Compiler-Assisted Instruction Fusion

Ravikiran Ravindranath Reddy
Computer Engineering Department
University of Murcia
Murcia, Spain
ravikiran.r.r@um.es

Sawan Singh
Computer Engineering Department
University of Murcia, AMD
Murcia, Spain
singh.sawan@um.es

Arthur Perais
Univ. Grenoble Alpes, CNRS,
Grenoble INP, TIMA
Grenoble, France
arthur.perais@univ-grenoble-alpes.fr

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

Alexandra Jimborean
Computer Engineering Department
University of Murcia
Murcia, Spain
alexandra.jimborean@um.es

***Abstract*—Hardware instruction fusion combines multiple architectural instructions into a single operation, improving performance by freeing up resources. While fusion typically involves consecutive instructions, there are proposals to fuse non-consecutive instructions to maximize potential. However, such approaches require complex and costly hardware to predict and either validate fusion or unfuse, which significantly increases the cost of fusion. In this work, we propose a compiler technique, CAIF - Compiler Assisted Instruction Fusion, for fusion-aware instruction scheduling. CAIF identifies fusible but non-consecutive memory operations and reorders eligible pairs of instructions such that they appear consecutively in the instruction stream.**

Our experiments demonstrate that for neural network workloads, a hardware that only fuses consecutive instructions obtains 1.2% average performance improvements over a no-fusion baseline when applications are compiled with a standard compiler and 19.6% when compiled with CAIF. In addition, when non-consecutive hardware fusion (Helios) is enabled, CAIF boosts performance from 6.6% to 20.3%. Moreover, CAIF can effectively handle the statically challenging general-purpose application and boost performance on SPEC CPU 2017 from 2.4% to 6.4%, and from 14.4% to 17.7%, respectively, on the hardware configurations mentioned above.

***Index Terms*—Compiler optimization, instruction fusion, neural networks**

I. Introduction

Bringing AI to the edge, IoT, and wearable devices is a key goal in the field nowadays. Deep learning is executed directly on mobile phones, edge nodes, and even highly resource-constrained microcontrollers (MCUs) [30]. On one hand, enabling AI capabilities on these devices greatly enhances cost-efficiency, scalability, and user privacy by performing data processing locally, thus eliminating the need to constantly send sensitive information to the cloud. This not only reduces latency and energy spent sending data wirelessly, but also improves reliability and supports offline operation in environments with limited connectivity.

On the other hand, these devices often face severe resource constraints, lacking the full hardware acceleration or vectorization support necessary for high-performance AI computations. This gap poses a critical challenge: how to deliver efficient AI inference without increasing hardware complexity or power consumption. In contrast, even on systems with advanced hardware acceleration or vectorization support, performance is often limited by inherent application characteristics—such as dependencies and complex control flow—which prevent software from fully utilizing available hardware capabilities.

Instruction fusion emerges as a compelling solution—a lightweight microarchitectural technique that combines multiple instructions into fused operations, boosting execution efficiency with limited additional hardware overhead. By enabling more efficient use of limited compute resources, instruction fusion (i) significantly improves AI inference performance on constrained edge devices and (ii) squeezes the performance of otherwise hard-to-optimize general-purpose applications.

Instruction fusion (IF) is a hardware optimization technique used in many modern microarchitectures [1, 5, 16]. Typically, an architectural instruction is translated into one or more internal micro-operations (μ-ops). IF merges a pair of consecutive micro-operations into a single one during decoding. The fused μ-ops typically remain combined in the pipeline, representing more work with fewer bits, freeing resources such as registers, scheduler entries, and execution units. It also reduces tracking overhead, saves pipeline bandwidth, and decreases cycles and energy [55]. A common example is fusing a comparison with a conditional branch (*cmp+br*), saving 10% (assuming 1 branch every 10 instructions) of scheduler entries and reducing branch latency by one cycle [5, 16].

Some fusion approaches require ISA extensions [3], while others compact multiple instructions into one using hardware support [58, 64] (see Section II).

More recently, an increasing number of proposals have analyzed the impact of fusion for various combinations of instruction types on RISC-V platforms [6, 10, 57, 58]. These works emphasize that (i) fusing memory operations yields higher benefits than arithmetic operations [10, 58], and (ii) fusing both consecutive and non-consecutive instructions is essential for fully exploiting fusion potential [58].

Problem: While consecutive fusion requires minimum hardware support [36], state of the art fusion of non-consecutive instructions entails prediction, runtime monitoring and validation, and support for unfusion upon a misprediction. Therefore, non-consecutive fusion is significantly more costly [58, 64].

Insight: To increase fusion potential without adding hardware complexity, one requires a code layout in which fusible instructions are placed together, thus converting non-consecutive into consecutive pairs.

Solution: We propose compiler support to identify fusible pairs and reorder instructions to bring fusible non-consecutive memory operations together. Our Compiler-Assisted Instruction Fusion (CAIF) replaces the hardware complexity for prediction and verification. Based on the compile-time guarantees, no support for misspeculation detection and rollback is necessary. Moreover, CAIF is entirely automatic.

CAIF is implemented in the widely used LLVM compiler framework [26], which offers a flexible and modular infrastructure for developing custom compiler analyses. Yet, LLVM does not provide passes dedicated to instruction reordering that maintain fine-grain order throughout the middle-end to final binary generation. Existing passes (e.g. LICM[31]) and prior work [67, 21] reorder instructions at LLVM IR (Intermediate representation) without maintaining instruction-level granularity order throughout code generation. CAIF introduces a novel fusion-friendly instruction ordering preserved across all compilation phases. This requires identifying a suitable method to pack instructions early enough such that the IR order is maintained without disturbing critical back-end optimizations such as redundant load elimination (using pseudo-instructions), and to capture potential fusible instructions past scheduling (late enough) to maximize fusion opportunity (DAG-mutations)(see subsection IV-B).

We demonstrate these techniques on neural networks (NNs), a class of applications amenable to precise static analysis. NNs are widely used and rapidly evolving across various domains such as machine learning, image and speech recognition, and natural language processing. NNs run on diverse types of hardware, including accelerators like GPUs, TPUs (Tensor Processing Units), NPUs (neural processing units), as well as specialized accelerators. They also operate on edge devices, which are smaller, resource-constrained devices such as IoT devices and other (embedded) systems. Many of these emerging platforms implement an underlying RISC-V ISA [13, 22, 33, 52, 70].

Our proposed compiler pass enhances neural network inference performance on edge-class general-purpose microarchitectures, particularly benefiting low-cost and low-power systems, such as an Intel Atom-class core (Gracemont). Moreover, while we demonstrate the CAIF with the RISC-V ISA, its compiler-based approach is universally applicable to microarchitectures capable of fusing consecutive memory accesses within a memory region (e.g., cacheline or half-cacheline), even when accesses are non-contiguous within the region. In practice, CAIF increases memory bandwidth without the complexity or expense of a dedicated accelerator or a vector engine, turning it into an ideal choice for resource-constrained systems. Nonetheless, at the other end of the performance spectrum, CAIF offers a cost-effective alternative to hardware-only non-consecutive fusion techniques that would be worth the silicon investment only in high-performance systems, such as Helios [58].

In particular, this work offers the following contributions.

- A compiler technique to identify non-consecutive fusible memory operations and their dependencies and place them consecutively through instruction reordering.
- A back-end compiler technique to maintain the order of fusible instructions across O3 optimizations.
- Implemented these techniques in LLVM and applied them on the most common neural networks operations.
- Through simulation, we evaluated the benefits of compiler-assisted instruction fusion on various hardware configurations, including Helios [58], a state-of-the-art microarchitecture that enables hardware fusion, and demonstrated performance improvements.

Our experiments on NNs demonstrate that CAIF boosts the fusion of consecutive memory operations from 1.94% (without compile-time reordering) to 61.67%. Compared to a no-fusion baseline, CAIF achieves a 19.6% performance improvement by fusing only consecutive instructions non-speculatively, outperforming the baseline compiler's 1.2% improvement and surpassing advanced hardware supporting non-consecutive fusion (Helios), which achieves a 6.6% gain. Furthermore, combining CAIF with Helios further boosts performance, achieving on average 20.3% reduction in execution time over the no-fusion baseline. On SPEC, CAIF increases the fraction of consecutively fused memory operations from 20% (without compile-time reordering) to 30%. By fusing only consecutive instructions non-speculatively, CAIF delivers a 6.6% performance improvement, surpassing the baseline compiler's 2.6% gain.

II. Background

This section describes the hardware support for instruction fusion and instruction scheduling in a mainstream compiler. We will leverage the compiler support to enforce an instruction schedule that favors simple hardware instruction fusion.

