Fast Instruction Selection for Fast Digital Signal Processing

Alexander J. Root
Stanford University
Stanford, CA, USA
ajroot@stanford.edu

Maaz Bin Safeer Ahmad
Adobe
Seattle, WA, USA
mahmad@adobe.com

Dillon Sharlet
Independent Researcher
Irvine, CA, USA
dsharlet@gmail.com

Andrew Adams
Adobe
San Francisco, CA, USA
anadams@adobe.com

Shoaib Kamil
Adobe
New York City, NY, USA
kamil@adobe.com

Jonathan Ragan-Kelley
Massachusetts Institute of Technology
Cambridge, MA, USA
jrk@mit.edu

ABSTRACT

Modern vector processors support a wide variety of instructions for fixed-point digital signal processing. These instructions support a proliferation of rounding, saturating, and type conversion modes, and are often fused combinations of more primitive operations. While these are common idioms in fixed-point signal processing, it is difficult to use these operations in portable code. It is challenging for programmers to write down portable integer arithmetic in a C-like language that corresponds exactly to one of these instructions, and even more challenging for compilers to recognize when these instructions can be used. Our system, Pitchfork, defines a portable fixed-point intermediate representation, FPIR, that captures common idioms in fixed-point code. FPIR can be used directly by programmers experienced with fixed-point, or Pitchfork can automatically lift from integer operations into FPIR using a term-rewriting system (TRS) composed of verified manual and automatically-synthesized rules. Pitchfork then lowers from FPIR into target-specific fixed-point instructions using a set of target-specific TRSs. We show that this approach improves runtime performance of portably-written fixed-point signal processing code in Halide, across a range of benchmarks, by geomean 1.31× on x86 with AVX2, 1.82× on ARM Neon, and 2.44× on Hexagon HVX compared to a standard LLVM-based compiler flow, while maintaining or improving existing compile times.

ACM Reference Format:
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 *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623278.3624768

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0394-2/23/03.
https://doi.org/10.1145/3623278.3624768

1 INTRODUCTION

Fixed-point computation is ubiquitous in high-performance digital signal processing (DSP) workloads, such as production camera systems [1, 21], machine learning kernels [15, 46], and recently in numerical simulations [22]. It offers a variety of advantages over floating point, including more efficient hardware and a uniform distribution of representable values.

Fixed-point computation is fundamentally different to the primitive integer arithmetic offered by most language front-ends. Most fixed-point operations increase precision, so the programmer must decide whether to drop the extra bits (potentially with rounding), promote to a wider type, use saturation, or simply wrap. All choices have their place, and expressing them with portable integer arithmetic is difficult and error-prone.

Dating back to at least Intel's MMX [36], processors have supported coarse-grained fixed-point instructions that implement these idioms. Modern ISAs such as ARM Neon [29], x86 AVX2 [23], and Hexagon HVX [14] provide a rich set of fixed-point instructions that support various rounding, saturating, and type-conversion modes, as well as fusion of multiple operations into a single instruction. These are implemented as vector instructions, as the applications are data-parallel.

Compilers that use pattern matching struggle to exploit these coarse-grained instructions due to their complex semantics: using primitive integer arithmetic, simple fixed-point idioms explode into large compositions of operations. State-of-the-art compilers such as LLVM [26] are unable to reliably pattern-match these large sequences of primitive integer operations back to the coarse-grained fixed-point instructions available in modern CPUs and DSPs. As a result, these systems leave significant performance on the table: failing to optimally map to fixed-point instructions increases critical path instruction count. Even worse, primitive integer implementations of fixed-point idioms often require high-bit-width intermediate values, which halves SIMD throughput, and can even require expensive emulation for types wider than what the hardware supports (see §5.1).

A core problem is the significant *impedance mismatch* between fixed-point and primitive integer arithmetic. Programmers writing portable fixed-point code must use primitive integer arithmetic to express their intent, which the compiler must then attempt to match back to fixed-point computation in order to properly target the fixed-point instructions that modern ISAs offer. Additionally,

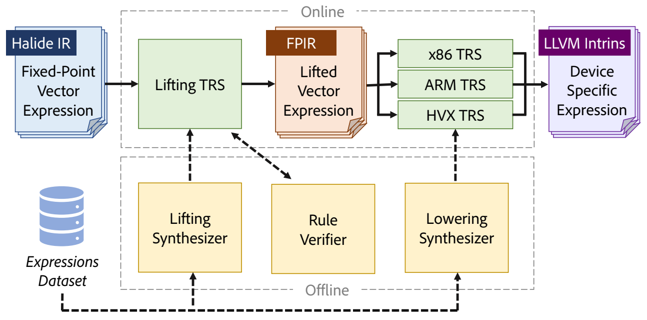

Figure 1: PITCHFORK System Architecture. At compile time, PITCHFORK performs instruction selection via a two-phased lift-then-lower algorithm. Offline, PITCHFORK leverages formal verification to establish the correctness of hand-written rules and program synthesis to infer missing translation rules.

there is a combinatorial explosion of effort required by compiler engineers to redundantly develop numerous complex patterns for backends that offer substantially-similar fixed-point instructions.

The limitations of existing rule-based systems motivate synthesis-based approaches to instruction selection, where an automated system performs more complex semantic reasoning than a rule-based system typically can. Rake [4] is one such system that uses program synthesis to target modern fixed-point instruction sets. While synthesis-based approaches are able to effectively bridge the gap between source code and hardware instructions, they increase compile times by multiple orders of magnitude, and are thus unsuitable for most real-world use-cases.

The inability of rule-based compilers to effectively exploit fixed-point instruction sets combined with prohibitively long compile times of synthesis-based solution forces developers to write performance-sensitive code directly using low-level hardware instructions. We see this in machine learning libraries such as XNNPACK [18] and production image processing systems such as `libjpeg-turbo` [28] and Adobe Photoshop [3]. Writing assembly allows programmers to directly express fixed-point computation, but produces non-portable code that requires significant and ongoing developer effort for each new hardware generation supported.

We aim to solve each of these limitations with our system, PITCHFORK, an instruction selector for digital signal processing. PITCHFORK's design is centered around a fixed-point intermediate representation, FPIR. Unlike existing rule-based compilers that attempt to directly translate primitive integer arithmetic into fixed-point hardware instructions, PITCHFORK uses a two-stage procedure where the computation is first lifted to FPIR before being lowered to hardware instructions. This design removes a combinatorial factor of rules from compilers, as each backend can share the lifting translation phase. The lowering systems from FPIR to target ISAs are also easier to develop: each backend is responsible only for compiling fixed-point computation to a fixed-point ISA. Both the target-agnostic lifting and the target-specific lowering systems are implemented as simple rule-based term-rewriting systems.

Unlike synthesis-based instruction selectors that use online synthesis to translate expressions into instructions, we take inspiration from prior work [10, 27, 38] and use *offline* program synthesis to strengthen PITCHFORK's lifting and lowering systems. By learning rewrite rules from real-world expressions using expensive offline program synthesis, PITCHFORK is able to perform powerful semantic reasoning cheaply at compile time.

Lastly, PITCHFORK caters to domain experts by allowing them to implement their algorithms directly using FPIR's fixed-point instructions, bypassing PITCHFORK's lifting phase. This enables experts to write portable high-performance code using the familiar idioms of fixed-point computation.

In summary, our contributions are:

- A domain-specific intermediate representation, FPIR, for fixed-point signal processing that unifies classes of instructions across multiple different ISAs.
- An instruction-selection system, PITCHFORK, that lifts integer code into FPIR before lowering it into target-specific instructions using fast term-rewriting systems partially synthesized from a corpus of real-world fixed-point expressions.
- A prototype of PITCHFORK that achieves state-of-the-art performance on three popular fixed-point instruction sets: ARM Neon, Hexagon HVX, and x86 AVX2.

We evaluate our approach using LLVM and Rake as baselines. Our results show that PITCHFORK achieves a geometric mean speed up over LLVM alone of 1.31× on x86 AVX2 (max 3.40×), 1.82× on ARM Neon (max 8.33×), and 2.44× on Hexagon HVX (max 5.76×), while coming within 2% of the performance of Rake [4] on ARM Neon and 13% on Hexagon HVX. Additionally, we show that PITCHFORK's compile times are comparable to or out-perform LLVM on all benchmarks, and are at least three orders of magnitude faster than Rake.

2 OVERVIEW

This section provides an overview of PITCHFORK's instruction selection algorithm, illustrated in Figure 1, using an example to illustrate the process.

