Postiz: Extending Post-Increment Addressing for Loop Optimization and Code Size Reduction

Enming Fan*[2], Xiaofeng Guan*[1,2], Fan Hu[2], Heng Shi[1,2], Hao Zhou[2], and Jianguo Yao[1,2]

[1]*Shainghai Jiao Tong University, Shanghai, China*
[2]*Shanghai Enflame Technology Co. Ltd, Shanghai, China*
(guanxiaof,heng.shi,jianguo.yao)@sjtu.edu.cn
(morgan.fan,frank.hu,hao.zhou)@enflame-tech.com

***Abstract*—Memory access instructions with auto-addressing modes are prevalent in various Instruction Set Architectures (ISAs), yet their use in compilers remains limited. Existing methods address code optimization in one of two ways: they either focus on reducing code size, but are constrained to basic block-level optimizations and may not fully exploit architectural benefits, or they optimize loop performance, often neglecting the advantages of post-increment instructions and focusing primarily on innermost loops while leaving outer loops unoptimized.**

To address these shortcomings and meet the needs of real-world Machine Learning (ML) applications, we introduce *Postiz*, a novel post-increment loop optimization technique. *Postiz* extends post-increment optimizations beyond traditional limits, incorporating enhancements for inner loops, cross-loop regions, and nested loop structures. Through a profitability analysis, *Postiz* optimizes code judiciously, leveraging architectural advantages and reducing code size without compromising improvement made by other optimizations.

Our experiments show that *Postiz* is effective, achieving an optimization coverage of 98.04% on *MobileNet* and *BERT* benchmarks. In comparison to default LLVM optimization, *Postiz* generates approximately four times more post-increment instructions. Moreover, it reduces code size by an average of 9.45% across various platforms. These improvements represent significant advancements over current methods, showcasing *Postiz*'s potential to enhance compiler optimizations in a meaningful way.

***Index Terms*—compiler, optimization, auto-addressing, post-increment, loop strength reduce, domain-specific architecture**

I. Introduction

Many Instruction Set Architectures (ISA) of existing processors incorporate a variety of memory access instructions with diverse addressing modes. One such mode that has garnered attention is the auto-addressing mode, where the memory access is combined with address computation within the same instruction.

Despite its potential for simple micro-architecture implementation, the effective utilization of auto-addressing instructions by compilers has remained limited. Most compilers focused on simple local optimizations, which converting memory accesses and their corresponding address calculations into single auto-addressing instructions within straight-line code. A major factor restricting the expanded use of auto-addressing mode is that auto-addressing code does not offer performance advantages on super-scalar machines with out-of-order execution pipelines. Consequently, auto-addressing optimization, which is commonly regarded as a code-size optimization method in many architectures, has not kept pace with modern compiler infrastructure developments.

However, within the domain of Digital Signal Processing (DSP) systems, sometimes with stringent code size constraints, presents a compelling use case for aggressive auto-addressing optimizations. Many previous researches aimed to maximize auto-addressing instruction utilization through solutions to the NP-complete problems, including Simple Offset Assignment (SOA) and General Offset Assignment (GOA) [2]. But their heuristic approaches often incurred costs far from optimal [10]. Moreover, these methods assume some register pre-assignment, rendering long live-ranges of the address registers and neglecting trade-offs between auto-addressing and other addressing modes.

Another weakness of all existing auto-addressing optimization approaches lies in the unexplored optimization opportunities beyond basic block scope. Particularly, in this work, we identified untapped potential within loop structures, particularly those found in Machine Learning (ML) kernels, applicable to both scalar and Single-Instruction-Multiple-Data (SIMD) vector extensions. By converting memory accesses inside loop structures to auto-addressing mode, it is possible to reduce code size across different architectures.

Nevertheless, when it comes to the address computation within loops, the traditional focus of optimization techniques, such as Loop Strength Reduction (LSR), which utilizes the mathematical tool Scalar Evolution [22] to formalize induction variable address computation, aims to minimize register usage for improved efficiency within innermost loops. However, this emphasis does not consider code size as a metric and sometimes misses optimization opportunities for exploiting hardware features and reducing the code size of outer loops.

In response, we present a novel loop transformation method, called "*Postiz*". This technique leverages post-increment instructions within nested loops, commonly found in machine learning scenarios, to address both the practical application of post-increment addressing and the code size optimization opportunities. *Postiz* enhances existing straight-line optimizers by applying post-increment transformations across loop regions

*These authors contributed equally to this work.

Table I: ISA Support for Different Auto-Addressing Categories Across Various Architectures

Vendor	Architecture	SIMD Ext.	Pre-increment		Post-increment		Note
			Scalar	Vector	Scalar	Vector	
IBM	POWER	Altivec	×	×	✓†	×	General Purpose, Servers
ARM	ARM/AArch64	NEON	✓	✓	✓	✓	General Purpose, Embedded
Motorola	M68k	-	✓	×	✓	×	Embedded, Legacy
Microchip	AVR	-	✓	×	✓	×	Embedded, Industrial Control
TI	TMS320	-	✓	×	✓	×	DSP, Telecommuication
ADI	Blackfin	EPIS	×	×	✓	×	DSP, Embedded

† Only *value-strided* is supported

Note: if not specified, *value-strided* and *register-strided* modes are both supported.
Note: *value-stride* may be limited to one or more fixed values.
Note: *pre-decrement* and *post-decrement* modes are treated as variants of *pre-increment* and *post-increment* modes.

and nests. Additionally, a profitability assessment process ensures that *Postiz* is applied only when necessary, enabling effective trade-offs with other optimizations like LSR. Benchmarking *Postiz* against *MobileNet* [9] and *BERT* [6], which are models used for machine learning, demonstrates the significant code size reduction compared to LLVM existing straight-line optimizations, both in *AArch64* and in our domain-specific architecture (DSA) hardware for ML workload accelerating.

We conclude our contribution below:

- We investigated the advantageous application of auto-addressing instructions to loops, extending beyond conventional optimization boundaries.
- We designed and implemented the novel loop optimization method *Postiz* within LLVM, generating post-increment code by analyzing nested loops. Notably, it achieves an optimization coverage of 98.04% on *MobileNet* and *BERT* benchmarks, generates approximately four times more post-increment instructions than LLVM default, and reduces code size by an average of 9.45% across various platforms. These results represent a significant advancement compared to existing LLVM auto-addressing optimizations.
- We devised a profitability analysis that selectively invokes the post-increment transformation based on different metrics, ensuring optimization only when warranted.

The subsequent chapters of this paper are structured as follows: In Chapter II, we provide background on auto-addressing mode and outline our motivation for *Postiz*. Chapter III details the overall workflow, including the memory address analysis, profitability assessment, post-increment transformation, and more. Experimental settings and results are presented in Chapter IV, followed by a discussion of related work in Chapter V. We conclude in Chapter VI.

II. Background and Motivation

A. Memory Accesses: The Auto-Addressing Mode

Compared to other memory access modes, auto-addressing enables memory access instructions to modify the register in parallel with memory accesses. While it is less flexible for random memory addressing, auto-addressing is particularly well-suited for sequential memory access scenarios, since it does not cost extra instruction to recompute the addresses.

We categorize auto-addressing instructions based on different traits that influence their behaviors:

- **Pre-Increment** vs. **Post-Increment**: Dictated by the order of memory address computation and modification, *pre-increment* addressing alters the address register before address computation while *post-increment* does the reverse.
- **Value-Strided** vs. **Register-Strided**: *Value-strided* instructions adjust memory addresses based on a compile-time known stride value embedded within the instruction. In contrast, *register-strided* mode utilizes a register operand for address modification, accommodating larger or runtime-defined values.
- **Scalar-Based** vs. **Vector-Based**: *Scalar-based* addresses single data elements, while *vector-based* suits SIMD data types, functioning on larger vectors.

