Customizing Scalable Instructions via Multi-Level E-Graph Anti-Unification

Yitian Sun
ytsun@stu.pku.edu.cn
School of Integrated Circuits
Peking University
Beijing, China

Youwei Xiao
shallwe@pku.edu.cn
School of Integrated Circuits
Peking University
Beijing, China

Yuyang Zou
yyzou25@stu.pku.edu.cn
Peking University
Beijing, China

Chenyun Yin
higgs@stu.pku.edu.cn
Peking University
Beijing, China

Yun Liang*
ericlyun@pku.edu.cn
School of Integrated Circuits
Peking University
Beijing, China

Abstract

Modern workloads demand computational capabilities that general-purpose processors cannot efficiently provide. Extending these architectures with domain-specific instructions and hardware offers a practical solution. The open RISC-V instruction-set architecture (ISA) significantly lowers the barrier to such customization. However, existing automated customization flows face two key limitations. First, they typically operate on low-level intermediate representations, obscuring higher-level computation patterns and thereby limiting both performance and reusability. Second, they decouple instruction identification from microarchitectural design, leading to suboptimal performance under on-chip resource constraints. These issues become more pronounced as instructions scale up and reusability becomes more critical.

We present `ScaleISA`, an end-to-end framework for customizing scalable instructions from domain applications. The major insight of `ScaleISA` is combining MLIR and e-graphs to identify reusable instructions at appropriate levels of abstraction and performing joint microarchitectural exploration. It introduces a dialect-agnostic encoding that brings MLIR into e-graphs. Its core methodology, multi-level instruction identification (`MLII`), applies microarchitectural transformations through e-graph rewrites, and then leverages multi-level e-graph anti-unification (`MLAU`) to identify reusable instruction candidates together with their microarchitectural implementations in a single unified search space. Final budget-aware e-graph extraction jointly chooses custom instructions and their microarchitectures. Evaluation shows that `ScaleISA` achieves an average 1.42× (up to 2.87×) speedup over the best baselines by performing multi-level instruction identification.

CCS Concepts: • **Software and its engineering** → *Compilers*; **Formal methods**; • **Hardware** → **Application specific instruction set processors**.

Keywords: Custom instruction, ASIP, E-graph, Hardware-software co-design, Compiler

ACM Reference Format:
Yitian Sun, Youwei Xiao, Yuyang Zou, Chenyun Yin, and Yun Liang. 2027. Customizing Scalable Instructions via Multi-Level E-Graph Anti-Unification. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3845814.3850048

1 Introduction

The rapid growth of modern computing domains, including large language model (LLM) [62] and point-cloud processing [41], is placing unprecedented pressure on processor computation power. General-purpose processors remain attractive for programmability, but their performance and energy efficiency are often insufficient for these emerging workloads. At the same time, fixed-function accelerators can deliver high efficiency but sacrifice flexibility. Application-specific instruction-set processors (ASIPs) occupy an important middle ground by retaining programmability while extending the processor with custom instructions (CIs) tailored to domain-specific computation. This design point has motivated a long line of work [10, 11, 34, 36, 40, 54–57] on automatic instruction-set customization and ASIP generation. More recently, ISAMORE [56] explores both instruction reusability and data-level parallelism (DLP) through unified

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850048

e-graph anti-unification [4], bringing automatic custom instruction generation closer to practical deployment.

At the same time, the custom instructions demanded by modern workloads are no longer limited to small scalar idioms. They increasingly resemble vector- or tensor-scale operations with substantial data movement, parallelism, and microarchitectural design freedom, as reflected by recent work [6, 27, 43] on generated vectorization support for customizable instruction sets and tensor-oriented extensions. This shift exposes weaknesses in existing automatic instruction customization flows:

First, existing work is constrained by the intermediate representation (IR) levels on which it operates. Prior customization methods [10, 11, 40, 54, 55] largely operate on low-level subgraphs or basic-block-local regions, which naturally biases the discovered instructions toward scalar-level idioms. When the target instructions become more scalable, this low-level view becomes increasingly limited: the loss of structured semantic information makes higher-level reusability invisible. For example, ISAMORE [56] operates at an LLVM-derived structured IR, which is low-level, and it struggles to find larger reusable instructions that would be more naturally visible at a higher level such as `linalg`.

Second, instruction identification does not adequately account for microarchitectural diversity. Current flows typically separate instruction identification from hardware exploration. For example, Cayman [55] combines instruction discovery with hardware design-space exploration for vector-level accelerators, but it fixes the instructions before exploration and assigns each only its individually best implementation. This sequential approach misses globally better tradeoffs because microarchitectural choices (e.g., tiling) can create new reuse opportunities through shared hardware components that are invisible if the instruction is fixed beforehand. More broadly, there has been substantial progress [7, 14, 38, 42, 52, 53] on automatic hardware generation and pragma-level high-level synthesis (HLS) exploration, but these advances are typically applied after the target computation has already been fixed, leaving ASIP performance on the table under realistic area and power budgets.

These observations suggest a natural insight to address both weaknesses together. Modern compiler flows such as Triton [44] and XLA [33] and HLS frameworks such as HECTOR [58] and Dynamatic [24] already meet on MLIR [26]: dialects preserve the high-level semantic structure needed to expose scalable instruction patterns, while many optimization techniques (e.g., algebraic optimizations, loop unrolling) are shared between software compilation and hardware synthesis. By reifying multi-level transformations as non-destructive e-graph rewrites, we can explore microarchitectural implementations and expose semantically equivalent patterns for reusability-aware instruction customization simultaneously. This joint identification addresses both challenges: it exposes scalable instruction candidates at the levels where structural semantics remain visible, and it lets instruction identification and microarchitectural exploration proceed together rather than as separate phases.

`ScaleISA` follows this insight with the framework of multi-level instruction identification (`MLII`). By integrating MLIR into e-graphs, `ScaleISA` performs multi-level microarchitectural transformations as rewrites and then applies multi-level e-graph anti-unification (`MLAU`) to identify reusable instruction candidates together with their microarchitectural implementations. It then uses e-graph extraction to jointly choose the final instruction set together with the corresponding microarchitectures. In this way, `ScaleISA` scales automatic instruction customization to modern workloads with more hardware sharing opportunities and better end-to-end performance.

Prior works [4, 55, 56] on automatic instruction customization operate on a single low-level IR and explore microarchitectures only after instruction candidates are fixed. `ScaleISA` advances the state of the art by identifying reusable instructions at multiple abstraction levels and by exploring multiple microarchitectural implementation choices during instruction identification. This is enabled by our unified MLIR and e-graph integration, microarchitectural transformations as non-destructive e-graph rewrites, and `MLAU`, which performs instruction identification at multiple IR levels. `ScaleISA` thereby identifies more instructions with better performance and reusability.

This work offers the following contributions:

- `ScaleISA` performs multi-level instruction identification via e-graph anti-unification and MLIR integration, enabling efficient identification of scalable instructions with multi-level reusability.
- `ScaleISA` formulates custom instruction identification and microarchitectural exploration as one joint optimization problem, where microarchitectural transformations are explored as e-graph rewrites to expose better tradeoffs and microarchitecture-aware reusability.
- `ScaleISA` provides an open-source[1], automated, and end-to-end framework that combines microarchitectural exploration, reusable instruction identification, and budget-aware instruction selection for scalable ISA customization.

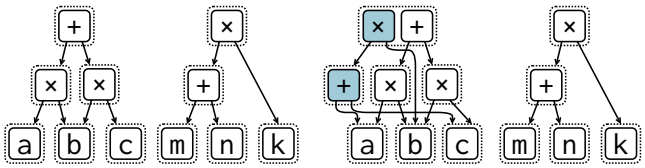
We evaluate `ScaleISA` against ISAMORE [56], NOVIA [45], and Cayman [55] on GTSAM [13], PCL [41], and TorchBench [20], using HECTOR [58] and OpenROAD-based hardware evaluation. At default area budgets, `ScaleISA` achieves a 5.73× geometric-mean speedup over the CPU baseline and an average 1.42× speedup over the strongest baselines selected per case. As the area budget grows, `ScaleISA` moves beyond the tight-budget regime where scalar reuse dominates and increasingly outperforms prior work by exploiting larger loop- and vector-level instructions with better performance-area tradeoffs. This high-level capability also

[1] https://github.com/pku-liang/aps-mlir/

Table 1. Comparison of instruction customization flows.

Work	Source	Scale	Reuse	μArch
NOVIA [45]	LLVM	Scalar	Syntax	No
CIDRE [40]	Assembly	Scalar	Syntax	Limited
Cayman [55]	LLVM*	Scalar, loop	Partial	Yes
ISAMORE [56]	LLVM*	Scalar, loop	Semantic	No
`ScaleISA` (this work)	High-level MLIR	Flexible	Multi-level Semantic	Yes

LLVM* denotes IRs derived from the LLVM IR.



(a) Initial e-graph **(b) Saturated e-graph**

Figure 1. Example of e-graph construction and equality saturation.

carries into our Allo [5] case study, where selecting instructions at the `affine` level enables more advanced optimizations. The `ScaleISA`-Allo flow achieves speedups of 13.7× on MobileNet [23] and 24.4× on ResNet-50 [21] compared to the hardware generated by `ScaleISA`'s default HECTOR flow. These results show that `ScaleISA` makes scalable ISA customization practical by identifying larger reusable instructions, jointly exploring their implementations, and preserving high-level structure when it unlocks further acceleration.

2 Background and Motivation

ASIPs extend general processors with domain-specific instructions that accelerate critical kernels without fully abandoning software flexibility. This design point has become increasingly attractive as open instruction-set architectures (ISAs) such as RISC-V make custom extension support more accessible to architects and system designers. These opportunities have motivated increasing efforts on automatic ISA customization.

2.1 Automatic Instruction Customization

Traditional ASIP design flows rely heavily on manual effort. Designers must inspect applications, decide which custom instructions are worth adding, describe those instructions using Register Transfer Language (RTL), and integrate the extensions into the processor and software stack. This workflow is time-consuming and requires substantial expertise on both the application and hardware sides.