A. Fusion Mechanisms for Memory Instructions

Multiple hardware fusion approaches exist, each balancing complexity and performance [53, 59]. Typically, fusing memory operations reduces latency only when both accesses map to the same cache line; otherwise, performance is similar to unfused instructions [1].

1) ISA-Level Fusion: ISAs like ARMv8 [3] provide fused instructions, such as load/store pairs, which do not need to be identified by hardware but require accesses to be contiguous in memory and use the same base register. This approach is not compatible with ISAs like RISC-V that restrict instructions to one destination register [10], as load pair has two.

2) Microarchitectural-Level Fusion: At the decode stage, hardware can pair valid memory instructions (same base register and suitable offsets) and emit a fused μ-op to optimize throughput [53]. Extensions allow fusion for any instructions accessing data within a 64-bytes region. This method introduces limited logic beyond standard cache design [58, 64].

3) Speculative Fusion: Predictive microarchitectures like Helios [58] fuse non-consecutive instructions based on runtime patterns, boosting fusion rates (e.g., from 1.94% to 27.12% in NNs). However, they incur significant hardware costs, including predictors, history buffers (9KB), and rename/issue validation to recover from misspeculation, avoid deadlocks, and ensure memory consistency. These mechanisms limit fusion depth to hardware queue sizes and delay retirement, as fused instructions spanning others can commit only after all in-between instructions do.

B. Compile-Time Instruction Scheduling

Instruction scheduling is a back-end compiler optimization that reorders instructions to avoid pipeline stalls and maximize parallelism [25]. LLVM's modular design allows easy customization of scheduling strategies tailored for fusion.

1) The SelectionDAG: In LLVM, the SelectionDAG [35] is a data structure representing data and instruction dependencies as a directed acyclic graph (DAG). It facilitates transforming intermediate representation (IR) into target-specific machine code through phases including construction, legalization, optimization, instruction selection, and scheduling.

2) DAG Mutations: Default heuristics may miss hardware-specific optimizations. By adding weak edges in the DAG, target-specific scheduling constraints can be introduced prior to scheduling. Although constraints reduce flexibility, the gains justify the cost, as some architectures run certain instructions faster when scheduled consecutively (e.g. AESE/AESMC [4]).

3) Memory Operations Clustering: LLVM's `BaseMemOpClusterMutation` [34] clusters memory operations within basic blocks using DAG mutations. It groups operations based on control dependencies while ensuring data dependencies are preserved, applying target-specific conditions such as alignment and offset ranges.

4) Pseudo-Instructions: LLVM uses pseudo instructions to represent complex patterns that do not correspond to real machine instructions [25]. These placeholders are expanded later, enabling alignment between IR and target-specific optimizations such as hardware instruction fusion.

III. Motivation

In this section, we characterize a hardware fusion approach for non-consecutive instructions, taking Helios as a state-of-the-art implementation. We conduct our analysis on a Gracemont-like processor [15], representative for edge computing, particularly in scenarios requiring energy and power efficiency.

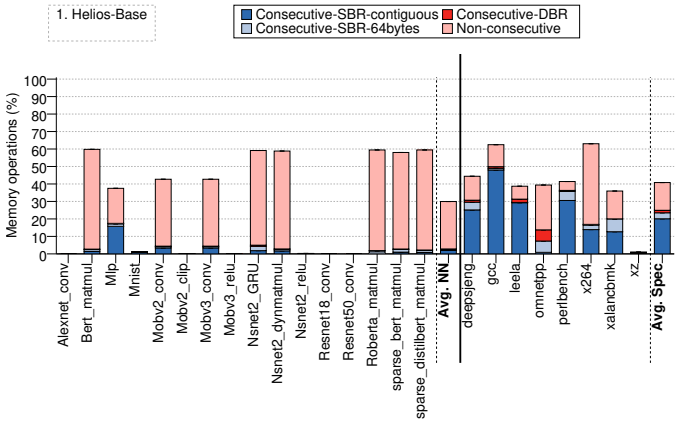

Fig. 1. Characterization of fused instructions in Helios

A. Characterization of Fused Pairs

Current solutions employ hardware prediction mechanisms to fuse both non same-base-register (SBR) and nonconsecutive memory operations. Figure 1 illustrates the characterization of fused instructions in Helios, applied on the most common neural network operations (see section V). For each application, the percentage of memory instructions fused at run-time is displayed on the y-axis. We categorize fusion pairs into one of the following four groups, listed by increasing hardware complexity, as detailed in section II:

- *Consecutive-SBR-contiguous* instruction pairs represent the simplest hardware fusion mechanism. Both instructions are consecutive, they use the same base register (SBR) to compute the address, and their memory locations are contiguous. These pairs are detected non-speculatively in the decode stage.
- *Consecutive-SBR-64bytes* The previous but non-contiguous access within a cacheline (64B) is allowed. This requires modifications in the LQ(Load queue) and the SQ(Store queue), but is also a non-speculative fusion mechanism.
- *Consecutive-DBR* The previous but the base register may differ. This requires the addition of a prediction mechanism and support for partially squashing fused instructions e.g., when the accesses span more than 64 bytes, as it is a speculative approach.
- *Non-consecutive* instruction pairs represent all pairs following any of the previous restrictions that are non-consecutive in the instruction stream. This requires extra logic at the Rename stage and more support for partially squashing fused instructions (e.g., a branch misprediction in the *catalyst*).

In NNs, Non-consecutive fusion has the potential that the benefits outweigh the (high) hardware costs. As shown in Figure 1, on average, 27.12% of the fused instructions are non-consecutive, while only 1.94% fall in the Consecutive-SBR-contiguous category which requires minimal hardware support. For SPEC, on average, 15.86% of instructions are fused non-consecutively, while 20% are fused as consecutive SBR-contiguous instructions.

Opportunity. A fusion-aware compiler can reschedule memory operations to convert non-consecutive fusible instructions into consecutive ones, reducing reliance on Helios' complex hardware while maintaining fusion and performance.

B. The Limitation of the Fusion Window

To fuse non-consecutive instructions, Helios searches the allocation queue for predicted-to-be-pairable instructions. If these instructions are not in the allocation queue –for instance because the processor is progressing well and has already allocated them– the fusion opportunities decrease. Although modern processors have large allocation queues (e.g., 192 entries in Lion Cove), these queues are often underutilized. Our experimental results show that the average distance between two fused instructions is generally low, averaging 4.16 instructions in NNs and 3.60 in SPEC, although for some pairs the distance can be as high as 64.

Opportunity. The compiler can potentially see a larger instruction window than the hardware, since it is not limited only to instructions in the allocation queue. For neural network applications, the average number of instructions in a basic block in LLVM Intermediate Representation (IR) is 20 to 40 instructions on average [39].

C. The Limitation of Prediction Mechanisms

Hardware fusion predictors can fail either by not predicting a fusible pair or by predicting an incorrect pair. In the former scenario, there is a missed opportunity, but in the latter, the performance impact can be significant, as the pipeline needs to be flushed. In the applications we analyzed, the number of mispredictions per kilo-instruction (MPKI) is generally negligible, reaching up to 0.27 in *xalancbmk* and 0.099 in *deepsjeng*. However, it can reach higher values in more complex applications [58] and potentially lead to performance degradation. In addition, although Singh et al. [58] did not quantify its impact, Helios requires the bulk commit of the fused instructions and instructions in the catalyst, for correctness. This can delay commit if the fused instructions are far apart and stall dispatch if the ROB becomes full as a consequence.

Opportunity. By arranging fusible instructions consecutively in the generated code, the compiler eliminates the need for hardware support for non-consecutive fusion. In addition to reducing hardware complexity, this approach avoids performance penalties caused by mispredictions and the delays introduced to enforce correctness (such as to commit).

IV. Compiler

We propose a compiler-based approach that (i) identifies pairs of fusible memory instructions, (ii) detects inter-instruction dependences, and (iii) reorders instructions to place the fusible instructions consecutively, thereby eliminating the need for a hardware predictor, runtime training, and support for misspeculation. In short, we shift the complexity from hardware to software.

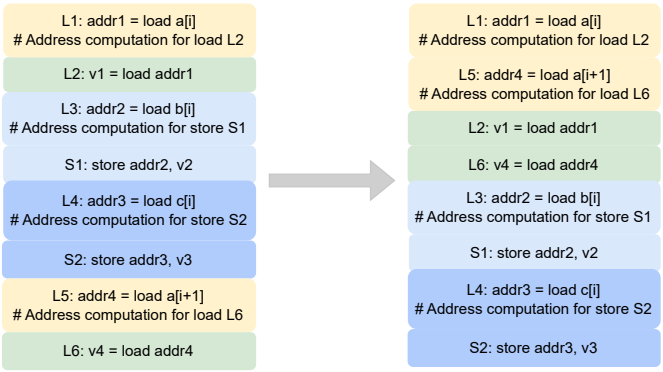

Fig. 2. Reordering instructions at LLVM IR requires complex *alias analyses* between memory instructions (*head*, *tail*, and *catalyst*) to determine the validity of the transformation and identifying instruction dependences (the *use-def chain*) to hoist the address computation together with the tail load.