As investigated, the detailed ISA support by different vendors and architectures are listed in Table I. It is observed that auto-addressing is widely supported by processors used in general purposes, embedded, and DSP areas. The most widely supported auto-addressing category is the post-addressing, scalar-based auto-addressing mode. As SIMD architectures dominate the exploitation of instruction-level parallelism (ILP) in many parallel programs, some ISAs with vector extensions, such as Arm NEON, also feature the vector-based auto-addressing mode.

In this work, we emphasize generating code for memory access instructions using the post-increment addressing mode. We utilize either the *value-strided* or *register-strided* mode to meet distinct needs, primarily catering to ML kernel functions that are highly likely to be vectorized.

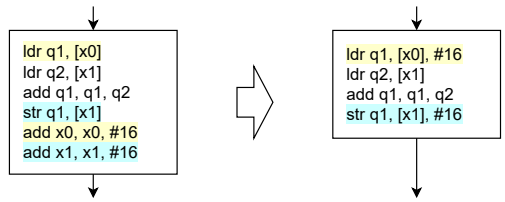
B. Merits of Post-Increment Addressing

Although supported by various hardware, the utilization of auto-addressing instructions is quite limited. As programming languages like C do not provide direct abstractions for auto-addressing, these instructions are typically generated by the compiler, usually as a result of specific optimizations.

One direct compiler optimization to generate auto-addressing instructions is the simplification of straight-line code. Architectures with auto-addressing capabilities often combine a memory access instruction and an associated address computation instruction into a single auto-addressing instruction using a peephole instruction combiner.

Figure. 1a illustrates an example of this fusion, where AArch64's ldr load instruction and str store instruction are

(a) Simplify the straight-line code



(b) Simplify the loops

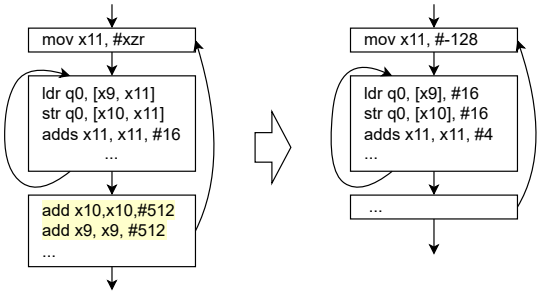


Figure 1: Apply Post-Increment Transformation to Optimize the Code Size

replaced with the corresponding add instructions with post-increment load and store. Notably, while this transformation may not consistently reduce cycles, especially in advanced pipeline setups with multi-issue and out-of-order execution, it always optimizes code size. Consequently, many compilers apply such local optimizations without considering address increments between basic blocks. However, as observed, particularly in loops of many compute-intensive kernels, there remain unexploited post-increment optimization opportunities.

Loop structures, which comprise multiple basic blocks and typically manipulate large amounts of data, often involve address calculation that can be optimized using post-increment instructions. However, existing compilers rarely apply post-increment transformations across basic blocks' boundaries.

Figure 1b showcases another example of applying post-increment transformation to simplify a loop. In this case, registers $x10$ and $x9$ serve as the bases of different memory addresses, with updates occurring at the loop latch. Notably, the data accesses are uniformly strided (always increasing by 16), both within and between loop iterations. Therefore, transforming the code to use post-increment form eliminates the address updating (add) instructions in the loop latch, thereby reducing the code size of the loop structure.

C. Post-Increment Addressing in ML Kernels

Many ML kernels are highly compute-intensive, typically operating on multi-dimensional tensors as inputs and processing chunks of data iteratively. Figure 2 showcases a simple Matrix Multiplication (Matmul) kernel and its corresponding assembly code with and without post-increment instructions generated. Note that the compiler has auto-vectorized code by grouping each 4 elements into a single vector register.

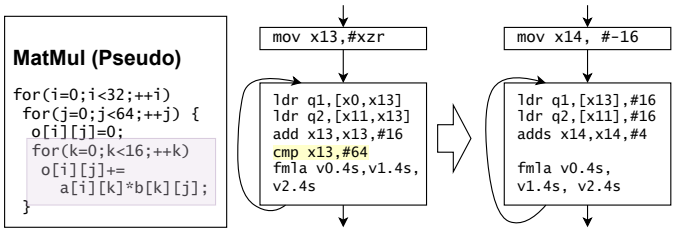


Figure 2: Implementations of Matrix Multiplication With and Without AArch64 Post-Increment Loads.

The assembly code for the innermost loops using 'base-offset' addressing is shown in the middle, while the code using post-increment addressing is shown on the right.

Notably, AArch64 add instruction sets the Z-flag utilized by $b.ne$ branch instruction when the result is 0. Therefore, transforming code using post-increment addressing results in fewer total instructions. It eliminates the need for the cmp instruction required to check the loop boundary in the 'base-offset' version, which is usually generated with compiler optimization focusing on the induction variable ($x13$) sharing, such as *loop strength reduction* (LSR). These optimizations aim to save registers by combining the loop iteration counter and the offset value of the array. While this approach minimizes register pressure within the innermost loop, it may underutilize architectural features. Consequently, auto-increment addressing exhibits an advantage in this scenario.

Furthermore, current loop optimizations applied in production primarily focus on the innermost loops. Optimizations like LSR are execution-time-oriented and do not prioritize the code size optimization of outer loops. This strategy can leave outer loop nests unoptimized, generating more code than expected.

Figure 3 shows a computation kernel implementation of a neural network layer and the associated sophisticated loop transformations. Loop fusion, tiling, and unroll-and-jam optimizations have been applied, which leaves substantial code in the outer loop nests. In the right-most code snippet of Figure 3, the code highlighted in green could result in an unexpectedly large binary size, as observed in the current LLVM implementation. This is due to the unoptimized address calculation code. This underscores the necessity of optimizing the code size of outer loop nests.

D. Scalar Evolution Analysis in LLVM

Scalar Evolution (abbreviated as "*SCEV*") is a mathematical framework designed to portray the evolution of scalar values/-variables across iterations of a loop. Compilers like LLVM utilize *SCEV* analysis to formalize the loop induction variables (*IV*) changes related to loop execution symbolically. In this framework, a simple loop IV is expressed as a *Recurrence*. In annotation, it consists of the expression of initial value "*base*", the operation "*op*" performed for value modification, and the expression of modification value "*step*" annotated as $\{base, op, step\}$ to reflects recurring alteration of the IV within a *for-loop* structure. In scenarios involving nested loops, a

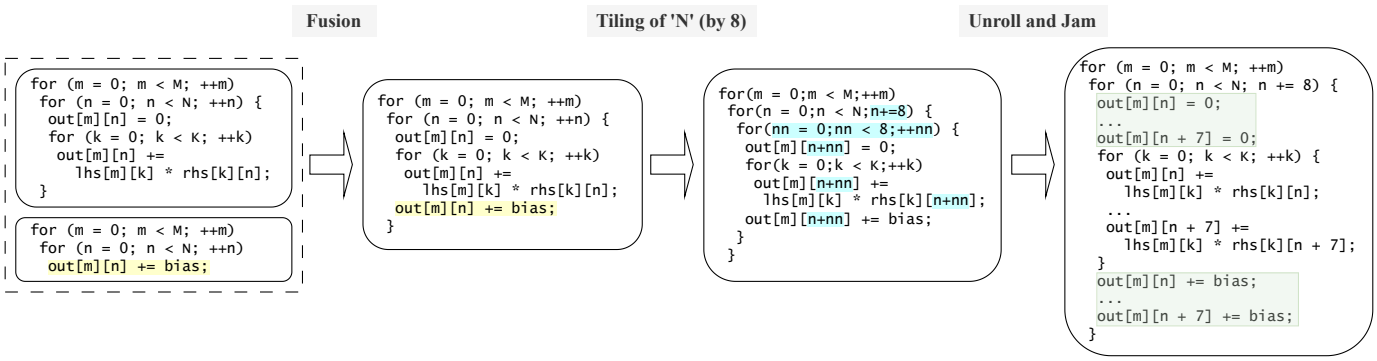


Figure 3: Computation Kernel Implementation of A Neural Network Layer and the Associated Loop Transformations

more complicated *Chain of Recurrence* (abbreviated as "*CR*") comes into play, such as $\{\{base, op_1, step_1\}, op_2, step_2, ...\}$, tailored to the specific nesting depth.

