bit is depicted in ⓫ of Figure 7.

Conversely, NCS *load pair* fusion may have loads and stores in the *catalyst* since loads already execute speculatively out-of-order with respect to other loads while respecting load order [9] and with respect to other stores while respecting sequential semantics [18].

5) Memory µ-ops with Different Base Registers: As pointed out in Section III-D, a non-negligible amount of pairs of loads access data that fit within a cacheline through a different base address register. In the context of a microarchitecture with register renaming, this can be either through a different *physical* base register or a different *architectural* base register. In the former case, the frontend can make a reasonable decision by inspecting the *nucleii* and the *catalyst*, but this requires inspecting an arbitrary number of µ-ops from the AQ, which is not desirable.

Moreover, in the other case (different architectural registers), the frontend cannot generally determine that two µ-ops are candidates for pairing with static information only. Fortunately, DBR *load pair* fusion is captured by the predictive scheme employed by *Helios*.

6) Load/Store Queue: NCSF by itself does not impact the baseline LQ/SQ designs as introduced in Section II-B. However, although that information is already encoded in the base address and the bitvector, the LQ/SQ entries may also store i) an offset from the base address for the second access, such that the base address of the second access can easily be re-generated if needed (6 bits) and ii) The access size of the second access if different access sizes are considered (2 bits). Given the processor configuration we consider, this amounts to 704 bits as we assume the LQ/SQ entries can already track 64B accesses (e.g., to support wide vector extensions such as AVX512). This storage is reported by ⓬ of Figure 7.

7) Summary: Supporting NCSF in the different stages of the pipeline requires a total of 4.77Kbits, or 0.60KB, for the processor configuration we consider (see Table II). Adding the Fusion predictor yields a grand total of 76.77Kbits (9.60KB).

C. Repairing Microarchitectural State

In Helios, several events require the microarchitectural state to be repaired:

1) *Rename:* An NCSF'd µ-op has at least one incorrect source name because of a RaW between the *catalyst* and the *tail nucleus*.
2) *Rename:* A dependency-based deadlock is discovered.
3) *Rename:* A store *tail nucleus* enters Rename and the *NCSF_StorePair* bit is set.
4) *Rename:* A *tail nucleus* enters Rename and the *NCSF_Serializing* bit is set.
5) *Execute:* The *head* and *tail nucleii* could not fuse because they span more than a *cacheline_size* region.
6) *Execute:* The memory access belonging to the *tail nucleus* faults.
7) *Execute:* A mispredicted µ-op (e.g., branch, memory dependency, fault) is discovered in the *catalyst*.

Case 1: The *pending* NCSF'd µ-op is kept in the IQ, so the *tail nucleus* can amend it in place using a dispatch slot.

Cases 2, 3 & 4: The NCSF'd µ-op is unfused in place using the same mechanism as in Case 1 and the fact that we attach one bit to each physical register identifier to inform whether the register belongs to the *head* or the *tail nucleus* (Section IV-B1). Unfusing also requires amending the NCSF'd µ-op Load Queue (resp. Store Queue) entry, which is also tracked in the dedicated Dispatch side buffer. In those cases, the *tail nucleus* also dispatches and therefore occupies two dispatch slots, preventing one younger µ-op to dispatch that cycle as an additional penalty.

Case 5, 6 & 7: In the two first cases, the NCSF'd µ-op *tail nucleus* became *validated* and the *tail nucleus* has left the pipeline after fused µ-op validation. Therefore, it needs to be refetched after flushing the pipeline. In the last case, the control flow is incorrect, hence a pipeline flush is needed. The only degree of freedom is the flush point. Indeed, we can either i) Unfuse any NCSF'd µ-op whose *catalyst* contains the mispredicted (resp. faulting) instruction and flush from the mispredicted (resp. faulting) instruction or ii) Flush from the oldest NCSF'd µ-op whose *catalyst* contains the mispredicted (resp. faulting) instruction.