Figure 2 illustrates these transformations, showing the original sequence of instructions, and highlighting how the non-consecutive load instructions (left) are identified as fusible and placed consecutively by our compiler (right).

Following the compilation chain described in Figure 3 and detailed in section V, we expose the compiler's intermediate representation (IR). Our compile-time analyses and code transformations are implemented in LLVM [25] and span the compiler middle-end and back-end, as shown in Figure 3. The middle-end is rich in high level information, such as loop structures, memory dependences, call graphs etc, and enables powerful static analysis and transformations (subsection IV-A). Nevertheless, instructions in the LLVM IR do not match one-to-one the final generated machine instructions, and their order is not guaranteed. Therefore, we extend the back-end to ensure that the fusible memory instructions remain paired in the final generated code (subsection IV-B).

A. Middle-End Analyses and Transformations

1) Identifying Fusible Memory Operations: In LLVM IR, we iterate each basic block's instructions — when a head load or store is found, we scan ahead for a corresponding tail in the same block. Next, we check if the head and tail operations are fusible, i.e. they access memory locations within a 64-byte range. For this, we compare two memory operations by extracting their pointer operands and identifying a potential common base pointer, using the `StripAndAccumulate` of each pointer. If they do, the offset between them is accumulated and used to determine the memory distance between the loads. If they do not share a common base, we compute the difference using Scalar Evolution (SCEV) analysis. This difference indicates if the two pointers fall within the 64-byte range and can be fused. If the selected memory operations cannot be fused, we reset the tail and continue searching for another potential tail until the end of the basic block. We repeat this process with each load (store) that has not been marked as fusible, in the role of the head. This ensures we systematically check all possible pairs of loads (or stores) within each basic

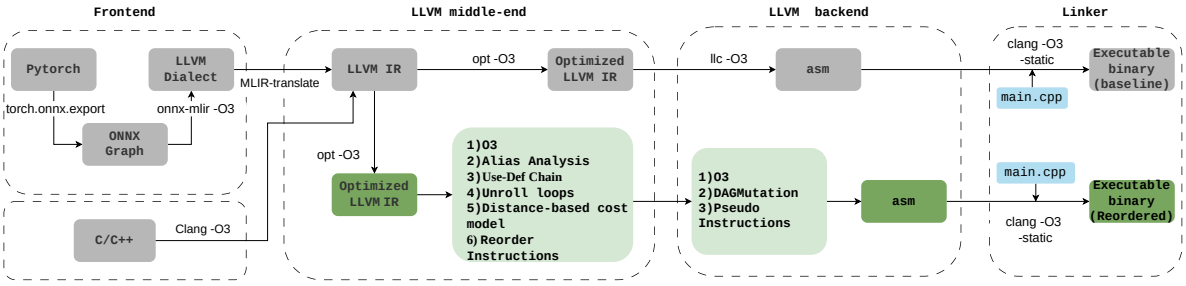

Fig. 3. Compilation flow: CAIF passes (shown in light green) are executed after the standard O3 optimization passes at each compilation stage.

block, maximizing the chances of identifying fusible memory operations.

If the head load (store) and tail load (store) are fusible, then we keep the instructions between the two fusible memory instructions in a queue named *Catalyst*. Next, we build Use-Def chains to find instructions from the Catalyst that are required for the execution of the tail instruction, such as the computation of its target address or the value to store (subsubsection IV-A2). Finally, using compile-time alias analysis, we identify memory dependences between the tail instruction and its use-def chain on one hand, and the memory instructions from the Catalyst and the head operation on the other hand. Intuitively, we ensure that loads and stores that are reordered do not cross any aliasing memory instructions. Any such memory dependence prohibits fusion (subsubsection IV-A3). These analyses ensure safe and correct instruction reordering. The analysis steps are summarized below:

1) Identify fusible loads/stores pair.
2) Collect intermediate instructions in the Catalyst.
3) Build the use-def chain for the tail instruction.
4) If the use-def chain of the Tail includes the Head, no reordering is performed.
5) Check memory dependences between the tail instruction and its use-def chain with the Catalyst, as follows:
   a) If tail is a load (Figure 4):
      - Check dependences with all the stores from the Catalyst.
      - If its use-def chain contains any loads, check dependences between each load and all the stores from the Catalyst.
   b) If tail is a store (Figure 5):
      - Check dependences with all the loads and stores from the Catalyst.
      - If its use-def chain contains any loads, check dependences between each load and all the stores from the Catalyst and the head store.

To broaden our analysis, we integrated advanced alias analyses, combining Andersen analysis [28], flow-sensitive methods [7] from SVF [62] with LLVM's built-in analysis. For interprocedural analysis, we used GraphAA [32], enabling memory operation reordering across function calls.

Fig. 4. Memory dependency to reorder a Tail load. The fusible instructions are L1 and L4. The use-def chain (target address computation) of L4 contains L2 and L3. CAIF verifies for potential conflicts between the loads from the use-def chain and other stores from the catalyst that preceed the use-def chain instructions, as shown in the gray box. If no conflicts exist, reordering can proceed (right).

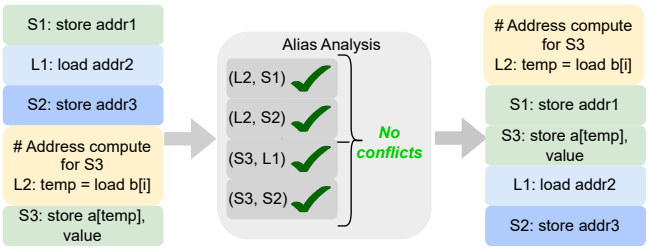

Fig. 5. Memory dependency to reorder a Tail store. The fusible instructions are S1 and S3. The use-def chain (target address computation) of L3 contains L2. Our compiler verifies for potential conflicts between the loads (L2) from the use-def chain and other stores (S2) from the catalyst that proceed the use-def chain instructions and any conflicts with the Head store. Also, it verifies the Tail store (S3) does not conflict with any load (L1) or store (S2) from the catalyst it is reordered with or with the Head store. The checks are shown in the gray box. If no conflicts exist, reordering can proceed (right).

2) Identifying Required Instructions for Hoisting: Along with the tail operation, the compiler must hoist the necessary instructions that the tail depends on. To identify these instructions from the Catalyst, we use the use-def chain from LLVM. The use-def chain provides a list of all values used by a user, in this case the Tail. If the use-def chain requires a store, we avoid reordering the tail due to the complexity involved in instruction reordering. Identifying the instructions from the Catalyst that belong to the use-def chain of the Tail operation, and hoisting them before the Head operation, is essential for preserving the correct data flow.

3) Validity of Reordering: Memory dependence analysis is crucial for correctly reordering instructions. Specifically, for instruction fusion, the compiler must ensure that:

1) The tail memory instruction does not conflict with other memory instructions from the Catalyst or with Head.
2) There are no memory dependencies between the tail's use-def chain from the Catalyst and any other memory instructions it is reordered with.

If no memory dependencies are detected, the compiler reorders the Tail's necessary instructions (use-def) to precede the Head. It then places the Tail after the Head, converting non-consecutive fusible operations into consecutive ones.

4) Increasing the Compiler's Reach: The compiler conservatively reorders memory operations only within the boundaries of a basic block, which by design keeps register pressure at bay. However, to expand the window of instructions the compiler can operate on, CAIF enables dynamic loop unrolling, which is not included in the default O3 compiler optimizations. Dynamic loop unrolling (DLU) unrolls loops with statically unknown trip counts by guarding the unrolled iterations with conditionals. DLU can increase the number of live variables, potentially leading to register spilling. To limit this effect, we restrict DLU to innermost loops that have not been previously unrolled by O3, have less than 50 instructions, and contain at least one memory operation. Moreover, the unroll factor is 2 (and not higher).

5) Correctness: Given that our compiler checks that all dependencies are satisfied and all CAIF transformations are valid, correctness is preserved. Moreover, we have empirically confirmed the correctness for all the selected benchmarks.

6) Limitations: The instruction order established in the compiler's middle-end might not be preserved by the LLVM backend optimizations (llc -O3). To preserve the order, the following changes are necessary in the LLVM backend.

B. Backend Analyses and Transformations

To preserve a fusion-friendly memory access order, we employ two mechanisms: Pseudo-Instructions and DAGMutation.