SCEV analysis widely serves LLVM transformations, including but not limited to loop vectorization, SLP vectorization, and loop strength reduction. In our work, *SCEV* analysis also establishes the foundations of related analysis and transformation. Therefore, in the following sections, we directly utilize LLVM terminologies and their abbreviations, such as *SCEV*, *CR*, and *IV*, etc. when necessary, to explore the topic more comprehensively.

III. Methodology

A. The Overall Workflow

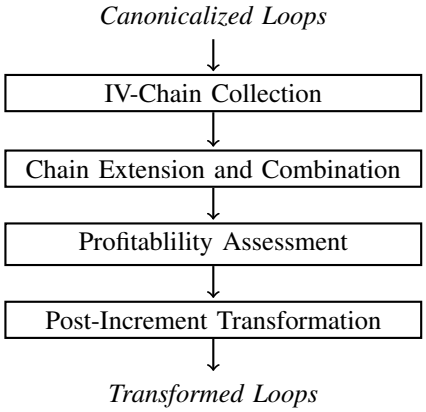


Figure 4: The Overall Workflow of *Postiz* Analysis and Transformation

Postiz is built on top of LLVM as a pass within the LLVM pipeline. To enable post-increment transformation for both straight-line code and nested loops, it utilizes existing LLVM analyses such as loop info and SCEV, and relies on LLVM transformations to provide canonicalized loops as inputs. Figure. 4 demonstrates the overall workflow, which consists of four phases.

The process begins by analyzing the innermost loops, which depicted as "*IV-Chain Collection*" phase in the figure. *Postiz* first collects memory access instructions within the loop and groups the associated induction variables into different chains, referred to as "IV-Chains", based on the SCEV expressions in each loop. The instructions corresponding to the IVs within these chains are candidates for transformation into post-increment form. However, as described in Section II, optimization opportunities are not limited to the innermost loops. The "*Chain Extension and Combination*" phase then attempts to convert the *IV-Chains* by grouping IVs across loop nests and different loop regions whenever possible. This approach allows *Postiz* to excel beyond existing local instruction combining methods and address common patterns seen in ML kernels.

Next, a "*Profitability Assessment*" is applied to filter only the *IV-Chains* considered beneficial based on various metrics. This assessment aims to avoid negative transformations as much as possible, including not only negative impacts on code size but also side effects that could disrupt other optimizations. Upon completing the analysis, *Postiz* proceeds to transform each candidate by applying newly generated stride values to the post-increment addressing form for the instructions, as shown in "*Post-Increment Transformation*" in the figure.

Note that while the functionality of *Postiz* is relatively independent, it is positioned almost at the end of the LLVM IR optimization pipline. Given that auto-vectorization is expected in real-world ML compute kernels, this positioning allows *Postiz* to transform both scalar and vector data access instructions.

B. IV-Chain Collection

1) Select Instructions in Innermost Loops: As *Postiz* focuses on ML kernels described in Section II-C, in this early phase, it targets the canonicalized innermost loop with a single latch and a single exit, which are usually observed in ML kernels. It collects the load and store instructions by traversing all the instructions in the *necessary basic blocks* from the loop header to the loop latch, where the *necessary basic blocks* between two basic blocks "A" and "B" are defined as the basic blocks from the intersection of dominators of the "A" and post-dominators of "B".

Memory access instructions inside other basic blocks are simply ignored. This is because such instructions are partially ordered in execution, making the computation of post-increment strides undecidable at compilation time. Despite

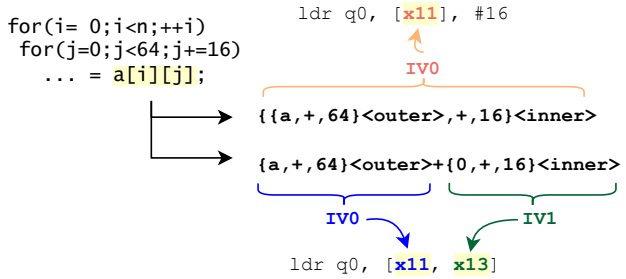


Figure 5: Different SCEV Expressions of the Same Memory Access

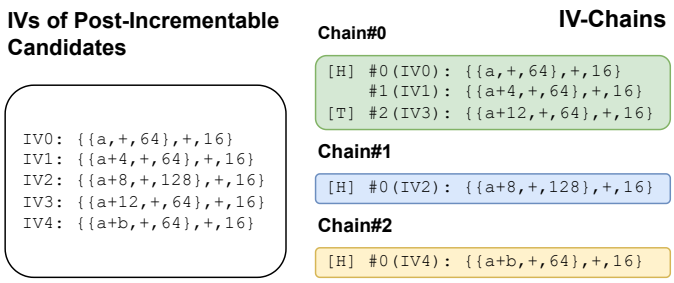


Figure 6: Group Induction Variables into Different IV-Chains

these limitations, *Postiz* addresses most ML kernels we studied, as branches are rarely found inside the innermost loops.

2) IV and CR of the Memory Access Instruction: Using LLVM's SCEV analysis, *Postiz* can derive SCEV expressions for memory address computation expressions of the collected instructions. However, the relationship between address computation expressions and SCEV expressions is not always one-to-one. As illustrated in Figure 5, accessing data inside the 2-dimensional array "*a*" can be represented either as a composition of two CRs or as a single CR with a depth of 2. The former results in two IVs, potentially leading to generating target load instructions that use base-offset addressing mode. The latter results in a single IV, which may be used for post-increment addressing mode, as shown in the figure.

To simplify our analysis, we focus on SCEV expressions with their purely recursive CRs. Consequently, the IV whose SCEV expression is the recursive CR corresponds directly to the unique address computation expression of a load/store instruction. This enables us to establish a one-to-one correspondence between IVs and their associated instructions. For brevity, in the subsequent paragraphs, we may refer to the 'IV of an instruction' to represent the address computation of that instruction, and 'CR of an instruction' to represent the unique SCEV expression that *Postiz* cares about.

3) Group IVs Into Different Chains: The ultimate target of this phase is to collect instructions that access nearby addresses, meeting the requirements of post-increment addressing. We name the IVs of these instructions as "*post-incrementable candidates*". Note that since we collect instructions from the loop header to the loop latch across the *necessary basic blocks*, the candidates are processed in program order. Figure 6 illustrates an example of grouping *post-incrementable candidates* into different chains. In this example, five IVs are collected as candidates, with their corresponding CRs also shown. *Postiz* considers `IV0` and `IV1` suitable for chaining together because their CRs have a constant delta value of 4. This means that, throughout all iterations, the addresses accessed by instructions using these IVs maintain a consistent gap, making them ideal for post-increment transformation. However, `IV2` cannot be appended to `Chain#0` because the *step* values of its outer loops differ from those of `IV0`, typically indicating different iteration patterns in the innermost loops. Similarly, `IV4` cannot be added to `Chain#0` because the subtraction `IV4 - IV1` results in a non-trivial value of `b - 4`. This implies that the memory address gap is a runtime variable, potentially leading to high costs if post-increment transformation is forced.

As a result, `IV2` and `IV4` are placed into separate chains, `Chain#1` and `Chain#2`, respectively. Each IV-Chain contains a *Head* (annotated as `[H]`) IV, and a *Tail* (annotated as `[T]`) IV. The corresponding instructions of the IVs inside a chain remain in program order, which facilitates later post-increment stride calculation.

More generally, an IV 'iv' can be grouped into an existing chain 'N' only when:

$$SCEV_{iv} - SCEV_{h(N)} = SCEV_{Const}$$

is satisfied, where $h(N)$ represents the head IV of `Chain#N`. And `SCEVConstant` stands for the loop invariant value, which can be either a compile-time literal or a variable unchanged when executing the nested loop.

