MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations

ABDUL RAFAE NOOR, University of Illinois at Urbana-Champaign, USA
DHRUV BARONIA, University of Illinois at Urbana-Champaign, USA
AKASH KOTHARI, University of Illinois at Urbana-Champaign, USA
MUCHEN XU, University of Illinois at Urbana-Champaign, USA
CHARITH MENDIS, University of Illinois at Urbana-Champaign, USA
VIKRAM S. ADVE, University of Illinois at Urbana-Champaign, USA

Using program synthesis to select instructions for and optimize input programs is receiving increasing attention. However, existing synthesis-based compilers are faced by two major challenges that prohibit the deployment of program synthesis in production compilers: exorbitantly long synthesis times spanning several minutes and hours; and scalability issues that prevent synthesis of complex modern compute and data swizzle instructions, which have been found to maximize performance of modern tensor and stencil workloads. This paper proposes Misaal, a synthesis-based compiler that employs a novel strategy to use formal semantics of hardware instructions to automatically prune a large search space of rewrite rules for modern complex instructions in an offline stage. Misaal also proposes a novel methodology to make term-rewriting process in the online stage (at compile-time) extremely lightweight so as to enable programs to compile in seconds. Our results show that Misaal reduces compilation times by up to a geomean of 16x compared to the state-of-the-art synthesis-based compiler, Hydride. Misaal also delivers competitive runtime performance against the production compiler for image processing and deep learning workloads, Halide, as well as Hydride across x86, Hexagon and ARM.

CCS Concepts: • **Theory of computation → Program semantics**; **Automated reasoning**; *Rewrite systems*; **Grammars and context-free languages**; • **Software and its engineering → Retargetable compilers**.

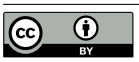
Additional Key Words and Phrases: Code Optimization, Static Analysis, Compilers, Synthesis, Semantics

ACM Reference Format:
Abdul Rafae Noor, Dhruv Baronia, Akash Kothari, Muchen Xu, Charith Mendis, and Vikram S. Adve. 2025. MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations. *Proc. ACM Program. Lang.* 9, PLDI, Article 198 (June 2025), 24 pages. https://doi.org/10.1145/3729301

1 Introduction

Domain-specific extensions to existing hardware architectures are emerging to meet the performance demands of modern workloads in domains such as deep learning, image processing, etc. For instance, Qualcomm's Hexagon DSP [5, 13], Intel's AVX ISA [8] and ARM's Neon ISA [3] have been extended with specialized instructions to optimize vector, tensor and stencil computations

Authors' Contact Information: Abdul Rafae Noor, University of Illinois at Urbana-Champaign, Urbana, USA, arnoor2@illinois.edu; Dhruv Baronia, University of Illinois at Urbana-Champaign, Urbana, USA, baronia3@illinois.edu; Akash Kothari, University of Illinois at Urbana-Champaign, Urbana, USA, akashk4@illinois.edu; Muchen Xu, University of Illinois at Urbana-Champaign, Urbana, USA, muchenx2@illinois.edu; Charith Mendis, University of Illinois at Urbana-Champaign, Urbana, USA, charithm@illinois.edu; Vikram S. Adve, University of Illinois at Urbana-Champaign, Urbana, USA, vadve@illinois.edu.



ACM 2475-1421/2025/6-ART198
https://doi.org/10.1145/3729301

including vector instructions that perform dot products, reductions, in-register data movement across vector lanes (swizzle operations), etc.

For nearly a decade, the conventional practice of supporting emerging ISAs and new extensions to existing ISAs has been to manually implement back ends for each hardware target in high-performance and/or domain-specific compilers such as Halide [14], TVM [4], MLIR [10], etc. This entails implementing a large set of pattern-matching rules to map sequences of input operations to complex target instructions — a practice that has three major limitations: it is error-prone; it requires an enormous amount of engineering time and effort; and pattern-matching rules are brittle – i.e., these rules dictate exactly what operations must appear and in what order in the input code to map to output code – and consequently are invariably incomplete and miss occasional high-performance cases.

In order to address these limitations, recent papers have proposed the use of program synthesis techniques to implement back end code generators for modern hardware targets [1, 9, 15, 17, 19]. These works eliminate the need to manually implement compiler back ends using pattern-matching rules since they leverage the semantics of an input language, an intermediate representation, and target hardware instruction, to automatically generate target code using some form of search. Moreover, these approaches provide formal correctness guarantees for the generated code which are not directly inferable from traditional methods. These correctness guarantees do not compromise performance, as program synthesis-based approaches offer competitive performance compared to hand-implemented compiler backends.

Prior related work offer point solutions to applying synthesis in automated compiler construction for specific limited contexts. Diospyros [19] is a vectorizer that uses a set of manually-defined rewrite rules for a set of 12 vector instructions to vectorize scalar code using equality saturation [16] for a Tensilica DSP. Isaria [17] improves on Diospyros by automatically generating rewrite rules using an ISA specification in an offline stage. However, both works don't support generation of complex compute and data swizzle instructions supported by modern ISAs. Rake [1] uses program synthesis to lower small sequences of input code operations to target vector instructions at compile time. Because it requires engineers to manually implement semantics of target instructions, it only supports small subsets of target ISAs (a few hundreds of instructions). It uses specialized heuristics to make generation of target cross-lane compute instructions, and data interleave and deinterleave instructions possible; however, synthesis still takes several minutes/hours to complete. Pitchfork [15] uses Rake in an offline stage to synthesize rewrite rules for short sequences of enumerated input IR operations with target-specific fixed-point compute vector instructions. It then uses a custom lightweight term rewriter to apply the rewrite rules on input code at compile time – this enables it to compile programs in a few seconds. However, Pitchfork does not support generation of complex data movement instructions unlike Rake.

Hydride [9] attempts to address some of the limitations of the aforementioned synthesis-based compilers. It automatically generates formal semantics of instructions from their pseudocode descriptions in official documentation to avoid the tedium and error-proneness of hand-implementing semantics for thousands of instructions, while maximizing instruction coverage and achieving competitive performance to manually written compiler backends. Hydride uses these semantics to create similarity classes of target ISA operations, and automatically generates a mid-level compiler IR called AutoLLVM using each class as one parameterized IR operation, representing all instructions in each class by different combinations of parameter values. Hydride uses program synthesis to translate the Halide front-end IR to AutoLLVM IR, greatly reducing complexity by targeting equivalence classes of instructions instead of individual ones. Nevertheless, Hydride still suffers from serious performance limitations like previous synthesis-based compilers: exorbitant synthesis

times spanning several minutes/hours; and using specialized heuristics to enable generation of complex data swizzle instructions.

Collectively, the recent synthesis-based compilers have achieved four major advances: (a) the use of synthesis techniques instead of pattern matching for code generation in compilers, with the added benefit of formal verification of the code generators; (b) automatic generation of translation components from formal semantic specifications of input languages, compiler IRs, and target ISA semantics; (c) partial progress towards better instruction coverage for complex instruction sequences, especially cross-lane vector compute and data movement operations; and (d) demonstrating that generating rewrite rules in an offline stage and performing term rewriting at compile time helps circumvent long program synthesis times at compile time.

Nevertheless, synthesis-based compilers are faced with a few remaining fundamental challenges that prohibit their deployment in real-world production systems, all essentially about obstacles to scalability for supporting large, real-world ISAs such as x86, Hexagon and ARM:

- ***Challenge 1:*** Program synthesis for large ISAs (and even for multiple ISAs) is intractable in today's approaches because it requires navigating an exponentially large search space with sufficient coverage in order to generate a large number of high-quality candidate rewrite rules.
- ***Challenge 2:*** Complex cross-lane vector instructions with implicit data movements exacerbate the scaling problem because they require enumerating relatively long sequences of compute and explicit data swizzle operations in order to generate high-quality rewrite rules for high performance code.
- ***Challenge 3:*** Generating rewrite rules using program synthesis enables the cost of synthesis to be moved offline instead of online (i.e., during compile time), but the size and complexity of target ISAs causes an explosion in the number of rewrite rules, which makes the rewriting systems used at compile time intractable with today's approaches.

We design a system, Misaal, that addresses these major challenges. Because enumerating equivalence relations between sequences of thousands of instructions from large ISAs (such as x86, Hexagon and ARM) to generate a system of rewrite rules is infeasible (challenge 1), in the offline stage, Misaal performs enumeration on the equivalence classes of target instructions generated by Hydride instead since the number of equivalence classes is significantly lower than the total number of supported instructions across different hardware targets. In order to support generation of complex rewrites for complex cross-lane compute and data swizzle instructions (challenge 2), Misaal uses the formal semantics of target-independent instructions to automatically derive the appropriate data swizzle patterns required to use those instructions and then runs Hydride's *similarity analysis* to represent those data swizzle patterns in the search space using equivalence classes to further aggressively prune the search space. This approach results a large number of candidate rewrite rules which when applied to input code using equality saturation at compile-time (the online stage) lead to formation of large e-graphs (i.e., large amount of memory consumption) and substantially long compilation times (challenge 3). Misaal addresses this challenge by using the knowledge of semantics of target instructions to automatically abstract target-specific rewrite rules to produce a significantly smaller set of target-independent rewrite rules. This prevents the explosion of the sizes of e-graphs and enables the overhead of performing term rewrites at compile time to be small.

In summary, the key contributions of this paper are:

- A novel methodology that uses the equivalence classes of target instructions to reduce the exponentially large search space to enumerate equivalence relationships between different

sequences of operations to generate a large system of candidate rewriting rules even for ISAs like x86 and ARM with thousands of instructions.
- A novel scalable semantics-based pruning strategy that automatically derives equivalence classes for complex data swizzle patterns in the search space to aggressively prune the search space and enable generation complex rewrite rules for complex cross-lane vector instructions with implicit data movements with large sequences of compute and explicit data swizzle operations. We observe that this approach reduces the number of data swizzle patterns in the search space by ~11x.
- A new methodology to abstract the automatically-generated target-specific rewrite rules into target-independent rewrite rules. We observe that Misaal significantly reduces size of the system of rewrite rules down to little over 2,200 rules, by ~19x.
- A comprehensive evaluation of Misaal for real-world image processing and deep learning kernels on x86, Hexagon, and ARM architectures. Our results show that Misaal compiles workloads 16x times faster than the state-of-the-art compiler, Hydride for x86, 9x times faster for Hexagon and 10x faster for ARM; and that Misaal achieves competitive performance relative to the carefully engineered production compiler for Halide and Hydride for all three architectures.