The prototype of PITCHFORK is implemented with the Halide [42] compiler, a popular domain-specific language for high-performance image and array processing. Halide allows users to separately specify the *algorithm*, i.e. what they would like to compute, from the *schedule*, i.e. how they would like to compute it. Halide compiles to LLVM IR [26] for most of its CPU and DSP backends.

PITCHFORK sits between Halide and LLVM, intercepting the translation of fixed-point vector expressions from Halide IR into LLVM IR. PITCHFORK takes these fixed-point vector expressions as input, which contain primitive integer arithmetic operations such as addition, multiplication, and bit-shifts, as well as any user-provided FPIR instructions. As output, PITCHFORK generates optimized target-specific implementations of the input expressions for three of Halide's supported DSP backends: x86 AVX2 (now referred to as x86), 64-bit ARM Neon (ARM) and Hexagon HVX (HVX). The output expressions use LLVM's target-specific intrinsics, thereby bypassing LLVM's instruction selection process.

2.1 Fixed-Point Considerations

The fixed-point representation is a digital encoding for approximating real numbers. In contrast to the more ubiquitous floating-point representation, fixed-point arithmetic can be performed using simpler and more efficient hardware. While floating-point offers dynamic scaling, fixed-point does not, so a fixed-point programmer has to explicitly reason about the magnitude of each number in a calculation, often needing to widen the types of operands to avoid overflow, or explicitly handle overflow with saturating operations.

For example, to add two values, a floating-point programmer writes `a + b`, and generally does not worry about overflow in DSP code. In contrast, the fixed-point programmer must handle a statically-limited range of values, and so needs to be more deliberate about handling overflow when the operands are large. Suppose the operands are stored as unsigned 8-bit fixed-point values. There are multiple reasonable options for addition: the user could widen the operands to represent the exact 9-bit sum, requiring a 16-bit result type for most architectures via `u16(x) + u16(y)`, or could use saturating arithmetic to handle overflow and stay in 8 bits, equivalent to `u8(min(u16(x) + u16(y), 255))`. Alternatively, if the programmer knows bounds on the inputs that mean the sum cannot overflow, they can use regular addition without widening or saturation. Each of these options have their use cases, and are often supported by DSP hardware: widening addition can be performed via ARM's `uaddl` instruction or HVX's `vaddubh`, and saturating addition is supported via ARM's `uqadd`, HVX's `vadd:sat`, and x86's `vpaddusb`.

Fixed-point considerations become more complicated when performing computation that produces results which cannot be exactly represented, such as division, square roots, and many other useful arithmetic operations. Rounding must occur for these operations, and different rounding modes are useful for different use cases. For example, consider taking the average of two unsigned 8-bit fixed-point values. The sum of the values might overflow, even though the average fits in 8 bits. Moreover, should the average of 4 and 3 be 3 or 4? Using round-down averaging[1], `u8((u16(x) + u16(y)) » 1)`, produces 3, while round-up averaging[2], `u8((u16(x) + u16(y) + 1) » 1)`, produces 4. Both of these options (and others) have perfectly valid places in various fixed-point algorithms, and need to be reasoned about when writing fixed-point code.

In simple integer arithmetic, the examples above produce remarkably verbose code that is difficult to write and subject to a number of compiler optimizations that might remove the original intent of the programmer. Compiler passes can obfuscate the code before reaching instruction selection, hindering the ability of instruction selection to properly target fixed-point ISAs.

2.2 Motivating Example

The Sobel filter [40] is a popular algorithm for approximating spatial gradients of images.

Figure 2a shows a fixed-point Sobel filter implemented in Halide. For brevity, we omit the Halide schedules governing the optimizations performed for each backend. All three schedules inline the entire computation and then vectorize it using vector-widths of 16,

[1]Implemented by ARM's uhadd and HVX's vavg.

[2]Implemented by ARM's urhadd, x86's vpavgb, and HVX's vavg:rnd.

```
in_u16(x, y) = u16(input_u8(x, y));

x_kernel(x, y) = in_u16(x-1, y) + 2 * in_u16(x, y) +
                 in_u16(x+1, y);
sobel_x(x, y) = absd(x_kernel(x, y-1), x_kernel(x, y+1));

y_kernel(x, y) = in_u16(x, y-1) + 2 * in_u16(x, y) +
                 in_u16(x, y+1);
sobel_y(x, y) = absd(y_kernel(x-1, y), y_kernel(x+1, y));

output(x, y) = u8(min(sobel_x(x, y) + sobel_y(x, y), 255));
```

(a) The Sobel filter algorithm expressed in Halide. The schedules (omitted for brevity) inline and vectorize the computation for all three backends.

```
// Syntax guide:
//  • u8(...) and u16(...) are vector casts to
//    uint8 and uint16, respectively
//  • x(c) broadcasts scalar c to a vector
//  • variables are suffixed with their types,
//    i.e. a_u8 is a vector of uint8s
//  • min, +, and * are vector integer operations
//  • absd (absolute difference) is an FPIR instruction
u8(
  min(
    absd(
      u16(a_u8) + u16(b_u8) * x(2) + u16(c_u8),
      u16(d_u8) + u16(e_u8) * x(2) + u16(f_u8)) +
    absd(
      u16(g_u8) + u16(h_u8) * x(2) + u16(i_u8),
      u16(j_u8) + u16(k_u8) * x(2) + u16(l_u8)),
    x(255)))
```

(b) The input to Pitchfork: A target-agnostic vector expression generated by lowering the Sobel filter to Halide IR.

```
saturating_cast<u8>(
  absd(
    widening_add(a_u8, c_u8) + widening_shl(b_u8, x(1)),
    widening_add(d_u8, f_u8) + widening_shl(e_u8, x(1))) +
  absd(
    widening_add(g_u8, i_u8) + widening_shl(h_u8, x(1)),
    widening_add(j_u8, l_u8) + widening_shl(k_u8, x(1))))
```

(c) The vector expression in (b) lifted into our fixed-point intermediate representation (FPIR).

Figure 2: The Sobel filter benchmark. Pitchfork performs instruction selection by lifting the lowered Halide IR expression into FPIR, before lowering the expression to taget-specific LLVM intrinsics. For (b) and (c), vector lengths are abstracted away as they vary across different backends.

32 and 128 for ARM, x86 and HVX respectively. Figure 2b shows the resulting Halide IR generated by lowering the Sobel filter, which is handed over to Pitchfork for instruction selection. Note that the lowered expression consists of primitive vector operations, with the exception of `absd`. The `absd` instruction implements the absolute-difference operation and is part of our proposed IR for fixed point arithmetic, discussed further in §2.3.

Compared to using LLVM [26] alone, Pitchfork recognizes *eight* missed optimizations in the Sobel filter across Pitchfork's three backends. With these optimizations, Pitchfork achieves a runtime speedup of 1.7× on ARM, 1.6× on x86, and 1.8× on HVX over LLVM, while compiling the benchmark faster than LLVM for

	Expression	ISA	Pitchfork Codegen	LLVM Codegen

(a)	`u16(a_u8) +`
	`u16(b_u8) * 2 +`
	`u16(c_u8)`
		ARM	`uaddl   v3.8h, v0.8b, v2.8b`
			`uaddl2  v4.8h, v0.16b, v2.16b`
			`umlal   v3.8h, v1.8b, 2`
			`umlal2  v4.8h, v1.16b, 2`
				`uaddl   v3.8h, v0.8b, v2.8b`
				`uaddl2  v4.8h, v0.16b, v2.16b`
				`ushll   v5.8h, v1.8b, 1`
				`ushll2  v6.8h, v1.16b, 1`
				`add     v7.8h, v3.8h, v5.8h`
				`add     v8.8h, v4.8h, v6.8h`

		HVX	`vzxt     v3.h, v0.ub`
			`vmpa.acc v3.h, v1.ub, v2.ub, 2, 1`
				`vzxt     v3.h, v0.ub`
				`vmpa     v4.h, v1.ub, v2.ub, 2, 1`
				`vadd     v5.h, v4.h, v3.h`

(b)	`// absd(x_u16, y_u16)`
	`select(`
	`a < b,`
	`b - a,`
	`a - b`
	`)`
		x86	`vpsubusw  %ymm2, %ymm0, %ymm1`
			`vpsubusw  %ymm3, %ymm1, %ymm0`
			`vpor      %ymm4, %ymm2, %ymm3`
				`vpminuw   %ymm2, %ymm0, %ymm1`
				`vpcmpeqw  %ymm3, %ymm2, %ymm1`
				`vpsubw    %ymm4, %ymm0, %ymm1`
				`vpsubw    %ymm5, %ymm1, %ymm0`
				`vpblendvb %ymm6, %ymm3, %ymm4, %ymm5`