C. Chain Extension and Combination

1) Chain Extension for Nested Loops: To minimize code size in nested loops, *Postiz* extends the IV-Chain collection to the basic blocks of outer loops. This approach targets the optimization shown in the rightmost code snippet of Figure 3, where the memory accesses in outer loops can optimized with post-increment addressing.

Unlike IV-Chain collection in innermost loops, *Postiz* handles outer loops by selecting basic blocks differently. It collects instructions from the *necessary basic blocks* between the outer loop's header and the inner loop's pre-header. Similarly, it gathers instructions from *necessary basic blocks* between the immediate successor of the inner loop's exit block and the outer loop's latch. The left diagram in Figure 7 illustrates an example of block selection when extending IV-Chains to two levels of outer loop nests. Any instructions outside the colored blocks are ignored.

Postiz use the same method as described in Section III-B3 to fill *IV-Chains* with supplemental blocks provided. Notably, it still follows the program order to work.

2) Chain Combination for Remainder Loops: In some scenarios, different inner loops are executed sequentially, and memory accesses exhibit adjacency among these loops. This

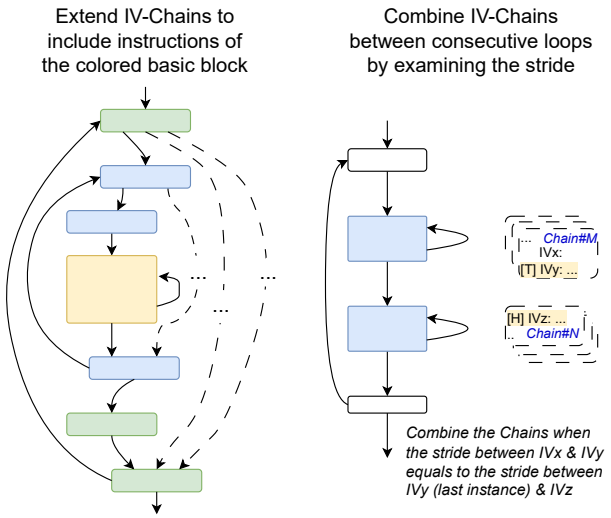


Figure 7: IV-Chain Extension and Combination

is sometimes a direct result of strip-mining optimization of the innermost loops. A "remainder loop" comes after the "main" loop in such situations.

To address the post-increment optimization opportunity for this scenario, *Postiz* attempts to connect IV-Chains if two loop regions exhibit a dominate/post-dominate relationship. Assuming the IV-Chain of two loop regions are "M" and "N", where the loop region of "M" comes before the region of "N" in program order, they can be combined when the bellow condition is satisfied:

$$SCEV_{h(M)} - SCEV_{t(N)} = SCEV_{Const}$$

In this equation, `h(M)` represents the head of IV `Chain#M` and `h(N)` represents the tail of IV `Chain#N`. When the relationship is proved, *Postiz* marks "M" and "N" as combined and computes the associated *cross-region stride*. The right figure of Figure 7 demonstrates an example of such a combination. However, it is only beneficial to combine loop regions when the *cross-loop stride* is equal to the *iteration stride*. We will next explain different stride calculations to make it clear.

3) Strides Calculation: Stride values are required to convert memory access instructions into post-increment addressing mode. In *Postiz*, we categorize the post-increment strides of an IV-Chain into three types, including:

- **In-Region Stride**. An *in-region stride* is the delta of two IVs. It represents the address delta between the corresponding instructions.
- **Iteration Stride**. The *iteration stride* is associated with the tail IV, representing the address difference between the final memory access of the last inner iteration and the next memory access. The stride value may vary depending on the loop nest level of the subsequent accessed addresses.

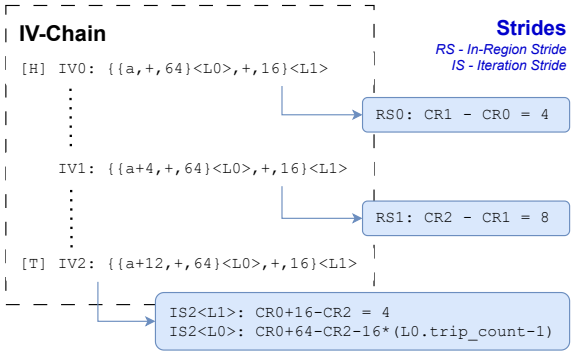


Figure 8: *In-Region Stride* and *Iteration Stride* Calculation

- **Cross-Region Stride**. An *cross-region stride* is defined as the address delta between the last and first memory accesses from loop regions that can be combined.

The *in-region stride* can be directly obtained by subtracting the CR expressions between IVs. Specifically, the *in-region stride* between two IVs is calculated using the following equation:

$$RS_i = C_{i+1} - C_i \tag{1}$$

In Equation 1 above,

- C_i represents the ith CR expression in the IV-Chain.
- RS_i represents the *in-region* stride value of ith IV.

For ML workloads, it is likely that memory accesses have identical *in-region strides* between IVs in an IV-Chain, as the instructions are typically generated through compiler optimizations. Even in cases of manual memory management, ML workloads tend to exhibit predictable, non-random access patterns, making consistent *in-region strides* likely. Therefore, post-increment optimization is especially beneficial for ML kernels.

However, calculating the stride of the tail IV is more complex, as it spans multiple iterations. Furthermore, since *Postiz* handles nested loops, the *iteration stride* consists of multiple values, each corresponding to a specific loop nesting level. In general, the *iteration stride* of the tail IV for a given loop nesting level can be calculated using Equation 2:

$$IS[k] = C_H - C_T + I[k] - \sum_{i=k+1}^{n} T[i] * I[i] \tag{2}$$

In this equation,

- k represents the loop nesting level, with the outermost loop as 0.
- $IS[k]$ is the *iteration stride* value of loop nest $L[k]$.
- C_H and C_T are the CR expressions of the head and tail IVs in the chain.
- $I[k]$ denotes the *step* value for loop nest $L[k]$.
- $T[i]$ denotes the trip count for loop nest $L[i]$.
- Loops from $k+1$ to n are inner loops of loop nest $L[k]$.

Figure 8 illustrates the calculation of both *in-region stride* and *iteration stride* through an example. In this case, a set of

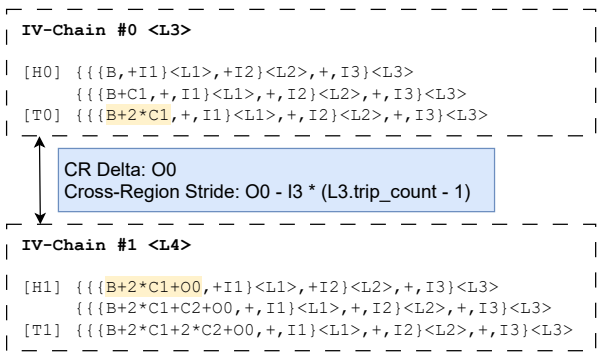


Figure 9: *Cross-Region Stride* Calculation

IVs forms an IV-Chain, with `IV0` as the head and `IV2` as the tail. The *in-region strides* (`RS0` and `RS1`) correspond to `IV0` and `IV1`, while the *iteration stride* (`IS2`) corresponds to `IV2`. The calculation of `RS0` and `RS1` is straightforward, involving simple subtraction between CR expressions. However, calculating `IS2` requires a more nuanced approach, as it involves subtracting the head from the tail, but the address access by head IV is from the next iteration.

Furthermore, with two levels of loop nests, the *iteration stride* of `IV2` consists of two values: `IS<L1>` for loop `L1`, and `IS<L0>` for loop `L0`. For the outer loops stride `IS<L1>`, it requires subtraction of the strides accumulation of `L0.trip_count - 1` inner iterations.