2 Background

Hydride [9] is a retargetable synthesis-based compiler infrastructure for automatically generating code generation support for different hardware ISAs. Hydride comprises two major components described below and depicted in Figure 1.

Hydride Automatic IR Generator. This component operates in an offline stage to automatically generate formal semantics of hardware instructions by parsing their pseudocode descriptions in official documentation. It eliminates the need to go through the tedious and error-prone process of hand-implementing semantics for thousands of instructions, while maximizing instruction coverage. Using these semantics, it performs a *similarity analysis* across the semantics of target-specific instructions to automatically put *similar* target instructions – i.e., instructions that perform represent similar computational and data movement patterns regardless of hardware-specific characteristics such as register sizes, precision types, etc. – in *equivalence classes*, each of which are then represented as parameterized target-independent instructions that represent multiple target-specific instructions as a part of a language-independent and target-independent intermediate representation, *AutoLLVM IR*. This component also automatically generates support for performing one-on-one translation from AutoLLVM IR to target-specific instructions in LLVM.

Hydride Code Synthesizer. This component is a program synthesizer that compiles expressions in a front end language, such as Halide IR or MLIR [10] dialects, to AutoLLVM IR expressions. It synthesizes code at compile time; it employs various heuristics to make synthesis tractable, such as partitioning large expressions into non-overlapping sub-expressions which are synthesized separately, selectively excluding AutoLLVM IR operation in synthesis, hand-crafting specialized swizzles to enable generating cross-lane operations, and caching synthesis results for reuse across expressions. While Hydride's Code Synthesizer is retargetable, the compilation with Hydride can several minutes and, in some cases, several hours despite using several heuristics primarily due to the inherent scalability issues associated with the synthesizer exploring a search space of thousands of target instructions to synthesize code.

Because Hydride provides formal semantics for AutoLLVM IR instructions, support for performing Similarity Analysis between target-specific instructions, and basic support for translating AutoLLVM IR instructions to target-specific instructions, Misaal builds on top of the Hydride to

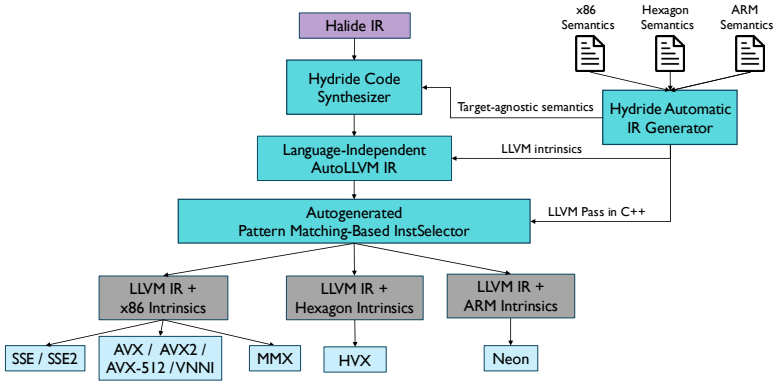


Fig. 1. Hydride's Workflow diagram largely consisting of two phases. Offline, the Hydride Automatic IR generator produces formal semantics for ISAs and abstracts the semantics into parameterized AutoLLVM. Online, the Hydride Code Synthesizer uses program synthesis to compile Halide IR expressions to concretely parameterized AutoLLVM IR.

replace its Code Synthesizer and generate code in a matter of a few seconds as opposed to several minutes or hours.

3 Design

In this section, we describe in detail how Misaal mitigates the challenges described above. Misaal operates in two phases: 1) An offline compiler construction phase which performs synthesis to derive retargetable rewrite rules, and 2) An online lightweight term rewriting phase which uses those rules to compile the application to the target architecture. Figure 2 illustrates the workflow for the offline phase.

Scaling Synthesis for Large ISAs. Misaal uses the formal semantics of the target ISAs and the frontend IR to derive retargetable rewrite rules. Enumeration is required to generate a rich collection of rewrite rules including complex cross-lane vector and data-swizzling operations. However, the large size of the ISAs with potentially thousands of ISA operations makes naive enumeration infeasible. To this end, Misaal employs a novel semantics-driven enumeration methodology to enable both enumeration and synthesis to scale to required expression depths. First, the ISA formal semantics are generated and abstracted into the AutoLLVM IR representation. The AutoLLVM IR representation, also referred to as equivalence classes, folds multiple semantically 'similar' operations into a target agnostic representation with parameterizations to exactly represent each of the folded operations. At the core of Misaal is a novel enumeration methodology (Section 3.1) that enumerates expressions using these exponentially more compact AutoLLVM IR representations to derive rewrite rules, which can be abstracted into retargetable rewrite rules.

After this, Misaal systematically decouples required data-swizzling generation from the ISA's computational instruction semantics to synthesize rewrite rules with both complex computation and data-swizzling semantics. To support enumeration of rewrite rules with complex data-swizzling operations (often realized by stitching together multiple simpler swizzles), Misaal uses the semantics of the target ISAs to extract the data-access patterns from each ISA operation. These access patterns are used to create complex data-swizzling operations which themselves can be folded into new AutoLLVM IR operations for use in enumeration (Section 3.3).

Synthesizing rewrite rules for large depths inevitably requires pruning the enumeration space. Misaal uses the formal semantics of the AutoLLVM IR operations to systematically prune the enumeration space such that it is automatically tailored to each target architecture's ISA. Only

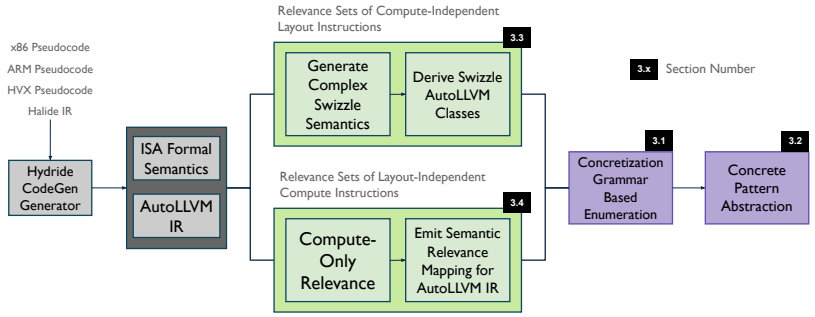


Fig. 2. Misaal Workflow for offline Compiler Construction phase

AutoLLVM IR operations that share some semantic relatedness, regardless of their data access patterns, are to be enumerated together (Section 3.4).

After the enumeration terminates, yielding rewrites on AutoLLVM IR operations with specific concrete parameterizations (referred to as concrete patterns), Misaal abstracts the generated rewrite rules into higher-order retargetable rewrite rules with symbolic parameterizations that are automatically verified to be correct. Additionally, the retargetable rewrite rules drastically compress the number of rewrite rules (Section 3.2).

Lightweight Online Compilation. Retargetable rewrites generated offline by Misaal are fed to a term rewriting system (EggLog [20] for this work) to compile applications to the target architecture (Section 5.1). The generated AutoLLVM IR is target-specific (i.e., parameter values for AutoLLVM IR operations have been selected) and is finally lowered to the equivalent target ISA operations. The remainder of this section elaborates on the described workflow in detail.

3.1 Equivalence Class Based Enumeration

Using enumeration offline to generate rewrite rules for compiling to target architectures is an increasingly common approach to enable reasonable compilation times online. However, for large ISAs containing thousands of instructions, enumeration itself is intractable even when performed offline. To address this inherent scalability problem in enumeration, Misaal uses AutoLLVM IR [9] as the target agnostic representation for target ISAs to perform enumeration. Hydride has shown that performing synthesis *online* using the AutoLLVM IR representation can enable synthesis to be tractable, but can still be expensive, taking hours of synthesis time for a single expression in some cases. We describe a novel enumeration and synthesis methodology using AutoLLVM IR to not only make enumeration for rewrite rules tractable, but also enable fast synthesis while importantly covering the equivalent space of programs as when using the target ISA.

A rewrite rule can be defined by a 3-tuple (LHS, RHS, $\simeq$) where LHS and RHS describe the left-hand-side and right-hand-side expressions respectively and $\simeq$ describes the relation under which the expressions are related. An example of $\simeq$ is the equivalence relation. Traditionally, $LHS \equiv RHS$ (is equivalent to) for all symbolic inputs with type T, where T are the input types for LHS. As a valid-rewrite rule requires equivalence (i.e. equality over all symbolic values of the required types), LHS and RHS are consequently target specific ISA operations with fixed types. These rewrite rules are not portable to other operations across different target ISAs or even within similar instructions of the same architecture. Consider an example of rewrites for dot-product on x86. Figure 3(a) shows the program rewrite rule (in Halide IR) for the 128-bit dot-product operation, while Figure 3(b) presents the corresponding 512-bit dot-product program. These programs represent distinct instructions in the x86 ISA, and operate on different vector register sizes. Additionally, the intermediate values span different intermediate vector sizes (256, and 1024 respectively). To derive these rewrite rules

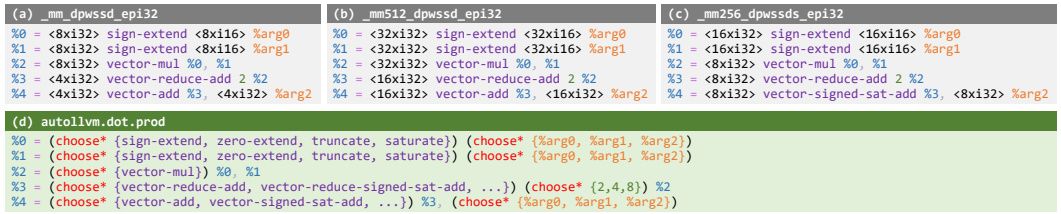


Fig. 3. Program representation of the semantics of multiple dot product instructions in x86. (a) and (b) describe the semantics of dot-product on different vector sizes while (c) describes a dot product operation with saturating accumulation. Subfigure (d) describes an example of the concretization grammar during enumeration which captures (a), (b), and (c).

under the ≡ equivalence relation, the enumeration must consist of a range of vector sizes. Coupled with the additional range of element bit-widths of the intermediate values in the programs and the required sequence length of 5 operations, it is easy to see how enumeration quickly becomes intractable.