		ARM	`uabd    v0.8h, v0.8h, v1.8h`	`cmhi     v2.8h, v0.8h, v1.8h`
				`sub      v3.8h, v0.8h, v1.8h`
				`sub      v4.8h, v1.8h, v0.8h`
				`bit      v4.16b, v3.16b, v2.16b`

		HVX	`vabsdiff v2.uh v0.uh v1.uh`	`vsub     v2.h v0.h v1.h`
				`vsub     v3.h v1.h v0.h`
				`vcmp.gt  q0 v1.uh v0.uh`
				`vmux     v4 q0 v3 v2`

(c)	`u8(min(z_u16, 255))`	x86	`vpackuswb %ymm2, %ymm0, %ymm1`	`vpminuw   %ymm2, %ymm0, 255`
				`vpminuw   %ymm3, %ymm1, 255`
				`vpackuswb %ymm4, %ymm2, %ymm3`

		ARM	`uqxtn   v2.8b, v0.8h`
			`uqxtn2  v2.16b, v1.8h`
				`umin    v2.8h, v0.8h, 255`
				`umin    v3.8h, v1.8h, 255`
				`uzp1    v4.16b, v2.1b, v3.16b`

		HVX	`vsat    v2.ub, v0.h, v1.h`	`vmin     v2.uh, v0.uh, 255`
				`vmin     v3.uh, v1.uh, 255`
				`vshuffeb v4.ub, v2.uh, v3.uh`

Figure 3: An illustration of the key differences between Pitchfork and LLVM's instruction selection on the Sobel filter. Pitchfork discovers eight optimizations resulting in runtime speed-ups of: $1.74\times$ **on ARM,** $1.58\times$ **on x86, and** $1.77\times$ **on HVX over LLVM, while compiling faster than LLVM alone. All assembly is written in Intel syntax: `instr dst, [operands]`.**

all three targets. Furthermore, when compared against Rake [4], a compiler that uses program synthesis to guide instruction selection, Pitchfork delivers matching runtime performance on the Sobel filter on ARM and only a 9% slow-down on HVX, while requiring only 1/100,000th of Rake's compilation time[3].

Figure 3 illustrates instances where Pitchfork achieves better instruction selection over LLVM alone on the Sobel filter. Expressions (a) and (b) both showcase examples where LLVM's instruction selection rules fail to match the input code patterns. In (a) LLVM converts the multiplication into a bit-shift, which in turn causes the multiply-add pattern to not be triggered. Similarly, LLVM does not have a rewrite rule that maps the implementation of absolute-difference in (b) to the appropriate hardware instructions. Expression (c) is a more interesting optimization, as it relies on bounds reasoning for both x86 and HVX backends. The saturating narrow instructions (`vpackuswb` and `vsat`) generated by Pitchfork are only correct if the `uint16` input can be represented as an `int16`. This requires predicated rules covering this edge-case use of these instructions.

The inability of a mature state-of-the-art compiler, LLVM, to match even basic fixed-point patterns illustrates the challenge of building and maintaining large rule-based instruction selectors for fixed-point computation. We believe that our two-phase system makes that task tractable.

[3]Rake currently does not support x86 as a backend.

2.3 Instruction Selection in Pitchfork

Pitchfork's contributions are centered on its language for fixed-point computation, FPIR. FPIR is designed to interoperate with primitive integer operations, as instruction sets provide both integer and fixed-point operations, and programmers write code in a mixture of modes as well.

At compilation, Pitchfork lifts primitive integer arithmetic inputs into FPIR via a target-agnostic lifting phase. We give an example of the lifted representation in Figure 2c. Note that the `u8` cast and the `min` are lifted to a single `saturating_cast<u8>` FPIR instruction (saturating cast to `uint8`). The kernel is similarly lifted to instructions using `widening_add` and `widening_shl`.

Lifting input expressions to FPIR presents three significant advantages. First, it acts as a normalization step, simplifying the subsequent lowering of expressions to device-specific instructions. Lowering from FPIR to instructions is easier, as the set of possible input patterns that must be handled is smaller due to the higher-level nature of FPIR. Second, operations like saturating cast and absolute-difference are almost ubiquitous across fixed-point backends. Lifting allows us to maintain a shared corpus of rewrite rules relevant across different backends, avoiding redundancy that would be necessary in a direct-translation system. Lastly, as FPIR is target-agnostic, domain experts who think in terms of these fixed-point idioms can express their computation using FPIR instructions in portable code. The Sobel example provides one such case, where the input code contains the absd instruction.

Pitchfork's lowering systems maps FPIR or fused combinations of FPIR to target instructions. These systems are made simple by the normalization of expressions in the lifting phase. Figure 3 provides the instructions chosen for the three key components of the Sobel algorithm, highlighting Pitchfork's wins over LLVM alone.

Pitchfork's lifting and lowering systems are implemented as term-rewriting systems (TRSs), described in detail in §3. This design enables offline synthesis to augment the hand-written TRSs, as in prior work [27, 38].

2.4 Synthesizing Rewrite Rules

While Pitchfork's two-stage compilation simplifies designing compilers for fixed-point computation, manually designing lifting and lowering rules remains challenging. Search-based instruction selectors [4, 12] and super-optimizers [44] are able to explore a richer space of optimizations, which can result in runtime performance improvements at the cost of compile-time performance. Pitchfork follows prior work on synthesizing rules offline [27, 38], which preserves the comprehensiveness of search-based approaches while avoiding their compile-time cost.

Pitchfork was initially designed with a manually-written set of rewrite rules. We augmented this set with additional rules learned from a corpus of sample expressions. Given this corpus, Pitchfork first searches for optimal translations to FPIR from primitive integer arithmetic and from FPIR to target instructions on specific expressions, and then performs rule generalization to produce symbolic rewrites from the input-output pairs. The automatically generated rules are added to Pitchfork's term-rewriting system to be used during online compilation.

The synthesized rules significantly improve runtime performance. When Sobel is compiled with Pitchfork's synthesized rules enabled, we see a 1.23× performance boost on ARM and a 1.10× performance boost on HVX. We describe Pitchfork's rule synthesis process in §4 and evaluate the impact of automatically synthesizing rules on all benchmarks in §5.3.

Verifying Hand-Written Rules. A desirable side-effect of synthesizing new rewrite rules is that it provides us with the machinery to verify existing hand-written rules. As synthesis requires the semantics of Halide IR and FPIR, we were able to use these semantics to formally verify Pitchfork's entire lifting TRS. This exercise unearthed a handful of subtle bugs that had escaped detection through testing and code-reviews. Examples include missing predicates over the range of constant values for which a rule is valid and a case where the documented semantics of an operation did not match its actual behavior. Verifying the semantics of the lowering TRS remains as future work.

3 AN IR FOR FIXED-POINT ARITHMETIC

Computation in fixed-point signal processing is rife with idioms that express a variety of common saturating, rounding, and widening fixed-point behavior. Instruction sets designed for fixed-point signal processing provide fused instructions that directly accelerate these idioms. FPIR attempts to capture these idioms as target-agnostic instructions that are both expressive enough to capture user intent and simple to map to complex hardware instructions.

Table 1 lists the full set of fixed-point instructions that comprise FPIR, as well as their semantics. Each FPIR instruction is defined as a composition of primitive integer arithmetic operations and implements a pattern typically found in DSP hardware. For instance, the FPIR instruction rounding_halving_add can be mapped to a single hardware instruction on all three backends currently supported by Pitchfork (vpavgb on x86, urhadd on ARM, and vavg:rnd on HVX), and various other ISAs as well (vavgub on PowerPC, vaaddu on RISC-V, etc.).

Although FPIR is designed specifically for fixed-point operations, in practice, fixed-point and integer code often mix. Therefore, to enable interoperability between fixed-point and integer Halide code, FPIR is implemented as an extension to Halide's existing integer IR. Note that while we implement FPIR within the Halide compiler, we believe that this set of intrinsics could (and should) be used for *any* compiler that aims to target fixed-point instructions, such as LLVM. See §7 for a discussion of LLVM's existing fixed-point operators.

3.1 Design

In this section, we highlight three key design objectives that shape FPIR, and provide concrete examples to justify our language design.