Moreover, the IV-Chain Combination introduces additional considerations for post-increment stride computation. For instance, in Figure 9, loops `L3` and `L4` are subloops of the parent loop `L2`. To compute the *cross-region* stride between `L3` and `L4`, the strides from all `L3` iterations must be accumulated. More generally, the *cross-region stride* ($CS[j][k]$) of the j^{th} IV-Chain at loop nesting level k is calculated as follows:

$$CS[j][k] = O[j] - \sum_{i=k+1}^{n} T[i] * I[i] \tag{3}$$

where $O[j]$ is the difference between the head CR of the $(j+1)^{th}$ IV-Chain and the tail CR of the j^{th} IV-Chain. Other terms follow from Equation 2.

D. Profitability Assessment

While earlier phases of *Postiz* focused on basic profitability conditions, the trade-off between applying post-increment transformations or not is crucial for effective optimization.

LLVM's existing LSR optimization employs an advanced solver to minimize address computation costs, particularly within innermost loops. To ensure *Postiz* performs on par with LSR, we compare both within innermost loops, evaluating optimization based on register usage and computation strength.

1) Register Usage Assessment: We model *Register Usage(RU)* for a innermost loop as follows:

- RU_{lsr}: The sum of CRs within all SCEV expressions in a loop. This reflects the induction variables and thus the registers required to calculate addresses, control loops, etc., across loop iterations. For example, if multiple SCEV expressions exist in a loop as follows:

$$\textit{SCEV \#0: } \overbrace{\{\{a,+,64\}<\text{L0}>,+,16\}<\text{L1}>}^{IV2} + \overbrace{\{0,+,4\}<\text{L2}>}^{IV1}$$

$$\textit{SCEV \#1: } \overbrace{\{\{a,+,64\}<\text{L0}>,+,16\}<\text{L1}>}^{IV2} + \overbrace{\{b,+,16\}<\text{L3}>}^{IV3}$$

  LSR attempts to find shareable CRs (like $IV2$) to minimize the IV count (e.g., to 3).
- RU_{postiz}: The count of *IV-Chains* plus CRs/IVs in SCEV expressions but not in *IV-Chains*. Each IV-Chain represents a single register usage across iterations, other IVs are calculated similar to RU_{lsr}. Additionally, if any variable strides are required, one extra register usage is added to the RU_{postiz}.

Postiz transformation is deemed profitable when $RU_{postiz} < RU_{lsr}$. If they are equal, further assessment of computation strength is required.

2) Computation Strength Assessment: We define the *Computation Strength (CS)* of a set of IVs(IS) as follows:

$$CS(IS) = |IS| + \sum_{i \in IS} NL(i)$$

Where:

- The function $NL(i)$ retrieves the nesting-level of IV i.
- $|IS|$ represents the size of the set, reflecting the cost of updating IVs in loop iterations (1 per IV).
- $\sum_{i \in IS} NL(i)$ reflects the initialization cost of IVs at different nesting levels.

Let IS_{loop} denote the IVs produced from all SCEV expressions in the loop. We define:

$$CS_{lsr} = CS(IS_{loop})$$

Next, let $Chains$ be the set of all IV-Chains in the loop. We define:

$$CS_{postiz} = CS(IS_{loop} - IVSet(Chains)) + \sum_{c \in Chains} IStride(c) + HStride(c)$$

Where:

- $IVSet(Chains)$ retrieves the set of IVs from the $Chains$.
- $IStrides(c)$ represents the cost of cross-loop-nest fix-ups when there is more than one *iteration stride*.
- $HStride$ reflects the cost of handling strides that exceed the target instruction's max value, necessitating a *register-based* post-increment.

If $CS_{postiz} \leq CS_{lsr}$, *Postiz* considers the transformation profitable and triggers post-increment transformation.

E. Post-Increment Transformation

After profitability accessments with computed post-increment strides, *Postiz* has determined the instructions to transform. It rewrite all instructions whose corresponding IVs are considered as profitable into post-increment addressing

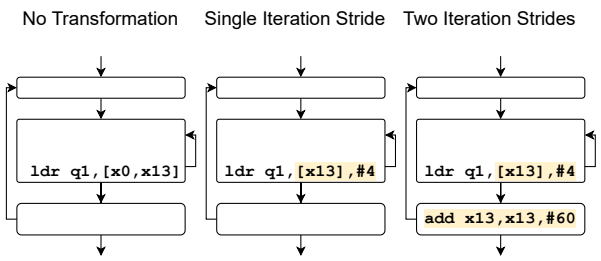


Figure 10: Transform Memory Access Instructions into Post-Increment Addressing Mode.

mode if possible, or generate IR what can be easily converted to post-increment loads/stores by later instruction combiner (implemented in AArch64).

The figure on the left of Figure 10 illustrates a nested loop with the last load instruction appearing in the loop body. No post-increment transformation has occurred. The middle figure depicts a scenario where all iteration strides are uniform. In this case, no further action is required for the outer loops; *Postiz* simply rewrites the instruction to a post-increment form to complete the transformation.

The figure on the right presents a situation with two different iteration stride values for different loop nest levels. Here, *Postiz* generates address fix-up instructions in the latch of the outer loop. While this address fix-up reduces the gain in optimizing code size, it ensures the correctness of the post-increment transformation. Similarly, if there are more than two iteration strides, the instruction rewrites may add more fix-up code into different loop latches. However, in such cases, the transformation may not be profitable.

IV. Experiment

A. Research Objectives

The primary objective of this study is to evaluate the optimization and coverage improvements achieved by *Postiz* in comparison to the default settings of the LLVM compiler. Specifically, our research objectives are as follows:

- **Analyze Code Size Reduction**: Measure the reduction in binary size achieved through the *Postiz* method and compare it to the LLVM default settings.
- **Examine Instruction Changes**: Investigate the effect of *Postiz* on instruction optimizations, with a focus on nested loop structures, and compare these results to the default LLVM behavior.
- **Estimate Register Usage**: Evaluate the changes *Postiz* introduces to the count of general-purpose registers used, approximating its potential impact on register pressure in benchmark functions.
- **Evaluate Benchmark Coverage**: Determine the breadth of coverage of *Postiz* method by identifying the number of cases impacted by loops and functions where post-increment instructions are generated and optimized, compared to the default LLVM implementation.

By pursuing these research objectives, we aim to provide a comprehensive understanding of the benefits offered by *Postiz* and its influence on the internal workings of the compiler.

B. Experiment Settings

Postiz optimization is integrated into LLVM for both *AArch64* (**Platform #1**) and a *DSA* (**Platform #2**) target [24]. As described in Section II, *AArch64* architecture with NEON SIMD vector extension support allows post-increment addressing for both scalar and vector types. We focus on the latter for *Platform #1*. The *DSA* (domain-specific architecture) is a hardware accelerator designed for ML workloads, supporting post-increment addressing for its SIMD instructions. However, due to the limited encoding bits in its instructions, the *DSA* lacks support for immediate values as addressing offsets, despite still supporting register-based addressing modes. This limitation makes post-increment the preferred choice for code generation on *Platform #2*.

We have conducted experiments with different setting on different platform.

1) Platform #1: We conduct the comparison among three different settings, including:

- **base**: Default LLVM-13 with AArch64 support, except for straight-line auto-addressing optimizations, like `aarch64ldstOpt` and `DagCombiner` are disabled.
- **llvm**: Similar to *base* but without turning off any optimizations. *.
- **postiz**: Similar to *llvm* but with *Postiz* optimization applied ahead of straight-line auto-addressing optimizations.

2) Platform #2: We conduct similar comparison between two different settings, including:

- **llvm**: Default LLVM-11 with DSA target code generation support. It is used as the baseline.
- **postiz**: Similar to *llvm* but with *Postiz* optimization.

In this experiment, we use LLVM `llc` tooling as the driver for compilation for all the above settings. We apply the `-O2` optimization level to both platforms. For *Platform #1*, we use the `-mtriple=aarch64 -mcpu=cortex-a72` options as the target options, while for *Platform #2*, we use the default options. `llc` option "`-stats`" is utilized for statistics collection. Further, `llvm-size` tool is employed for code size analysis.