To illustrate the complexity, Figure 3(c) shows the rewrite rule for a similar x86 dot-product variant using signed saturating accumulation. The programs are not equivalent but are structurally similar. Figures 3(a) and 3(b) use different vector register sizes, while Figures 3(a) and 3(c) differ in vector sizes and accumulation methods. This redundancy worsens as the rewrite length increases. Manually identifying and folding these cases is error-prone and impractical for large ISAs.

To mitigate this redundancy in enumeration, we employ Hydride's Similar Instruction analysis [9] to automatically create an abstraction which represents multiple semantically *similar* instructions, namely, the AutoLLVM IR. We leverage the insight that rewrite rules capture semantic equivalence while AutoLLVM IR captures semantic similarity. Therefore similar rewrite rules on specific ISA operations representing different element bitwidths, vector sizes and bitvector operation variants (as shown in Figures 3(a-c)) can be represented under a single rewrite rule on AutoLLVM IR operations (with different parameterizations). The pair of AutoLLVM IR expressions in such a rewrite rule represent a **template** under which similar rewrite rules can be represented.

To formalize this notion, we define two functions.

$$Abstraction(TargetInstExpr\ C) = \text{AutoLLVM IR representation of C}$$

For example, $Abstraction($`__mm_dpwssd_epi32`$) =$ `autollvm.dot.prod`, where `autollvm.dot.prod` is the automatically derived target agnostic abstraction for dot-product-like target instructions. The abstraction function can also be applied to expressions of target instructions. Conversely we define:

$$Concretization(AutoLLVMInstExpr\ A) =$$

$$\{\text{ Set of all concretely parameterized programs represented by expression A }\}.$$

Note that the abstraction function returns a single AutoLLVM IR program parameterized with symbolic parameters, while the concretization function returns a set of programs.

In existing approaches for deriving rewrite rules, two expressions are sampled from the ISA operation enumeration space and tested for equality using symbolic inputs. Instead of enumerating using target specific ISA operations, Misaal enumerates using automatically generated AutoLLVM IR to generate rewrite rule templates. Table 1 describes the size of AutoLLVM IR with respect to multiple targets. The smaller IR size enables far more scalability in enumeration by defining a new ≃. In this setting we define LHS_{IR} and RHS_{IR} to be the left-hand-sided and right-hand-sided **AutoLLVM IR** expression for a possible rewrite rule. Misaal proposes a new definition of ≃,

Table 1. AutoLLVM IR statistics across different target architectures [9].

Architecture	ISA Size	AutoLLVM IR Size	Reduction in ISA size
x86	2,029	136	14.9x
HVX	307	115	2.7x
ARM	1,221	177	6.9x

denoted $\simeq_{IR}$ such that $(LHS_{IR}, RHS_{IR}, \simeq_{IR})$ is a valid **template** for a rewrite rule if and only if:

$$\exists l \in Concretization(LHS_{IR}), \exists r \in Concretization(RHS_{IR}), l \equiv r \quad (1)$$

Finding these l and r which satisfy the equivalence in Problem 1 can be efficiently solved by formulating it as a SyGus (Syntax Guided Synthesis) problem. The syntax for both expressions is derived from the structure of the AutoLLVM IR programs being enumerated, but we introduce a symbolic choice (denoted with `choose*` according to Rosette's[18] convention) for which target instruction (i.e. concretizations) to use for each AutoLLVM IR operation. Formally, the `choose*` function is used to create symbolic choices from a set of possible values and must evaluate to exactly one of the possible provided choices while satisfying some specified constraints. The symbolic choices may be from a set of operands, intermediate values, or AutoLLVM IR concretizations. We refer to the synthesis grammar for such a SyGus formulation as Concretization Grammar for AutoLLVM IR. The corresponding *Concretization Grammar* for the dot-product expressions described previously is shown in Figure 3(d). The structure of the concretization grammar shown is equivalent to the programs described previously, but the choice of which operations to use is expressed as symbolic choices. Note that while `sign-extend` is listed once in Figure 3(d) for brevity, the symbolic choice includes all variants of sign-extension for different element-bitwidths and vector register sizes. Additionally, the concretization grammar folds in the choice of input argument combinations into a symbolic choice (i.e. `(choose* {%arg0, %arg1, %arg2)}`) so that multiple enumerations are not required for aligning terminals of the source and target expressions. All x86 (2-point) dot product variants on different element bit-widths, accumulation operations, signedness, and vector sizes are captured in the above single concretization grammar providing equivalent coverage to the separate multiple explicit enumerations. Figure 4 describes the AutoLLVM IR rewrite rule template derived from the x86 dot-product instructions in Figure 3(a-c) with a valid concretization of the template corresponding to `_mm_dpwssd_epi32`.

More generally, for any given rewrite rule $LHS_{IR} \simeq_{IR} RHS_{IR}$, the synthesis problem jointly searches for any concretizations of both expressions under which they produce symbolically equivalent outputs. This greatly reduces the enumerated terms by a factor of $|Concretization(LHS_{IR})| \times |Concretization(RHS_{IR})|$. The output of this phase produces only a single valid concretization as a template, if it exists, for a pair of AutoLLVM IR expressions. Other concretizations may exist for the pair of AutoLLVM expressions which would need to be captured. Section 3.2 describes how the derived concretization template is abstracted to apply across these (potentially) multiple concretizations.

3.2 Target-Agnostic Rewrite Rule Abstraction

After Equivalence class based enumeration (Section 3.1) identifies any *single* concretizations for pairs of AutoLLVM IR expressions for which the rewrite would be semantically equivalent, Misaal uses synthesis to abstract the rewrite into a retargetable rewrite rule. The existence of at least one possible valid pair of concretizations signals that others may also exist. Misaal uses this concretization as a template to extract the possible concretizations which may exist for which the rewrite rule may be valid (i.e. the pair of concretized expressions are equivalent symbolically). First, other potential

(a) LHS_{IR} for abstract rewrite rule template

```
%0 = @autollvm.dot.prod %arg0 %arg1 %arg2 %accType %vsize %bitwidth
```

(b) RHS_{IR} for abstract rewrite rule template

```
%0 = @autollvm.extend %arg0 %vsize %bitwidth <numeric params>
%1 = @autollvm.extend %arg1 %vsize %bitwidth <numeric params>
%2 = @autollvm.vector-mul %0, %1 <numeric params>
%3 = @autollvm.vector-reduce-add 2 %2 <numeric params>
%4 = @autollvm.vector.add %3, %arg2  %accType <numeric params>
```

(c) Possible Concretization for LHS_{IR}

```
//  _mm_dpwssd_epi32
%0 = @autollvm.dot.prod %arg0 %arg1 %arg2 0 128 16
```

(d) Possible Concretization for RHS_{IR}

```
%0 = <8xi32> sign-extend <8xi16> %arg0
%1 = <8xi32> sign-extend <8xi16> %arg1
%2 = <8xi32> vector-mul %0, %1
%3 = <4xi32> vector-reduce-add 2 %2
%4 = <4xi32> vector-add %3, <4xi32> %arg2
```

Fig. 4. Example for an abstract rewrite rule template for x86's dot-product. (a) and (b) denote the LHS and RHS AutoLLVM IR expressions template. Some numeric parameters have been elided for space and simplicity. (c) and (d) describe a valid concretization for the 128-bit dot-product instruction for which (a) and (b) would be symbolically equivalent.

concretizations are extracted by issuing synthesis queries from Section 3.1 with additional semantic constraints. These constraints include different vector register sizes for the inputs and/or outputs, as well fixing the usage of a given parametrization a given AutoLLVM IR operation within the source and / or target expression. Once these expressions have been enumerated, yielding potentially tens of various valid concrete rewrite rules, Misaal abstracts the numeric parameters such that a single higher order complex rewrite represents all the enumerated concretizations for the pair of AutoLLVM IR expressions in the template. This is achieved by synthesizing arithmetic expressions for the numeric parameters which appear in the target expression using the numeric parameters from the source expression. Consider the example of performing element-wise addition on 256 bit-vectors with 8-bit elements shown in Figure 5(a). Another semantically equivalent program is shown in Figure 5(b). This program, slices the vector inputs into two halves, adds the corresponding halves of the inputs separately, and then finally concatenates the results.

This rewrite rule corresponds to a single concretization derived by Misaal. It is apparent that this rule corresponds to a more general rule which can be abstracted. Specifically, the slicing offset, number of elements, bitwidth, and vector sizes can be lifted to symbolic expressions. The numeric parameters are extracted at the corresponding parameter position across the multiple

(a) Element-wise add on 256-bit vector

```
%0 = <32 x i8 > vec-add %arg0 %arg1
```

(b) Concatenation of two 128-bit vector additions

```
// Slice Low Half ...
%0 = <16 x i8> slice_vector %arg0
            /* offset */       0,  /* Stride */ 1,
            /* Num Elements */ 16,
            /* Bitwidth */     8,
            /* Vector Size */  256
%1 = <16 x i8> slice_vector %arg1
            /* offset */       0,  /* Stride */ 1,
            /* Num Elements */ 16,
            /* Bitwidth */     8,
            /* Vector Size */  256
// Elided Slice High Half for space ...
%4 = <16 x i8> vector-add    %0 %1
%5 = <16 x i8> vector-add    %2 %3
%6 = <32 x i8> concat_vector %4 %5 8 128
```

(c) Abstracted Element-wise add on full vector

```
%0 = vec-add %arg0 %arg1 %bitwidth %vectorsize
```

(d) Abstracted vector addition by slicing and concatenating

```
// Slice Low Half ...
%0 = slice_vector %arg0
    /* offset */       0,  /* Stride */ 1,
    /* Num Elements */(%vectorsize/(%bitwidth*2)),
    /* Bitwidth */     %bitwidth,
    /* Vector Size */  %vectorsize
%1 = slice_vector %arg1
    /* offset */       0,  /* Stride */ 1,
    /* Num Elements */(%vectorsize/(%bitwidth*2)),
    /* Bitwidth */     %bitwidth,
    /* Vector Size */  %vectorsize
// Elided Slice High Half for space ...
%4 = vector-add    %0 %1 %bitwidth (%vectorsize/2)
%5 = vector-add    %2 %3 %bitwidth (%vectorsize/2)
%6 = concat_vector %4 %5 %bitwidth (%vectorsize/2)
```

Fig. 5. (a) and (b) describe the rewrite rule derived from Halide IR programs which compute element-wise vector addition 512-bit vectors with 8-bit elements. (a) computes the sum directly from the inputs, while (b) computes the results by splitting the operands and concatenating the partial sums. (c) and (d) describe the abstracted version of the same rewrite rule with symbolic parameters. The appropriate symbolic expression is synthesized for corresponding parameters as shown in (d). This abstracted rule captures multiple different vector sizes and element bitwidths in a single rule compactly.

concrete rewrite instances, and are replaced with a symbolic expression which 'fit' the parameters. Figures 5(c) and 5(d) describe the abstracted rewrite rule for the concrete 256-bit vector add example described previously. This abstraction enables Misaal to represent a large number of rewrite rules compactly while covering the larger number of potentially valid concretizations. In the next subsections, we describe how Misaal's enumeration incorporates generating rewrite rules with complex data-swizzling operations (Section 3.3) and how we can further scale generating rewrite rules for large program sequences with systematic semantics-based pruning of the enumeration space (Section 3.4).