3.1.1 Portable. Programmers should be able to program in a close-to-assembly but portable language that offers reliable performance. As a result, FPIR unifies instruction classes across architectures by providing portable instructions that implement common fixed-point idioms. For example, FPIR exposes non-rounding averaging via halving_add, which directly maps to ARM's uhadd and shadd instructions, HVX's vavg* instructions, and RISC-V's vaadd and vaaddu instructions. If a backend lacks support for a particular FPIR instruction, Pitchfork is still able to offer portable performance for these instructions by providing the most efficient emulation available. For example, x86, WebAssembly, and PowerPC do not support halving_add, and therefore share Pitchfork's fast non-widening implementation of this instruction [17].

A portable fixed-point IR also decreases the cognitive load on the compiler writers and reduces the size of the compiler. For example, if there are k ways for a programmer to write rounding_halving_add and n backends that implement rounding_halving_add, without rounding_halving_add in the IR itself, a compiler requires $k * n$ rules to map from primitive integer IR to backend instructions. Instead, FPIR requires only $k + n + 1$ rules to be written: k patterns that map integer arithmetic to rounding_halving_add, n mappings from rounding_halving_add to the target instructions, and one efficient lowering for targets that don't support this operation.

Note that this paper evaluates the FPIR x86, ARM, and HVX backends; subsequently, developers have adopted FPIR for all of Halide's CPU backends, including WebAssembly, PowerPC, XTensa, and RISC-V. See Section §8 for details.

3.1.2 Curated. FPIR is a useful and minimal set of fixed-point operations. We aim to be judicious about which instructions to include in FPIR to prevent users from accidentally writing buggy

FPIR Instruction	Semantics	
`extending_add(x, y)`	`x * widen(y)`	/* x must have double the bits of y */
`extending_sub(x, y)`	`x - widen(y)`	/* x must have double the bits of y */
`extending_mul(x, y)`	`x * widen(y)`	/* x must have double the bits of y */
`widening_add(x, y)`	`widen(x) * widen(y)`	
`widening_sub(x, y)`	`widen(x) - widen(y)`	/* x and y are cast to the wider signed type */
`widening_mul(x, y)`	`widen(x) * widen(y)`	/* x and y may have different signedness */
`widening_shl(x, y)`	`widen(x) « widen(y)`	
`widening_shr(x, y)`	`widen(x) » widen(y)`	
`abs(x)`	`select(x > 0, x, -x)`	/* The output is always unsigned */
`absd(x, y)`	`select(x > y, x - y, y - x)`	/* The output is always unsigned */
`saturating_cast<t>(x)`	`cast<t>(min(max(x, t.min()), t.max()))`	
`saturating_narrow(x)`	`saturating_cast<type(x).narrow()>(x)`	/* This is just shorthand that is useful in later semantic definitions */
`saturating_add(x, y)`	`saturating_narrow(widening_add(x, y))`	
`saturating_sub(x, y)`	`saturating_cast<type(x)>(widening_sub(x, y))`	
`halving_add(x, y)`	`narrow(widening_add(x, y) / 2)`	
`halving_sub(x, y)`	`narrow((widen(x) - widen(y)) / 2)`	/* Widening preserves signedness */
`rounding_halving_add(x, y)`	`narrow((widening_add(x, y) + 1) / 2)`	
`rounding_shl(x, y)`	`saturating_narrow(widening_add(x, select(y < 0, 1 » (y + 1), 0)) « y)`	
`rounding_shr(x, y)`	`saturating_narrow(widening_add(x, select(y < 0, 1 « (y + 1), 0)) » y)`	
`mul_shr(x, y, z)`	`saturating_narrow(widening_mul(x, y) » widen(z))`	
`rounding_mul_shr(x, y, z)`	`saturating_narrow(rounding_shr(widening_mul(x, y), widen(z)))`	

Table 1: Semantics of FPIR instructions, designed to unify common fixed-point idioms across multiple ISAs.

code. This means not including superfluous instructions such as `saturating_halving_add` (equivalent to `halving_add`, as this operation cannot overflow), which would be generated if the set of instructions were a Cartesian product of modes and operations. Likewise, FPIR purposefully excludes instructions with dubious semantics that are likely to be used in a bug-prone way, such as `rounding_halving_sub`, which has surprising overflow behavior at the signed extremes. A clean and minimal set also reduces the cognitive load on both programmers and compiler writers, by reducing the number of instructions either programmer needs to consider. Further, this property enables synthesis by reducing the *branching factor* of the search space of possible programs.

3.1.3 Expressive. The language should allow programmers, compiler writers, and synthesizers to write short and readable, yet powerful, expressions. For example, to write a non-rounding averaging instruction in a C-like language, the user would need to write `narrow((widen(x) + widen(y)) » 1)`, while a user of FPIR can simply write `halving_add(x, y)`. The widening is never performed when the instruction selection system targets ARM (`uhadd` and `shadd`) or HVX (`vavg*`)[4]. Likewise, compiler writers (and synthesizers) should be able to write short patterns in FPIR that map to complex instructions. This property also improves the tractability of synthesis by reducing the necessary *depth* of the search tree of programs.

3.2 Lifting to FPIR

Pitchfork lifts Halide IR expressions to FPIR using a rule-based term-rewriting system. The lifting TRS traverses the expression tree bottom up, greedily applying a set of ordered rules to fuse multiple operations (both Halide IR operations and FPIR instructions) into a cheaper expression that incorporates at least one FPIR instruction. The lifting TRS repeats this process until the expression converges to a fixed point. Convergence is guaranteed by requiring that each rule strictly reduces a target-agnostic cost, described below. Rules that could match on the same input are also ordered using this cost, with the lower-cost output preferred. The lifting TRS was implemented using approximately 50 hand-written rules, and augmented with a further 25 synthesized rules, which we discuss later in §4. Figure 4 shows a handful of lifting rules required to lift the expression in Figure 2b.

Pitchfork's cost model is a lexicographic order, which first sums the bit-widths of the inputs to each instruction. This favors fewer, narrower-bit-width instructions. Ties are resolved using an ordering over operations designed to capture their average cost on real targets. For example, `rounding_halving_add` for `u8` is slightly lower cost than `halving_add`, because while ARM and HVX support both, x86 only supports the former.

[4]As noted in §3.1.1, `halving_add` does not need widened intermediates on any backend [17].

3.3 Lowering to Target ISA

The lowering of expressions from FPIR to the target ISA is implemented using several target-specific term-rewriting systems. As Halide is built on top of LLVM, the output of the lowering process is not device-specific assembly code but rather LLVM IR. With Pitchfork disabled, Halide lowers to target-agnostic LLVM IR and lets LLVM select the correct device-specific instructions. In order to punch a hole through this and perform its own instruction selection, Pitchfork instead emits LLVM's device-specific intrinsics, a set of functions that correspond directly to instructions available in the target ISAs. For some instructions LLVM does not provide an intrinsic, and instead relies on device-specific pattern-matching. For these, we emit precisely the pattern that LLVM expects for that instruction, using LLVM's unit tests as a guide. If all else fails, we emit inline assembly, though that was not necessary for the instructions used in this work.

Lowering rules are designed using target-specific cost models provided by processor documentation [23, 29, 41] to maximize throughput. We discuss the five main classes of lowering translation rules below:

```
// Syntax guide:-
// •  Rule format: before -> after [predicate]
// •  Wildcards are suffixed with their types,
//    i.e. x_u8 is a vector of uint8s.
// •  c0 is a wildcard that matches only to constants
u16(x_u8) * c0
  -> widening_shl(x_u8, log2(c0)) [is_pow2(c0)]

u16(x_u8) + y_u16
  -> extending_add(y_u16, x_u8)

extending_add(extending_add(x_u16, y_u8), z_u8)
 -> widening_add(y_u8, z_u8) + x_u16

u8(min(x_u16, 255))
 -> saturating_cast<u8>(x_u16)
```

Figure 4: Examples of rules triggered in lifting Sobel's Figure 2b to Figure 2c. Note that many of these rules are polymorphic in nature, but we present them with concrete types for illustrative purposes.

Direct mappings. These are one-to-one translations from FPIR to concrete target instructions. In the Sobel example, the widening_-add is directly compiled to a ARM's uaddl instruction, and absd is directly compiled to the corresponding unsigned absolute-difference instructions on ARM (uabd) and HVX (vabsdiff).