Since we only consider at most two nested/interleaved NCSF'd µ-op, any mispredicted (resp. faulting) instruction may need to unfuse at most two NCSF'd µ-op (only one for cases 5 & 6 since the mispredicted/faulting µ-op is one of the two nested NCSF'd µ-ops). As a result, in this paper, we consider solution i) by having all µ-ops that can potentially trigger a pipeline flush keep two pointers to their associated "encompassing" NCSF'd µ-ops. The pointers are obtained at Rename by first grabbing the *NCS_Tag* of valid entries in the Rename buffer used to handle WaR and then using this tag to retrieve a ROB entry number or sequence number from the Dispatch buffer used to handle RaW dependencies of NCSF'd µ-ops. The pointers are then stored in a FIFO queue that stores the information needed to recover from pipeline flushes (e.g., PC). Note that since most µ-ops fuse with a µ-op that is close by, the impact of flushing more than necessary (solution ii)) will remain limited. However, even solution ii) requires µ-ops that can trigger a pipeline flush to be able to determine the older NCSF'd µ-ops whose *catalyst* they belong to (e.g., through a pointer). As pointed out in Section IV-B3, a mispredicted (resp. faulting) µ-op will always find relevant *head-nucleii* in the ROB and be able to unfuse them through the pointers, which guarantees precise exceptions and correct branch misprediction recovery.

An upper bound cost for solution i) is therefore two 9-bit ROB pointers per ROB entry,[4] amounting to 6336 bits and increasing the total cost of supporting NCS fusion in Helios to around 83Kbits (around 10.4KB).

V. Evaluation

A. Simulation Methodology

Our simulation infrastructure employs a modified version of the RISC-V Spike Simulator [21] and an in-house cycle-level simulator modeling a seven-stage pipeline as described by González et al. [10]. Spike runs in full-system mode with a Linux kernel and injects instructions into our out-of-order processor model, modeled after an Intel Icelake microarchitecture. A first insight obtained from evaluating Helios is that using such a deep machine with equally wide frontend (Fetch/Decode/Rename) stages prevents the Allocation Queue from getting filled and greatly limits fusion opportunities. As a result, Helios features 8-wide Fetch and

[4] A dedicated queue smaller than the ROB may be implemented to track this information for potential flushers only.

Table II
BASELINE CONFIGURATION.

Out-of-order processor	
Model	Intel Icelake
Predictors	L-TAGE [25], Store-set [8]
Stages	Fetch/Decode/Rename/Allocation /Issue/Execution/Memory/Commit
Frontend Stages	8-wide Fetch/Decode, 5-wide Rename
Allocation Queue	140 entries
Backend Stages	5-wide Alloc., 10x Exec. Ports, 2x loads 2x stores, 4x AGU 4x ALU, 1x DIV, 2x FP Add/Sub 1x SQRT, 20-wide Commit
ROB/IQ/LQ/SB	352/160/128/72 entries
Memory hierarchy	
L1I	32KB, 8 ways, 4-cycle hit lat., pipelined
L1D	48KB 12 ways, 4-cycle hit lat., pipelined
L1D prefetcher	Stride, degree 3
L2	256KB, 8 ways, 12 hit cycles
LLC	8MB, 8 ways, 35 hit cycles
RAM	160-cycle latency

Decode stages to ensure that the Allocation Queue gets filled even in high IPC workloads. Our model implements a Total-Store-Order consistency model, thereby being compliant with the RISC-V TSO extension (*Ztso*). The high level characteristics of the system used in our simulations are displayed in Table II.

We evaluate Helios with the SPEC CPU 2017 [28][5] and MiBench [11] benchmark suites. We skip the Linux kernel boot and setup for all the applications. Then, SPEC applications skip an additional 10B instructions and report results for the next 500M instructions. MiBench applications run until completion. SPEC workloads run using *reference* inputs while MiBench workloads use the *large* input set. The binaries were compiled with GCC 10.2.0 targeting the RV64G ISA with flags *-O3 -static*.

We consider five configurations. **RISCVFusion** fuses µ-ops using the non-bold idioms of Table I (i.e., no memory pairs), as suggested by Celio *et al.* [7]. **CSF-SBR** fuses only consecutive memory instruction that access contiguous data through the same base register, but may be asymmetric (different access sizes). **RISCVFusion++** fuses all instructions from Table I. **Helios** implements a predictor and machinery to fuse consecutive as well as non-consecutive memory pairs. Finally, **OracleFusion** is an upper bound configuration that fuses all eligible memory pairs as well as non memory pairs from Table I.