3.3 Automatic Complex Swizzle Discovery

To discover efficient rewrite rules using complex cross-lane operations, rewrite rules must include the required data-swizzling. The issue however is that is not entirely obvious what the required data-swizzle is and when to use such a data-swizzle. Synthesizing this required swizzle from a general permute instruction which takes the swizzle mask is intractable. The swizzle patterns are often not only target-dependent, but also instruction dependent within a target architecture. Compiler writers have to manually reason about the semantics of the ISA operations and relate swizzling instructions to computation instructions. For architectures such as x86, multiple simpler swizzling ISA operations have to be stitched together to realize a complex data movement. It is easy to see how scaling to the required depths of expressions for these architectures is infeasible.

Existing works either do not generate rewrite rules with swizzles [15, 17], or used fixed sets of crafted swizzle patterns [1, 9] to enable synthesizing programs with data-movement. For new architectures, new swizzles patterns would need to be created which requires expert knowledge of the target semantics. Additionally, there is little guarantee of completeness of the swizzles being sufficient to capture the many complex data-movements which may arise.

To address this limitation and enable generating rewrite rules with complex data-swizzling, Misaal automatically derives a set of complex data-swizzles offline purely from the semantics of each target architecture ISA. This is performed by leveraging the availability of the pseudo-code of the ISA operations to analyze the data-access patterns and to create swizzles which produce vectors in the required data-access patterns.

Consider the illustrative example of HVX's widening multiplication operation shown in Figure 6(a). The target instruction *vmpybv* accesses the operands in an 'even-odd' interleaved manner after sign-extending the values. To leverage this instruction for an input program which needs to apply widening multiplication and accumulation on contiguous values C0-C3, the input program must apply some form of swizzling within the above sequence such that the final result is correct with respect to the ordering of the input contiguous values (i.e. A0-A3). Misaal analyzes the bitvector slices accessed across output lanes and creates two categories of swizzles for this instruction since either swizzling transformation is valid and different programs may require either (or both) variants:

(1) Interleaves (i.e. interleaving even-odd elements) the operands of the widening multiplication (Figure 6(b))
(2) Deinterleaves (i.e. interleaving elements from first half and second half) the result of the widening multiplication intrinsic ((Figure 6(c))).

Sets of swizzle patterns are automatically derived from each target instruction's semantics access patterns while accounting for different variants of vector register sizes, element bit-widths, offsets, and number of operands. In cross-lane vector operations, multiple elements (of the same operand) may be accessed to produce a single output lane value. For these instructions, some form of packing is necessary to layout the data. The input vectors for this packing operation may come from a single vector operand, or across multiple vector operands. For such instructions,

(a) Simplified version of HVX's widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A1.s, A2.s, A3.s]
%arg1.sext = [B0.s, B1.s, B2.s, B3.s]
// Interleaved Access Pattern
%widen.mul = [(A0.s x B0.s), (A2.s x B2.s), (A1.s x B1.s), (A3.s x B3.s)]
```

(b) Interleave Operands before widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Interleave Operands
%arg0.i = [A0, A2, A1, A3]
%arg1.i = [B0, B2, B1, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A2.s, A1.s, A3.s]
%arg1.sext = [B0.s, B2.s, B1.s, B3.s]
%widen.mul = [(A0.s x B0.s),(A1.s x B1.s),
              (A2.s x B2.s),(A3.s x B3.s)]
%result = vector-add %widen.mul [C0, C1, C2, C3]
```

(c) Deinterleave result after widening multiplication

```
%arg0 = [A0, A1, A2, A3]
%arg1 = [B0, B1, B2, B3]
// Sign Extend Inputs
%arg0.sext = [A0.s, A1.s, A2.s, A3.s]
%arg1.sext = [B0.s, B1.s, B2.s, B3.s]
%widen.mul = [(A0.s x B0.s), (A2.s x B2.s),
              (A1.s x B1.s), (A3.s x B3.s)]
// Deinterleave product after widening
%widen.mul.d = [(A0.s x B0.s), (A1.s x B1.s),
                (A2.s x B2.s), (A3.s x B3.s)]
%result = vector-add %widen.mul.d [C0, C1, C2, C3]
```

Fig. 6. (a) Simplified illustration of the implicit data-swizzling in the semantics of HVX's widening multiplication. Operands are accessed in an even-odd interleaved fashion. (b) describes the swizzling of the vector operands before widening multiplication. (c) describes performing the deinterleaving swizzle on the output of widening multiplication before accumulation.

(a) Dot-product with non-cross lane access with elements from across operands

```
%0 = <8xi32> sign-extend <8xi16> %arg0
%1 = <8xi32> sign-extend <8xi16> %arg1
%2 = <8xi32> vector-mul %0, %1
%3 = <8xi32> sign-extend <8xi16> %arg2
%4 = <8xi32> sign-extend <8xi16> %arg3
%5 = <8xi32> vector-mul %3, %4
%6 = <8xi32> vector-add %2, %5
%7 = <8xi32> vector-add %6, %accum
```

(b) Dot-product with cross-lane accesses requiring swizzles

```
// Interleave 16-bit elements 2 from 128-bit
vectors
%0 = misaal.interleave.16b.2op.128b %arg0 %arg2
%1 = misaal.interleave.16b.2op.128b %arg1 %arg3

// Apply dot product ( cross-lane access )
// on interleaved vectors
%2 = _mm256_dpwssd_epi32 %accum %0 %1
```

Fig. 7. Equivalent programs for realizing the dot-product semantics. (a) Implements the dot-product using element-wise SIMD operations without data-swizzling. (b) Implements the dot-product using Misaal automatically generated swizzles and cross-lane target dot-product instruction.

Misaal explicitly generates the swizzle which produces the desired packing from n input vector registers (where $1 \leq n \leq 4$). Figure 7 illustrates an example of the derived packing swizzles in the context of x86 dot-product instructions. The programs shown produce equivalent output, however Figure 7(b) interleaves alternating elements of $\%arg0$ and $\%arg2$ using Misaal derived swizzles to produce a packed vector which is twice as long as the individual operands. This enables the higher performance `_mm256_dpwssd_epi32` instruction to perform a 2-point horizontal dot-product on the packed vectors producing equivalent output to the non-swizzled program in Listing 7(a).

After deriving these swizzle patterns from the target instruction semantics, Misaal applies Hydride's Similarity analysis on these swizzles to create AutoLLVM IR of target specific swizzles to abstract away parameters such as element-bitwidth, swizzle-offsets, number of elements, and vector register sizes. Figure 8 shows an example of the AutoLLVM IR representation of the swizzle used in Figure 7 with different parameterizations implementing different swizzling behavior.

Table 2 describes the number of concrete swizzles with their AutoLLVM IR representation. This compact representation captures the possible data swizzling required while marginally increasing the size of the IR (up to a maximum of 7% for HVX) to be enumerated. Misaal also maintains a mapping of which swizzle AutoLLVM IR equivalence class was derived from which target instruction. As a result of this analysis, Misaal can now enumerate expressions with the required

Dot-product with cross-lane accesses requiring swizzles

```
// AutoLLVM parameterization for complex swizzle
@autollvm.interleave(%arg0, %arg1, %bitwidth, %vectorsize, %offset, %numElem)
// Fully interleave alternating 16-bit elements from 128-bit vectors
@autollvm.interleave(%arg0, %arg1, 16,        128,         0,       8)
// Interleave the first half of alternating 32-bit elements from 1024-bit vectors
@autollvm.interleave(%arg0, %arg1, 32,        1024,        0,       16)
// Interleave the second half of alternating 32-bit elements from 1024-bit vectors
@autollvm.interleave(%arg0, %arg1, 32,        1024,        16,      16)
```

Fig. 8. AutoLLVM IR representation of complex data-swizzles created from the automatically derived data-swizzling semantics generated by Misaal from ISA semantics. The prototype is described in blue with the named parameters. The parameterization immediately below the prototype corresponds to the target specific swizzle used in Figure 7(b).

Table 2. Derived swizzles AutoLLVM IR statistics across different target architectures.

Architecture	# Swizzles	Swizzle AutoLLVM IR Size	Reduction in # Swizzles
x86	243	10	24.3x
HVX	74	9	8.2x
ARM	59	8	7.4x
Total	**376**	**27**	**13.9x**

swizzle operations. The methodology of enumerating using equivalence classes from the previous subsection is extended to include these additionally derived swizzle AutoLLVM IR. Note that enumeration captures swizzle lowering rules as well which are needed to compile these high-level complex swizzles into sequences of target supported instructions.

3.4 Semantics-Based Search Space Pruning

Enumerating using equivalence classes greatly reduces the enumeration space by orders of magnitude. For example, enumerating x86 expressions to depth 2 (i.e. sequence length up to 4) with target instructions produces $3x10^{12}$ terms, while using equivalence classes results in $2.9x10^{8}$ terms (approximately 9000x reduction). Scalability issues of enumerating deeper expressions still persist. For example, enumerating AutoLLVM IR expressions derived from x86 to depth 3 results in a search space exceeding 10^{32} terms. This implies some manner of pruning is inevitable. The challenge however is how to prune such that desirable rewrite rules are not pruned out, while at the same time keeping enumeration tractable.

The ideal design would automatically derive a pruning strategy directly from the target ISA semantics. Hence Misaal proposes the use of the ISA semantics in automatically deriving pruning policies. We motivate deriving these policies by example. Consider the program corresponding to the 4-point dot product instruction `_mm256_dpbusd_epi32` in x86 shown in Figure 9.