To reduce this burden, one line of work [22, 34, 36, 57, 63] provides instruction description languages and toolchains that can automatically generate hardware and integrate new instructions into a processor once the target extension is known. These systems substantially accelerate instruction customization and processor integration, but the instructions themselves still need to be designed manually.

Table 1 summarizes another line of work, which focuses on automatic ISA customization itself. Early fine-grained and coarse-grained work [10, 11, 15, 17, 18, 39, 54], such as NOVIA [45], generally performs instruction identification on relatively low-level source representations, such as LLVM IR [25], and therefore mainly finds only scalar instructions. CIDRE [40] pushes this direction down to RISC-V assembly and also targets scalar instructions. More recent systems move to structured LLVM-derived IRs. ISAMORE [56] uses e-graph anti-unification to make reusable instruction finding practical and uses packing-based vectorization, while Cayman [55] can target larger loop-level instructions. However, these flows still rely on relatively low-level program views and may lose opportunities for identifying more complex instructions. As instruction scale grows from scalar-level to vector-level and tensor-level operations, low-level IRs become a less suitable source level for pattern identification because the structure needed to expose those instructions has already been removed during the lowering phases.

2.2 E-Graph Anti-Unification for Instruction Customization

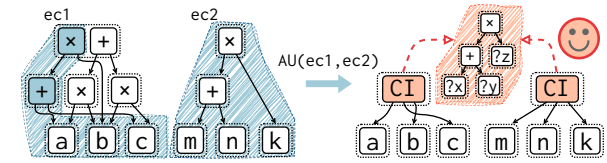
E-graphs [51] compactly represent many equivalent program fragments without forcing one canonical form. Figure 1 shows this process on two expressions, $a \times b + b \times c$ and $(m + n) \times k$. Starting from the initial terms in Figure 1(a), equality saturation applies semantics-preserving rewrites such as $x \times z + y \times z \rightarrow (x + y) \times z$. The resulting saturated e-graph in Figure 1(b) therefore keeps both the original and rewritten forms alive in shared e-classes instead of committing to one choice.

Anti-unification (AU) [37] computes the least general pattern of two terms. It subsumes both terms by keeping their common structure and abstracting the differences into variables. For example, AU of the two terms 1 + 1 and 2 + 1 yields the pattern $?x + 1$, where $?x$ is a variable that can be instantiated to any value. E-graph AU [4] extends AU from single terms to e-graphs, and can be formulated as follows:

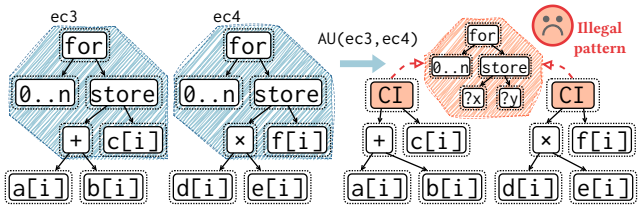
$$\text{E-class AU} \quad \mathrm{AU}(\mathrm{ec}_1, \mathrm{ec}_2) = \bigcup\nolimits_{\mathrm{en}_1 \in M(\mathrm{ec}_1),\, \mathrm{en}_2 \in M(\mathrm{ec}_2)} \mathrm{AU}(\mathrm{en}_1, \mathrm{en}_2)$$

$$\text{E-node AU} \quad \mathrm{AU}(o_1(\mathrm{ec}_{1,1}, \ldots, \mathrm{ec}_{1,k}), o_2(\mathrm{ec}_{2,1}, \ldots, \mathrm{ec}_{2,l})) = \begin{cases} \{o_1(p_1, \ldots, p_k) \mid p_i \in \mathrm{AU}(\mathrm{ec}_{1,i}, \mathrm{ec}_{2,i})\}, & o_1 = o_2 \\ ?x, & o_1 \neq o_2 \end{cases}$$

In the formulation, $M(\mathrm{ec}_1)$ denotes the member e-nodes of e-class ec_1, o denotes an operator, and $?x$ denotes a new variable. AU between two e-classes ec_1 and ec_2 considers pairs of member e-nodes from $M(\mathrm{ec}_1)$ and $M(\mathrm{ec}_2)$, keeps a common operator when the paired e-nodes match, and introduces a new variable when they do not. We call this standard



(a) E-graph anti-unification on flat IR



(b) E-graph anti-unification on structured IR

Figure 2. E-graph anti-unification on flat and structured IR. (a) On flat IR, AU over the e-classes `ec1` and `ec2` extracts a pure data-flow pattern. (b) On structured IR, abstracting a mismatch inside a loop body into a variable yields an illegal pattern.

formulation *vanilla e-graph AU*. Because the e-graph keeps all semantically equivalent variants alive, e-graph AU can identify patterns that are semantically reusable across the program, which makes it a natural fit for instruction customization [56]. For example, Figure 2(a) applies AU over the e-classes `ec1` and `ec2`, extracting the pure data-flow pattern $(?x{+}?y){*}?z$. Such a pattern can be lifted into a custom instruction (CI) that the program calls directly. However, the way vanilla e-graph AU generates variables is not always correct on *structured IRs*. Figure 2(b) shows two loops performing element-wise vector add and multiply. Vanilla e-graph AU over `ec3` and `ec4` returns a vector element-wise store as the common pattern, because the store is the only structure the two loops share. Instantiating this pattern as a CI breaks the loop's integrity: the loads and the element-wise operations not included in the CI are left dangling, producing an illegal program. Such failures occur frequently when e-graph AU runs on structured IR, limiting the scale of patterns that can be identified as CIs and calling for new methods for customizing scalable instructions.

2.3 Scaling to Larger Instructions

Besides the failure of vanilla e-graph AU on structured IR discussed in Section 2.2, scaling instruction customization to larger instructions faces two more challenges:

***Challenge A**: larger reusable patterns are often visible only in higher-level IRs where vector, tensor, or structured control semantics have not yet been lowered away.*

***Challenge B**: larger instructions admit richer microarchitecture choices, so instruction identification can no longer be cleanly separated from design-space exploration.*

Figure 3 illustrates why larger instructions require both higher-level IR and microarchitectural awareness. Figure 3(a) shows two loops whose bounds differ and whose bodies have the same semantics up to a reordering of independent statements. As shown in Figure 3(b), in ISAMORE, these loops are encoded to an LLVM-derived DSL as a `dowhile` skeleton with threaded memory-state dependencies that preserve the order of effectful operations. This representation makes AU fragile: even a small change in statement order changes the state-thread structure which is hard to recover. For this case, the different statement orderings lead to different state-thread structures, causing AU mismatches that introduce variables; as a result, an incomplete loop is identified by AU and cannot be actually used in practice. Furthermore, the standard `dowhile` form also makes loop-level equivalence transformations awkward because the loop bound is buried in the low-level predicate term. For example, loop tiling or unrolling rewrites are difficult to apply in this representation due to the challenges in analysis.

As instruction scale grows, the microarchitectural design space also grows quickly. A loop-level instruction with one fixed implementation is often not very useful; the practical speedup depends on whether the loop can be pipelined, unrolled, or otherwise reorganized into an efficient hardware structure for parallel execution. Cayman [55] provides a useful contrast here. For the example in Figure 3(c), a Cayman-style flow could still identify both loops as accelerator candidates and automatically choose hardware optimizations such as loop unrolling and pipelining for each one. Cayman could also use syntactic merging to merge the two loop bodies. However, this still leaves three limitations. First, syntactic merging introduces extra multiplexers to reconcile the merged bodies. Second, Cayman can merge the DFGs of the loop bodies but not the control flows, namely the loop FSMs in the figure, which have microarchitectural reuse potential by loop tiling. Third, Cayman commits each accelerator to one fixed, performance-optimal optimization setting, which can miss useful performance-area tradeoffs.

2.4 Unifying Abstractions in MLIR

MLIR [26] is a multi-level compiler infrastructure organized around dialects and staged lowering. In recent years, a substantial body of compiler research has been built on MLIR, including machine learning compilers such as Triton [44] and XLA [33], accelerator-oriented compiler frameworks such as SNAX-MLIR [46], and high-level synthesis (HLS) frameworks such as HECTOR [58], Dynamatic [24], and superoptimizers like SEER [7].

MLIR organizes compilation as a multi-level IR composed of dialects, providing a principled basis for performing analysis and transformations at the abstraction levels where the relevant program structure and semantics are explicit. This property is essential for solving *Challenge A* in Section 2.3, where instruction identification should be conducted at the

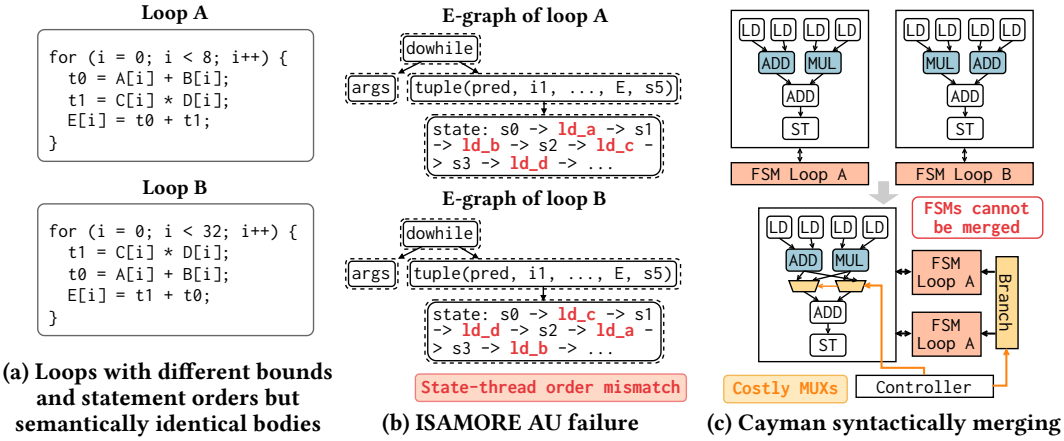


Figure 3. Failure in identifying reusable loop instructions at low-level IRs without microarchitectural awareness.