B. Results

1) Fusion pairs: Figure 8 shows the percentage of CSF and NCSF in Helios and OracleFusion with respect to total dynamic memory instructions. On average, the total number of fused pairs in Helios is close to the upper limit of OracleFusion. *Helios* delivers 6.7% CSF and 5.5% NCSF pairs. It is interesting to note that the number of CSF in *Helios* is slightly higher when compared to OracleFusion which fuses only 6.1%. The reason is that OracleFusion will immediately be able to fuse distant µ-ops the first time it encounters them. Conversely, *Helios* has a training phase during which it may fuse two consecutive µ-ops, the *head nucleus* of which is actually the *tail nucleus* of the NCSF'd µ-op identified by OracleFusion. Since CSF µ-ops are not inserted in the UCH, *Helios* will "favor" CSF over NCSF in this case. In our experiments, the distance between *head nucleus* and *tail nucleus* averages at 10.5 dynamic instructions (amean), suggesting that currently implemented decode width are not sufficient to perform "brute force" NCSF at Decode.

[5]Due to a known Spike limitation at the time of the study (https://github.com/riscv-collab/riscv-gcc/issues/175), we were unable to run application 625.x264_s in full system mode.

Table III
HELIOS FUSION PREDICTOR'S COVERAGE, ACCURACY AND MPKI FOR ALL THE APPLICATIONS USED IN THIS PAPER.

Benchmark	Coverage	Accuracy	MPKI
600.perlbench_s_1	76.56	99.96	0.0183
600.perlbench_s_2	75.30	99.90	0.0592
600.perlbench_s_3	71.99	99.97	0.0170
602.gcc_s_1	63.32	99.51	0.1776
602.gcc_s_2	62.73	99.52	0.1709
602.gcc_s_3	63.43	99.52	0.1722
605.mcf_s	62.24	98.60	0.8350
620.omnetpp_s	67.86	99.40	0.3018
623.xalancbmk_s	82.94	99.97	0.0269
631.deepsjeng_s	58.82	98.68	0.4602
641.leela_s	62.15	97.74	0.8172
648.exchange2_s	50.04	99.56	0.1595
657.xz_s_1	99.99	99.99	0.0000
657.xz_s_2	73.90	99.90	0.0207
adpcm	58.90	99.99	0.0005
basicmath	61.68	99.99	0.0013
bitcount	74.54	99.56	0.1083
blowfish	48.00	99.89	0.0254
crc32	66.49	99.99	0.0046
dijkstra	85.64	99.99	0.0058
fft	57.05	99.93	0.0210
gsm_toast	65.34	99.51	0.3765
gsm_untoast	67.89	99.99	0.0010
jpeg	72.74	99.99	0.0061
patricia	62.80	99.99	0.0036
qsort	66.97	99.77	0.0965
rijndael	62.22	99.85	0.0471
rsynth	64.23	99.99	0.0047
sha	69.22	99.99	0.0023
stringsearch	67.76	99.97	0.0115
susan	91.36	99.99	0.0010
typeset	69.26	99.17	0.5758
Average	**68.23**	**99.68**	**0.1416**

2) Helios Fusion predictor: Table III summarizes the performance of the fusion predictor. Coverage includes only the pairs that need predictions: NCSF and CSF load pairs that use different base registers.

On average, the predictor is able to correctly fuse 68.2% of the eligible dynamic memory µ-ops. The relative increase in CSF compared to OracleFusion is one of the factors for