Figure 9 states the individual Halide IR operations required in synthesizing this complex instruction. Intuitively, we know that a dot-product-like instruction would consist of extension

4-point dot product in x86

```
// _mm256_dpbusd_epi32
%0 = <32xi16> sign-extend <32xi8> %arg0
%1 = <32xi16> zero-extend <32xi8> %arg1
%2 = <32xi16> vector-mul %0, %1
%3 = <8xi16> vector-reduce-add 4 <32xi16> %2 // Horizontal vector reduction with reduction factor 4
%4 = <8xi32> sign-extend <8xi16> %3
%5 = <8xi32> vector-add %4, %arg2
```

Fig. 9. Halide IR program 4-point dot product in x86. `%arg0` is zero-extended, while `%arg1` is sign-extended. Another sign-extension is applied on the product of the two extended vectors. Finally the extended vector is reduced and added to the accumulation registers. The following program exhibits multiple vector size, bitwidths, extension variants.

```
Compute-Only Instruction Relevance: _mm_cvtepu8_epi16 for _mm256_dpbusd_epi32
%0 = <4xi16> rdsl-sext <4xi8> BVSlices(_mm256_dpbusd_epi32, /* output lane */ 0, /* %arg0 */ 0)
%1 = <4xi16> _mm_cvtepu8_epi16 <4xi8> BVSlices(_mm256_dpbusd_epi32, /* output lane */ 0, /* %arg1 */ 1)
%2 = <4xi16> rdsl-mul %0, %1
%3 = <1xi16> rdsl-vector-reduce-add 4 <4xi16> %2 // Horizontal vector reduction with reduction factor 4
%4 = <1xi32> rdsl-sext <1xi16> %3
%5 = <1xi32> rdsl-add %4, %arg2
```

Fig. 10. RDSL program containing __mm_cvtepu8_epi16 (i.e. zero-extend 8-bit elements to 16-bit) instruction which implements the semantics of _mm256_dpbusd_epi32 (i.e. 4-point dot-product) output lane 0. Note that since the reduction factor for this dot-product instruction is 4, the __mm_cvtepu8_epi16 instruction is uniformly scaled down to operate on 4 vector lanes.

instructions, multiplication, reduction and addition. Hence, when enumerating for complex dot-product instructions, for larger sequence lengths we should include these operations, and therefore other operations are reasonable to exclude when enumerating for `_mm256_dpbusd_epi32`. While this is sufficient for dot-product, this 'intuition' may not be portable to other complex instructions. Furthermore, it requires expert knowledge of the semantics of the instructions to define this pruned set of operations to enumerate.

The high-level algorithm Misaal employs is inspired by this motivation. We create distinct sets (possibly overlapping) of AutoLLVM IR operations to enumerate rather than enumerating all AutoLLVM IR operations with respect to each other. These sets are referred to as Relevance Sets. Briefly, given AutoLLVM IR Expression R, if $I \in \mathit{RelevanceSet}(R)$ then AutoLLVM IR I can be used, either as part of or fully, in an AutoLLVM IR expression E such that $E \simeq_{IR} R$. Intuitively, I represents part of the required semantics which may be necessary when synthesizing AutoLLVM IR expression R.

Creating these relevance sets is non-trivial as:

- AutoLLVM expression R may contain various intermediate vector register sizes and element bit-widths.
- AutoLLVM expressions may only produce the required semantics only if the data are swizzled in a particular layout.
- Relevance sets need to be derived using equivalence over symbolic inputs which is not entirely straightforward to evaluate with partial expressions.

Misaal yet again uses program synthesis to create these relevance sets by defining a '*Compute-Only*' synthesis problem . For this problem, we define the Relevance Domain Specific Language, (RDSL) which consists of vectorized bit-vector operations and reduction operations described in Figure 11. We then define the following equivalence over symbolic inputs as synthesizing a relevance program E such that:

$$I \in \mathit{RelevanceSet}(R) \iff \exists E \in (\mathit{RDSL} \cup \{I\}) \wedge (I \in E) \wedge (R \simeq_{IR} E)$$

RDSL does not contain any data swizzle operations, hence it does not immediately rectify all of the relevance set creation challenges described. Synthesis complexity grows exponentially with bitvector sizes. Therefore, we would like the AutoLLVM IR operation and the synthesized relevance program use minimal bitvector sizes. Misaal resolves both these limitations by leveraging the following insight. For every vector operation, simple or complex, there is a repeating pattern for groups of lanes. The group may be a single lane for simple vector operations (e.g element-wise addition) or 2-4 (e.g. horizontal subtraction, dot-product). If the synthesized program E can be restricted to this small subset of lanes, the relevance property still holds but is more efficient to evaluate. This formulation requires automatically scaling down the vector lanes of the AutoLLVM IR operation semantics to operate on the required vector lanes. We leverage this transformation

RDSL Specification

```
Expr ::=
# Operations between vector operands producing vector
      BinaryOperator Expr Expr ElementBitwidth VectorSize
# Operation on single vector operand producing vector
    | UnaryOperator Expr ElementBitwidth VectorSize
# Folding operation on single vector operand producing scalar
    | ReductionOperator Expr ElementBitwidth VectorSize
# Terminals are either slices of Operands, Intermediate
# values per lane, or constants
    | BVSlices | IntermediateValues | Const

BinaryOperator ::=  ReductionOperator | concatenate
                  | absolute-signed-difference
# Fold left these operations across vector-lanes

ReductionOperator ::=
        add | signed-sat-add | unsigned-sat-add
      | sub | signed-sat-sub | unsigned-sat-sub
      | unsigned-div | signed-div |  bitwise-or
      | bitwise-and | bitwise-xor | arithmetic-shift-right
      | logical-shift-right | shift-left | signed-max
      | unsigned-max | unsigned-min | mul
      | signed-remainder | unsigned-remainder

UnaryOperator ::=
        bitwise-not | zero-extend | sign-extend | extract
      | signed-saturation | unsigned-saturation
```

Fig. 11. Description of the Relevance Domain Specific Language (RDSL) for Compute-Only Instruction Relevance. RDSL primarily contains unary and binary arithmetic operations on scalar and vector operands. It also provides operations for reducing vectors into scalars to capture cross-lane vector ISA semantics. Finally, the terminals in RDSL denote constants, bitvector slices from the AutoLLVM IR operation's vector operands, as well as intermediate values extracted from the semantics of the AutoLLVM IR operation.

from Hydride. However, even for a single output lane, AutoLLVM IR operations may access vector data arbitrarily requiring data-swizzling.

To this end, we define a semantics-based property of AutoLLVM IR expression: *BVSlices*(R, OIdx, IIdx) as the set of bitvector slices accessed by AutoLLVM IR expression R for output lane index OIdx by input operand at operand index IIdx. Consequently, the Relevance synthesis problem directly includes the corresponding input operands bitvector slices accessed for the given output lane as terminals rather than synthesizing expensive data swizzling instructions. This enables efficiently relating (according to our definition of relevance defined previously) the semantics of two AutoLLVM IR operations independently of their respective data movements. For the example in Figure 9, by analyzing the semantics of the dot-product instruction, we can infer that for output vector lane 0:

$$BVSlices(\texttt{\_mm256\_dpbusd\_epi32}, /*output\ lane*/\ 0, /*\%arg1*/\ 1) = \%arg1[0:3].$$

Different subset of lanes may implement different computations. Thus to provide sufficient coverage across lanes, we define another semantics-based property of AutoLLVM IR expression: $UniqueLanes(R)$ = {Minimal set of output lanes indices which perform distinct bitvector operations in deriving the lane result}

For example, x86 addsub instructions perform an add on odd lanes indices and a sub on even lanes indices. $UniqueLanes$(addsub) = $\{0, 1\}$ as lane 0 and 1 are the first instances where `bvadd` and `bvsub` are performed (in increasing order of lane indices), and these operations can be replicated across the lanes of the entire vector. This provides coverage across a minimal set of output lanes. Finally, Misaal analyzes the AutoLLVM IR semantics to extract the reduction factors for a given AutoLLVM IR operation. For Figure 9, the reduction factor is 4. Hence scalar and up to 4-long vector RDSL operations are included when synthesizing the relevance program. Now that we have defined these utilities we can update the definition of materializing relevance sets according to Algorithm 1.

Figure 10 illustrates the concretized relevance program synthesized for identifying that a specific x86 zero-extension instruction is relevant to the 4-point x86 dot-product instruction. The RDSL operations used include 4-long vector and scalar RDSL operations. The x86 zero-extension vector operation is uniformly scaled to 4-long vectors accordingly. Note that in practice, evaluating this 'Compute-Only' relevance needs to occur only once for pairs of AutoLLVM IR equivalence classes.

Generalized Instruction Relevance. As the compute-only relevance problem operates on extracted bitvector slices and is required to produce a single scalar, the synthesis problem can be efficiently solved. However, for more complex operations such as the one shown in Figure 9, the synthesized program itself can be large resulting in expensive synthesis times. Misaal enables flexibility in the compute-only program to more rapidly converge on the relevance problem. This is achieved by optionally enabling the intermediate values derived in the complex instruction semantics as inputs to the compute-only program (in addition to the original extract slices). This enables the relevance program to start from some intermediate state of the complex program semantics. Additionally, the desired output of the compute-only synthesis problem can be made to be some (subsequent) intermediate value (in addition to the final output scalar). This generalization captures the same relevance property while making the synthesized program size small. The generalized algorithm for this property is expressed in Algorithm 2.

Algorithm 1 Relevance Set Computation

```
Relevance(R) ← {}
for all Inst in AutoLLVM IR do
  for OIdx in UniqueLanes(R) do
    Inputs ← {}
    for all IIdx in NumOperands(R) do
      Inputs ← Inputs ∪ BVSlices(R, OIdx, IIdx)
    OutputValue ← R[OIdx]
    if ∃E ∈ (RDSL∪{Inst}), E ≃IR OutputValue then
      Relevance(R).insert(Inst)
```

4 Implementation

We implement a prototype version of Misaal on top of Rosette [18], a solver aided programming language with constructs for synthesis and verification. Transformations on the AutoLLVM IR , Concretization Grammar Generation, and Rewrite Rule Enumeration are implemented in Python. We leverage the EggLog [20] programming language for expressing rewrite rules and applying term rewriting using equality saturation. For end-to-end compilation, we integrate Misaal into the Halide [14] compiler.