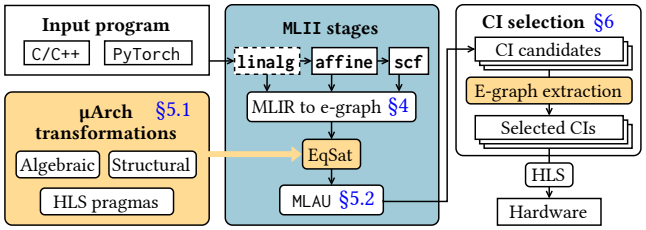


Figure 4. Overview of ScaleISA. C/C++ programs skip the linalg stage.

levels where such patterns naturally emerge. This observation naturally forms the central insight behind ScaleISA: *by integrating MLIR with e-graph AU, we systematically identify reusable and scalable custom instructions at appropriate abstraction levels.* Although prior works such as SEER [7], DialEgg [60], HEC [59], and the eqsat dialect [30, 31] have explored integrating MLIR with e-graph-based optimization, they do not investigate this combination in the context of instruction customization. Moreover, this formulation naturally unifies instruction identification with microarchitectural exploration. *Microarchitectural optimizations can be expressed as transformations across different IR levels and encoded as rewrite rules within the e-graph, enabling their co-exploration during e-graph-based instruction identification and thereby solving Challenge B.* As a result, building ScaleISA upon MLIR not only exposes semantically reusable instructions across multiple program abstractions, but also captures microarchitectural optimization opportunities, leading to improved performance-area tradeoffs.

3 Framework Overview

Figure 4 illustrates the end-to-end ScaleISA flow. Starting from a program written in PyTorch or C/C++, ScaleISA progressively lowers it into MLIR at multiple abstraction levels. Each abstraction level is one identification stage targeting a different instruction granularity: operator-level instructions are most naturally exposed in linalg, loop-level instructions in affine, and scalar data-flow-level instructions in scf.

ScaleISA identifies instructions one stage at a time, encoding only the current stage's MLIR into an e-graph through the ScaleISA DSL (Section 4). ScaleISA does not explicitly preserve any cross-dialect information, such as rewrite-introduced e-nodes or identified CI candidates in prior stages. Each stage is independent and starts from a fresh e-graph containing only information from the current level.

At each stage, ScaleISA applies microarchitectural transformations as rewrites with equality saturation (Section 5.1). Then MLAU identifies reusable instruction candidates with both semantics and microarchitectural implementations (Section 5.2). After all stages, ScaleISA lowers every candidate to an scf left-hand-side pattern and applies them as rewrite rules in one unified scf e-graph of the original program, where software execution and all candidates compete regardless of their identification level. Budget-aware e-graph extraction then selects the latency-optimal expression under the area budget (Section 6). The instructions that remain determine the final instruction set and their microarchitectures. By default, ScaleISA uses HLS to generate the hardware. All the steps mentioned above are automatic.

Compared with state-of-the-art approaches [4, 55, 56], ScaleISA's multi-level stages identify more complex instructions at higher abstraction levels. Representing microarchitectural transformations as rewrites allows ScaleISA to optimize hardware implementations before fixing the instruction set, which leads to better performance.

End-to-end running example. We use the fully connected layer of ResNet-50 as the running example as shown in Figure 5, which is the final classifier of the network. It multiplies the pooled feature vector by the transposed weight matrix

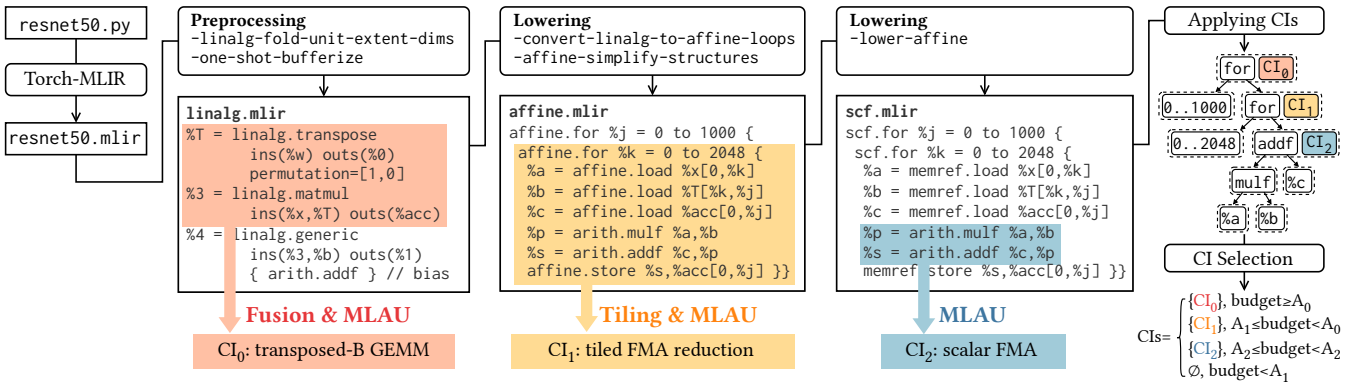


Figure 5. End-to-end running example: the fully connected layer of ResNet-50 through Torch-MLIR, preprocessing, the three-stage MLII, each with a representative MLIR snippet, the applied rewrites, and the identified instruction, and the unified selection e-graph.

and adds a bias. The frontend lowers the PyTorch model to `linalg-on-tensors` with Torch-MLIR [28]. We then preprocess the exported MLIR code with a few routine cleanup passes followed by bufferization, because all stages model memory uniformly with the `memref` dialect.

ScaleISA lowers the preprocessed MLIR to the target dialects. Each stage is independent and constructs a fresh e-graph from its corresponding lowering. Figure 5 shows, for each stage, a representative MLIR snippet, an instruction identified by MLAU, and the transformations that expose it. At the `linalg` stage, ScaleISA encodes the three `linalg` operations to an e-graph, absorbs the weight transpose into the GEMM through `linalg` fusion, and identifies transposed-B GEMM ($\texttt{CI}_0$). At the `affine` stage, ScaleISA encodes the `affine` GEMV loop nest, tiles the loop nest, and identifies tiled FMA-reduction loop ($\texttt{CI}_1$), which is highly reusable across the network. At the `scf` stage, ScaleISA encodes the `scf` GEMV loop nest and identifies the scalar FMA ($\texttt{CI}_2$) inside the loop body.

The saturated e-graph during the instruction selection process is shown in Figure 5. In this example the three candidates match at overlapping positions of the original program, so ScaleISA keeps the best-performing one that fits the current budget, shifting from the scalar FMA to the tiled loop to the GEMM as the budget grows.

4 MLIR-E-Graph Integration

This section introduces how ScaleISA integrates MLIR with e-graphs to support MLII for instruction customization. The goal is not merely to materialize MLIR in e-graphs. Instead, the integration is designed around two requirements imposed by MLII:

Uniform multi-level encoding. ScaleISA must encode `linalg`, `affine`, and `scf` stages with one methodology rather than with separate ad hoc encodings. This keeps the downstream MLII procedure stage-generic.

$x \in$	Names	(Block Arguments)
$i \in$	$\mathbb{N}$	(Projection Index)
$v \in$	**Attrs**	**(MLIR Attribute)**
$o \in$	**OpNames**	**(MLIR Operation Name)**
$e ::=$	$\texttt{arg}(x)$	(Block argument)
	$\mid \texttt{attr}(v)$	(MLIR Attribute)
	$\mid \texttt{seq}(e_1, \dots)$	(Ordered Statements)
	$\mid \texttt{block}(e_1, \dots, e_{\text{body}})$	(Block)
	$\mid \texttt{tuple}(e_1, \dots)$	(Create Tuple)
	$\mid \texttt{proj}(e_{\text{tuple}}, i)$	(Project Tuple Element)
	$\mid \mathbf{o}(\mathbf{e_1}, \dots, \mathbf{e_n})$	**(MLIR Operation)**
	$\mid \mathbf{CI}(e_{\text{pat}}, e_1, \dots)$	**(Apply Instruction)**

Figure 6. Formal syntax of the ScaleISA DSL.

Explicit structural semantics. The encoding must preserve region boundaries, block arguments, and other structural cues explicitly. MLII relies on that structure to identify legal reusable patterns rather than illegal region skeletons, as Section 5.2 discusses.

The rest of this section describes the ScaleISA DSL and the stage-local translation, integrating MLIR and e-graphs.

4.1 ScaleISA DSL

ScaleISA bridges MLIR and e-graphs with a small dialect-agnostic DSL. The DSL is intentionally minimal: most of its syntax exists only to satisfy the requirements above, while the remaining forms are auxiliary nodes needed by e-graph materialization and final instruction selection.

Figure 6 defines the syntax of the ScaleISA DSL. Its central design is to encode every MLIR operation, regardless of dialect, with the same syntax whose children are its operands, attributes, and regions. This treatment gives ScaleISA uniform encoding methodology across `linalg`, `affine`, and

scf, and it also keeps the e-graph terms close to the original MLIR shape instead of translating each dialect into a different style.

The DSL also makes information that matters to MLII explicit. Attributes appear as ordinary child nodes rather than as external metadata, because constants, bounds, iterator annotations, and similar properties often determine whether two structures can be anti-unified legally. The DSL also keeps block nodes and arg nodes explicit to expose structural scope: block marks the boundary of a structured region, and arg represents the block arguments. ScaleISA does not need a separate region syntax because it targets structured dialects in which a block already serves the same meaning as a region.

Beyond these core forms, the remaining syntax is auxiliary. Nodes such as tuple and proj support multi-result values and bookkeeping that arise during encoding, while the CI form is reserved for the final instruction-selection stage to represent CI candidates identified by MLII.