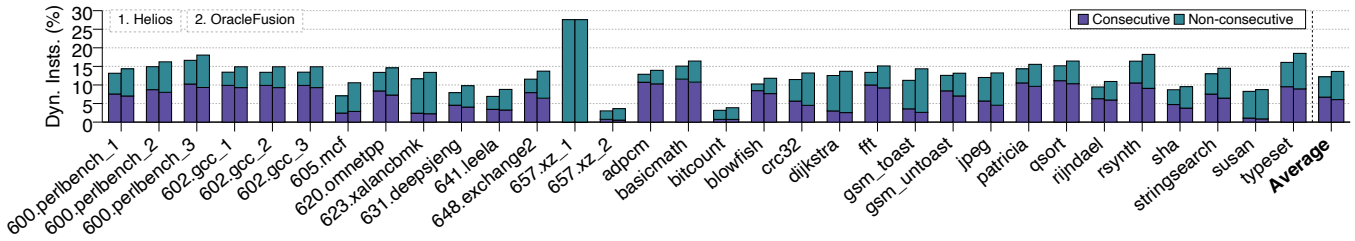

Figure 8. Number of CSF and NCSF pairs in Helios and OracleFusion.

missing coverage as it results in fewer available µ-ops for NCSF/CSF-DBR. A related reason is the delay caused by the training phase of the fusion predictor itself. Overall, *Helios* fuses 12.2% of the dynamic µ-ops, approaching the 13.6% of OracleFusion.

Prediction accuracy in a deeply pipelined processor is crucial, especially since the misprediction penalty may be higher than the expected gains of a correct fusion prediction. Through tagging and confidence estimation, the *Helios* fusion predictor provides high accuracy: 99.7% on average. *641.leela* has the lowest accuracy among all the application, at 97.7%. Note that higher accuracy may always be traded for lower coverage using better confidence estimation e.g., probabilistic counters [20]. Average mispredictions per kilo instructions (MPKI) in *Helios* stands at 0.1416.

3) Processor Stalls & IPC: Figure 9 shows the percentage of Rename and Dispatch structural stalls with respect to the total execution cycles for baseline, *Helios* and OracleFusion. Figure 10 reports the IPC for all configurations.

In applications that encounter many dispatch stalls in the baseline, fusion generally provides high IPC gains, for example applications (e.g., *600. perlbench_1 & 2*, *602.gcc*, *657.xz_1*, *rsynth*). In *657.xz_1*, Dispatch spends 88% of the execution cycles waiting for an SQ entry. Thanks to *Helios*, *657.xz_1* gets a high IPC improvement of 70% due to 27.6% additional NCSF pairs (Figure 8). Application *602.gcc* (resp. *rijndael*, *typeset*) also suffer from many SQ stalls in the baseline, a significant part of which are avoided in *Helios*, yielding an IPC improvement of 14.8% (resp. 11.94%, 20.6%).

Nonetheless, applications that do not have significant stalls in the baseline still benefit from *Helios* on the IPC front due to reduced execution latency of paired loads and the potential doubling of the load/store throughput (e.g., most MiBench workloads). One interesting case that does not follow this trend is *605.mcf*, who suffers few stalls in the baseline and features a reasonable number of fused memory pairs in *Helios*. Yet, a fusion MPKI slightly higher than average results in an overall IPC degradation of 1%.

On average Helios provide an IPC improvement of 14.2% over a baseline without fusion, and 8.2% over only fusing consecutive and contiguous memory pairs that use the same base register (CSF-SBR). *Helios* achieves most of the benefits of OracleFusion which stands at 16.3% improvement. Part of the gap could undoubtedly be bridged by tuning the fusion predictor or considering other algorithms [27]. Other configurations discussed in the paper achieve 0.8% (RISCVFusion), 6% (CSF-SBR), and 7% (RISCVFusion++) IPC improvement, respectively.

VI. Related work

Coalescing memory operations. Wilson et al. suggest to leverage the access FIFO in front of the L1D in some designs to serve multiple loads in one go by reading the whole cacheline and letting entries of the FIFO match against returning data [31]. Rivers et al. follow a similar –in spirit– path in the pursuit of efficient data cache multiporting, in which each bank of the data cache is augmented with a single highly ported line buffer from which multiple loads can be served each cycle [22]. Similarly, Baoni et al. introduce *Fat loads* [6] in which an initial *fat load* brings a full cacheline in a dedicated buffer to allow subsequent loads to that cacheline to be served from this fast buffer. *Fat loads* is a very effective technique to reduce pressure on the L1D and DTLB while also speeding up performance since buffers can be accessed faster than the L1D.