1) Pseudo Instructions: Pseudo-instructions are a powerful mechanism to enforce clustering of memory operations. Through pseudo-instructions, two or more scalar memory operations are packed into one, preventing the default scheduler and register allocator from breaking the desired order of the selected memory operations. We created new custom instructions for integer and floating-point pairs of loads and stores, respectively. Each pseudo-instruction is associated with a profile describing its operands and specific properties, which determine the behavior and constraints of the corresponding custom instruction. During compilation, the original fusible memory instructions are replaced by these pseudo-instructions, after which the standard back-end optimizations are applied. However, O3 optimizations treat pseudo-instructions as single units, which restricts per-instruction optimization and may lead to redundant memory operations, because a pseudo-instruction can only be removed if both of its memory operations are redundant. This may hinder peephole optimization and dead code elimination.

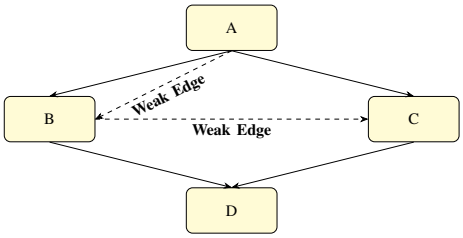

Fig. 6. Schedule nodes A and B consecutively

To mitigate this limitation, unused nodes are pruned during instruction selection, before forming pseudo-instructions from candidate memory-operation pairs. Next, a top-down scheduling policy is applied to minimize register pressure while preserving the relative order of the fused memory operations, since they are already packed into pseudo-instructions. After instruction scheduling and register allocation, pseudo-instructions are expanded back into their original scalar instructions, ensuring that the intended sequence of consecutive memory accesses is preserved. Note that additional memory operations (such as stack pushes and pops) are generated in later back-end phases and cannot be packed, because pseudo-instructions operate before register allocation. These late memory operations are instead co-scheduled using the DAGMutation mechanism in CAIF.

2) DAGMutation: To insert scheduling constraints in the later backend stages for fusible memory instructions missed by pseudo-instructions, we use DAGMutation.

The steps for DAGMutation for instruction fusion are:

- Find candidates for fusion by inspecting the predecessors of each instruction. Specifically, eligible candidates are scalar memory operations that share the same base register and have memory offsets within a 64 byte range.
- Add a weak Cluster edge (Constraint) between A and B (shown in dotted line, Figure 6). In top-down scheduling, when node A is encountered, node B will be chosen next over C, due to the weak edge between (A,B).
- Add a weak Cluster edge (Constraint) between B and C (shown in dotted line). In bottom-up scheduling, the weak edges ensure no instructions are scheduled between A and B. By marking C as dependent on B, C must be processed before B, keeping nodes B and A together.

In both scenarios, the schedule order will be A, B, C and D.

DAGMutation fusion pairs are generated using TableGen. In this process, we specify the memory operations and constraints such as ensuring the same base pointers and that the difference between offsets is less than or equal to 64.

Next, we leverage the `BaseMemOpClusterMutation` class to cluster memory operations that target the same cache line. The clustering process is guided by the `shouldClusterMemOps` method, which is customized for each target architecture.

DAGMutation introduces weak dependencies that can be disrupted by register pressure. This drawback is addressed

by complementing DAG Mutation with the use of pseudo-instructions, that impose stricter scheduling constraints, as mentioned above. The two approaches work in tandem to balance the default scheduling heuristics with the order constraints required for fusion: pseudo-instructions pack instructions early enough such that the IR order is maintained without disturbing critical back-end optimizations, while DAG-mutations capture potential fusible instructions past register allocation, to maximize fusion opportunities.

V. Methodology

A. Workloads and Compilation Flow

To evaluate our proposal, we selected twelve neural network models: Alexnet [47], Resnet18 [50], Resnet50 [51], MobileNet v2 (Mobv2) [48], MobileNet v3 (Mobv3) [49], Mlp [45], Mnist [46], Nsnet2 [9, 40], Google Bert [8], Roberta [54], Sparse Bert [12, 17] and Sparse DistilBert [18, 69] pre-trained models. These benchmarks are amenable to precise alias analysis. The training and testing data sets used for these models are detailed in the corresponding references. Due to storage space constraints caused by large trace files, we split the neural networks into layers. We profiled each layer by instrumenting and counting load/store instructions and total instructions using Pin [37]. For most models, we chose the top 2–3 representative layers; if they were identical, we used only one, and in some cases included all layers for Mnist, Mlp. Details can be found in the Appendix.

The steps to generate LLVM IR from a neural network implemented with PyTorch are shown in Figure 3. In a nutshell, we convert PyTorch code to an ONNX graph [43], and translate the ONNX graph to the LLVM dialect of MLIR [27]. Finally, we use clang to produce an executable optimized for the RISCV with RISCV64G extensions. All compilation steps use the O3 optimization level.

We also evaluate SPEC CPU 2017 [61] (int, speed) by selecting a region of interest (ROI) based on a thorough application analysis to delimit the amount of work rather than a fixed number of instructions, thus providing comparable results across compile-time transformed code. We select the functions that are most representative of the application's total execution, provided they execute at least 100 million instructions. If no single instance reaches this threshold, we evaluate multiple invocations of that function instead.

Details are given in the Appendix.

We conduct all experiments with two compilation flows: Base –the baseline compilation with standard O3 optimization level and no reordering– and CAIF. We also report our findings on the impact of dynamic loop unrolling on Helios and CAIF.

B. Simulation Model and Configurations

The produced binaries are executed on top of the RISC-V Spike Simulator [11] in full-system mode running a Linux kernel [63]. Spike has been modified to generate the stream of instructions (and the memory locations and size they access) executed by the program, which are used to feed the timing simulator. Our cycle-level in-house simulator uses the GEMS memory model with a three-level cache hierarchy and a detailed out-of-order core that resembles an Intel Gracemont architecture. The processor configuration is summarized in Table I. We use CACTI [29] to model the energy consumption of the LQ, SQ and L1D. The LQ and the SQ are modeled as a CAM and use the high-performance (hp) model.

TABLE I
Baseline configuration

Branch pred.	64KB TAGE-SC-L [56], minimum branch misprediction penalty 8 cycles
Front end	6-wide Fetch/Decode, 5-wide Rename/Dispatch, 64-entry decode queue
Back end	17-wide Execution (2 read ports, 2 write ports), 8-wide Commit
ROB/IQ/LQ/SB	256/221/80/50 entries
L1I	64KB, 8 ways, 3-cycle hit cycles, pipelined
L1D	32KB, 8 ways, 3-cycle hit cycles, pipelined, IP-Stride prefetcher with degree 3
L2	2MB, 16 ways, 20 hit cycles
LLC	750KB per bank, 8 banks, 65 hit cycles
RAM	100-cycle latency

Our out-of-order core model has been extended to implement both consecutive and non-consecutive instruction fusion as described in Helios. In particular, we are able to run several configurations of the Helios hardware with different complexity requirements (see subsection III-A for the description of the fusion types):

- *NoFusion*: Fusion is disabled in the hardware.
- *ConsSBR*: This hardware is able to fuse only Consecutive-SBR-contiguous instruction pairs. It is a non-speculative fusion implementation.
- *ConsSBR64*: This hardware is able to fuse both Consecutive-SBR-contiguous and Consecutive-SBR-64bytes instruction pairs. It is also a non-speculative implementation.
- *Cons*: This hardware is able to fuse all consecutive instruction pairs. It requires speculative support.
- *Helios*: This hardware enables all kinds of fusion, including non-consecutive pairs. It is also a speculative solution.

VI. Evaluation

This section evaluates the impact of compiler assistance on instruction fusion, and investigate (1) the degree to which CAIF shifts fusion from non-consecutive (highest hardware complexity) to consecutive (minimum hardware complexity); (2) the number of mispredictions in Helios after reordering; (3) analysis of stall cycles; (4) the overall performance with varying hardware support for fusion; (5) energy savings.