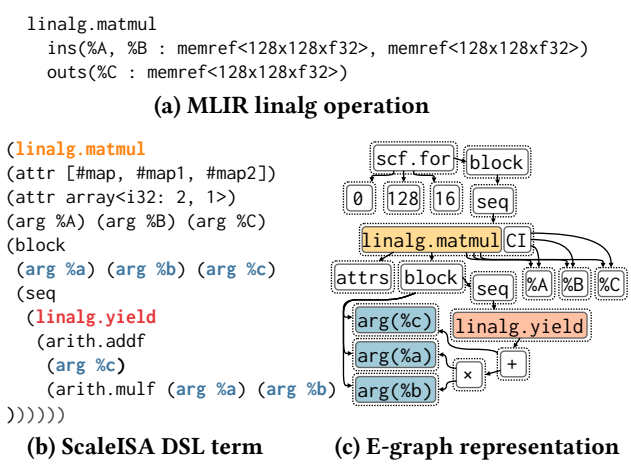


Figure 7. Encoding a simple linalg.matmul operation from MLIR into the ScaleISA DSL and e-graph.

4.2 Encoding MLIR to E-Graph

For each instruction identification stage, ScaleISA lowers the program into IR of the corresponding level and translates it into the DSL before building the e-graph, which starts from the MLIR module and proceeds recursively. When encoding an operation, ScaleISA uses its attributes, operands, and regions as children, while keeping its type information as metadata rather than part of the structural term. When encoding a region, ScaleISA first creates a block node with args and a seq as children. It then analyzes which operations in the block must remain explicitly ordered and places only those operations as direct children of that seq. Starting from each such operation, the translation proceeds recursively through its operands and nested structure, so data-producing dependencies become child nodes instead of siblings in the seq node. This keeps the subsequently generated e-graph clean and easy to rewrite, while preserving the correct semantics. After that, ScaleISA performs common-subexpression elimination to share identical subterms and translates the resulting DSL term into an e-graph.

Figure 7(a) shows a simple linalg.matmul operation. Figure 7(b) shows the resulting DSL term, where the index mappings and operand segment sizes are encoded as the children of linalg.matmul, because they are attributes. The computation region becomes an explicit block with arguments and a seq that retains only the linalg.yield operation as an ordered child; the other operations are instead embedded through the linalg.yield's data dependencies. Figure 7(c) then shows the e-graph after MLII and CI rewrite rules. The full-size linalg.matmul is tiled and the tile-level linalg.matmul is identified as a CI candidate through MLII, which is later applied and creates the CI e-node.

5 Multi-Level Instruction Identification

To address the scale and microarchitectural challenges in Section 2.3 together, ScaleISA proposes multi-level instruction identification (MLII). It first reifies microarchitectural transformations as rewrites to saturate the e-graph (Section 5.1), and then applies multi-level anti-unification (MLAU) to identify reusable instructions effectively (Section 5.2).

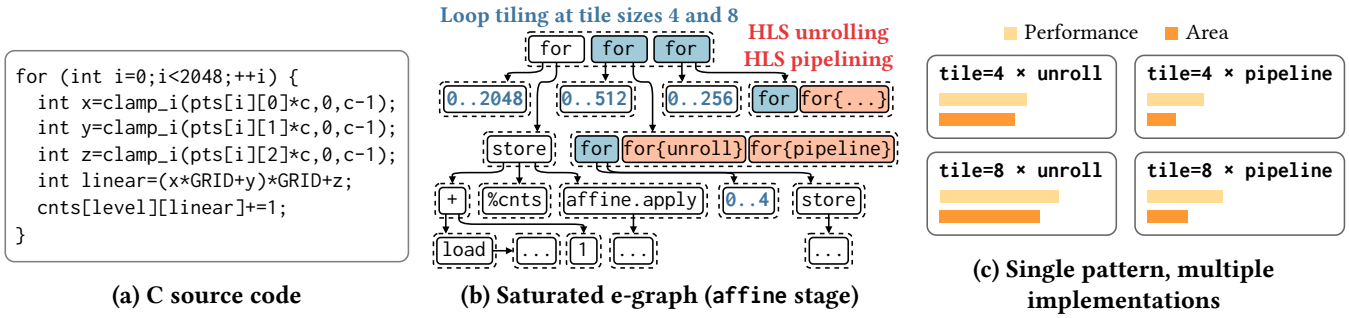
5.1 Microarchitectural Transformations

To address *Challenge B* in Section 2.3, ScaleISA reifies microarchitectural transformations as non-destructive e-graph rewrites. This formulation is possible because microarchitectural optimizations are transformations across abstraction levels, so they can be represented naturally as rewrites over ScaleISA's multi-level IR views.

Saturation with these rewrites creates optimized structures on which MLAU can later identify more scalable microarchitectural choices while exposing more reuse opportunities. Concretely, ScaleISA organizes these rewrites into three rulesets: algebraic transformations, structural transformations, and HLS pragmas.

Algebraic transformations. First, we use *Enumo* [35] to automatically generate algebraic rules verified by Satisfiability Modulo Theories (SMT) solvers [12]. These rules provide foundational transformations for optimizing the data-flow and matching semantically equivalent patterns across different program segments.

Structural Transformations. Second, we incorporate external structural transformations through upstream MLIR passes and ScaleISA MLIR passes. Mechanically, ScaleISA attempts each structural rewrite at every plausible e-graph site, extracts the corresponding sub-program as a standalone



(a) C source code **(b) Saturated e-graph (`affine` stage)** **(c) Single pattern, multiple implementations**

Figure 8. Microarchitectural transformations on the point-to-cell counting loop from the PCL octree workload. (a) The C source code of the loop. (b) The `affine`-stage e-graph after loop tiling rewrites at tile sizes 4 and 8, and HLS pragma rewrites. (c) Four possible microarchitectural implementations of the loop with different performance-area tradeoffs.

Table 2. Structural rewrites used at each identification stage.

Level	Rewrite	Pass
`linalg`	op generalization	`-linalg-generalize-named-ops`
	operator fusion	ScaleISA pass
	operator tiling	ScaleISA pass
`affine`	loop tiling	`-affine-loop-tile`
	loop fusion	`-affine-loop-fusion`
	loop flattening	`-affine-loop-coalescing`
	loop interchange	ScaleISA pass
	loop perfection	ScaleISA pass
`scf`	if-correlation	ScaleISA pass
	if-conversion	ScaleISA pass
	control-flow muxing	ScaleISA pass

MLIR IR segment, processes it with the targeted pass, imports the transformed IR back into the e-graph, and unions it with the original e-class.

Heuristics. ScaleISA applies these rewrites heuristically because extracting MLIR and invoking a pass at every e-graph node is expensive. ScaleISA only extracts sub-programs that satisfy the necessary conditions required by each rewrite to apply its corresponding transformation. For example, for `linalg` tiling, it derives the iteration-space dimensions from the operation's indexing maps and operand shapes, then requires each tiled dimension to have a static size divisible by the configured tile size. For `linalg` fusion, it requires a producer that writes a memref read by another `linalg` operation and compatible indexing maps that map the consumer tile to the producer iteration space.

Table 2 lists the structural transformations. For `linalg`, ScaleISA implements operator tiling and fusion with the `transform` dialect. At the `affine` level, ScaleISA's interchange pass permutes nested loops, while its perfection pass restructures imperfect loop nests into perfect nests. At the `scf` level, if-correlation merges sequential `scf.if` operations with correlated conditions, if-conversion replaces `scf.if` with `arith.select`, and control-flow muxing extracts common branch operations to enable hardware resource sharing.

HLS Pragmas. Third, we derive *lazy* microarchitectural rules from the *HECTOR* HLS framework [58]. These rules inject HLS pragmas, such as loop unrolling and pipelining, as metadata into loop nodes. To keep instruction identification efficient, ScaleISA applies these transformations lazily: each rewrite adds only a new e-node with pragma metadata to the *existing* matched e-class.

Parameters. Four of the rewrites above carry parameters: `linalg` tiling, `affine` tiling, HLS unrolling, and HLS pipelining. For both `linalg` and `affine` tiling, ScaleISA tries all tile sizes {2, 4, 8, 16, 32} on every matched loop during saturation. This exposes different microarchitectural choices in the e-graph. Because tiling already controls loop size, the HLS unrolling rewrite in ScaleISA fully unrolls each loop instead of enumerating unroll factors. For HLS pipelining, ScaleISA leaves the initial interval (II) unspecified and lets the HLS tool automatically attempt the smallest II it can implement.

Example. Figure 8 shows the rewrites on one pattern from the PCL octree workload, which we also evaluate in Section 8. Figure 8(b) shows the `affine`-stage e-graph after saturation with the loop-tiling rewrites at tile sizes 4 and 8, followed by HLS unrolling and pipelining. By retaining all the transformations in the e-graph, ScaleISA allows the loop to be implemented with different microarchitectural choices combining different tile sizes and HLS pragmas, as shown in Figure 8(c). MLAU can identify CIs from each of these implementations to achieve different performance-area tradeoffs.

5.2 Multi-Level Anti-Unification

As discussed in Section 2.2, vanilla e-graph AU is effective at identifying reusable patterns on flat, pure data-flow IR, but its way of generating variables does not carry over to structured IR. ScaleISA's IR views nest regions at arbitrary depth and carry semantics of different levels, so instruction identification needs a new method that works uniformly across all of them. To this end, ScaleISA proposes multi-level anti-unification (MLAU).