Fused mappings. These rules map combinations of FPIR instructions to a single equivalent target instruction. For example, an addition with a widening_shl can use ARM's widening multiply-accumulate instruction. This pattern is triggered when compiling the Sobel example on ARM.

```
x_u16 + widening_shl(y_u8, c0)
  -> umlal(x_u16, y_u8, (1 << c0))
```

Compound instructions. Not all backends support all instructions in FPIR. For these cases, we provide efficient lowering from the FPIR instruction to multiple target instructions. We give one such example below, which implements a fast unsigned absd implementation on x86 (due to [17]), and is triggered when compiling the Sobel example on x86.

```
absd(x_u16, y_u16)
 -> vpor(vpsubusw(x_u16, y_u16),
         vpsubusw(y_u16, x_u16))
```

Predicated Rules. Pitchfork supports predicated lowering rules that compile to specific instructions if facts can be proven at compile time. We offer two such rules below:

```
HVX:
sat_cast<u8>(x_u16) -> vsat(x_u16)
 [upper_bounded(x_u16, INT16_MAX)]
x86:
sat_cast<u8>(x_u16) -> vpackuswb(x_u16)
 [upper_bounded(x_u16, INT16_MAX)]
```

While some predicate checks are simple, the most powerful that Pitchfork offers are bounds-related queries, as shown above. For these queries, we use interval analysis, and for performance reasons, a simple expression cache for bounds queries. This is only a small modification to the existing bounds inference engine in Halide [42], as we only needed to add support for bounds queries on FPIR instructions. These rules allow Pitchfork to perform better compilation of the saturating cast performed in the Sobel example on x86 and HVX.

Specific Constants. Some rules apply to usages of FPIR instructions with specific constants. For example, a particular shift value for a mul_shr instruction maps to the vpmulhw instruction on x86:

```
mul_shift_right(x_i16, y_i16, 16)
 -> vpmulhw(x_i16, y_i16)
```

4 SYNTHESIZING TERM-REWRITING RULES

In this section, we discuss how Pitchfork synthesizes term-rewriting rules for both lifting and lowering TRSs. Pitchfork's offline modules are implemented in Rosette [47], a solver-aided programming language that offers rich libraries for constructing and solving both synthesis and verification queries. We use Z3 [16] as the underlying solver.

The space of possible lifting and lowering rules is large. We aim to only include rules that may trigger on real code - this is why we do not use randomly-generated expressions, and instead choose a data-driven approach.

4.1 Synthesizing Lifting Rewrite Pairs

We follow a similar algorithm to those used in prior works for synthesizing term-rewriting systems [27, 38]. Given a set of fixed-point expressions in Halide IR, Pitchfork enumerates all sub-expressions of size up to 10 IR nodes to generate the left-hand-sides of possible lifting rules. We keep left-hand-sides small in order to generate less-specific rules that might be more broadly applicable, and to keep synthesis manageable. To generate the right-hand-side of a lifting rule, Pitchfork uses syntax-guided program synthesis (SyGuS) [6] to lift the sub-expressions into FPIR, guided by the target-agnostic cost model described in §3.2. This required implementing an interpreter for Halide IR and FPIR in Rosette. Left-hand-sides that successfully lift to cheaper right-hand-sides are added to the rule-set that Pitchfork will try to generalize. Generated lifting rules are ordered using the cost model.

We give a small example of a synthesized lifting rule from the *add* benchmark below, but note that this is before generalization. We discuss generalization further in §4.3.

```
i16(x_u8) << 6
 -> reinterpret(widening_shl(x_u8, u8(6)))
```

The original hand-written lifting system failed to include a rule that lifts a signed widening shift left on an unsigned variable into a reinterpret of an unsigned widening_shl instruction. While the hand-written lifting system contained a version of this rule where the widen cast is u8 -> u16, the signed case was missed. Rules such as this are difficult for human compiler engineers to enumerate.

4.2 Generating Lowering Rewrite Pairs

Pitchfork's system for generating lowering rules is dependent on an instruction selection oracle. We use Rake [4], which has backends for both ARM and HVX - we do not currently automatically generate lowering rules for x86. This is due to the small number of interesting fused operations on x86. Future work could generate rules using synthesis-based x86 instruction selectors such as STOKE [45].

Using the same set of fixed-point expressions in §4.1, Pitchfork generates the left-hand-sides of lowering rules by using the lifting system to lift a full example expression into FPIR and enumerating small sub-expressions of the lifted expression, again up to a limit of

10 IR nodes. Optimal right-hand-sides for these rules are provided by our oracle – Rake. Lowering rules are ordered using Rake's target-specific cost model.

We give an example of a synthesized lowering rule from our motivating example below, which corresponds to **(a)** in Figure 3. Again, note that this rule is before generalization.

```
x_u16 + widening_shl(y_u8, 1)
 -> umlal x_u16 y_u8 2
```

The hand-written lowering system for ARM produced a widening shift left `ushll` followed by a vector `add`, and this is also what LLVM alone produces. However, it is faster to use ARM's fused widening multiply-add instruction, `umlal` for this particular computation.

4.3 Generalizing Rewrite Pairs to Rules

We generalize lifting and lowering rules using a set of techniques described below. Note that these are only generalization *attempts* – Pitchfork verifies the attempt at generalization to confirm that the generalized rule is still correct.

(1) Replace all instances of a constant with a symbolic constant.
(2) Require one constant to be the two to the power of another.
(3) Require safe-reinterpretations for a variable, i.e. when a `uint16` can be safely reinterpreted as a `int16`.
(4) Safe truncation or saturation. Multiple processors implement only saturating or only truncating versions of narrowing operations. In some cases, a saturating variant of an instruction can be used for a truncating computation if Pitchfork can prove that the saturation won't trigger, and vice versa.

For bounds on symbolic constants, we perform a simple binary search on the space of possible integer values for that constant's type.

This small set of techniques is enough to generalize most rules. We give the generalized versions of the lifting and lowering examples used in §4.1 and §4.2 below.

```
i16(x_u8x) << c0
 -> reinterpret(widening_shl(x_u8x, u8(c0)))
 if (0 < c0 < 256)
```

This generalized lifting rule requires a predicate that constrains the range of possible constant-values that the rule is correct for, found via the approach described above.

```
x_u16x + widening_shl(y_u8x, c0)
 -> umlal x_u16x y_u8x (1 << c0)
```

This generalized lowering rule requires no predicate, but creates a relationship between symbolic constants in the left-hand-side and the right-hand-side of the rule.

5 EVALUATION

We evaluate Pitchfork on the set of Rake [4] benchmarks with fixed-point computation[5]. These benchmarks span quantized machine learning, computational photography, image processing, and computer vision workloads, and are all written as portable Halide code.

We do not alter the existing Halide schedules found in the implementation, and only change the instruction selection algorithm. All benchmarks were run with a single thread. Benchmarks for ARM Neon were compiled and run on an Apple M1 Pro (3.2 GHz, 8 cores) with 16 GB of RAM. Benchmarks for x86 target AVX2 instructions and were compiled and run on an Intel Xeon Platinum 8275CL CPU @ 3.0 GHz running Ubuntu 20.04.4 with 96 cores and 192 GB of RAM. Hexagon HVX runtime numbers were computed using Qualcomm's cycle-accurate Hexagon Simulator v8.3.07 without cache-modeling enabled (to simulate a compute-limited system). We use mainline LLVM as of October 12, 2022[6]. For the comparison against Rake, we use the most recent version made publicly-available by the authors[7].

Due to the limited nature of open-source fixed-point benchmarks, a full train-test-validate benchmark partition is not possible. Instead, we perform cross-validation via a leave-one-out approach. The performance numbers reported are produced via compiling each benchmark *without* the synthesized rules generated from *that benchmark's* expressions.

5.1 Runtime Evaluation

Compared to using LLVM alone for instruction selection, Pitchfork achieves geometric mean speedups of 1.31× on x86 AVX2 (maximum 3.40×), 1.82× on ARM Neon (maximum 8.33×), and 2.44× on Hexagon HVX (maximum 5.76×), as shown in Figure 5. For most benchmarks, Pitchfork matches the performance of Rake, while in a few cases Pitchfork produces code that exceeds the performance obtained by Rake.

On ARM, Pitchfork obtains performance on average 2% and at most 9% slower than Rake. For HVX, Pitchfork is on average 13% and at most 50% (on *matmul*) slower. On the HVX platform, the impact of Rake's optimization of data swizzling operations has a large impact; Pitchfork does not optimize swizzles, as Pitchfork's design is focused on optimizing fixed-point computation patterns. We discuss this limitation further in §6.

There are three benchmarks (*depthwise_conv*, *matmul*, and *mul*) that use 64-bit types when expressed using primitive integer operations, which HVX does not support and LLVM fails to compile. For the LLVM evaluation of these three benchmarks, we use Pitchfork's lowering of `rounding_mul_shr` that stays within 32-bit arithmetic, intended to be used only for x86, as ARM and HVX contain instructions for this particular usage of `rounding_mul_shr`[8]. This implementation is only used to ensure that LLVM can compile all benchmarks for comparison purposes, but note that this means that Pitchfork can compile programs that LLVM alone cannot.