Other techniques to coalesce memory accesses include to always read double words (8 bytes) from the data cache, even for smaller accesses, and store the result in a load-load forwarding capable Load-Store Queue [14], keeping retired loads and stores alive longer than needed to create more load-store and load-load forwarding opportunities [5], [19], and coalescing non-consecutive stores after they commit while guaranteeing total store order [23].

These techniques are oblivious to consecutivity, dependencies, and whether the accesses use different base or offset registers. In addition, they do not require augmenting the µ-ops with additional source and/or destination register fields. Although they may coalesce more than two accesses into a single one, all coalesced µ-ops are still treated as distinct entities in the pipeline, preventing them from providing savings in the ROB/IQ/LQ/SQ. Conversely, microarchitectural fusion and especially Helios reduces occupancy in the pipeline structures while also coalescing two accesses into a single one. Nevertheless, Helios and such prior proposals are quasi-orthogonal in the sense that if the Helios prediction scheme

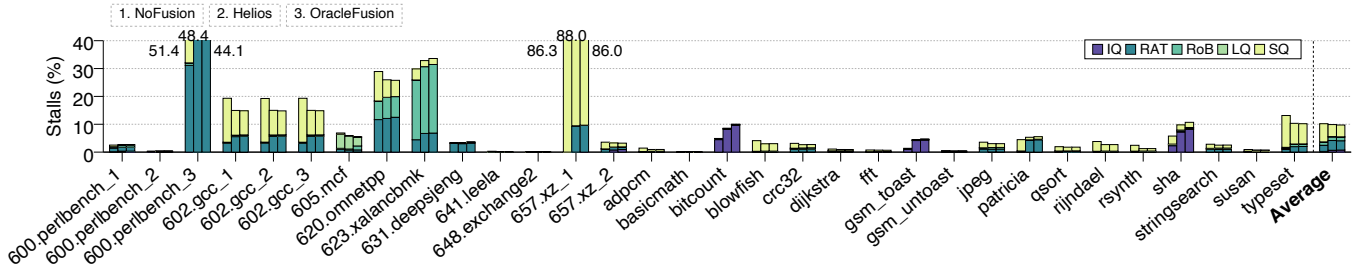

Figure 9. Percentage of processor stalls with respect to total execution cycles.

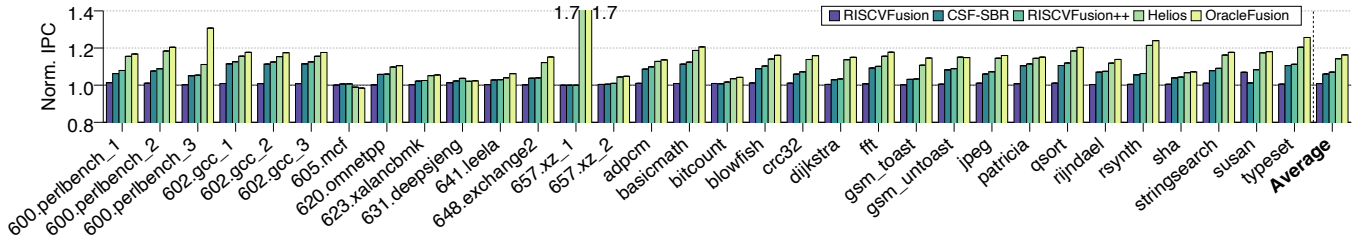

Figure 10. Normalized IPC with respect to baseline configuration with no instruction fusion.

were built to fuse only accesses that fall within the same cacheline, then fused µ-ops would be able to leverage Fat Loads line buffers [6].

Fusion. Kim and Lipasti introduce macro-op scheduling for the purpose of simplifying the IQ logic [16]. In this work, candidate pairs of µ-ops are scheduled as a unit to relax the wakeup & select loop since the IQ has to schedule half as often if it schedules "fused" pairs of µ-ops. While macro-ops occupy a single slot in the IQ, thereby saving capacity, they do not execute faster.

Celio et al. [7] make the case that fusion can be used to improve "work per µ-op" in RISC-V designs, rather than adding more powerful –yet common– instructions to the ISA, and discuss a number of potential idioms amenable to fusion. In this case, fusion is restricted to consecutive and contiguous (for memory) instructions.