Algorithm 1: MLAU: Region-Aware E-Node AU

input : e-node pair n_a, n_b; reference regions ρ_a, ρ_b
output : matched e-node, a fresh variable, or FAIL

function MLAUENode(n_a, n_b, ρ_a, ρ_b):
  if Op(n_a)≠Op(n_b) ∨ Attr(n_a)≠Attr(n_b) **then**
    if InDescendantRegion(n_a, ρ_a) ∨ InDescendantRegion(n_b, ρ_b) **then return** FAIL ;
    return variable()
  if ¬(InRegionOrDescendants(n_a, ρ_a) ∧ InRegionOrDescendants(n_b, ρ_b)) ∧ ¬(IsConst(n_a) ∧ IsConst(n_b)) **then**
    return variable()
  $C := \emptyset$;
  for $(c_a, c_b) \in$ Children(n_a, n_b) **do**
    $C := C \cup \{$AUEClass(c_a, c_b, ρ_a, ρ_b)$\}$;
  return Op(n_a)(C);

MLAU is shown in Algorithm 1. The e-class-level AU is unchanged from vanilla AU, so the pseudocode only shows the e-node case. MLAU starts from a pair of e-classes and records the regions ρ_a and ρ_b that contain the two root e-classes as *reference scopes* for the recursive matching. Region membership is attributed per e-class: an e-class belongs to the region in which its SSA value is produced, and an e-class holding a compile-time constant belongs to no region, since no operation in the program produces it. For each node pair n_a and n_b, MLAU first compares their operations and attributes. When any of them differ, the pair is a candidate for abstraction: if either e-class belongs to a descendant region of ρ_a or ρ_b, the pair lies inside the nested structure that the pattern must keep intact, so abstracting it would break the region skeleton and MLAU returns FAIL; otherwise the mismatch sits in the reference scopes themselves and becomes a variable, which is the usual pure data-flow AU case. If the operations and attributes match, MLAU checks scope: if n_a does not belong to ρ_a or one of its descendant regions, or n_b does not belong to ρ_b or one of its descendants, the value is produced outside the scope and the pair becomes a fresh variable, corresponding to the $?x$ notation in Section 2.2. Pairs of compile-time constants are exempt from this rule because they can be abstracted safely regardless of scopes. In the remaining case MLAU recurses into the children pairs.

Stage-specific AU pairing. The insight behind ScaleISA is to identify instructions at the abstraction level where they are most visible. MLAU therefore does not anti-unify every e-class pair at every stage. MLAU runs only on e-classes that contain specific whitelisted operations at each stage. The linalg stage whitelists the linalg operations with computational or data-movement semantics; the affine stage whitelists affine.for together with the loop operations of the scf dialect; the scf stage whitelists scf.if, the scalar compute operations, and the memory-access operations.

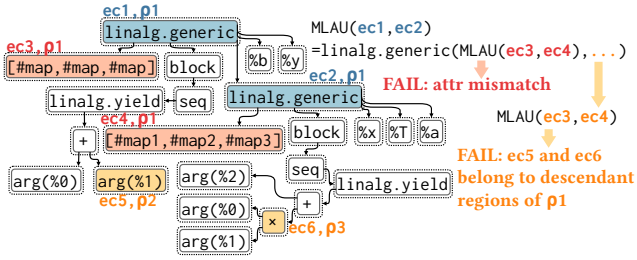


Figure 9. MLAU on two linalg.generic e-classes from the ResNet-50 fully connected layer. The pair fails on an indexing_maps attribute mismatch (ec3/ec4) and on a region-body mismatch inside descendant regions (ec5/ec6).

Some program properties encoded in MLIR can prevent MLAU from identifying useful instructions. ScaleISA does not model memory aliasing in the e-graph. Instead, preprocessing reduces aliasing uncertainty where possible. For example, function inlining exposes memory accesses across call boundaries that would otherwise be opaque. Affine indexing expressions usually appear as operation attributes. Preprocessing and lowering simplify and canonicalize these expressions, after which MLAU compares them as ordinary operation attributes and rejects pairs whose attributes differ. Finally, ScaleISA DSL represents each SSA producer as a child of its consumers, so MLAU follows SSA dependence chains recursively. E-nodes without an explicit SSA dependence path are unreachable from one another, so a candidate identified by MLAU cannot span them.

Example. Figure 9 applies MLAU to two e-classes from the ResNet-50 fully connected layer of Figure 3. MLAU between ec1 and ec2 fails at two places. First, ec3 and ec4 hold different indexing_maps attributes, so an attribute mismatch fails the pair. Second, even if their attributes match, because ec5 and ec6 differ in their body operations, and both e-classes belong to descendant regions of the reference region ρ_1, MLAU will still fail instead of abstracting them. Vanilla e-graph AU would abstract every mismatch into a variable and return a linalg.generic pattern whose indexing map and body operation are both variables, which cannot be used in practice. MLAU rejects the pair and never materializes it.

With MLAU, larger instructions become practical to identify. The microarchitectural transformations described in Section 5.1 further increase the chance for MLAU to find reusable instruction candidates with scalable microarchitectural choices.

6 Instruction-Microarchitecture Selection

The final step jointly selects which instructions to keep as well as their microarchitectures. Unlike prior instruction customization flows that are constrained by selecting candidates only at a fixed low-level IR (e.g., basic blocks or LLVM IR), ScaleISA can flexibly pick the abstraction level

where selection happens. By selecting candidates at higher levels (e.g., `affine` loop nests or `linalg` tensor operations), ScaleISA can preserve the high-level structural information of the custom instructions. This capability is crucial for connecting ScaleISA to advanced compiler and hardware frameworks that can exploit such structured information to enable further optimizations. We carry out a case study in Section 8.4 to further demonstrate this. In our default flow, after MLII, ScaleISA lowers all custom instruction candidates to rewrites over a unified `scf` IR view of the original program. Software execution and all instruction candidates thus compete in one common e-graph, regardless of the IR level at which each candidate was originally identified.

This formulation is important because the true optimization problem is closer to a grouped knapsack than a simple 0–1 knapsack. Suppose there are N instruction candidates, and candidate i has M_i implementation variants. Under a global budget, the optimizer must choose whether to use candidate i at all and, if so, which one of its variants to keep. This unified view enables the possibility for more high-level optimizations that are applied on the high-level form of generated instructions.

ScaleISA uses a budget-aware e-graph extraction objective. For a software expression e, it estimates latency as

$$L_{\mathrm{sw}}(e) = \sum_{o \in e} \mathrm{exec}(o) \cdot \ell_{\mathrm{sw}}(o), \tag{1}$$

where $\ell_{\mathrm{sw}}(o)$ is the software latency of operation o and $\mathrm{exec}(o)$ is its execution count. For a custom-instruction candidate v, ScaleISA uses the latency and area obtained from HLS:

$$L_{\mathrm{hw}}(v) = \mathrm{exec}(v) \cdot \ell_{\mathrm{hw}}(v), \qquad A(v) = a_{\mathrm{hw}}(v). \tag{2}$$

The final extraction minimizes total latency under an area budget B:

$$\min_{E} \; L(E) \qquad \text{s.t.} \qquad \sum_{v \in \mathcal{V}(E)} A(v) \le B, \tag{3}$$

where $\mathcal{V}(E)$ is the set of selected instructions and their microarchitecture choices in the extracted expression E. In this way, ScaleISA benefits from the diverse hardware implementations exposed during MLII.

7 Implementation

ScaleISA is implemented as a multi-front-end flow on top of MLIR [26] and egg [51]. It currently takes Polygeist [32]- and Torch-MLIR-generated programs, which provide structured MLIR suitable for different identification stages. The egg engine serves as the e-graph engine for equality saturation, e-graph anti-unification, and final extraction, where we extend MLII features as introduced in Section 5. For hardware generation and evaluation, ScaleISA uses HECTOR [58] to generate implementations and report latency-area tradeoffs. In total, ScaleISA comprises 11,587 LOC in Python for MLIR-e-graph roundtrip and end-to-end workflow control, 10,489 LOC in Rust for implementing MLII, and 1,070 LOC in C++ for the structural transformations and lowering custom-instruction candidates to HECTOR inputs. The resulting custom instructions are integrated through a RoCC-style coprocessor interface, which provides a concrete path from selected instruction candidates to deployable hardware-backed ISA extensions.

Table 3. Summary of benchmarks.

Suite	Workload	Description
GTSAM	Planar	2D planar SLAM.
	Pose2	2D pose-graph SLAM.
	Pose3	3D pose-graph SLAM.
	SFM	Batch structure-from-motion.
	StereoVO	Stereo visual odometry.
PCL	ISS	ISS keypoint detection.
	KDTree	k-d tree nearest-neighbor search.
	Morph	Morphological filtering.
	Normals	Surface-normal estimation.
	Octree	Octree-based spatial search.
	ICP	Iterative closest point registration.
	RANSAC	Sample-consensus fitting and outlier rejection.
	RegGrow	Region-growing segmentation.
	Poisson	Poisson surface reconstruction.
TorchBench	MobileNet	Lightweight CNN inference.
	ResNet-50	Residual CNN inference.

8 Evaluation

Our evaluation addresses the following research questions:

1. Can ScaleISA identify instructions that deliver higher fixed-budget performance than prior methods? (Section 8.1.1)
2. How do different methods scale performance as the area budget increases? (Section 8.1.2)
3. Does ScaleISA scale performance by selecting more scalable instructions, rather than by simply growing the instruction set without bound? (Section 8.1.3)
4. How much do ScaleISA's multi-level instruction identification and microarchitectural exploration each contribute to solution quality? (Section 8.2)
5. How close is ScaleISA's automatically generated hardware to hand-optimized hardware? (Section 8.3)
6. Does ScaleISA enable more advanced optimizations with instructions selected at a high level? (Section 8.4)

8.1 Comparison with Baselines

We compare ScaleISA against ISAMORE [56], NOVIA [45], and Cayman [55] on three benchmark suites: GTSAM [13], PCL [41], and TorchBench [20]. GTSAM is a C++ library for factor-graph-based estimation and optimization in robotics and vision workloads such as SLAM and structure from motion. PCL is a C++ library for 3D point-cloud processing, covering workloads such as keypoint detection, filtering, registration, and surface reconstruction. TorchBench represents PyTorch model workloads. We retrieve the evaluated workloads from their corresponding upstream repositories and

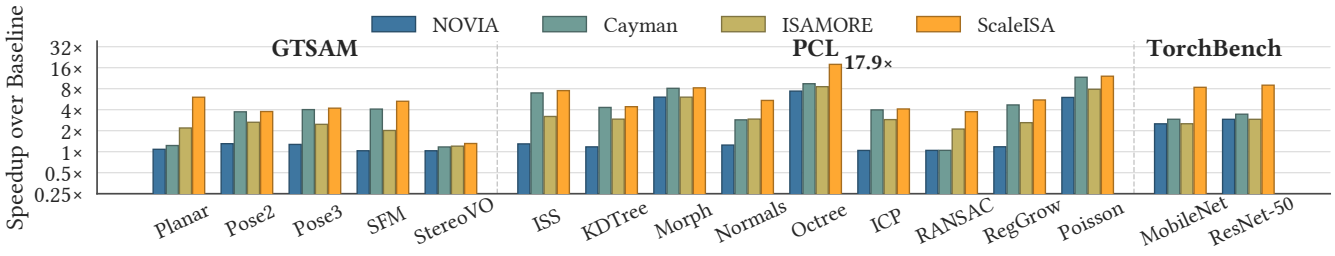


Figure 10. End-to-end speedup at fixed area budgets: 20,000 μm^2 for GTSAM and PCL, and 50,000 μm^2 for TorchBench.