A. Characterization of Fused Pairs

To evaluate the efficiency of the compile-time reordering, we analyzed the percentage of fused memory operations across different hardware configurations of increasing complexity, as detailed in subsection V-B and with two compiler optimizations (Base and CAIF), leading to the eight combinations shown in Figure 7. Thus, Figure 7 extends Figure 1 (which corresponds to the bar labeled Helios-base) with all

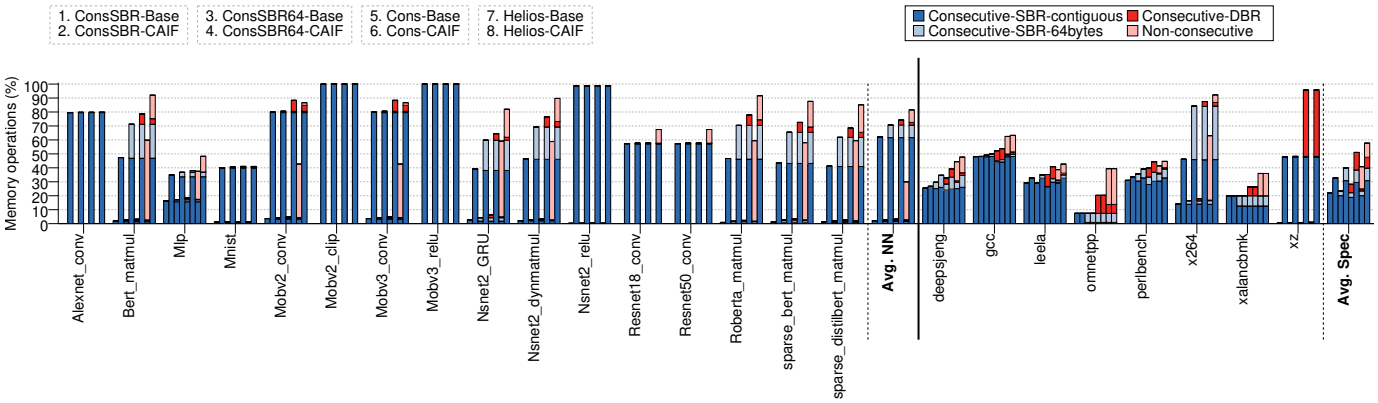

Fig. 7. Percentage distribution of fusion types. Bars are paired (without and with CAIF for each configuration)

the configurations tested. It shows the breakdown of fusion per category: Consecutive-SBR-contiguous, Consecutive-SBR-64bytes, Consecutive-DBR, and Non-consecutive. The Oy axis shows the percentage of fused instructions.

Focusing on the last two bars (average bar), we can observe that for Helios, CAIF converted Non-consecutive fusion to consecutive fusion hence on average CAIF managed to decrease Non-consecutive fusion from 27.12% to 8.96% on NNs. On SPEC, with CAIF Non-consecutive fusion reduced from 15% to 10%. In addition, on average for NNs, CAIF increases Consecutive-SBR-contiguous from 1.94% to 61.67%. Fusing consecutive memory operations requires relatively simple hardware compared to Helios, i.e., no need for prediction and no misprediction penalty.

For all layers in Alexnet, Resnet18, Resnet50 and Mobv2.clip there is no fusion when using Base, but CAIF can fuse more than 50% of instructions in these layers, with all cases being Consecutive-SBR-contiguous fusion. For instance, for Mobv2_relu, CAIF fuses 100% of the memory operations.

CAIF delivers high performance particularly on layers such as Matmul, GRU, and Relu, where dynamic unrolling and instruction reordering bring together fusible pairs, effectively converting a significant fraction of the non-consecutive fused instructions from different iterations into consecutive pairs (up to 100% in some cases).

More importantly, when focusing (Avg. NN) on a ConsSBR64 hardware, we can observe that the number of fused instructions with CAIF ConsSBR64-CAIF (4th bar, 70.61%) increases by more than 40% compared to Helios running the code generated by the base compiler (7th bar, 29.93%). For SPEC (Avg. Spec), CAIF with ConsSBR64 (4th bar), we see that fused instructions have increased from 23.51% to 39.8% when compared to ConSBR64-Base (3rd bar). ConsSBR64 CAIF (4th bar, 39.8%) is on par with Helios-Base (7th bar) which achieves 40.69%. This shows that CAIF on NNs, even with simple non-speculative fusion hardware, can outperform the more complex speculative Helios hardware while achieving comparable performance on SPEC.

B. Fusion Mispredictions

We analyzed the MPKI of non-consecutive fused pairs across the neural network benchmarks, comparing Helios using the baseline compiler versus Helios using CAIF. Generally, when using CAIF, the MPKI reduces since there are fewer pairs to speculate on, as a consequence of turning non-consecutive pairs into consecutive pairs. MPKI was reduced from 0.31 to 0.29 with CAIF. MPKI of deepsjeng with CAIF decreases from 0.099 to 0.089 and xalancbmk has the same mpki in CAIF as Helios-base this correlates with the trends in non-consecutive pairs as shown in Figure 7.

We underline that CAIF performs non-speculative fusion (consecutive only), where no mispredictions occur.

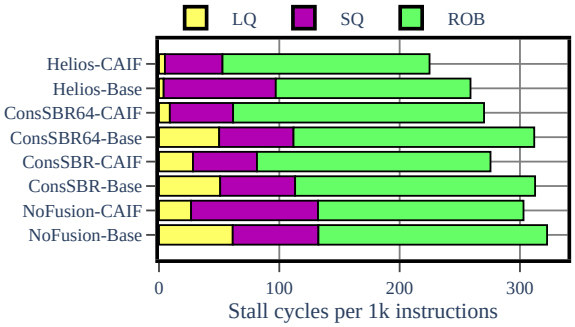

Fig. 8. Average LQ/SQ/ROB stalls for the fusion types

C. Analysis of Backend Stalls

Fusing instructions leads to lower occupation of resources, namely ROB, LQ, and SQ, as a fused pair uses a single entry in those queues. This results in reduced stall cycles at the allocation stage where the instructions wait for free slots in the queues. Figure 8 shows the average stall cycles caused by ROB/LQ/SQ being full at the allocation stage per 1K instructions, for the evaluated configurations. CAIF consistently reduces stall cycles in all hardware configurations. ConsSBR64-CAIF achieves lower allocation in LQ (-27.5% / -6.34%), SQ (-24.14% / -22.7%), and ROB (-9.26% / -4.25%) when compared to ConsSBR64-Base / Helios-Base for NNs and SPEC combined.

D. Execution Time

Reductions in stalls cycles translates to better execution time. Figure 9 shows the impact of CAIF versus `Base` (-O3) for the different fusion configurations on execution time (y-axis) and for the neural network kernels explored

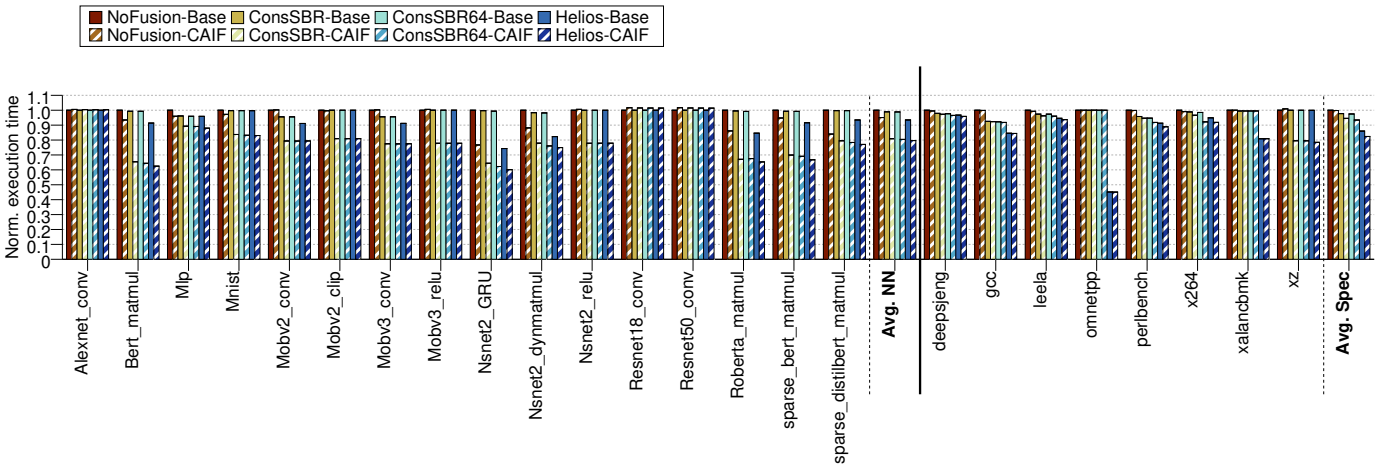

Fig. 9. Normalized execution time of evaluated fusion types

in this work (x-axis). The execution time is normalized against our compiler baseline (-O3) with hardware fusion disabled (`NoFusion-Base`). Since Consecutive-DBR instructions represent a negligible (0.12%) fraction of the fused instructions (see Figure 1), here we omit the Cons-DBR hardware configuration for clarity. We discuss the execution time by pair, with and without compiler-assisted fusion.