C. Benchmark Programs

We selected the ML model *MobileNet* and *BERT* benchmark for our study due to their significance in deep learning and their computational complexity. *MobileNet* is a *Convolutional Neural Network* (CNN) model used for computer vision, focusing on efficient convolutions and other commonly used kernels for image classification in resource-constrained environments. *BERT*, based on the *Transformer* architecture, is designed for natural language understanding and represents state-of-the-art attention structures. Together, these models

*A subtle change is made to let *AArch64* `DagCombiner` to prioritize post-increment over pre-increment in optimization

Table II: Computation Kernel Information of *MobileNet* and *BERT*

Kernel Category	Cnt	Label	Loop Depth
MobileNet: 34 Kernels			
Fused Conv2d	20	*#4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20, #21, #22, #23*	*2/3/4*
Fused Depthwise Conv2d	10	*#25, #26, #27, #28, #29, #30, #31, #32, #33, #34*	*2/3*
Average pooling	1	*#3*	*2*
Dense	1	*#24*	*2*
Layout Transform	1	*#1*	*1*
Others	2	*#2*	*0*
BERT: 21 Kernels			
Batch Matmul	5	*#5, #6, #7, #8, #9*	*3/4*
Transpose & Broadcast	4	*#14, #15, #16, #17, #18*	*2/3*
Dense	1	*#10*	*2*
Softmax	1	*#11*	*2*
Arithemtic	7	*#2, #3, #4, #13, #19, #21*	*1/2*
Others	3	*#1, #12, #20*	*0*

cover a wide range of kernel types found in diverse ML workloads, spanning computer vision, NLP (Natural Language Processing), and emerging LLMs (Large Language Models).

We utilize TVM to generate LLVM IR kernels for *MobileNet* and *BERT*. In TVM, we apply loop transformations, including vectorization, unrolling, and more, to optimize memory locality, exploit instruction-level parallelism, and reduce memory access latency on both platforms. The resulting IR is then provided to `llc` for *Postiz* optimizations and additional lower-level code optimization and generation.

TABLE II illustrate the detail of different kernel categories. In *MobileNet*, 31 out of 34 kernels are *fused 2D convolutions*, which are highly compute-intensive and often involve deeply nested loops, typically with a nesting depth of up to 4. Moreover, all loop bounds are constants, allowing the trip count and post-increment strides to be determined at compile time. While the *Batch MatMul* kernels in *BERT* share a similar structure, the computation within deeply nested loops is less dominant compared to *MobileNet*. In general, most cases involve a loop nesting level greater than one, which *Postiz* is expected to optimize better.

D. Results

1) Code Size Comparison: In this metric, we measure the size of the .text section to compare code size differences in binaries. Figure 11 illustrates a comparison of normalized code sizes between *postiz* (bars) and *llvm* (lines). *Platform #1* uses *base* as its baseline, whereas *Platform #2* uses *llvm* as its baseline since there is no LLVM straight-line optimizer for this target. The accompanying chart showcases the proportion of the enhanced, unchanged, and degraded cases for the two benchmarks.

As observed, *postiz* achieved an average code size reduction of 9.45% compared with *llvm*. Specifically, on *Platform #1*, *postiz* reduced code size by 5.31% for *MobileNet* and 6.73%

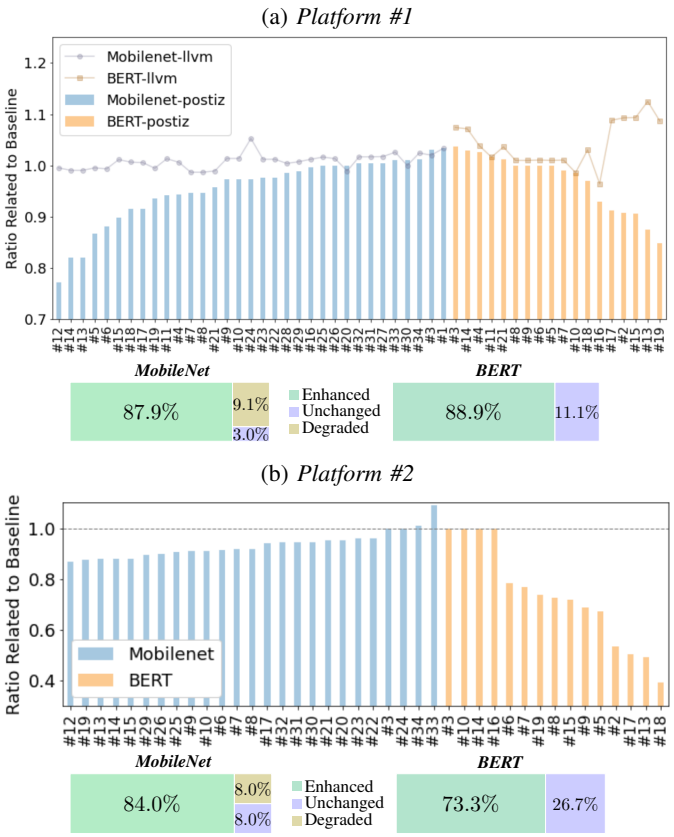


Figure 11: Comparison of Code Sizes Between *llvm* and *postiz* for the *MobileNet* and *BERT* Model (Normalized, Excluding Unaffected Cases)

for *BERT*. On *Platform #2*, *postiz* reduces code size by 6.63% for *MobileNet* and 26.55% for *BERT*. From the perspective of test cases, on *Platform #1*, *postiz* improved code size reduction of 87.9% cases for *MobileNet* and 88.9% cases for *BERT*, compared with *llvm*. On *Platform #2*, improvements were seen in 84.0% of cases for *MobileNet* and 73.3% of cases for *BERT*. These results underscore the effectiveness of *postiz* in code size reduction over existing straight-line optimizers.

Certain cases, such as `MobileNet#12` and `BERT#13`, exhibited significant reductions in code size. Further investigation revealed that the improvement in `MobileNet#12` resulted from post-increment transformation of the outer loops, which the *LLVM LSR* optimization missed, as it focuses solely on the innermost loop. In the case of `BERT#13`, *postiz* generated fewer induction variables than *llvm*, reducing the number of induction variable updates required across iterations, leading to a significant decrease in code size. Additionally, *postiz* achieved a higher reduction ratio in *BERT* on *Platform #2* because, without *Postiz* optimizations, *LLVM* generated more induction variables as offset registers. Since the platform lacks a "base-offset" addressing mode, the compiler frequently refilled offset registers, which increased the code size. This underscores the importance of *Postiz* for optimizing such ISAs.

Moreover, we conducted a comparison between the scenar-

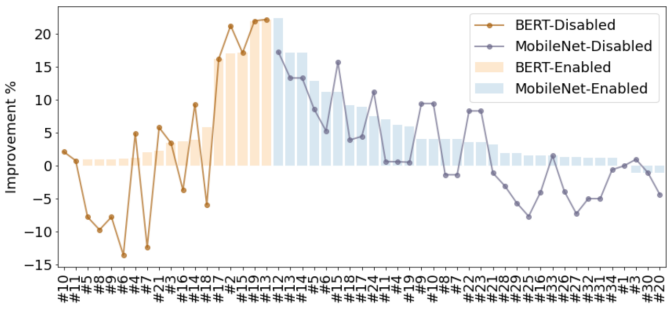


Figure 12: *Platform #1* - Comparison of Code Size Improvements With Profitability Assessment Enabled And Disabled

ios where *Profitability Assessment* is enabled and disabled on *Platform #1* to examine its effectiveness. Figure 12 illustrates the improvement ratio of the two results compared to the *llvm* baseline for *MobileNet* and *BERT*. While not always superior in all cases, enabling *Profitability Assessment* results in an average additional code size reduction of 2.74% for *Postiz*. Notably, without *Profitability Assessment*, *postiz* experiences degraded optimization in 39.22% cases (20 cases), compared to only 5.88% cases (3 cases) with *Profitability Assessment* enabled. Thus, profitability assessment enhances both the overall optimization degree and the stability of optimizations.