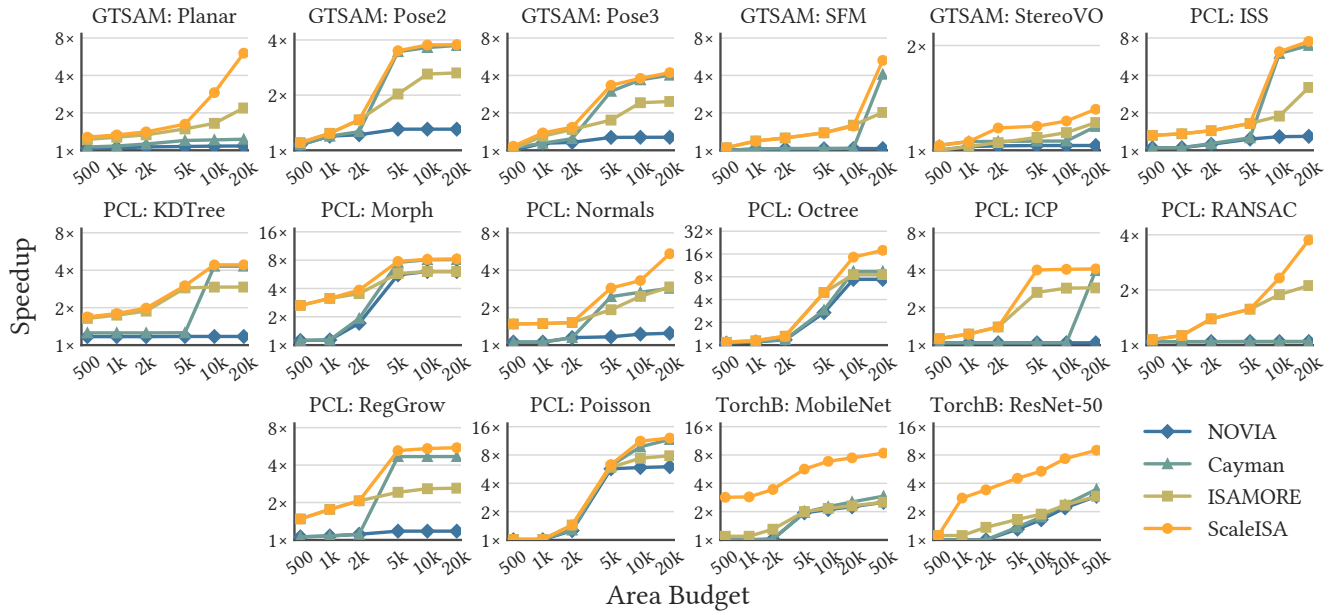


Figure 11. Performance-area Pareto frontiers for the evaluated workloads under different area budgets.

feed them into our flow in the MLIR form. The baseline CPU execution cycles are obtained by running the workloads on a Chipyard-generated system. For CIs, we use HECTOR [58] to generate hardware, evaluate latency with RTL simulation, and evaluate area with OpenROAD targeting the ASAP7 PDK [9]. Table 3 briefly summarizes the function of each workload. Unless otherwise noted, all results report end-to-end application speedup over baseline CPU execution.

8.1.1 Performance Comparison. Figure 10 reports one comparison point per workload, taken at fixed area budgets of 20,000 μm^2 for GTSAM and PCL and 50,000 μm^2 for TorchBench. At these budgets, `ScaleISA` reaches a 5.73× geometric-mean speedup, compared with 3.72× for Cayman, 3.01× for ISAMORE, and 1.77× for NOVIA. Overall, `ScaleISA` achieves an average 1.42× speedup (up to 2.87×) over the strongest baselines picked for each workload.

Specific workloads show where these gains come from. On Planar, `ScaleISA` reaches 6.04×, while the best baseline reaches only 2.19×; this workload contains repeated structured SLAM updates that benefit from `ScaleISA`'s multi-level identification. On Octree, `ScaleISA` reaches 17.88× versus Cayman's 9.44×, indicating the benefit of scalable loop-level instructions for hierarchical search kernels. Cayman is competitive on several GTSAM and PCL workloads, including Pose2, KDTree, ICP, and Poisson, because it can already apply automatic microarchitectural optimization once a candidate accelerator has been fixed. The largest gap appears on TorchBench, however. `ScaleISA` reaches 8.38× on MobileNet and 8.99× on ResNet-50, while Cayman reaches only 2.92× and 3.45×, respectively. These large deep-learning workloads contain many reuse opportunities that become visible only when microarchitectural exploration exposes better tensor- and loop-level structure. `ScaleISA` can effectively utilize these acceleration opportunities under the area budgets, because it couples multi-level identification with microarchitectural exploration, exploiting the high-level reusability exposed.

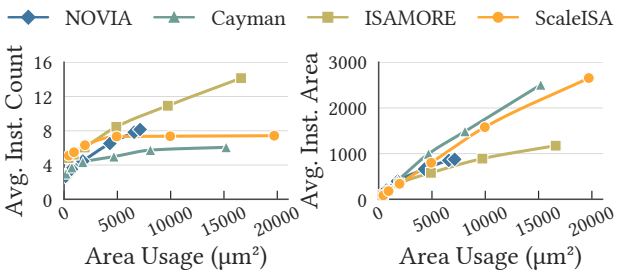


Figure 12. Trends of selected instruction set by area budgets.

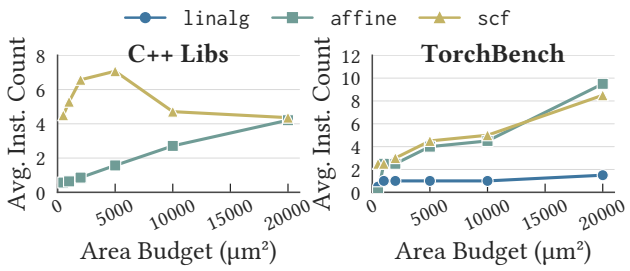


Figure 13. Average number of selected instructions by granularity and area budget for C++ libraries and TorchBench.

8.1.2 Performance Scalability Analysis. Figure 11 shows the full performance-area tradeoff behind these fixed-budget comparisons. Across all workload-budget points, `ScaleISA` is best or tied in 97 of 98 cases; the only exception is SFM at 10,000 μm^2, where ISAMORE is slightly higher (1.59× vs. 1.56×). Thus, `ScaleISA` usually remains the strongest method throughout the budget range rather than only at one selected comparison point.

At tight budgets, many workloads remain close to ISAMORE. On GTSAM and PCL, `ScaleISA`'s geometric-mean advantage over ISAMORE is only 1.02×–1.04× up to 2,000 μm^2. This is expected because both systems can use equality saturation to discover highly reusable scalar instructions, which are the most affordable options when the budget is tight. As the budget grows, however, `ScaleISA` pulls away from both ISAMORE and NOVIA because it can expose and exploit larger loop- and tensor-level instructions. By 20,000 μm^2, `ScaleISA` improves the geometric mean over ISAMORE by 1.81× on GTSAM and 1.75× on PCL, and by 50,000 μm^2 it improves over ISAMORE by 3.20× on TorchBench.

The comparison with Cayman highlights a different advantage. Although Cayman can be close at the highest budget on some workloads, it often has a clear disadvantage at lower budgets. For example, on KDTree at 5,000 μm^2, `ScaleISA` reaches 3.00× while Cayman reaches 1.26×; on ICP, `ScaleISA` reaches 4.02× while Cayman remains at 1.04×. This pattern indicates that Cayman can optimize fixed candidates well near the loosest budget, but `ScaleISA`'s joint flow finds better performance-area tradeoffs throughout the budget range.

8.1.3 Instruction Scale Analysis. Higher performance does not necessarily imply a better ISA extension if it is achieved only by selecting many more instructions. Figure 12 therefore examines how the selected instruction set evolves as total area usage grows. Unlike the other three methods, `ScaleISA`'s selected instruction count increases only in the low-budget regime and then quickly stabilizes over most of the operating range. In contrast, ISAMORE and NOVIA keep adding instructions as more area becomes available, and Cayman also grows the selected set when the budget becomes large.

At the same time, the average area of a selected `ScaleISA` instruction keeps increasing with total area usage. This means `ScaleISA` is not buying performance by letting the ISA explode; instead, it spends additional budget on making each selected instruction itself more capable. In other words, the identified instructions are genuinely scalable: as the target performance increases, `ScaleISA` tends to keep roughly the same instruction-set size while scaling up the microarchitectural strength of the instructions that remain selected.