Applying our instruction reordering techniques without fusion (NoFusion-CAIF) improves execution time (by 5% on average in NNs), thanks to dynamic loop unrolling and instruction reordering, which can help increase the scope of optimizations and enhance memory level parallelism [68]. Clustering memory instructions enhances both memory-level and instruction-level parallelism by enabling simultaneous memory requests and parallel execution of independent loads. This reduces latency, improves bandwidth utilization, and keeps the pipeline full. NoFusion-CAIF outperforms NoFusion-Base by leveraging higher ILP and MLP, demonstrating that CAIF enables a more efficient instruction schedule. Through careful engineering to reduce overheads (see section IV, our compiler introduces no register pressure: CAIF constrains DLU as described in subsubsection IV-A4; only reorders within a basic block, which imposes a natural boundary and limits the distance of reordering; and performs backend optimizations to eliminate redundant operations, prior to packing mem-ops in pseudo-instructions.

When employing the simplest fusion hardware (ConsSBR) on NNs, CAIF reduces the average execution time by 19.1% with respect to NoFusion-Base, while the baseline compiler achieves a 1.1% reduction. CAIF also excels when both non-speculative fusion techniques are leveraged (ConsSBR64). In this case, CAIF reduces execution time by approximately 19.6% with respect to NoFusion-Base, while execution time reduces by an average of 1.2% with the baseline compiler with respect to NoFusion-Base. ConsSBR64-CAIF further reduces execution time on SPEC, improving from 2.6% with ConsSBR64-Base to 6.6% relative to NoFusion-Base.

The overall reduction in execution time suggests that instruction reordering indeed enhances the potential of consecutive instruction fusion even on simple hardware.

Helios on NNs, fusing all possible pairs including non-consecutive, shows an execution time reduction of 6.6% when using the baseline compiler. However, combining Helios with instruction reordering (Helios-CAIF) yields the most significant improvement, reflecting a 20.3% reduction in execution time compared to NoFusion-Base.

While matrix multiplication (MatMul) shows consistent performance gains, convolution workloads benefit less. This is primarily due to a higher number of cache misses in convolution, which causes processor stalls waiting for memory.

In SPEC, as seen in x264, CAIF reports many pairs where the distance between the head and tail is 120 instructions, which is not reachable in hardware. Hence, CAIF increases fusion opportunities that were not possible with Helios with a gracemont-like configuration. In addition, CAIF reduces the average distance between head and tail instructions, known as catalyst depth, as shown in Figure 10. By lowering the average catalyst depth from 4.16 to 1.68 in NNs and from 3.60 to 2.88 in SPEC, the commit time of instructions decreases because the head instruction waits less for the catalyst instructions to become ready. This alleviates pressure on the ROB, reducing stalls and ultimately enhancing performance. In applications with more backend stalls, fusion provides higher benefits. Notably, xz shows larger performance improvements than x264, even though both benchmarks have a similar percentage of fused pairs. This is because xz suffers from more pronounced backend bottlenecks, making it more sensitive to increased fusion opportunities from CAIF.

Overall, CAIF strikes an optimal balance between hardware complexity and performance without increasing binary size. By controlling register pressure, CAIF produces binaries of comparable size to the baseline compiler, with the maximum increase observed in the xalancbmk benchmark at just 0.31%.

When executing on non-speculative fusion hardware, CAIF-generated binaries surpass the performance of baseline binaries leveraging more intricate hardware with speculative fusion techniques. These results underline the potential of simple fusion mechanisms in hardware, particularly when complemented by tailored compiler instruction reordering.

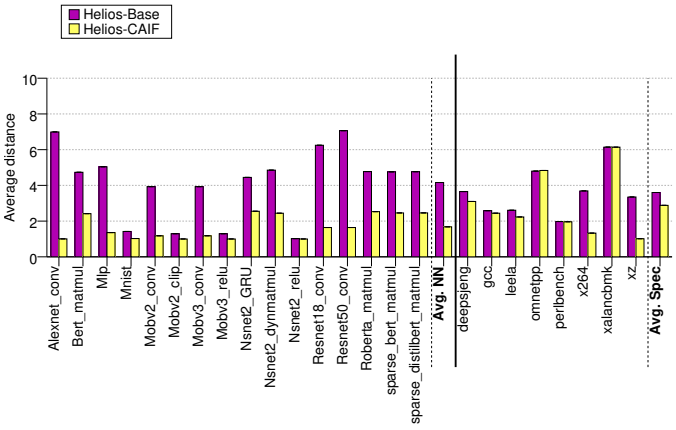

Fig. 10. Helios-reported catalyst depth

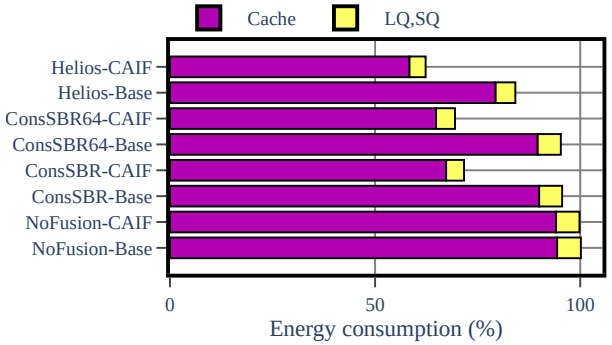

Fig. 11. Normalized energy consumption of the cache, LQ, and SQ

E. Energy Savings

CAIF reduces energy consumption in both the L1D cache, by minimizing accesses, and the processing core, by combining two instructions into one. Figure 11 highlights its impact on the energy usage of the L1D cache, as well as the LQ and SQ—key structures affected by fusion. As before, results are shown for four hardware configurations and two compiler versions. CAIF proves highly effective in lowering L1 cache accesses, with ConsSBR64-CAIF achieving a 31.28% reduction and Helios-CAIF reaching 38.17%, compared to NoFusion-Base. Additionally, CAIF reduces LQ and SQ energy expenditure across all setups, as fused loads and stores only require a single search in these queues. The highest reductions are seen with Helios-CAIF (32.41%) and ConsSBR64-CAIF (20.17%) compared to their Base versions.

Comparing Helios-Base and ConsSBR64-CAIF, which deliver comparable performance, ConsSBR64-CAIF stands out by using 18.2% less L1D energy and 3.54% less LQ/SQ energy. This makes it not only a simpler but also a more energy-efficient choice.

F. Analysis on High-Performance Cores

Until now, the evaluation has focused on low-power processors such as Gracemont (256 ROB entries). This section analyses the performance improvements on a high-performance core, a Golden Cove processor (512 ROB entries), modifying the parameters shown in Table I accordingly [23]. Golden cove has larger ROB, LQ and SQ than Gracemont, leading to fewer backend stalls, hence less performance opportunities. We analyse it both without and with vectorization support.

1) Without Vectorization Support: Our results on SPEC reveal that CAIF boosts performance by 2.7% with ConsSBR and by 4.2% with ConsSBR64. Energy savings follow the same trend as on Gracemont-like cores: on SPEC, Helios-Base cuts energy by 21.2% vs. NoFusion-Base, while Helios-CAIF reaches 24.14%, mainly by reducing cache accesses, in line with its higher fraction of fused memory operations.

2) With Vectorization Support: CAIF is applied after O3 (Vectorization) and does not reorder vectorized memory operations. Vector-friendly kernels such as Relu and Conv are completely vectorized, hence fusion gains are minimal. However, for Matmul layers, which dominate execution in Transformer models, our analysis shows that only about 40% of the memory operations are vectorized, leaving substantial room for fusion.

CAIF on NNs decreases non-consecutive fusion from 15.4% to 3.1% and increases consecutive fusion from 3.3% to 26.4%. This improves performance by 4% with ConSBR64, whereas the compiler baseline only gains 2.2% with ConsSBR64. CAIF with ConsSBR64 reaches a maximum of 19.6% in Sparse_distilbert_matmul and a minimum of 1.97% in bert_matmul, whereas the compiler baseline delivers 0.4% in Sparse_distilbert_matmul and 0.4% in bert_Matmul.

On SPEC with vectorization, CAIF reduces non-consecutive fusion from 17.8% (baseline) to 13.5% and boosts consecutive fusion from 26.14% to 35.79%, yielding 3.7% speedup with ConsSBR64 (vs. baseline's 2.5%). Across benchmarks, CAIF with ConsSBR64 gains maximum improvement of 8.9% for GCC versus 4.9% with the baseline compiler, and its minimum is 2.62% on Deepsjeng versus 2.25% for the baseline.

G. Limitation of Reordering for Fusion

While compiler-driven instruction reordering enables macro fusion by grouping memory operations into a single macro-operation, this optimization can introduce a trade-off. It can increase dependency lengths in the pipeline. If a short-latency instruction (head) is fused with a long-latency one (tail), the head may wait longer, delaying dependent operations. This can serialize execution and reduce performance, especially under contention for backend resources (e.g., caches, memory ports, functional units) or in long dependency chains.