We now discuss a some of the optimizations that Pitchfork achieves through its combination of hand-written and synthesized rewrite rules.

5.1.1 Fused multiply-add instructions. These fused optimizations fall into two sub-categories: direct fused multiply-adds (e.g. ARM's `umlal`) and dot-product instructions (e.g. ARM's `udot`, HVX's `vrmpy`, and x86's `vpmaddwd`). Many benchmarks make heavy use of patterns that can take advantage of each of these instructions, including *add*, *mul*, *matmul*, *sobel3x3*, and the *gaussian* benchmarks. For many of these benchmarks, LLVM either fails entirely to generate the fused instructions or fails to properly utilize dot-product instructions.

[5] 16 of Rake's 21 benchmarks perform fixed-point computation.

[6] Commit c7dd7f20b099d98c1941240e5c8586290dd53182.

[7] https://github.com/uwplse/rake/tree/hvx-arm-x86-artifact

[8] This is used for signed 32-bit fixed-point multiplication within $[-1, 1]$.

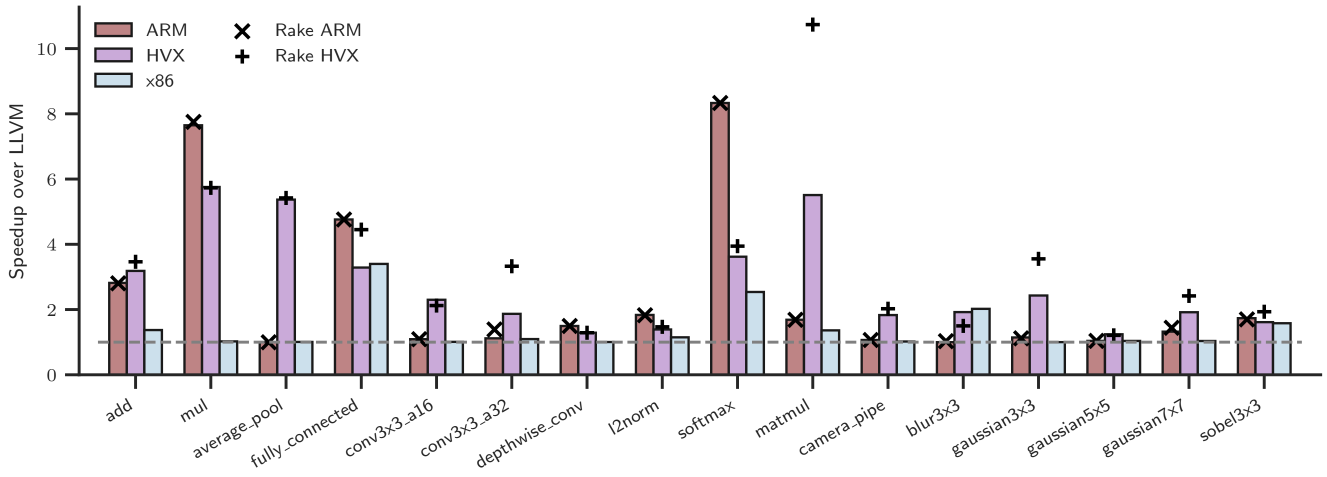

Figure 5: Runtime speedups over LLVM instruction selection. Pitchfork achieves geometric mean speedups of 1.31× for x86, 1.82× for ARM, and 2.44× for Hexagon HVX, with a maximum speedup of 8.33×. For most benchmarks, Pitchfork matches the performance of Rake at a fraction of the compile time. (Rake does not currently support x86 compilation.)

5.1.2 Saturating and rounding instructions. All of the quantized ML benchmarks perform saturating and rounding operations, which LLVM generally fails to produce[9]. Image processing benchmarks like *camera_pipe* and *gaussian3x3* feature rounding averaging and rounding shifts respectively, for which Pitchfork produces the correct target instructions but LLVM does not.

5.1.3 Direct mappings. As discussed in §3.3, many FPIR instructions have direct mappings to concrete target instructions. This includes `absd` (used in *camera_pipe* and *sobel3x3*), `rounding_mul_shr` (used in *mul* and *depthwise_conv*, and lifted to in *matmul*), and many others. These are mapped to their corresponding target instructions, but LLVM generally fails to match their semantic definitions entirely.

5.1.4 Compound instructions. While ARM and HVX implement most of the instructions defined in FPIR, x86 implements far fewer. Thus, Pitchfork lowers to optimized implementations of many operations on x86, as discussed in §3.3. These have particular impact on several benchmarks, including *sobel3x3* and *matmul*. Efficient implementations of these instructions are particularly relevant for extending Pitchfork to other backends that do not have the large number of fixed-point instructions that ARM and HVX support, such as WebAssembly.

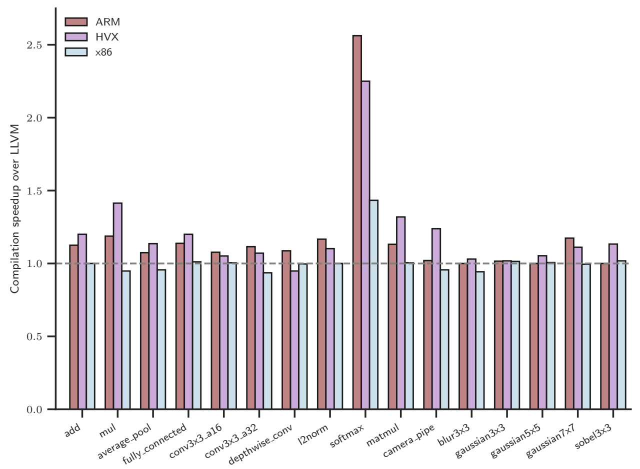

Figure 6: Compile-time speed-up over LLVM alone for the three backends. Pitchfork improved runtime performance without compile-time cost – compile times are in fact slightly improved in most cases by passing less code to the downstream LLVM compiler.

5.2 Compilation Speed

Figure 6 shows compilation time speedup over LLVM alone for our three backends. Despite existing on top of LLVM, Pitchfork compiles most benchmarks in less time, due to generating less LLVM IR. This reduces time spent in LLVM optimization passes, which more than makes up for the time spent in Pitchfork's lifting and lowering. The largest speedup is compiling *softmax*, which requires a large amount of LLVM IR to express using primitive integer operations, but is more compact in FPIR. Pitchfork and LLVM are both orders of magnitude faster than compiling via Rake.

5.3 Ablation: Impact of Synthesized Rules

In order to empirically measure improvements due to generating rewrite rules for Pitchfork, we perform two ablation studies (for ARM and HVX) where we compare the performance of Pitchfork to a version of Pitchfork with only hand-written rules. Figure 7 shows the speedups due to learned rules, showing that synthesized rules increase performance by up to 4.99× over handwritten rules alone.

[9]With the exception of cases where the program explicitly uses `saturating_add`, which is lowered to `llvm.(u|s)add.sat`.

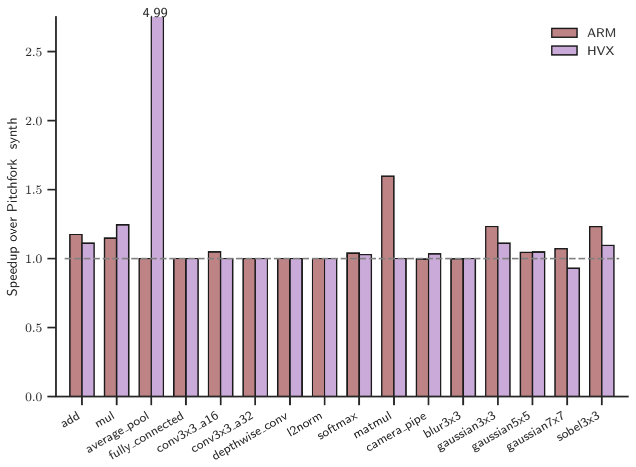

Figure 7: Speed-up over hand-written rules. Adding synthesized rules results in geomean speed-ups of 1.09× **on ARM and** 1.14× **on HVX, with up to a** 4.99× **speedup for *average_pool* on HVX.**

5.3.1 Synthesized rules for ARM. Many of the synthesized lowering rules on ARM correspond to missed patterns for fused multiply-accumulate instructions. These correspond to usages of `umlal` on *add*, *mul*, and *sobel3x3*, and usages of `udot` for *matmul* and *gaussian7x7*, where convolution can use ARM's dot-product instruction. Pitchfork also generates some shift-right-narrow patterns that use bounds-inference-derived predicates for more efficient instruction selection on *gaussian3x3*.