Figure 13 breaks down the selected instructions by dialect. For C++ libraries, `affine` instructions increase with the area budget while `scf` instructions first rise and then decline. This trend confirms that performance gains arise from improved instruction granularity rather than from simply increasing instruction count. In contrast, for TorchBench, `linalg` instructions stay roughly constant while `affine` and `scf` instructions increase. This behavior comes from the computational nature of deep learning workloads: most of the reusable computation is already captured by a small number of coarse-grained instructions, so further speedup comes primarily from accelerating additional fine-grained operations.

8.2 Ablation Study

Table 4 ablates the two major ingredients of `ScaleISA` that remain meaningfully separable in the current formulation: multi-level instruction identification and joint microarchitectural exploration. Accordingly, *w/o μArch* removes the rewrite-driven microarchitecture exploration and lets each identified instruction keep only one best fixed microarchitectural design, while *w/o Multi-Level* restricts instruction identification to the `scf` stage.

The largest effect comes from removing multi-level instruction identification. The geometric-mean speedup drops from 5.73× to 3.00×, decreasing by 47.6%. Especially on MobileNet, ResNet-50, Planar, and SFM, the performance losses exceed 60%. These results show that access to high-level IR is the most important factor for identifying larger-scale instructions and translating them into higher end-to-end performance. Without those higher-level program views, the system falls back toward smaller low-level candidates and

Table 4. Ablation of ScaleISA on the evaluated workloads. The default column reports the end-to-end speedup of the full system. Each ablated configuration reports its raw speedup, with the relative change versus the default configuration shown in parentheses.

Benchmark	Workload	Default	*w/o Multi-Level*	*w/o μArch*
GTSAM	Planar	6.04×	2.16× (-64.2%)	2.33× (-61.4%)
	Pose2	3.77×	2.64× (-29.9%)	3.77× (-0.1%)
	Pose3	4.21×	2.46× (-41.7%)	4.07× (-3.3%)
	SFM	5.29×	2.02× (-61.9%)	4.47× (-15.4%)
	StereoVO	1.31×	1.21× (-7.9%)	1.26× (-4.2%)
PCL	ISS	7.52×	3.20× (-57.5%)	7.35× (-2.3%)
	KDTree	4.42×	2.93× (-33.7%)	4.46× (+0.9%)
	Morph	8.24×	6.06× (-26.5%)	8.11× (-1.5%)
	Normals	5.44×	2.94× (-46.0%)	3.72× (-31.6%)
	Octree	17.88×	8.56× (-52.1%)	11.42× (-36.1%)
	ICP	4.10×	2.89× (-29.6%)	4.08× (-0.6%)
	RANSAC	3.75×	2.06× (-45.0%)	2.06× (-45.0%)
	RegGrow	5.52×	2.61× (-52.7%)	5.44× (-1.5%)
	Poisson	12.09×	7.85× (-35.1%)	11.92× (-1.4%)
TorchBench	MobileNet	8.38×	2.52× (-70.0%)	7.35× (-12.3%)
	ResNet-50	8.99×	2.92× (-67.6%)	7.00× (-22.2%)
GeoMean		5.73×	3.00× (-47.6%)	4.73× (-17.4%)

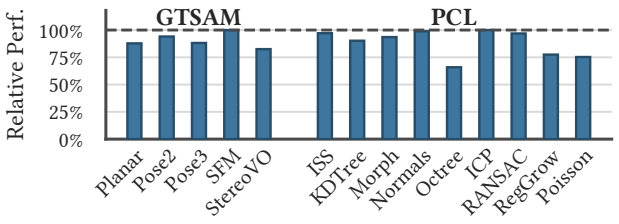


Figure 14. Relative performance of the instructions generated by ScaleISA against hand-optimized HLS (dashed line: hand-optimized = 100%).

loses much of the scalability advantage. Removing joint microarchitectural exploration also causes severe performance degradation. The geometric-mean speedup drops from 5.73× to 4.73×, decreasing by 17.4%, with especially clear losses on Planar, RANSAC, Octree, and Normals. This result validates the significance of jointly exploring microarchitectural choices during instruction customization rather than postponing them until after candidate identification.

8.3 Comparison with Hand-Optimized Hardware

To estimate how close our automatically generated hardware comes to hand-written hardware, we manually tuned the HLS code of the instructions ScaleISA selected on the 14 GTSAM and PCL workloads at the default 20,000 μm^2 budget. The manual pass only adds memory partitioning optimization, without re-identifying or re-selecting instructions, so both designs implement the same instruction set.

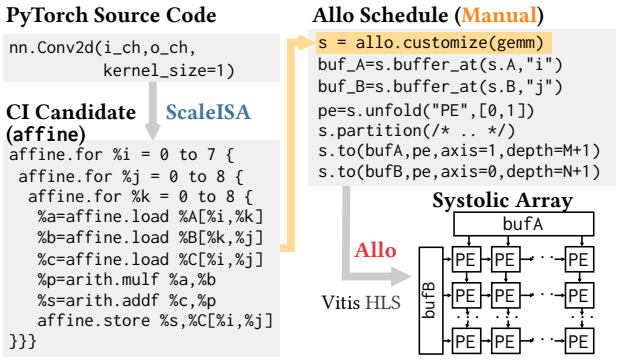


Figure 15. Case study of the ScaleISA-Allo[5] flow.

Figure 14 shows the performance of ScaleISA's automatically generated hardware relative to the manually optimized hardware. Across the 14 workloads it reaches an average relative performance of 91%.

8.4 Case Study: Integration with Allo

To explore the potential of integrating ScaleISA with high-level hardware optimization, we implement the CI candidates identified by high-level ScaleISA stages (`linalg` and `affine`) with Allo [5], a high-level schedule language for accelerator programming. More specifically, we manually design the CIs in Allo and enable various dataflow schedules. Figure 15 presents an example of designing GEMM with systolic-array construction using Allo. We skip the `scf` stage and lift instruction selection to the `affine` stage. For the selected CIs, the semi-automatic ScaleISA-Allo flow achieves speedups of 13.7× for MobileNet and 24.4× for ResNet-50 compared to the default HECTOR [58] backend. This case study demonstrates the broader potential of ScaleISA: its capability to customize instructions at a high level not only enables the identification of larger candidates but also preserves the high-level information necessary to enable more optimizations, fully unveiling acceleration potentials.

9 Related Work

Algorithms for enumerating data-flow subgraphs for fine-grained custom instructions have been well-studied [10, 11, 15, 39, 54] to handle convexity and I/O constraints. Later work [2, 3] explored instruction recurrence and isomorphism-aware identification to unveil syntactic merging opportunities. Systems such as C-cores [48], GreenDroid [19], and QsCores [49] offload functions and loops to custom accelerators. PICO [1] and FINDER [16] identify parallel instructions for VLIW architectures, while APEX [29] and RADISH [50] customize processing elements from common patterns for domain-specific reconfigurable accelerators. ScaleISA differs by identifying reusable scalable instructions at multiple IR levels with joint microarchitectural exploration.

MLIR [26] has become a popular infrastructure for building both software and hardware compilers. On the software side, MLIR compilers such as Triton [44] and XLA [33] optimize tiled neural network computations and machine learning graphs. Van Delm et al. [46] introduces MLIR-based abstractions to eliminate configuration overhead towards domain-specific accelerators. On the hardware side, HECTOR [58], Dynamatic [24], and the CIRCT project [8] use MLIR for high-level synthesis and EDA tools. `ScaleISA` leverages this ecosystem by using the multi-level structure as a search space for scalable instruction customization.

E-graphs and equality saturation [51, 61] have been widely used for rewrite-based optimization. Diospyros [47] and Isaria [43] apply equality saturation for vectorization and DSP compiler generation. Recent efforts combine e-graphs directly with MLIR, including SEER [7] for high-level synthesis super-optimization, DialEgg [60] for dialect-agnostic optimization, HEC [59] for equivalence checking, and the `eqsat` dialect [30] for non-destructive rewriting. `ScaleISA` builds on these foundations by extending multi-level e-graph anti-unification [4, 56] with MLIR-e-graph rewrites to jointly identify scalable custom instructions and microarchitectures.

10 Conclusion

This paper presents `ScaleISA`, an end-to-end framework for multi-level custom instruction identification and joint microarchitectural exploration. It introduces multi-level e-graph anti-unification to identify reusable instructions across MLIR abstraction levels and incorporates microarchitectural transformations into the identification process. `ScaleISA` achieves an average 1.42× speedup over the strongest baseline per workload, with gains up to 2.87×. Evaluation demonstrates its ability to exploit larger instructions as area budgets grow, while an Allo case study illustrates the potential for further hardware optimization.

Acknowledgments

We would like to thank our anonymous reviewers for their constructive feedback. This work was supported in part by the National Science Foundation of China (Grant No. T2325001).