VII. Related Work

A. Software Instruction Fusion

Modern processors like ARMv7/v8 offer instructions such as Load-Pair (LDP) and Load Multiple Increment After (LDMIA), which can load multiple registers from consecutive memory locations. Those can be cracked to simplify hardware but increase latency, or executed as a unit to minimize latency at hardware cost [10] (e.g., two register file write ports to perform LDP as a unit). Fusion idioms are limited to what the architectural instructions provide.

Micro-op Fusion (μ-ops) is gaining traction on RISC-V processors. Balasubramanian et. al. [6] introduced three new instructions—Load Word and Add (LWA), Load Word

and Multiply (LWM), and Load Word and Subtract (LWS) —which fuse these common load- arithmetic pairs into single instructions, at the ISA level. In contrast, we target the fusion of memory operations and opt for compiler-assisted fusion in hardware, without modifying the ISA.

However, the availability of fused instructions at the ISA level is useful only if the compiler is able to emit them. Shen et al. tailor the LLVM instruction scheduler and register allocator for instruction fusion in a manner that resembles the initial steps of our approach [57]. They leverage existing LLVM mechanisms – DAGMutations – to place fusible load-add and shift-add instructions together, but DAGMutations alone introduce weak constraints that are not guaranteed to remain in the final generated code.

Six et al. [60] provide a certified instruction scheduler for fusing memory operations on AArch64 platforms (*load pair* and *store pair*). However, the technique is limited to contiguous memory accesses only, as required by the AArch64 instruction semantics. Moreover, as it reorders instructions post register allocation, it is more restricted than CAIF by register-introduced dependences.

This work focuses on memory instruction fusion as they have been shown to provide the highest potential [10, 58], leveraring leveraging advanced memory aliasing analysis and balancing the soft constraints of DAGMutations with the hard constraints imposed through pseudo-instructions, to enable O3 back-end optimizations, but also guarantee the desired order in the final generated code.

B. Hardware Fusion

Kim and Lipasti propose Macro-op scheduling [20] pairs two dependent instructions in hardware and schedules them as a unit. This is a limited form of hardware fusion that saves scheduler entries but still performs the two instructions distinctly, i.e., the latency is the sum of both and two functional units are required over two cycles. It is orthogonal to our focus on fusing independent instructions. Singh et al. propose non-consecutive load and store fusion in hardware with the Helios microarchitecture, which relies on speculation and several points of validation to confirm that two instructions could indeed be fused. CAIF targets some of the potential of Helios without incurring hardware cost over consecutive fusion.

C. Instruction Reordering

Beyond instruction fusion, static instruction scheduling (reordering) has been employed in prior work for improving performance through various methods.

Static instruction schedulers [2, 38, 24] aim to hide memory latency by reordering instructions to keep the processor busy while waiting for memory operations to complete. However, these schedulers face limitations due to register pressure and basic block boundaries. Recent advancements in static instruction scheduling have focused on overcoming these limitations and improving performance. For example, through loop pipelining and cross basic block boundary instruction reordering, the compiler separates load instructions from their uses to increase memory and instruction level parallelism [66, 68, 65]. This technique involves monitoring the latency of load instructions in real-time. By learning their latencies, processors can orchestrate the execution of the compiler-predefined phases, achieving performance close to that of out-of-order processors while maintaining energy efficiency.

Evolutionary Algorithms, such as genetic algorithms, are used to optimize the scheduling of instructions and the allocation of registers simultaneously. This method has shown promise in improving the overall performance and efficiency of processors by finding optimal schedules that traditional techniques might miss [14].

CAIF employs a two-level reordering to leverage the rich-in-information middle-end and preserve the order in the back-end, while balancing register pressure.

D. Vectorization

CAIF differs from the LLVM Loop Vectorizer[42, 41, 19], which relies on predictable loop patterns and widens instructions [44], and from SLP vectorization, which bundles isomorphic operations, instead of reordering. This lightweight mechanism for preserving instruction order from middle-end till binary generation, currently absent in LLVM and in the literature, is key to software-hardware co-design contracts.

VIII. Conclusion

Compiler reordering enhances fusion by scheduling fusible instructions consecutively. Through static analysis, the compiler can identify fusible instructions, check memory and instruction dependences, and validate the transformation. This approach incurs no additional costs and replaces complex hardware mechanisms that perform these predictions and validations at runtime.

For NN workloads, CAIF reduces execution time by 19.6% on average, on hardware that fuses only consecutive instructions, surpassing Helios's 6.6% for both consecutive plus non-consecutive fusion. Furthermore, CAIF enhances Helios, achieving an average reduction of 20.3%. Since most pairs can be fused with minimal hardware modifications with CAIF, the added complexity of the predictor does not justify the limited performance gains. For SPEC workloads, CAIF reduces execution time by 6.6% while the compiler baseline reaches 2.6% on hardware that fuses only consecutive instructions. Thus, CAIF improves performance on general-purpose workloads while maintaining simple hardware.

Acknowledgement

This work has been funded by the EU's Horizon 2021 research and innovation program (CONVOLVE, g.a. no. 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO, g.a. no 819134), the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00), and the Ramón y Cajal Research Contract RYC2018-025200-I.

References

[1] Advanced RISC Machines. *Arm® Neoverse™-N2 Core Software Optimization Guide*. [Online; accessed June.-2024], p. 88.

[2] Alexander Aiken, Alexandru Nicolau, and Steven Novack. "Resource-Constrained Software Pipelining". In: *IEEE Trans. Parallel Distrib. Syst.* 6.12 (1995), pp. 1248–1270.

[3] *ARM Architecture Reference Manual ARMv8-A*. 2015.

[4] *Armaesfusion*. 2020. URL: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://documentation-service.arm.com/static/5fb7d95ed77dd807b9a80c79%3Ftoken%3D&ved=2ahUKEwiThu-v4e-JAxVuTaQEHTqtEUkQFnoECC8QAQ&usg=AOvVaw0pw8cbaH-XCggPBMVX5BzF.

[5] *Arm® Cortex™-A77 Core Software Optimization Guide*. [Online; accessed Nov.-2024]. Advanced RISC Machines. Chap. Section 4.13, p. 68. URL: https://developer.arm.com/documentation/swog011050/latest/.

[6] Karthikeyan Kalyanasundaram Balasubramanian et al. "Designing RISC-V Instruction Set Extensions for Artificial Neural Networks: An LLVM Compiler-Driven Perspective". In: *IEEE Access* 12 (2024), pp. 55925–55944.

[7] Mohamad Barbar, Yulei Sui, and Shiping Chen. "Object versioning for flow-sensitive pointer analysis". In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '21. Virtual Event, Republic of Korea: IEEE Press, 2021, pp. 222–235. ISBN: 9781728186139. DOI: 10.1109/CGO51591.2021.9370334. URL: https://doi.org/10.1109/CGO51591.2021.9370334.

[8] *Google Bert*. 2020. URL: https://huggingface.co/textattack/bert-base-uncased-yelp-polarity.

[9] Sebastian Braun and Ivan Tashev. "Data Augmentation and Loss Normalization for Deep Noise Suppression". In: *Speech and Computer*. Ed. by Alexey Karpov and Rodmonga Potapova. Cham: Springer International Publishing, 2020, pp. 79–86. ISBN: 978-3-030-60276-5.

[10] Christopher Celio et al. "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V". In: (July 2016).

[11] RISC-V Software Development. *RISC-V ISA Simulator*. https://github.com/riscv-software-src/riscv-isa-sim. Accessed: 2024-06-25. 2024.

[12] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[13] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th Annual Design Automation Conference (DAC)*. 2021.

[14] Florian Giesemann, Lukas Gerlach, and Guillermo Payá-Vayá. "Evolutionary Algorithms for Instruction Scheduling, Operation Merging, and Register Allocation in VLIW Compilers". In: *Journal of Signal Processing Systems* 92.7 (July 2020), pp. 655–678. ISSN: 1939-8115.

[15] *Gracemont Microarchitecture (online)*. 2021. URL: https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/4.

[16] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. [Online; accessed June 2024]. 2024, pp. 3–12. URL: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[17] Intel. *Sparse bert*. https://huggingface.co/Intel/bert-base-uncased-sparse-70-unstructured. Hugging Face. 2025. URL: https://huggingface.co/Intel/bert-base-uncased-sparse-70-unstructured.

[18] Intel. *Sparse Distilbert*. https://huggingface.co/Intel/distilbert-base-uncased-sparse-90-unstructured-pruneofa. Hugging Face. 2025. URL: https://huggingface.co/Intel/distilbert-base-uncased-sparse-90-unstructured-pruneofa.

[19] Ralf Karrenberg and Sebastian Hack. "Whole-function vectorization". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. USA: IEEE Computer Society, 2011, pp. 141–150. ISBN: 9781612843568.