The works closest to this paper are Kim and Lipasti [17] and Thakker et al. [29]. Both refer to non-consecutive fusion of memory instructions. The key differences with our proposal is that they remain conservative in the content of the catalyst. Kim and Lipasti only handle a catalyst made up of ALU instructions and do not handle non-consecutive store combining. Thakker et al. is more aggressive but still does not fuse in the presence of a RaW or WaR with the catalyst, contrarily to Helios. Conservative catalyst content limits the potential of non-consecutive fusion (18.89% of the NCSF'd µ-ops have a RaW or WaR in the catalyst, 82.25% have a µ-op that may cause a pipeline flush). Moreover, our approach uses a predictive scheme and can therefore scale to large Allocation Queues, whereas the two proposed mechanism appear to rely on combinatorial logic to identify candidates.

VII. Conclusion

This work demonstrates that there exists significant potential for fusing non-consecutive memory instructions in RISC-V binaries and introduces the Helios microarchitecture to leverage this potential. Helios relies on a predictive scheme to fuse distant µ-ops and tackles numerous challenges to guarantee correct execution. Helios achieves a significant performance uplift over microarchitectures supporting various flavours of fusion, notably 14.2% over no fusion and 8.2% over consecutive and contiguous only memory fusion. While Helios was introduced in the context of the RISC-V ISA, we expect that similar potential exists in general purpose programs regardless of ISA, and Helios could therefore benefit other widely available processor families (e.g., x86, Arvm8).

Acknowledgment

This work was supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), the Ramón y Cajal Research Contract (RYC2018-025200-I), and the Vicerrectorado de Investigación e Internacionalización of the University of Murcia under the Talento 2021 programme.

References

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

[2] Advanced RISC Machines, *ARM Architecture Reference Manual ARMv8-A*, [Online; accessed Apr.-2022].

[3] Advanced RISC Machines, "Arm® Cortex™-A77 Core Software Optimization Guide, Issue 3," p. 68 Section 4.13, [Online; accessed Apr.-2022].

[4] Advanced RISC Machines, "Arm® Neoverse™-N2 Core Software Optimization Guide, issue 3," p. 88 Section 4.13, [Online; accessed Apr.-2022].

[5] 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.

[6] 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)*, Oct. 2021, pp. 366–379.

[7] 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*, Jul. 2016.

[8] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1998, pp. 142–153.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[10] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[11] 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. 03–14.

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

[13] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Pub 325383-076us," [Online; accessed Apr.-2022].

[14] L. Jin and S. Cho, "Reducing cache traffic and energy with macro data load," in *Proceedings of the 2006 Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Oct. 2006, p. 147–150.

[15] 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.

[16] I. Kim and M. Lipasti, "Macro-op scheduling: Relaxing scheduling loop constraints," in *36th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2003, pp. 277–288.

[17] I. Kim and M. H. Lipasti, "Implementing optimizations at decode time," in *29th Int'l Symp. on Computer Architecture (ISCA)*, May 2002, pp. 221–232.

[18] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *30th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1997, pp. 235–245.

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

[20] N. Riley and C. Zilles, "Probabilistic counter updates for predictor hysteresis and stratification," in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 110–120.

[21] RISC-V Software, "Spike RISC-V ISA Simulator." [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

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

[23] 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.

[24] A. Seznec, "A 256 kbits L-TAGE branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, pp. 1–6, Dec. 2007.

[25] A. Seznec, "The L-TAGE branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, pp. 1–13, May 2007.

[26] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," in *29th Int'l Symp. on Computer Architecture (ISCA)*, May 2002, pp. 295–306.

[27] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism (JILP)*, p. 23, Feb. 2006.

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

[29] 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," U.S. Patent 20 200 004 550A1, Feb., 2020.

[30] A. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, Jan. 2016.

[31] K. M. Wilson, K. Olukotun, and M. Rosenblum, "Increasing cache port efficiency for dynamic superscalar microprocessors," in *23rd Int'l Symp. on Computer Architecture (ISCA)*, May 1996, p. 147–157.

[32] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *24th Int'l Symp. on Microarchitecture (MICRO)*, Nov. 1991, pp. 51–61.