References

[1] S. Aditya, B. Ramakrishna Rau, and V. Kathail. 1999. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings 12th International Symposium on System Synthesis*. 107–113. doi:10.1109/ISSS.1999.814268

[2] Junwhan Ahn and Kiyoung Choi. 2013. Isomorphism-Aware Identification of Custom Instructions With I/O Serialization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2013), 34–46. doi:10.1109/TCAD.2012.2214033

[3] Paolo Bonzini and Laura Pozzi. 2008. Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 10 (2008), 1259–1267. doi:10.1109/TVLSI.2008.2001863

[4] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-Unification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023). doi:10.1145/3571207

[5] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 593–620. doi:10.1145/3656401

[6] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. doi:10.1145/3445814.3446692

[7] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 1029–1044. doi:10.1145/3620665.3640392

[8] CIRCT community. 2025. CIRCT. https://circt.llvm.org/

[9] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006

[10] Nathan Clark, Jason A. Blome, Michael L. Chu, Scott A. Mahlke, Stuart Biles, and Krisztián Flautner. 2005. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. doi:10.1109/ISCA.2005.9

[11] Nathan Clark, Hongtao Zhong, and Scott A. Mahlke. 2003. Processor Acceleration Through Automated Instruction Set Customization. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. doi:10.1109/MICRO.2003.1253189

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24

[13] Frank Dellaert and Michael Kaess. 2017. *Factor Graphs for Robot Perception*. Vol. 6. Foundations and Trends in Robotics. doi:10.1561/2300000043

[14] Aggelos Ferikoglou, Andreas Kosmas Kakolyris, Vasilis Kypriotis, Dimosthenis Masouros, Dimitrios Soudris, and Sotirios Xydis. 2024. CollectiveHLS: Ultrafast Knowledge-Based HLS Design Optimization. *IEEE Embedded Systems Letters* 16, 2 (2024), 235–238. doi:10.1109/LES.2023.3330610

[15] Emanuele Giaquinta, Anadi Mishra, and Laura Pozzi. 2015. Maximum Convex Subgraphs Under I/O Constraint for Automatic Identification of Custom Instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 3 (2015), 483–494. doi:10.1109/TCAD.2014.2387375

[16] Vikkitharan Gnanasambandapillai, Jorgen Peddersen, Roshan G. Ragel, and Sri Parameswaran. 2020. FINDER: Find Efficient Parallel Instructions for ASIPs to Improve Performance of Large Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3577–3588. doi:10.1109/TCAD.2020.3012211

[17] Cecilia González-Álvarez, Jennifer B. Sartor, Carlos Álvarez, Daniel Jiménez-González, and Lieven Eeckhout. 2013. Accelerating an Application Domain with Specialized Functional Units. *ACM Transactions on Architecture and Code Optimization* 10, 4 (2013), 47:1–47:25. doi:10.1145/2541228.2555303

[18] Cecilia González-Álvarez, Jennifer B. Sartor, Carlos Álvarez, Daniel Jiménez-González, and Lieven Eeckhout. 2016. MInGLE: An Efficient

Framework for Domain Acceleration Using Low-Power Specialized Functional Units. *ACM Transactions on Architecture and Code Optimization* 13, 2 (2016), 17:1–17:26. doi:10.1145/2898356

[19] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Taylor. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro* 31, 2 (2011), 86–95. doi:10.1109/MM.2011.18

[20] Yueming Hao, Xu Zhao, Bin Bao, David Berard, Will Constable, Adnan Aziz, and Xu Liu. 2023. TorchBench: Benchmarking PyTorch with High API Surface Coverage. *arXiv preprint arXiv:2304.14226* (2023). https://arxiv.org/abs/2304.14226

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778. doi:10.1109/CVPR.2016.90

[22] Kari Hepola, Tharaka Ranasinghe Arachchige, Joonas Multanen, and Pekka Jääskeläinen. 2025. Automatically Retargeting Hardware and Code Generation for RISC-V Custom Instructions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2025). doi:10.1109/TVLSI.2025.3586902

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017). doi:10.48550/arXiv.1704.04861

[24] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2020. Dynamatic: From C/C++ to Dynamically Scheduled Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 1–10. doi:10.1145/3373087.3375391

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*. 75–88. doi:10.1109/CGO.2004.1281665

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. 2–14. doi:10.1109/CGO51591.2021.9370308

[27] Fei Liu, Yanping Shao, Zhouquan Liu, Jing Zhang, Junbo Tie, Mingche Lai, Gang Chen, and Libo Huang. 2025. RISC-TAE: Instruction Set Extension for Transformer Model Acceleration. In *Proceedings of the 2025 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. doi:10.1145/3742872.3757073

[28] LLVM. 2026. Torch-MLIR. https://github.com/llvm/torch-mlir. https://github.com/llvm/torch-mlir

[29] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A. Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 33–45. doi:10.1145/3582016.3582070

[30] Jules Merckx, Alexandre Lopoukhine, Samuel Coward, Jianyi Cheng, Bjorn De Sutter, and Tobias Grosser. 2025. eqsat: An Equality Saturation Dialect for Non-destructive Rewriting. *arXiv preprint arXiv:2505.09363* (2025). https://arxiv.org/abs/2505.09363

[31] Jules Merckx, Alexandre Lopoukhine, Samuel Coward, Jianyi Cheng, Bjorn De Sutter, and Tobias Grosser. 2026. E-Graphs as a Persistent Compiler Abstraction. arXiv:2602.16707 [cs.PL] https://arxiv.org/abs/2602.16707

[32] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. IEEE Press, 45–59. doi:10.1109/PACT52795.2021.00011

[33] OpenXLA Project. 2026. XLA: Optimizing Compiler for Machine Learning. https://openxla.org/xla/tf2xla. https://openxla.org/xla/tf2xla

[34] Julian Oppermann, Brindusa Mihaela Damian-Kosterhon, Florian Meisel, Tammo Mürmann, Eyck Jentzsch, and Andreas Koch. 2024. Longnail: High-Level Synthesis of Portable Custom Instruction Set Extensions for RISC-V Processors from Descriptions in the Open-Source CoreDSL Language. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. doi:10.1145/3620666.3651375

[35] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622834

[36] Weijie Peng, Youwei Xiao, Yuyang Zou, Zizhang Luo, and Yun Liang. 2025. Clay: High-Level ASIP Framework for Flexible Microarchitecture-Aware Instruction Customization. In *Proceedings of the 44th IEEE/ACM International Conference on Computer-Aided Design*. doi:10.1109/ICCAD66269.2025.11240669

[37] Gordon D. Plotkin. 1970. *Lattice Theoretic Properties of Subsumption*. Technical Report. Edinburgh University, Department of Machine Intelligence and Perception. https://books.google.com/books?id=2p09cgAACAAJ

[38] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. *ACM Transactions on Design Automation of Electronic Systems* 30, 2 (2025), 26:1–26:44. doi:10.1145/3711847

[39] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. 2006. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1209–1229. doi:10.1109/TCAD.2005.855950

[40] Evgenii Rezunov, Niko Zurstraßen, Lennart M. Reimann, and Rainer Leupers. 2025. Automatic Microarchitecture-Aware Custom Instruction Design for RISC-V Processors. In *Proceedings of the 44th IEEE/ACM International Conference on Computer-Aided Design*. doi:10.1109/ICCAD66269.2025.11240781

[41] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA)*. 1–4. doi:10.1109/ICRA.2011.5980567

[42] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (2022), 32:1–32:27. doi:10.1145/3494534

[43] Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. doi:10.1145/3617232.3624873

[44] Philippe Tillet, HT Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19. doi:10.1145/3315508.3329973

[45] David Trilla, John-David Wellman, Alper Buyuktosunoglu, and Pradip Bose. 2021. NOVIA: A Framework for Discovering Non-Conventional Inline Accelerators. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 507–521. doi:10.1145/3466752.3480094

[46] Josse Van Delm, Anton Lydike, Joren Dumoulin, Jonas Crols, Xiaoling Yi, Ryan Antonio, Jackson Woodruff, Tobias Grosser, and Marian Verhelst. 2026. The Configuration Wall: Characterization and Elimination of Accelerator Configuration Overhead. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3760250.3762225

[47] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. doi:10.1145/3445814.3446707

[48] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 205–218. doi:10.1145/1735970.1736044

[49] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 163–174. doi:10.1145/2155620.2155640

[50] Max Willsey, Vincent T. Lee, Alvin Cheung, Rastislav Bodík, and Luis Ceze. 2019. Iterative Search for Reconfigurable Accelerator Blocks With a Compiler in the Loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2019), 407–418. doi:10.1109/TCAD.2018.2878194

[51] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021). doi:10.1145/3434304

[52] Nan Wu, Yuan Xie, and Cong Hao. 2021. IRONMAN: GNN-Assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning. In *Proceedings of the 2021 Great Lakes Symposium on VLSI*. 39–44. doi:10.1145/3453688.3461495

[53] Nan Wu, Yuan Xie, and Cong Hao. 2023. IronMan-Pro: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2023), 900–913. doi:10.1109/TCAD.2022.3185540

[54] Chenglong Xiao, Emmanuel Casseau, Shanshan Wang, and Wanjun Liu. 2014. Automatic Custom Instruction Identification for Application-Specific Instruction Set Processors. *Microprocessors and Microsystems* (2014). doi:10.1016/j.micpro.2014.09.001

[55] Youwei Xiao, Fan Cui, Zizhang Luo, Weijie Peng, and Yun Liang. 2025. Cayman: Custom Accelerator Generation with Control Flow and Data Access Optimization. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference*. doi:10.1109/DAC63849.2025.11132875

[56] Youwei Xiao, Chenyun Yin, Yitian Sun, Yuyang Zou, and Yun Liang. 2026. Finding Reusable Instructions via E-Graph Anti-Unification. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. doi:10.1145/3779212.3790162

[57] Youwei Xiao, Yuyang Zou, Yansong Xu, Yuhao Luo, Yitian Sun, Chenyun Yin, Ruifan Xu, Renze Chen, and Yun Liang. 2025. Invited Paper: APS: Open-Source Hardware-Software Co-Design Framework for Agile Processor Specialization. In *Proceedings of the 44th IEEE/ACM International Conference on Computer-Aided Design*. 1–9. doi:10.1109/ICCAD66269.2025.11240817

[58] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. 2022. HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ACM, 54:1–54:9. doi:10.1145/3508352.3549370

[59] Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, and Cunxi Yu. 2025. HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association. https://www.usenix.org/conference/atc25/presentation/yin

[60] Abd-El-Aziz Zayed and Christophe Dubach. 2025. DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog. In *Proceedings of the 2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. doi:10.1145/3696443.3708957

[61] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. In *Proceedings of the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. doi:10.1145/3591239

[62] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL] https://arxiv.org/abs/2303.18223

[63] Yuyang Zou, Youwei Xiao, Yansong Xu, Chenyun Yin, Yuhao Luo, Yitian Sun, Ruifan Xu, Renze Chen, and Yun Liang. 2026. Aquas: Enhancing Domain Specialization through Holistic Hardware-Software Co-Optimization based on MLIR. In *Proceedings of the 45th IEEE/ACM International Conference on Computer-Aided Design*. ACM. doi:10.1145/3831252.3834028