Algorithm 2 Generalized Relevance Set Computation

```
Relevance(R) ← {}
for all Inst in AutoLLVM IR do
  for all OIdx in UniqueLanes(R) do
    Inputs ← {}
    for all IIdx in NumOperands(R) do
      Inputs ← Inputs ∪ BVSlices(R, OIdx, IIdx)
    IntValues ← getIntermediateValues(R, OIdx)
    for all Idx in 0 . . . IntValues.size() do
      Inputs' ← Inputs ∪ IntValues[:Idx]
      for all FIdx in (Idx + 1) . . . IntValues.size() do
        OutputValue ← IntValues[FIdx]
        if ∃E ∈ (RDSL ∪ {Inst}), E ≃IR OutputValue then
          Relevance(R).insert(Inst)
```

5 Evaluation

We evaluate Misaal against the state-of-the-art production compiler Halide (version 13) with its back ends for x86, Hexagon and ARM [7, 14] and also against synthesis-based compilers Hydride [2, 9] and Rake [1], using 33 benchmarks from image processing and deep learning domains. We can not do an evaluation against Isaria [17] (which builds on Diospyros [19]) as it does not target the architectures Misaal compiles for, nor Pitchfork [15] as it does not support generation of data-swizzling operations. We choose Halide's production-quality back ends for x86, Hexagon and ARM as baselines because they have been developed and aggressively optimized for commercial users (e.g., YouTube, Google Photos, Android mobile devices, Adobe Photoshop, etc.) by teams of engineers at Google, Qualcomm, Adobe, and others over nearly a decade. We consider three hardware targets – x86, Hexagon and ARM. Halide programs must be tuned manually for each hardware target by optimizing their schedules; these benchmarks have been hand-tuned by us for

x86, and by Qualcomm and Adobe for ARM and Hexagon. The image processing kernels include image dilation, blurs and edge detection filters across a range of filter sizes, and others. We also include important deep learning kernels, such as matrix multiplication on tensors of low batch sizes (1, 2 and 4) which have low arithmetic density and are commonly found in large language models; and we also evaluate some fused versions of deep learning kernels that are commonly found in various neural networks (such as average/max pool + add), and in MLP blocks, in particular (matmul + bias + activation + matmul).

For our experiments, for x86, we use an Intel Xeon Silver 4216 CPU (16 cores, 2.1GHz, 22 MB L3 cache) with hyperthreading disabled; for HVX, a cycle-accurate simulator in Hexagon SDK v3.5.2 provided by Qualcomm; and for ARM, an Apple M2 CPU (3.49GHz, 16 GB memory, 16 MB L3 cache). We use AMD EPYC 7453 28-Core Processor 2.7 GHz for generating the offline components of Misaal.

5.1 EggLog Usage

Misaal uses EggLog [20] for online rewrite rule application to compile programs in Halide IR to AutoLLVM IR. Abstracted rewrite rules are generated separately per architecture. The relevant architecture-specific abstracted rewrite rules and the abstracted Halide rewrite rules are included when compiling for a particular architecture (x86, HVX or ARM). Halide IR operations are given cost 10,000 and AutoLLVM IR operations are given cost 1 (to ensure that the extracted expression is in terms of AutoLLVM IR). Bursts of 5 iterations of equality saturation are run until the extracted expression is purely in AutoLLVM IR. If the extracted expression contains abstract swizzles, then equality-saturation is re-run with only swizzle lowering rules (with abstract swizzles now having cost 10,000) to ensure the swizzles are legalized to operations in the target IR. We terminate when the extracted expression is purely in terms of AutoLLVM IR corresponding to the target ISA.

5.2 Compilation Times

One of the two fundamental advances in Misaal is in speed and scalability (the other is in more powerful support for swizzles).

We compare how long it takes to compile programs with Misaal against the state-of-the-art synthesis-based compiler, Hydride. Columns 1 and 2 of Table 4 list the compilation times with Hydride and Misaal across different benchmarks. We observe that Misaal achieves order of magnitude reduction in compilation times against Hydride with a geomean reduction of 16x for x86, 9x for HVX and 10x for ARM. Most notably, for kernels such as convolution and l2norm where data swizzling and cross-lane vector operations such as dot-product are needed to achieve performance, Hydride can compile a single application for over 5 hours, whereas Misaal completes compilation in the order of seconds resulting in a maximum reduction in compilation time of 224x, illustrating the dramatic improvement in scalability. This is expected as the underlying technology leveraged in Hydride (and most other program synthesis based tool chains) is SMT (satisfiability modulo theory) solvers whose complexity increases with the number of arithmetic operations such as multiplication and division, where as Misaal instead leverages equality saturation at compile time to perform program translation on AutoLLVM IR patterns.

Hydride does achieve comparable compilation times on a few benchmarks such as the dilate kernel and even faster compilation time on a single kernel of max pool for HVX. As Hydride prunes its synthesis grammar online according to each expression it compiles at a time, it achieves lower overall compilation time for simpler kernels like Max Pool where only the target instructions involving element-wise maximum and minimum are needed. On the other hand, Misaal does not perform any pruning during compilation and includes all the front end and target rewrite

rules during compilation, which a fixed overhead which results in Misaal taking slightly longer compilation times for simpler benchmarks.

It's worth noting that, due to inherent scalability limitation with program synthesis, Hydride compiles large Halide IR programs by decomposing an expression into small windows and synthesizing each window separately. On the other hand, Misaal reasons about the entire expression at once, resulting in optimizations over larger sequences of operations, potentially yielding better code and also simplifying the compiler design. Despite this more powerful approach, Misaal shows dramatically faster compilation time than Hydride (and Rake).

Table 3. Comparing compilation times of Misaal against Rake, a synthesis-based compiler for HVX (in seconds). Rake fails to compile 27 out of 33 benchmarks. Speedups in compilation times achieved by Misaal shown in parentheses.

Benchmark	Rake	Misaal
sobel3x3	5988	300 **(20x)**
dilate3x3	3398	100 **(34x)**
avgpool	3648	329 **(11x)**
maxpool	63	413 **(0.2x)**
add	1879	517 **(3.6x)**
fully_connected	3669	193 **(19x)**
Geomean	1784	272.2 **(6.8x)**

We also compare Misaal against Rake [1, 11], a synthesis-based compiler for HVX and ARM. Rake fails to compile 27 out of 33 benchmarks due to crashes in the Rosette interpreter for HVX and all of the benchmarks for ARM despite our best efforts. We show the comparisons between the compilation times of Rake and Misaal in Table 3. Misaal compiles 6.84x faster than Rake for all the benchmarks for HVX we could evaluate, except max pool which Misaal slightly slower because of the same reason why Hydride is slower at compiling it.

5.3 Peak Memory Utilization

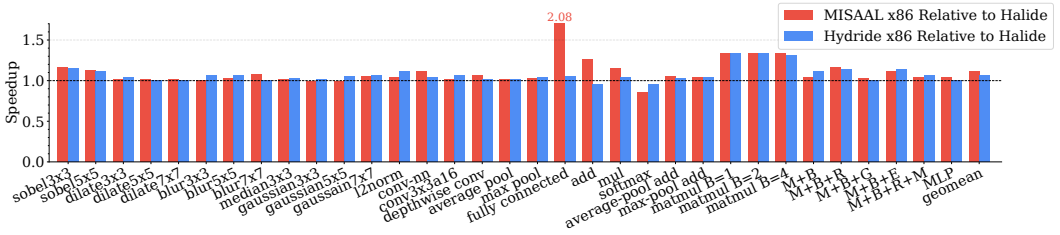
Excessive memory consumption is a major obstacle to scalability for previous synthesis-based compilers, like Hydride. The last two groups of columns in Table 4 show the peak physical memory usage for Hydride and Misaal respectively. Misaal requires orders of magnitude less memory to compile than Hydride, which uses online synthesis. Hydride achieves retargetability at the cost of up to 15 Gigabytes (Gb) of memory for benchmarks such as depthwise convolution on x86. In contrast, Misaal requires few megabytes to compile. For x86 and ARM, Misaal achieves a geomean memory reduction of 18x and 26x compared to Hydride respectively, where as for HVX Misaal achieves a relatively modest geomean memory reduction of 3x. This can be attributed to the characteristics of the ISA themselves; x86 and ARM provide mostly element-wise SIMD vector operations with select cross-lane operation variants where all operations are supported across various element bitwidths and vector registers whereas HVX is a DSP architecture with highly specialized, complex instructions. As a result, compiled HVX programs require more complex data swizzling operations in addition to vector slicing and concatenations. Despite this, Misaal reaches a maximum memory usage of 2.5 Gb which remains feasible for edge devices whereas Hydride exceeds 15 Gb, limiting its practical use to server-class machines. Furthermore, Misaal is able to scale compilation across much larger program sequences, unlike Hydride which must split programs into more manageable synthesis queries. We additionally report the peak memory usage for the offline enumeration and synthesis workflow for Misaal. The offline flow parallelizes the enumeration across 64 processes. Across the target architectures ISAs, we observe a sustained peak memory usage of 8 Gb demonstrating that Misaal's concretization-based grammar can be solved efficiently with regards to memory usage enabling a high degree of parallelism. In contrast, a single synthesis task for Hydride may require gigabytes of memory.

5.4 Performance Evaluation

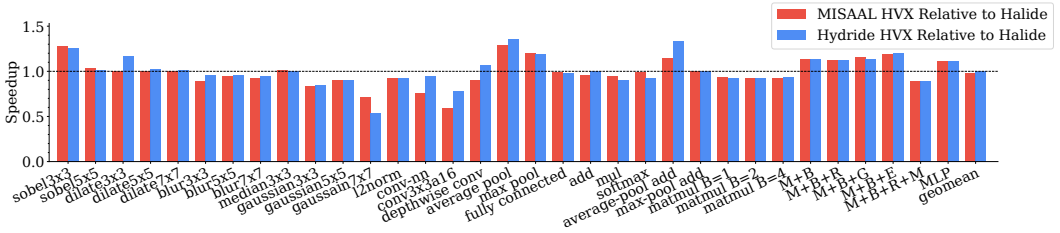
We evaluate the performance of code generated by Misaal and Hydride against the common baseline of Halide's manually-written and optimized target-specific back ends. Figure 12a, 12b, 12c show the relative performance of the three compilers for x86, HVX and ARM respectively.