5.3.2 Synthesized rules for HVX. Pitchfork generates all of the missed optimization patterns that Rake finds for HVX [4]. This corresponds to missing fused multiply-accumulate instructions for *add*, *average_pool*, and *sobel3x3*, as well as fused saturate-shift instructions on *camera_pipe* and *gaussian3x3*. Note that on one benchmark, *gaussian7x7*, Pitchfork performs worse when including synthesized rules. This is due to a synthesized lifting rule that re-orders a series of widening additions and multiplications in a way that interacts poorly with HVX's challenging swizzling patterns; the re-ordered computation requires an additional shuffle operation.

6 LIMITATIONS AND FUTURE WORK

An important design goal of Pitchfork is efficient compilation – the majority of Pitchfork's limitations are direct results of prioritizing this goal.

Local Optimizations. Term-rewriting systems cannot perform global optimizations, which can further improve performance. This can be seen when comparing the performance of Pitchfork and Rake on the *gaussian7x7* benchmark for ARM Neon. Rake performs a global computation reordering that Pitchfork does not. Pitchfork would require impractically-large rules to perform the same re-ordering.

Data Swizzling. As noted in §5.3.2, Hexagon performance is highly dependent on swizzling patterns. Future work could co-optimize data computation patterns and data swizzling, though this is very Hexagon-specific.

Predicates. Pitchfork could support more complicated predicates, or use more powerful theorem proving systems to prove compile-time facts, at the cost of compile-time. For example, Pitchfork's bounds inference fails to prove the same bound that Rake proves on an expression for the *gaussian3x3* benchmark on Hexagon – this leads to worse code generation. Additionally, Rake is able to prove facts about index relationships that allow it to generate sliding window instructions (i.e. `vtmpy` on *sobel3x3*), but Pitchfork does not currently support this.

x86 Offline Optimization. We only implement the generation of lowering rules for Pitchfork's ARM and HVX backends via Rake, which does not offer an x86 backend. We leave offline generation of x86 lowering rules as future work, but note that this backend lacks the number of interesting fused instructions that ARM and HVX offer.

Verified Lowering Systems. As noted in §2.4, we verify the rules that translate primitive integer arithmetic into FPIR. We could feasibly verify the rules that lower into target instructions, but this requires formal models of every target instruction. We leave this as future work.

Implementation in LLVM. Pitchfork was developed as part of the Halide compiler for ease of implementation. However, we believe our work makes a case for how fixed-point instruction selection should be done inside LLVM, and what fixed-point intrinsics LLVM should expose in its IR.

7 RELATED WORK

Fixed-Point Intermediate Representations. Despite attempts to standardize a set of portable fixed-point operators (e.g. [33]), to our knowledge the vast majority of fixed-point code in production is still expressed as primitive integer arithmetic, or non-portable intrinsics or assembly (e.g. [28], [18]). LLVM exposes a handful of generic fixed-point multiplication intrinsics [19]. However these have deliberately unspecified rounding behavior, so they are not useful for writing portable code and cannot be lifted to exactly from integer code.

Vectorization. There are decades of work on various vectorization methods [5, 8, 11, 12, 25, 30, 34, 39]. This work is largely complementary to our work as it generates vectorized IR, while our system takes already-vectorized IR and chooses concrete target instructions to perform computation.

Program Synthesis. Program synthesis is the task of constructing a program that satisfies a high-level specification [20]. There is extensive prior work in using online program synthesis for domains such as optimizing small tensor kernels [15], auto-vectorization of general-purpose loops [9], and optimizing distributed memory kernels [52]. These techniques do not perform offline synthesis in order to improve online performance, as our system does.

Verified Lifting. Prior work in verified lifting has proven successful at lifting legacy code into new DSLs and frameworks [2, 13, 24], including lifting legacy image processing code to Halide [3]. Our lifting system can be seen as a limited verified lifter, using offline instead of online synthesis.

Vectorization and Instruction Selection via Synthesis. Diospyros [48] is an e-graph based auto-vectorizer for synthesizing efficient linear algebra kernels for DSP architectures. It does not perform instruction selection, relying on a vendor-supplied compiler toolchain to lower vectorized expressions. Like auto-vectorization in general, this is complementary to our work in choosing optimal instructions for already-vectorized IR. Rake [4] is a synthesis-based compiler for instruction selection. We compare to Rake in §5, and it is our oracle for lowering-rule synthesis §4.2.

Term-Rewriting Systems. Recent work in building term-rewriting systems leverages synthesized rules for numerical simplifications [38] and optimization of fully-homomorphic encryption circuits [27]. These systems use offline program synthesis to learn a TRS, as we do. However, each of these systems learns a single TRS operating within a single IR, whereas we use two TRSs translating between IRs.

Buchwald et al. [10] proposed synthesizing a term-rewriting system for instruction selection, but only attempts to do direct translation to x86 for simple integer IR, and does not scale to the size of rules that Pitchfork enables.

Likewise, the Alive project [31, 32] verifies compiler optimizations, Alive-Infer [35] learns preconditions for optimization rules, and Souper [44] is a super-optimizing compiler for LLVM IR. Each of these work as rewrites within LLVM IR, and do not perform instruction selection.

A large amount of recent work [48, 51, 53, 54] uses e-graphs for term optimization, and for learning inference rules themselves [37]. Our investigation into these techniques showed that equality saturation is not well suited for translating between languages (necessary for both our lifting and lowering phases), does not efficiently support constraints or rule predicates (e.g. enforcing type-matching of operands), and is not efficient enough for online use.

8 DISCUSSION

Pitchfork is open-source[10], and Halide now uses FPIR (with hand-written TRSs) for *all* of its production CPU and DSP backends, including other x86 variants (SSE2, SSE4.1, and AVX512), 32-bit ARM, PowerPC, WebAssembly, XTensa, and the experimental RISC-V backend. Many of the synthesized rules for ARM and HVX have been merged into the Halide compiler. While these additional backends are not contributions of this paper, they illustrate FPIR's generality and success as a portable fixed-point language.

8.1 Common Threads

Usage of FPIR for instruction selection was straightforward for most of the ISAs listed above. Supporting WebAssembly, PowerPC, x86 variants, and ARM32 required no extensions to FPIR. WebAssembly SIMD was specifically designed to take advantage of common hardware capabilities [50], and therefore is similar to the x86 and ARM ISAs. PowerPC is similar to x86, and x86 variants do not introduce new fixed-point operations beyond the instruction classes that FPIR supports. The same is true for ARM32.

[10]https://github.com/halide/Halide/tree/rootjalex/trs-codegen

XTensa is similar to both ARM and HVX; however, it did reveal a shared instruction class between XTensa and ARM that prompted adding a new instruction to FPIR. We discuss this extension below, in §8.4. Lastly, the RISC-V vector extensions, while mostly covered by the existing design of FPIR, does support some fixed-point instructions that are not easily expressed in FPIR, which we discuss in §8.2.

While FPIR encapsulates most fixed-point instruction sets, it may need to evolve as hardware backends evolve. We believe (based on the work required to incorporate FPIR into the Halide compiler) that this work will be minimal.

8.2 Rounding Modes

The RISC-V Vector Extensions support a number of rounding modes for averaging and shifting instructions: round-up, round-down, round-towards-even, and round-towards-odd [43]. FPIR exposes only the first two variants for shifts and averaging instructions, as these variants are supported by multiple hardware targets[11]. These additional modes are rarely used in practice in portable code [1] because no other architectures support them and they are expensive to emulate, conflicting with FPIR's *portability* goal. We note in §6 that improving the performance of emulated instructions is interesting future work, which could enable supporting these new rounding modes with portable performance.

As discussed in §3.1.2, FPIR purposefully does not include `rounding_halving_sub`, the round-up subtracting averaging instruction. RISC-V does support this instruction; it has been excluded from FPIR as it conflicts with our design goals by offering an instruction with dubious semantics that only one ISA supports[12]. However, as we discuss in §8.4, FPIR can be easily extended to support new instructions, so it could be added if needed.

8.3 Targeting Relaxed Instructions

The WebAssembly Relaxed SIMD proposal [49] includes three new fixed-point instructions with relaxed determinism related to overflow semantics. These instructions can be easily matched using Pitchfork's pattern-matching in conjunction with its bounds inference machinery to prove that the original code cannot overflow, therefore allowing deterministic use of the relaxed instruction. For example, the `i16x8.q15mulr_s` instruction can be matched to a `rounding_mul_shr(x_i16, y_i16, 15)` pattern if either `x_i16` or `y_i16` cannot be `INT16MIN`. This shows that Pitchfork's machinery can be used for ensuring determinism, even if instructions may include non-deterministic behavior with certain input ranges.