2) Instruction Count Change Comparison: In this metric, we measure the instruction counts within loops. We performed a detailed analysis of instruction count changes across two different scopes: the innermost loop and the entire loop nest, as illustrated in Figure 13. The lines in the figure represent the reduction ratio of instruction counts for the innermost loops, while the bars represent the reduction ratio for the entire loop nests. The reduction ratio is calculated by comparing the data between *postiz* and *llvm*.

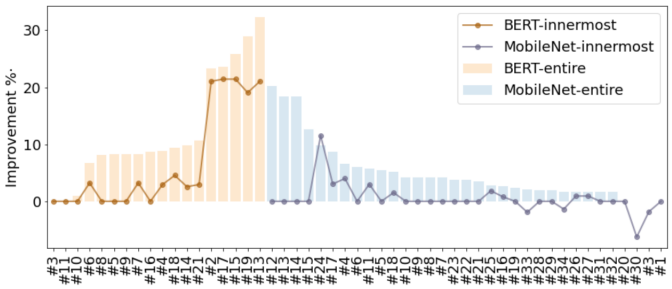
On *Platform #1*, *postiz* achieved an average reduction of 2.67% in instruction counts for innermost loops and an average reduction of 9.76% for entire loop nests. On *Platform #2*, *postiz* achieved an average reduction of 11.73% in instruction counts for innermost loops and an average reduction of 13.61% for entire loop nests. Clearly, *Postiz* is capable of effectively reducing instruction counts in both innermost and outer loops.

Table III: Instruction Count Reductions: *postiz* over *llvm*

Reduction/Total Count	Platform #1		Platform #2	
	BERT	MobileNet	BERT	MobileNet
Innermost Loops	24/586	7/1823	1081/3718	446/4167
Entire Loops	94/899	206/4169	1090/4122	866/8176

TABLE III provides the reduction and instruction counts. The instruction reduction outside of inner loops is significantly higher in *MobileNet*, which primarily contains deeply nested loops. Additionally, the figures show that *postiz* does not reduce instructions inside innermost loops for cases with high loop depth. These findings indicate that the outer loop optimization addressed by *postiz* plays an important role.

(a) *Platform #1*



(b) *Platform #2*

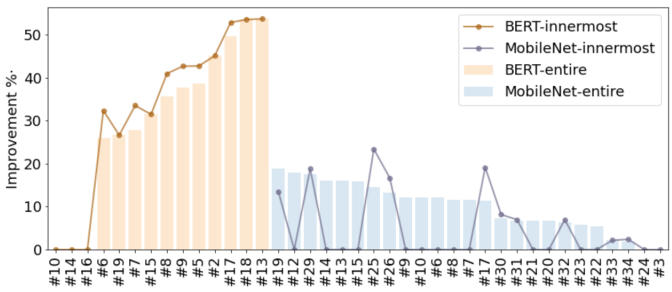


Figure 13: Improvement by *postiz* on Instruction Changes in Innermost Loops and Entire Loops Nests Compared to *llvm* (Normalized, Excluding Unaffected Cases)

3) Register Usage Comparison: Since *Postiz* affects the choice of induction variables, it may impact scalar-integer register pressure inside the loops. We estimated the changes in general purpose register usage in kernel functions to reflect how it could affect the register pressure internally.

TABLE IV shows the accumulated usage count for different benchmarks and platforms, as well as the reduction count of *postiz* compared to *llvm*. As observed, on *Platform #1*, a reduction of 10.27% in register usage was observed for *MobileNet*, and a 8.85% reduction is for *BERT*. On *Platform #2*, while there was a slight increase in register usage by 0.03% for *MobileNet*, it reduced register usage by 18.52% for *BERT*. This reflects that *Postiz* optimization can reduce register usage and positively impact register pressure.

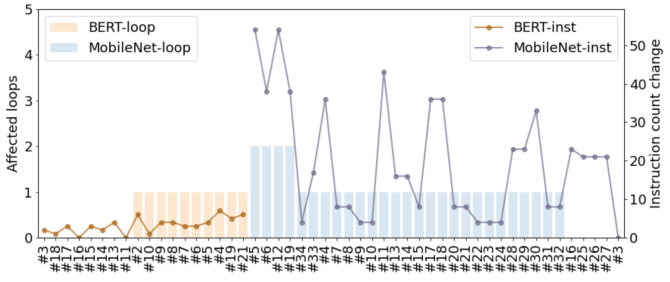
Table IV: Register Usage Reductions: *postiz* over *llvm*

Reduction/Total Count	Platform #1		Platform #2	
	BERT	MobileNet	BERT	MobileNet
GPs	25/395	106/984	100/444	-1/743

Additionally, we tested the influence of *Profitability Assessment* on register usage. The comparison results show that *Profitability Assessment* reduces register usage by 0.49% in *MobileNet* and by 0.79% in *BERT*, reflecting *Profitability Assessment* plays an postive role in reducing register usage.

4) Optimization Coverage: We conducted an evaluation of optimization coverage for both *postiz* and *llvm*. In Figure 14, we show the influence of benchmark cases by affected instructions and affected loops.

(a) *Platform #1*



(b) *Platform #2*

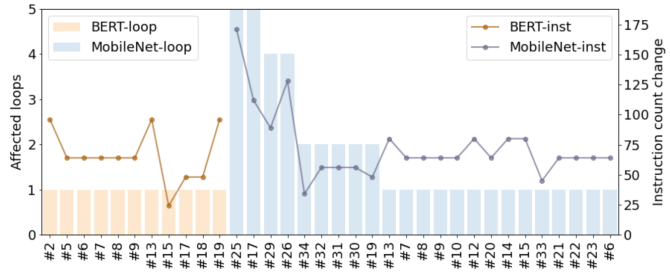


Figure 14: Improvements by *postiz* on Affected Loops and Instructions Compared to *llvm* (Excluding Unaffected Cases)

For *Platform #1*, *postiz* generated post-increment instructions in 50 out of 51 cases across all benchmarks, with 47 cases (92.16%) where *postiz* transformed more instructions than *llvm*, with a increment of up to 55. Accumulating results from all the benchmark cases, *postiz* generated 828 post-increment instructions, while *llvm* produced only 141. This clearly demonstrates *postiz* is better at exploiting optimization opportunities than *llvm*.

Regarding affected loops, *postiz* influenced a total of 118 loops, while *llvm* affected only 77. This represents a 53.25% increase in loop coverage, further highlighting the broader scope of optimization coverage provided by *postiz*.

5) Impact on Runtime Performance: Lastly, we evaluated the runtime performance of *Postiz* on *Platform #1*. The average performance improvement for *MobileNet* was 0.13%, while *BERT* saw a slight decline of 1.05%, primarily due to differences in instruction scheduling. Despite this, we conclude that *Postiz* effectively optimizes code size with minimal impact on performance.

V. Related works

1) Research on Auto-Addressing: Much previous research on optimizing post-increment addressing has focused on minimizing memory access overhead, particularly in DSP architectures. However, unlike *Postiz*, which focuses on nested loops with relatively regular memory access patterns inside inner loops, most of these studies optimize sequential code where memory accesses are inherently irregular. This drives the research differently from *Postiz*.

In this direction, Bartley et al. [2] first introduced the *Simple Offset Assignment* (SOA) problem, aiming to minimize address arithmetic in restricted addressing modes. Liao et al. [12] extended this to the *General Offset Assignment (GOA)* problem, providing graph-based methods to optimize variable placement. Later, Salamy et al. [18] further improved code efficiency by reducing address modification instructions using coalescing techniques. Zhuang et al. [26] also contributed to the offset assignment problem by presenting a framework that leverages auto-increment/decrement modes to minimize memory access overhead and improve variable coalescing.

Memory access optimizations were explored by Hartley et al. [8] and Sudarsanam et al. [19], who demonstrated the benefits of auto-increment and auto-decrement addressing modes in reducing explicit address calculations. Wess et al. [23] proposed an Address Generation Unit (AGU) model that further minimized address computation,while Udayanarayanan et al. [20] enhanced memory layout strategies for DSPs.