Table 4. Compilation times and Memory Usage with Hydride and Misaal on x86, HVX, and ARM. Fused kernels are abbreviated as M: Matmul, B: Bias Add, R: ReLU, G: GeLU and E: Element-wise Add

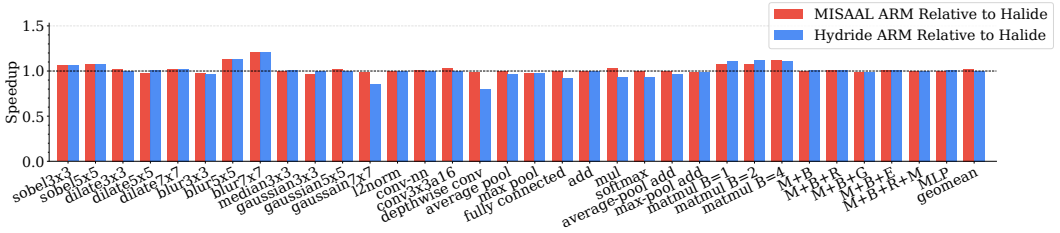
	Hydride Compilation Times (s)			Misaal Compilation Times (s) (Compilation Speedup)			Hydride Peak Memory (Mb)			Misaal Peak Memory (Mb) (Memory Improvement)		
Benchmark	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*	*x86*	*HVX*	*ARM*
sobel 3x3	8300	4740	590	81 **(103x)**	301 **(16x)**	86 **(7x)**	2.5K	1.6K	8.5K	146 **(17x)**	1.2K **(1x)**	82 **(103x)**
sobel 5x5	10511	9171	332	416 **(25x)**	629 **(15x)**	81 **(4x)**	2.3K	1.6K	1.9K	190 **(12x)**	2.3K **(1x)**	82 **(23x)**
dilate 3x3	90	80	251	45 **(2x)**	100 **(1x)**	36 **(7x)**	863	1.6K	913	124 **(7x)**	677 **(2x)**	78 **(12x)**
dilate 5x5	90	120	86	33 **(3x)**	111 **(1x)**	25 **(3x)**	849	1.9K	814	113 **(8x)**	630 **(3x)**	79 **(10x)**
dilate 7x7	45	120	41	36 **(1x)**	110 **(1x)**	27 **(2x)**	856	1.9K	676	119 **(7x)**	663 **(3x)**	80 **(8x)**
box blur 3x3	724	450	82	118 **(6x)**	210 **(2x)**	26 **(3x)**	667	1.1K	857	101 **(7x)**	767 **(1x)**	79 **(11x)**
box blur 5x5	943	226	8832	159 **(6x)**	268 **(1x)**	99 **(89x)**	868	1.1K	12.5K	101 **(9x)**	917 **(1x)**	96 **(130x)**
box blur 7x7	1155	6900	8274	204 **(6x)**	453 **(15x)**	224 **(37x)**	876	7K	12.3K	102 **(9x)**	1.7K **(4x)**	96 **(129x)**
median 3x3	429	960	7247	93 **(5x)**	37 **(26x)**	47 **(155x)**	2K	1.6K	7.5K	184 **(11x)**	1.4K **(1x)**	80 **(93x)**
gaussian 3x3	2600	11760	133	54 **(48x)**	225 **(52x)**	42 **(3x)**	3K	5.5K	5.4K	101 **(30x)**	904 **(6x)**	81 **(66x)**
gaussian 5x5	5326	10800	776	57 **(93x)**	558 **(19x)**	52 **(15x)**	2.4K	1.9K	6.5K	102 **(23x)**	2.5K **(1x)**	81 **(80x)**
gaussian 7x7	12041	39480	1574	92 **(130x)**	500 **(79x)**	92 **(17x)**	5K	2.6K	5.3K	110 **(46x)**	588 **(5x)**	96 **(55x)**
l2norm	6000	20600	7068	75.9 **(79x)**	224 **(92x)**	102.12 **(69x)**	14K	5K	8.8K	121 **(116x)**	693 **(7x)**	98 **(89x)**
conv_nn	22000	54000	18270	144 **(153x)**	454 **(119x)**	110 **(167x)**	4.1K	1.4K	6.7K	142 **(29x)**	1.2K **(1x)**	112 **(60x)**
conv3x3a16	23940	25200	304	131 **(183x)**	281 **(90x)**	105 **(3x)**	2.4K	2K	7K	114 **(21x)**	970 **(2x)**	99 **(71x)**
depthwise_conv	11000	60274	1672	59 **(186x)**	620 **(97x)**	55 **(30x)**	15K	1.7K	5.7K	118 **(128x)**	1.2K **(1x)**	98 **(58x)**
avgpool	640	4487	203	50 **(13x)**	329 **(14x)**	37 **(6x)**	1.6K	1K	877	116 **(14x)**	885 **(1x)**	82 **(11x)**
maxpool	100	68	58	49 **(2x)**	413 **(0.2x)**	30 **(2x)**	838	1.3K	613	110 **(8x)**	909 **(1x)**	82 **(8x)**
fullyconnected	8563	36000	727	57 **(150x)**	193 **(187x)**	53 **(14x)**	2.4K	3.4K	4.2K	135 **(18x)**	668 **(5x)**	111 **(38x)**
add	2261	9480	259	98 **(23x)**	517 **(18x)**	38 **(7x)**	2.4K	1.9K	8.2K	118 **(20x)**	1.3K **(2x)**	98 **(83x)**
mul	6000	45000	2210	86 **(70x)**	245 **(184x)**	67 **(33x)**	2.2K	5.8K	7.6K	123 **(18x)**	860 **(7x)**	98 **(78x)**
softmax	4925	14000	3107	138 **(36x)**	549 **(26x)**	186 **(17x)**	5K	1.4K	2K	116 **(43x)**	1.5K **(1x)**	142 **(14x)**
matmul[b = 1]	125	500	97	48 **(3x)**	170 **(3x)**	30 **(3x)**	989	5.2K	628	102 **(10x)**	634 **(8x)**	81 **(8x)**
matmul[b = 2]	125	500	97	48 **(3x)**	170 **(3x)**	29 **(3x)**	989	5202	628	102 **(10x)**	634 **(8x)**	80 **(8x)**
matmul[b = 4]	125	500	97	48 **(3x)**	170 **(3x)**	29 **(3x)**	989	5202	628	102 **(10x)**	634 **(8x)**	80 **(8x)**
avgpool+add	3500	5000	15277	49 **(71x)**	364 **(14x)**	37 **(412x)**	1.6K	1.4K	4.5K	117 **(14x)**	964 **(1x)**	82 **(54x)**
maxpool+add	40	1000	83	52 **(1x)**	393 **(3x)**	31 **(3x)**	1.1K	545	1.4K	114 **(9x)**	909 **(1x)**	81 **(18x)**
M + B	198	500	142	49 **(4x)**	210 **(2x)**	31 **(5x)**	974	3.8K	679	106 **(9x)**	634 **(6x)**	81 **(8x)**
M + B + R	228	1000	188	62 **(4x)**	214 **(5x)**	36 **(5x)**	1.1K	5.4K	732	108 **(10x)**	634 **(9x)**	88 **(8x)**
M + B + G	2980	450	815	83 **(36x)**	235 **(2x)**	78 **(10x)**	9.8K	3.7K	4.6K	106 **(93x)**	634 **(6x)**	81 **(56x)**
M + B + E	227	1000	174	65 **(4x)**	223 **(5x)**	36 **(5x)**	1.4K	4K	765	106 **(13x)**	634 **(6x)**	88 **(9x)**
M + B + R + M	6274	350	868	87 **(72x)**	223 **(2x)**	85 **(10x)**	13.3K	5.2K	634	104 **(128x)**	638 **(8x)**	81 **(8x)**
M + B + G + M	3100	300	848	85 **(37x)**	223 **(1x)**	88 **(10x)**	8.4K	3.9K	635	112 **(76x)**	638 **(6x)**	81 **(8x)**
Geomean (33 benchmarks)	1196	2222	556	75 **(16x)**	258 **(9x)**	54 **(10x)**	2.1K	2.4K	2.3K	116 **(18x)**	891 **(3x)**	88 **(26x)**



(a) Performance on x86



(b) Performance on HVX



(c) Performance on ARM

Fig. 12. Performance of code generated by Misaal and Hydride relative to Halide's different target-specific back ends. Bars show speedups compared to Halide, so higher is better. Fused kernels are abbreviated as M: Matmul, B: Bias Add, R: ReLU, G: GeLU and E: Element-wise Add.

Performance on x86. Misaal achieves a geomean 10% performance improvement on x86 compared Halide's x86 back end, with a maximum speedup of 2.08x, a slight improvement from Hydride which outperforms the same back end by a geomean of 8% with a maximum speedup of 1.35x for matrix multiplication benchmarks. In these benchmarks, Misaal effectively leverages efficient x86 dot-product instructions, delivering competitive performance compared to the optimized Halide x86 backend. In the batched matrix multiplication kernels, Misaal identifies that a dot product instruction can be repurposed as a widening multiply-add. Halide (as of version 13) does not know this trick, and generates a longer sequence of instructions. For the fully connected benchmark, both generate dot-product instructions, but Misaal achieves superior performance. This is because Misaal identifies opportunities to generate saturation instructions for 64-bit values, which Halide cannot due to limited support. Additionally, Misaal reduces the number of shuffle vector operations (data-swizzling) compared to the Halide generated code. Similar optimizations occur in the Add and Sobel benchmarks, where Misaal utilizes target-specific x86 instructions for better data movement. Although Halide supports some x86-specific optimizations, its capabilities are limited by the large size of the x86 ISA.

Performance on Hexagon. Compared to the x86 and ARM Halide backends, the Halide HVX back end is the most aggressively optimized manually. It performs many optimizations, including aggressively moving data swizzling instructions across multiple basic blocks in order to eliminate redundant interleave and deinterleave swizzle patterns. However, these optimizations require multiple years of development and fine-tuning, and can be tedious and error prone to develop and maintain as the ISA grows.