8.4 Extending Pitchfork

We illustrate the extensibility of Pitchfork by showing how an instruction can be added to FPIR, and separately to Pitchfork's synthesis system. As mentioned above, adding the XTensa backend

[11]Note that while all targets support round-up averaging and round-down shifting, only XTensa, RISC-V, ARM, and HVX support round-down averaging and round-up shifting. Targets that do not support round-down and round-up shifts use Pitchfork's efficient lowering [17].

[12]32-bit ARM documentation [7] pre-armv8 shows a similar instruction but does not define overflow semantics, and no compiler that we are aware of generates this instruction.

to Halide revealed a shared instruction[13] class with the following semantics:

```
saturating_shl<T>(x, y)
  = saturating_cast<T>(widening_shl(x, y))
```

Extending FPIR is straightforward: a one-line definition of `saturating_shl` is added, one line of code is added to the lifter to pattern match the above pattern, nine lines are added to the ARM backend to map `saturating_shl` to its variants, and four lines to the XTensa backend. Lastly, one line is added to the lowering system that handles FPIR instructions on backends that do not directly support them. Note that a more efficient lowering may exist, but existing fixed-point bit-trick resources do not provide efficient lowering for this pattern. As noted in §6, super-optimizing the emulation of unsupported instructions is interesting future work.

Extending Pitchfork's synthesis system to support a new instruction requires only a few lines of Rosette code to encode the semantics of the instruction, and a log in the synthesis engine's list of available instructions.

9 CONCLUSION

We present Pitchfork, an instruction selector for fixed-point signal processing code written in Halide. Pitchfork is constructed around our intermediate representation FPIR, which captures the idioms of fixed-point computation in a target-agnostic way. Given portable fixed-point code written using primitive integer operations or FPIR directly, our system achieves the runtime performance of recent research compilers with the compile-time performance of production compilers.

ACKNOWLEDGMENTS

We thank the Halide contributors who have improved instruction selection over the years, especially Volodymyr Kysenko, Zalman Stern, Steven Johnson, Pranav Bhandarkar, and Ankit Aggarwal. We also thank Manya Bansal, Jacob Kahn, Fredrik Kjolstad, Amanda Liu, Katherine Mohr, Kevin Mu, Rohan Yadav, Bobby Yan, and David Zhang, for their helpful feedback on drafts of this paper. Alexander J. Root was supported by the NSF Graduate Research Fellowship, and part of this work was done while he was an intern at Adobe Research and a student at MIT. This work was supported by NSF awards CCF-1723445, CCF-1846502, and funding from the SRC JUMP ADA center.

REFERENCES

[1] Andrew Adams and Dillon Sharlet. 2022. Better Fixed-Point Filtering with Averaging Trees. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3 (July 2022), 1–8. https://doi.org/10.1145/3543869

[2] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220.

[3] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics* 38 (11 2019), 1–13. https://doi.org/10.1145/3355089.3356549

[4] 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*. Association for Computing Machinery. https://doi.org/10.1145/3503222.3507714

[5] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (oct 1987), 491–542. https://doi.org/10.1145/29873.29875

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

[7] ARM. [n.d.]. *Learn the architecture - Neon programmers' guide: D.3.13. VRHSUB*. Technical Report. ARM Developer. https://developer.arm.com/documentation/den0018/a/NEON-Intrinsics-Reference/Arithmetic/VRHSUB

[8] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-Vectorization for Irregular Loops. *SIGPLAN Not.* 51, 6 (jun 2016), 697–710. https://doi.org/10.1145/2980983.2908111

[9] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/2442516.2442529

[10] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/3168821

[11] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[12] 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 902–914. https://doi.org/10.1145/3445814.3446692

[13] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA.

[14] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* 34, 02 (mar 2014), 34–43. https://doi.org/10.1109/MM.2014.12

[15] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic Generation of High-Performance Quantized Machine Learning Kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM. https://doi.org/10.1145/3368826.3377912

[16] 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.

[17] Henry Gordon Dietz. [n.d.]. *The Aggregate Magic Algorithms*. Technical Report. University of Kentucky. http://aggregate.org/MAGIC/

[18] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2019. Fast Sparse ConvNets. https://doi.org/10.48550/ARXIV.1911.09723

[19] LLVM Foundation. 2022. LLVM Fixed Point Arithmetic Intrinsics. https://llvm.org/docs/LangRef.html. Accessed: 2022-10-18.

[20] S. Gulwani, O. Polozov, and R. Singh. 2017. *Program Synthesis*. Now Publishers. https://books.google.com/books?id=mK5ctAEACAAJ

[21] Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. 2016. Burst Photography for High Dynamic Range and Low-Light Imaging on Mobile Cameras. *ACM Trans. Graph.* 35, 6, Article 192 (nov 2016), 12 pages. https://doi.org/10.1145/2980179.2980254

[22] Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Transactions on Graphics* 40, 4 (aug 2021), 1–16. https://doi.org/10.1145/3450626.3459671

[23] Intel. [n.d.]. *Intel Intrinsics Guide*. Technical Report. Intel. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[24] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 711–726.

[13] ARM Neon's `sqshl`, `uqshl`, and `sqshlu`, and XTensa's `IVP_SLSN*` intrinsics.

[25] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not.* 35, 5 (may 2000), 145–156. https://doi.org/10.1145/358438.349320

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

[27] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery. https://doi.org/10.1145/3385412.3385996

[28] libjpeg turbo. 2022. *libjpeg-turbo*. Technical Report. https://github.com/libjpeg-turbo/libjpeg-turbo/tree/5446ff88d617b2d2768456d9be1a8c47c4606c92/simd

[29] ARM Limited. 2011. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ga

[30] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A Compiler Framework for Extracting Superword Level Parallelism. *SIGPLAN Not.* 47, 6 (jun 2012), 347–358. https://doi.org/10.1145/2345156.2254106

[31] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[32] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[33] John McFarlane. 2018. Fixed-Point Real Numbers. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0037r5.html. Accessed: 2022-10-18.

[34] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (oct 2018), 28 pages. https://doi.org/10.1145/3276480

[35] David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. *SIGPLAN Not.* 52, 6 (jun 2017), 49–63. https://doi.org/10.1145/3140587.3062372

[36] Millind Mittal, Alex Peleg, and Uri Weiser. 1997. MMX Technology Architecture Overview. *Intel Technology Journal* Q3 (1997), 12. http://developer.intel.com/technology/itj/q31997/articles/art_2.htm;http://developer.intel.com/technology/itj/q31997/pdf/archite.pdf

[37] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

[38] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428234

[39] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. *SIGPLAN Not.* 41, 6 (jun 2006), 132–143. https://doi.org/10.1145/1133255.1133997

[40] William K. Pratt. 2007. *Digital Image Processing: PIKS Scientific Inside*. Wiley-Interscience, USA. https://doi.org/10.1002/0470097434

[41] Qualcomm Technologies 2018. *Qualcomm Hexagon V66 HVX Programmer's Reference Manual* (80-n2040-44 rev. b ed.). Qualcomm Technologies.

[42] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[43] RISC-V. [n.d.]. *RISC-V "V" Vector Extension*. Technical Report. RISC-V. https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#v-vector-extension-for-application-processors

[44] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. https://doi.org/10.48550/ARXIV.1711.04422

[45] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (Jan. 2016), 114–122. https://doi.org/10.1145/2863701

[46] Manu Mathew Thomas, Karthik Vaidyanathan, Gabor Liktor, and Angus G. Forbes. 2020. A Reduced-Precision Network for Image Reconstruction. *ACM Trans. Graph.* 39, 6, Article 231 (nov 2020), 12 pages. https://doi.org/10.1145/3414685.3417786

[47] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[48] 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[49] WebAssembly. [n.d.]. *Relaxed SIMD proposal for WebAssembly*. Technical Report. WebAssembly. https://github.com/WebAssembly/relaxed-simd

[50] WebAssembly. [n.d.]. *WebAssembly 128-bit packed SIMD Extension*. Technical Report. WebAssembly. https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md

[51] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[52] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. 2014. MSL: A Synthesis Enabled Language for Distributed Implementations. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 311–322. https://doi.org/10.1109/SC.2014.31

[53] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. https://doi.org/10.48550/ARXIV.2101.01332

[54] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational E-Matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (jan 2022), 22 pages. https://doi.org/10.1145/3498696