Compiler-based optimizations have also played a significant role. Liem et al. [13]'s ArrSyn tool and Zivojnovic et al. [28]'s DSPstone benchmarks showcased significant improvements in code size and performance by optimizing memory access patterns. Ottoni et al. [14] contributed optimizations in storage assignment using post-increment instructions. Huynh et al. [10] further evaluated various offset assignment heuristics and introduced the Memory Layout Permutation (MLP) problem, emphasizing the importance of memory layouts in optimizing post-increment addressing efficiency.

2) Scalar Evolution: Postiz relies on LLVM's scalar evolution analysis to function effectively. Numerous studies have explored chains of recurrence and scalar evolution analysis to build a foundation for this approach. Bachmann et al. [1] demonstrated its effectiveness for periodic function evaluation, while Van et al. [21], [22] used it to analyze induction variables, allowing loop optimizations such as strength reduction and parallelization. Brich et al. [3], [4] applied it to nonlinear array dependence testing. Zima et al. [27] further developed tools for manipulating chains of recurrences.

3) Research on Code Size Reduction: Two widely used techniques, including function merging and function outlining, address code size reduction effectively. The function merging method, as explored by Edler et al. and Rocha et al. [7], [17], reduces redundancy by merging similar functions. On the other hand, the function outlining [5], [25] extracts replicated or noncritical code (e.g., code not in the hot path) into separate functions, helping to streamline the primary code flow.

Another approach introduced by RoLAG [15] identifies isomorphic code within straight-line segments using SSA graph alignment, generating loops to optimize size. More recently, Rocha et al. proposed HyBF [16], a framework that reduces code size by merging conditional branches with similar code across both paths.

In addition, machine learning has emerged as a tool for code size optimization. Liang et al. [11] applied machine learning to predict optimal compiler pass sequences, yielding further code size improvements by tailoring pass orders to specific programs.

VI. Conclusion

In this study, we have introduced a novel loop optimization technique called *Postiz*. This approach effectively incorporates the usage of post-increment instructions into loops and focuses on optimizing nested loops to reduce code size. By analyzing SCEV expressions, extending and combining IV-Chains, and including a profitability assessment, *Postiz* has successfully applied to the most frequently used machine learning kernels derived from *MobileNet* and *BERT*. The results demonstrate significant reductions in code size and enhancements in optimization coverage, indicating *Postiz*'s capability to bridge post-increment hardware and compute-intensive workloads.

Our work delves into areas rarely explored by prior research, shedding light on the potential implications for ISA design in processors, particularly in scenarios suitable for machine learning. Looking ahead, studying a wider range of workloads could be useful to broaden the applicability of post-increment addressing. Furthermore, it is possible to merge *Postiz* and *LSR* to better support more architectures with auto-addressing capabilities.

VII. Ackownledgement

This work was supported in part by National Key Research & Development Program of China (No. 2022YFB4500103), the Programs for NSFC (No. 62032008), STCSM (No. 23511100100), and the Shanghai Rising-Star Program (22QB1404600). Jianguo Yao is the corresponding author.

References

[1] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–249, doi:10.1145/190347.190423.

[2] D. H. Bartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Softw. Pract. Exp.*, vol. 22, pp. 101–110, 1992, doi:10.1002/spe.4380220202.

[3] J. Birch, R. van Engelen, K. Gallivan, and Y. Shou, "An empirical evaluation of chains of recurrences for array dependence testing," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 295–304, doi:10.1145/1152154.1152198.

[4] J. L. Birch, "Using the chains of recurrences algebra for data dependence testing and induction variable substitution," Ph.D. dissertation, Florida State University, 2002.

[5] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production ios mobile applications," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 363–377, doi:10.1109/CGO51591.2021.9370306.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019, doi:10.48550/arXiv.1810.04805.

[7] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, 2014, pp. 85–94, doi:10.1145/2597809.2597811.

[8] D. H. Hartley, "Optimizing stack frame accesses for processors with restricted addressing modes," *Software: Practice and Experience*, vol. 22, no. 2, pp. 101–110, 1992, doi:10.1002/spe.4380220202.

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, doi:10.48550/arXiv.1704.04861.

[10] J. Huynh, J. N. Amaral, P. Berube, and S.-A.-A. Touati, "Evaluation of offset assignment heuristics," in *High Performance Embedded Architectures and Compilers: Second International Conference, HiPEAC 2007, Ghent, Belgium, January 28-30, 2007. Proceedings 2*. Springer, 2007, pp. 261–275, doi:10.1007/978-3-540-69338-3_18.

[11] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. Leather, and Y. Tian, "Learning compiler pass orders using coreset and normalized value prediction," 2023, doi:10.48550/arXiv.2301.05104.

[12] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Storage assignment to decrease code size," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 235–253, may 1996, doi:10.1145/229542.229543.

[13] C. Liem, P. Paulin, and A. Jerraya, "Address calculation for retargetable compilation and exploration of instruction-set architectures," in *Proceedings of the 33rd annual Design Automation Conference*, 1996, pp. 597–600, doi:10.1145/240518.240631.

[14] D. Ottoni, G. Ottoni, G. Araujo, and R. Leupers, "Improving offset assignment through simultaneous variable coalescing," in *International Workshop on Software and Compilers for Embedded Systems*. Springer, 2003, pp. 285–297, doi:10.1007/978-3-540-39920-9_20.

[15] R. C. O. Rocha, P. Petoumenos, B. Franke, P. Bhatotia, and M. O'Boyle, "Loop rolling for code size reduction," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 217–229, doi:10.1109/CGO53902.2022.9741256.

[16] R. C. O. Rocha, C. Saumya, K. Sundararajah, P. Petoumenos, M. Kulkarni, and M. F. P. O'Boyle, "Hybf: A hybrid branch fusion strategy for code size reduction," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 156–167.

[17] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 149–163, doi:10.1109/CGO.2019.8661174.

[18] H. Salamy and J. Ramanujam, "An effective heuristic for simple offset assignment with variable coalescing," in *Languages and Compilers for Parallel Computing*, G. Almási, C. Caşcaval, and P. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 158–172, doi:10.5555/1757112.1757131.

[19] A. Sudarsanam, S. Liao, and S. Devadas, "Analysis and evaluation of address arithmetic capabilities in custom dsp architectures," in *Proceedings of the 34th Annual Design Automation Conference*, 1997, pp. 287–292, doi:10.1023/A:100886991532.

[20] S. Udayanarayanan and C. Chakrabarti, "Address code generation for digital signal processors," in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 353–358, doi:10.1145/378239.378521.

[21] R. van Engelen, "Symbolic evaluation of chains of recurrences for loop optimization," Technical Report TR-000102, Computer Science Department, Florida State ..., Tech. Rep., 2000.

[22] R. A. Van Engelen, "Efficient symbolic analysis for optimizing compilers," in *International Conference on Compiler Construction*. Springer, 2001, pp. 118–132.

[23] B. Wess and M. Gotschlich, "Minimization of data address computation overhead in dsp programs," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 5, 1998, pp. 3093–3096 vol.5, doi:10.1109/ICASSP.1998.678180.

[24] J. Yao, H. Zhou, Y. Zhang, Y. Li, C. Feng, S. Chen, J. Chen, Y. Wang, and Q. Hu, "High performance and power efficient accelerator for cloud inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1003–1016, doi:10.1109/HPCA56546.2023.10070941.

[25] P. Zhao and J. N. Amaral, "Ablego: a function outlining and partial inlining framework," *Software: Practice and Experience*, vol. 37, no. 5, pp. 465–491, 2007, doi:10.1002/spe.774.

[26] X. Zhuang and S. Pande, "An optimization framework for embedded processors with auto-addressing mode," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 4, Apr. 2010, doi:10.1145/1734206.1734208.

[27] E. V. Zima, "On computational properties of chains of recurrences," in *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, 2001, p. 345, doi:10.1145/384101.384148.

[28] V. Zivojnovic, "Dspstone: A dsp-oriented benchmarking methodology," *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994*, pp. 715–720, 1994.