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

[21] Konstantinos Koukos et al. "Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs". In: *Proceedings of the 25th International Conference on Compiler Construction*. CC '16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 121–131. ISBN: 9781450342414. DOI: 10.1145/2892208.2892209. URL: https://doi.org/10.1145/2892208.2892209.

[22] *KULeuven-MICAS*. 2023. URL: https://github.com/KULeuven-MICAS/snax_cluster.

[23] Chester Lam. "Popping the Hood on Golden Cove". In: *Chips and Cheese* (Dec. 2021). URL: https://chipsandcheese.com/p/popping-the-hood-on-golden-cove.

[24] Monica S. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 318–328.

[25] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *Int. Symp. Code Gener. Optim. (CGO '04)*. 2004.

[26] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *2nd Int’l Symp. on Code Generation and Optimization (CGO)*. Mar. 2004, pp. 75–88.

[27] Chris Lattner et al. “MLIR: scaling compiler infrastructure for domain specific computation”. In: *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’21. IEEE Press, 2021, pp. 2–14. ISBN: 9781728186139.

[28] Yuxiang Lei and Yulei Sui. “Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis”. In: *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings*. Porto, Portugal: Springer-Verlag, 2019, pp. 27–47. ISBN: 978-3-030-32303-5. DOI: 10.1007/978-3-030-32304-2_3. URL: https://doi.org/10.1007/978-3-030-32304-2_3.

[29] Sheng Li et al. “CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques”. In: *2011 Int’l Conf. on Computer-Aided Design (ICCAD)*. Nov. 2011, pp. 694–701.

[30] Edgar Liberis and Nicholas D. Lane. “Neural networks on microcontrollers: saving memory at inference via operator reordering”. In: *arXiv e-prints*, arXiv:1910.05110 (Oct. 2019), arXiv:1910.05110. DOI: 10.48550/arXiv.1910.05110. arXiv: 1910.05110 [cs.LG].

[31] *LICM.cpp — Loop Invariant Code Motion Pass*. LLVM Project. 2025. URL: https://llvm.org/doxygen/LICM_8cpp_source.html (visited on 12/02/2025).

[32] Dinghao Liu et al. “Improving {Indirect-Call} Analysis in {LLVM} with Type and {Data-Flow}{Co-Analysis}”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 5895–5912.

[33] Qiankun Liu, Sam Amiri, and Luciano Ost. “Exploring RISC-V Based DNN Accelerators”. In: *2024 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. 2024, pp. 1–6.

[34] *LLVM DAGMutation*. 2000. URL: https://reviews.llvm.org/D85517.

[35] *LLVM SelectionDAG*. 2000. URL: https://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process.

[36] Yaojie Lu and Sotirios G Ziavras. “Instruction Fusion for Multiscalar and Many-Core Processors”. In: *International Journal of Parallel Programming* 45 (2017), pp. 67–78.

[37] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 190–200. ISBN: 1595930566.

[38] Scott A. Mahlke et al. “Effective compiler support for predicated execution using the hyperblock”. In: *Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, November 1992*. Ed. by Wen-mei W. Hwu. ACM/IEEE, 1992, pp. 45–54.

[39] Sandya Mannarswamy and Dibyendu Das. *Learning to Combine Instructions in LLVM Compiler*. 2022. arXiv: 2202.12379 [cs.LG]. URL: https://arxiv.org/abs/2202.12379.

[40] Riccardo Miccini et al. “Dynamic nsNET2: Efficient Deep Noise Suppression with Early Exiting”. en. In: *2023 IEEE 33rd International Workshop on Machine Learning for Signal Processing (MLSP)*. Rome, Italy: IEEE, Sept. 2023, pp. 1–6. ISBN: 9798350324112. (Visited on 01/24/2024).

[41] Dorit Nuzman and Richard Henderson. “Multi-platform Auto-vectorization”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’06. USA: IEEE Computer Society, 2006, pp. 281–294. ISBN: 0769524990. DOI: 10.1109/CGO.2006.25. URL: https://doi.org/10.1109/CGO.2006.25.

[42] Dorit Nuzman, Ira Rosen, and Ayal Zaks. “Auto-vectorization of interleaved data for SIMD”. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 132–143. ISBN: 1595933204. DOI: 10.1145/1133981.1133997. URL: https://doi.org/10.1145/1133981.1133997.

[43] *ONNX GitHub Repository (online)*. 2016. URL: https://github.com/onnx/onnx.

[44] LLVM Project. *LoopVectorize.cpp*. https://github.com/llvm/llvm-project/blob/ae11c5c2c4d7ae4cba4a8e05f0c7d85b316a2cf0/llvm/lib/Transforms/Vectorize/LoopVectorize.cpp. Commit ae11c5c. 2024.

[45] *PyTorch MLP (online)*. 2022. URL: https://github.com/christianversloot/machine-learning-articles/blob/main/creating-a-multilayer-perceptron-with-pytorch-and-lightning.md.

[46] *PyTorch MNIST (online)*. 2020. URL: https://github.com/onnx/onnx-mlir/blob/main/docs/mnist_example/mnist.onnx.

[47] *AlexNet Pretrained Model (online)*. 2016. URL: https://pytorch.org/hub/pytorch_vision_alexnet.

[48] *PyTorch Vision MobileNet V2 (online)*. 2016. URL: https://pytorch.org/hub/pytorch_vision_mobilenet_v2.

[49] *PyTorch Vision MobileNet V2 (online)*. 2016. URL: https://pytorch.org/hub/pytorch_vision_mobilenet_v2.

[50] *PyTorch RESNET18 (online)*. 2016. URL: https://pytorch.org/hub/pytorch_vision_resnet/.

[51] *PyTorch RESNET50 (online)*. 2016. URL: https://pytorch.org/hub/nvidia_deeplearningexamples_resnet50/.

[52] Eric Qin et al. “SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 58–70.

[53] Chiara Riatti. "Design and Implementation of Instruction Fusion Techniques in a RISC-V Out-Of-Order Core". Open access, UPC's institutional repository. Treball Final de Grau (Bachelor's Thesis). Barcelona Supercomputing Center, Sept. 2024. URL: https://hdl.handle.net/2117/425653.

[54] *Roberta*. 2016. URL: https://huggingface.co/docs/transformers/en/model_doc/roberta.

[55] Ronny Ronen, Alexander Peleg, and Nathaniel Hoffman. *System and method for fusing instructions*. US Patent 6,675,376. Jan. 2004.

[56] André Seznec. "TAGE-SC-L Branch Predictors Again". In: *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. June 2016.

[57] Jian-Yu Shen and Shih-Wei Liao. "Evaluating and Enhancing Performance through Macro-Op Fusion Optimization with RISC-V". In: *Workshop Proceedings of the 53rd International Conference on Parallel Processing*. ICPP Workshops'24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 33–37.

[58] Sawan Singh et al. "Exploring Instruction Fusion Opportunities in General Purpose Processors". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 199–212.

[59] Sawan Singh et al. "Exploring Instruction Fusion Opportunities in General Purpose Processors". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 199–212.

[60] Cyril Six et al. "Formally verified superblock scheduling". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2022, pp. 40–54.

[61] Standard Performance Evaluation Corporation. *SPEC CPU2017*. 2017. URL: http://www.spec.org/cpu2017.

[62] SVF-tools. *SVF: Static Value-Flow Analysis Framework for Source Code*. https://github.com/SVF-tools/SVF. GitHub repository. Latest release: (2025-09-08). 2025. URL: https://github.com/SVF-tools/SVF.

[63] Sycuricon. *riscv-spike-sdk*. https://github.com/sycuricon/riscv-spike-sdk. Accessed: 2024-06-25. 2024.

[64] Harsh Thakker et al. *Combining load or store instructions*. US Patent 11,593,117. Feb. 2023.

[65] Kim-Anh Tran. "Static Instruction Scheduling for High Performance on Energy-Efficient Processors". Licentiate thesis. Uppsala University, 2018.

[66] Kim-Anh Tran et al. "Clairvoyance: Look-ahead compile-time scheduling". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 171–184.

[67] Kim-Anh Tran et al. "Clairvoyance: look-ahead compile-time scheduling". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pp. 171–184. ISBN: 9781509049318.

[68] Kim-Anh Tran et al. "Static Instruction Scheduling for High Performance on Limited Hardware". In: *IEEE Transactions on Computers* 67.4 (2018), pp. 513–527.

[69] Ofir Zafrir et al. "Prune Once for All: Sparse Pre-Trained Language Models". In: *arXiv preprint arXiv:2111.05754* (2021).

[70] Florian Zaruba et al. " Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads ". In: *IEEE Transactions on Computers* 70.11 (Nov. 2021), pp. 1845–1860. ISSN: 1557-9956.