Misaal is able to synthesize strong enough rewrite rules offline to achieve competitive performance against this highly optimized baseline, delivering a geomean relative performance of 0.98 against the Halide HVX back end, which is also what Hydride delivers. For kernels such as sobel and average pooling, Misaal and Hydride generate appropriate scalar-vector widening multiplication operations whereas Halide's HVX back end's pattern matching rules fail to generate these operations, and end up generating more expensive vector-vector widening multiplication operations using broadcasts instead. Conversely, for conv3x3a16, gaussian7x7 and a few other gaussian variants, both Hydride and Misaal incur similar, significant slowdowns compared with Halide. For gaussian7x7, Misaal and Hydride generate efficient cross-lane reduction operations with swizzle operations. However, Hydride is unable to optimize swizzle layouts across the entire program due to limited scalability of program synthesis; but Misaal is able to automatically propagate swizzles across operations such as interleaving and de-interleaving across instructions, resulting in a 20% speedup over Hydride. However, Halide is able to perform the same optimization more aggressively so as to emit a 4-point dot-product, achieving significantly higher relative performance. Because HVX's arithmetic operations do not support all vector register sizes, compilers must split large vector registers into smaller vector sizes and concatenate them as needed. For the kernels such as conv_nn, Hydride and Halide generate code with vector addition operations operating on the larger vector sizes; however, Misaal generates vector addition operations on the smaller vector registers since it does not concatenate them, thereby leading to slowdowns.

Performance on ARM. Misaal achieves a geomean speedup of 1.02x against Halide on ARM, with a maximum speedup of 1.21x over Halide's production ARM backend. Meanwhile, Hydride gets a geomean speedup of 3% and a maximum speedup of 1.21x. Misaal achieves similar performance to Hydride in most cases. Additionally, Misaal can recognize and rewrite dot product patterns more flexibly compared to Hydride. For fully connected benchmark, Hydride fails to synthesize dot product instructions since it requires synthesis over a larger sequence of input operations than Hydride supports; on the other hand, Misaal is able to synthesize the dot product operations by reasoning about longer sequences of input operations. In Sobel kernels, Misaal and Hydride reduce the number of addition instructions by leveraging the `umlal` (unsigned multiply and accumulate) instruction to combine addition and multiplication. In contrast, Halide generates `ushll` for constant multiplication, missing the opportunity to combine it with addition. Speedups observed over Hydride for depthwise convolution and gaussian7x7 are serendipitous: Hydride happens to generate sequences of operations that LLVM's ARM back end is unable to optimize, thereby leading to slowdowns against Misaal and Halide.

5.5 Rewrite Rules Abstraction

Table 5 shows the numbers of rewrite rules identified across the various target architectures in Misaal, and the number in Halide IR (the same Halide rewrite rules are used across the target architectures). Across the target architectures, we observe the largest number of concrete rewrites for x86, and is due to the relative ISA size of x86 compared to the other targets. The largest number of concrete patterns are found in Halide IR, because it supports scalar and vector operations across different vector register sizes separately (from 8-bit up to 4096-bit vectors). Misaal automatically abstracts these rewrites into parameterized, retargetable rewrite rules for both simple and complex

Table 5. Number of rewrites derived from each target and their corresponding number of abstracted rewrites.

Target	# Concrete Rewrites	# Abstracted Rewrite	Reduction in # Rewrite
x86	5,806	329	17.6x
HVX	4,586	412	11.1x
ARM	4,085	190	21.5x
Halide	27,364	1,284	21.3x
Total	**41,841**	**2,215**	**18.9x**

Table 6. Compiler Construction times for automatically generated components of Misaal. Enumeration is done up to a depth of 4 with up to 5 terminals. NA : Not applicable.

Category	x86	HVX	ARM	Halide
Compute-Only Relevance	16 hours	14 hours	21 hours	10 hours
Swizzle-Generation	37 mins	9 mins	8 mins	NA
AutoLLVM IR Enumeration	1 month+	1 month+	1 month+	1 month+
Extracting Concrete Rewrite Rules	5.9 hours	6.3 hours	3.5 hours	52 hours
Rewrite Rules Abstraction	10 mins	20 mins	6 mins	56 mins

cases. Across the target architectures, x86 and ARM observe the largest reduction of 17.6x and 21.5x in the number of rewrites rules which is to be expected; x86 and ARM's ISA are mostly generic and provide various versions of the same operations with different element-bitwidths and vector register sizes enabling Misaal to abstract more concrete rewrite rules. In contrast, HVX exhibits a relatively modest reduction of 11.1x due to it being a DSP architecture with specialized instructions with fewer variations of element-types and vector register sizes.

5.6 Compiler Construction

We provide the compiler construction times across different hardware targets and Halide in Table 6 for reference. Enumeration is performed up-to a depth of 4 (with up-to 5 terminals) which results in over a month long enumeration time per target. Further engineering optimizations can be taken to improve this time. For example, pruning commutative, associative and distributive variants of equivalent expressions during enumeration can significantly improve enumeration efficiency. Halide IR requires significantly higher time for extracting concrete rewrite rules and rewrite rules abstraction, which is directly a product of the large variation of 'concretizations' for Halide IR operations with respect to AutoLLVM IR. While we exhaustively extract all possible concretizations for rewrite rule abstraction, a sufficient cut-off in the number of concretizations which are to be extracted can be provided as a parameter to reduce the times for these two phases.

Halide uses greedy term-rewriting systems in its backends. Naturally, one might ask whether the synthesized rewrite rules can be integrated. This presents challenges, particularly in determining the rule application order. These systems often require careful tuning of both the rules and their application order to ensure good performance and fast compilation times. However, insights from e-graph-based rewriting can guide faster rule application in other systems. For example, compiling benchmarks with e-graphs can reveal the most used rules during equality saturation. Abstracting these rules reduces their number compared to concrete rules, especially when factors like vectorization, element types, or bitvector operation variant vary. Although equality saturation makes it difficult to extract rule order directly, manual analysis might uncover heuristics for applying these rules in other systems.

6 Related Work

Autovectorization. Porcupine [6] uses hand-implemented semantics for a small set of SIMD instructions to synthesize code for homomorphic encryption. It requires users to provide reference and sketch implementation with holes for their input programs. Diospyros [19] is a vectorizer for Tensilica DSP; it uses a set of manually-defined rewrite rules to vectorize scalar code using equality saturation at compile time. It defines a large space of rewrites that leads to an explosion in the size of e-graphs and causes the equality saturation at compile time to take several minutes, if not hours, even for small tensor kernels. Isaria [17] improves on Diospyros by automatically generating rewrite rules using an ISA specification in an offline stage and using heuristics to contain the explosion in the size of e-graphs which enables it to reduce compilation times to a few minutes. Neither Diospyros nor Isaria supports generation of complex compute and data swizzle instructions which contribute enormously to the size and complexity of the synthesis process.

Instruction Selection. Rake [1] uses program synthesis to lower small sequences of input code operations to target vector instructions at compile-time. Because it requires manually-implemented semantics of target instructions, it only supports small subsets of target ISAs (a few hundreds of instructions). Rake uses specialized heuristics to make generation of target cross-lane compute instructions, and data interleave and deinterleave patterns tractable; however, synthesis still takes several minutes and, in some cases, several hours. Pitchfork [15] uses Rake in an offline stage to synthesize rewrite rules for short sequences of enumerated input IR operations with target-specific fixed-point compute vector instructions. It then uses a custom lightweight term rewriter to apply the rewrite rules on input code at compile time – this enables it to compile programs in a few seconds. However, Pitchfork does not support generation of complex data swizzles. The rewrite rules from Pitchfork have been partially merged into Halide's main branch, though this occurred after the Halide version we compare to in this work (version 13), so we do not directly compare to Pitchfork. Enumo [12] is a DSL that can guide rewrite rule inference using equality saturation-driven enumeration. Enumo can be used to infer rewrite rules for instruction lowering, however it does not scale for larger ISA sizes and depths over the space we require. Hydride [9] uses program synthesis to generate code in a target-independent IR to target x86, ARM and Hexagon. Hydride uses several heuristics to make synthesis of complex compute and swizzle instructions feasible; however, it still takes several minutes and hours, in some cases, to generate code.

7 Conclusion

We have presented Misaal, a retargetable compiler for Halide based on offline synthesis and online (compile-time) rewriting for multiple target architecture families: x86, HVX and ARM. Most Misaal components are generated fully automatically from vendor-provided pseudocode specifications of the target ISA semantics and an existing formal semantics of the Halide front-end IR. Misaal addresses three major challenges faced by previous compilers based on program synthesis: scalable enumeration for synthesis of rewrite rules; optimized code for complex cross-lane compute and data movement operations, without hurting scalability; and greatly reducing the number of rewrite rules in order to make compile-time rewriting fast and efficient. Moreover, Misaal delivers competitive and sometimes even better performance than the highly tuned, production compiler for Halide on all three architectures, except for a few cases on HVX. Overall, we believe Misaal takes the development of synthesis-based compilers forward to a major new milestone, in particular, showing that such compilers can be fast and scalable enough for deployment in real-world compiler systems. Our results further show that mostly-automatically generated compilers can be competitive with hand-crafted and heavily tuned manually implemented ones, even when performance is at a premium.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful feedback on this paper. This work was supported by funding from Amazon Research Awards programs, Qualcomm, Intel, the University of Illinois Urbana-Champaign, and PRISM and ACE, two of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Maaz Bin Safeer Ahmad, Alexander J Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 2022. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1004–1016.

[2] Akash Kothari. [n. d.]. Hydride. https://github.com/akothen/Hydride.

[3] Arm. 2024. Neon. https://developer.arm.com/Architectures/Neon.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.

[5] Lucian Codrescu. 2015. Architecture of the Hexagon™ 680 DSP for mobile imaging and computer vision. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–26.

[6] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. 2021. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 375–389.

[7] Halide. [n. d.]. Halide. https://github.com/halide/Halide.

[8] Intel. 2019. Intel Deep Learning Boost. https://www.intel.com/content/dam/www/public/us/en/documents/product-overviews/dl-boost-product-overview.pdf.

[9] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 514–529.

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

[11] Maaz Ahmad, Hongpu Ray Gong, Andrew Adams. [n. d.]. Rake. https://github.com/uwplse/rake/tree/hvx-artifact.

[12] 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. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 258 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622834

[13] Qualcomm. 2020. Exploring the AI capabilities of the Qualcomm Snapdragon 888 Mobile Platform [video]. https://www.qualcomm.com/news/onq/2020/12/02/exploring-ai-capabilities-qualcomm-snapdragon-888-mobile-platform.

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[15] Alexander J Root, Maaz Bin Safeer Ahmad, Dillon Sharlet, Andrew Adams, Shoaib Kamil, and Jonathan Ragan-Kelley. 2023. Fast Instruction Selection for Fast Digital Signal Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 125–137.

[16] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[17] 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, Volume 1*. 19–34.

[18] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[19] 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*. 874–886.

[20] 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. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (June 2023), 25 pages. https://doi.org/10.1145/3591239

Received 2024-11-15; accepted 2025-03-06