Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors

Samuel Thomas
sgt@cs.utexas.edu
The University of Texas at Austin
Austin, TX, USA

James Bornholt
bornholt@cs.utexas.edu
The University of Texas at Austin
Austin, TX, USA

Abstract

Embedded applications extract the best power–performance trade-off from digital signal processors (DSPs) by making extensive use of vectorized execution. Rather than hand-writing the many customized kernels these applications use, DSP engineers rely on auto-vectorizing compilers to quickly produce effective code. Building these compilers is a large and error-prone investment, and each new DSP architecture or application-specific ISA customization must repeat this effort to derive a new high-performance compiler.

We present Isaria, a framework for automatically generating vectorizing compilers for DSP architectures. Isaria uses equality saturation to search for vectorized DSP code using a system of rewrite rules. Rather than hand-crafting these rules, Isaria automatically synthesizes sound rewrite rules from an ISA specification, discovers *phase structure* within these rules that improve compilation performance, and schedules their application at compile time while pruning intermediate states of the search. We use Isaria to generate a compiler for an industrial DSP architecture, and show that the resulting kernels outperform existing DSP libraries by up to 6.9× and are competitive with those generated by expert-built compilers. We also demonstrate how Isaria can speed up exploration of new ISA customizations by automatically generating a high-quality vectorizing compiler.

CCS Concepts: • **Hardware → Digital signal processing**; • **Software and its engineering → Retargetable compilers**; • **Theory of computation → Equational logic and rewriting**.

Keywords: Vectorization, DSPs, Equality Saturation, Rewrite Rules, Program Synthesis

ACM Reference Format:
Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3617232.3624873

1 Introduction

Low-power embedded applications such as remote sensing and virtual reality make extensive use of *digital signal processors* (DSPs) to meet tight energy and performance targets. DSPs meet these targets by using simple but heavily vector-oriented architectures. Large DSP kernels make easy use of these vector units by calling into off-the-shelf linear algebra libraries like Nature or Eigen [10]. But recent work [35] has observed an Amdahl's law effect in which DSP applications are increasingly bottlenecked by a long tail of small, custom kernels not available off the shelf. This effect means that the barrier to deploying DSP applications is compiling custom kernels to efficient vectorized code. The compilation challenge is amplified by the wide range of DSP instruction-set variations offered by vendors, each of which requires its own vectorizing compiler, and application-specific ISA customizations that necessitate building a compiler for a bespoke instruction set.

To address this problem, recent work has developed new techniques for automatically vectorizing these kernels, including program synthesis [1], pattern matching [7], and autotuning [30]. In this paper we focus on the promising *equality saturation* [32] approach to auto-vectorization pioneered by Diospyros [35]. With Diospyros, the compiler writer defines a system of *rewrite rules* to transform scalar programs into vectorized ones. To compile a program, Diospyros searches the space of vectorized programs using equality saturation, which effectively applies the rewrite rules in all possible orders simultaneously [37] using congruence closure on an E-graph data structure [19]. Diospyros then extracts the most efficient vectorized program from the E-graph using an abstract cost model. This approach achieves excellent performance, with kernels 3.1× faster than off-the-shelf DSP libraries and competitive with expert-written custom kernels.

While these results are encouraging, the equality saturation approach places several burdens on the compiler writer (or the DSP engineer, for application-specific extensions). First, crafting effective rewrite rules for equality saturation is a delicate balancing act [18]: the author must develop rules whose congruence closure reaches interesting programs, while using fragile heuristics to avoid rules that might loop infinitely or provoke NP-complete associativity and commutatively problems [4]. In practice, the best way to design a system of rewrite rules for vectorization today is through trial and error, but even this is difficult as "error" often means waiting for hours-long timeouts and manually examining E-graphs with millions of nodes. Second, the rewrite rules must be sound for the compiler to be correct, but correctness bugs in compiler rewrite rules are common even for widely adopted toolchains and architectures [13, 15, 38]. Finally, a system of rewrite rules is necessarily specific to an instruction set, and so this manual effort slows the exploration of customized instructions for an application and of new instruction set designs for future DSPs. In short, low-power DSP applications require not just a single effective vectorizing compiler, but a productive approach to designing and evolving the compiler itself.

This paper introduces the Isaria framework for developing vectorizing compilers for DSP architectures. At a high level, Isaria follows the approach of (and is based on) the Diospyros compiler [35], reducing the vectorization problem to a search problem using equality saturation. But Isaria takes a radically different approach to developing the compiler itself: rather than manually crafting rewrite rules and heuristics for applying them, Isaria automatically generates a suitable system of rewrite rules offline, and then schedules their application during equality saturation at compile time to generate a vectorized program. Concretely, the Isaria framework takes as input a specification of the target instruction set, implemented as an executable interpreter in the Rosette solver-aided language [33], together with an abstract cost model for the ISA's instructions. Given these inputs, Isaria automatically generates a system of rewrite rules and a schedule for them, and builds these results into Diospyros to produce a vectorizing compiler for the target architecture.

Isaria's key insight is that rewrite rules for vectorization fall into natural *phases*: some rules expand the program to expose vectorization opportunities, others simplify the program to make vector execution faster, and still others perform the actual vectorization transformations. On the surface, this insight contradicts the common motivation for equality saturation in compilers, as an antidote to the "phase ordering" problem of choosing a good ordering for applying destructive program transformations [32]. But unlike existing compilers, Isaria discovers these phases automatically, from only an ISA specification and an abstract cost model. The resulting phases are far more coarse-grained than traditional compilers, and Isaria's equality saturation uses these automatic phases to make vectorization tractable at compile time.

The Isaria workflow comprises three parts: generation of candidate rewrite rules, analysis and phase selection for those rules, and application of the rules at compile time. To generate candidate rewrite rules, Isaria extends the Ruler synthesis engine [18], which synthesizes sound rewrite rules from a language specification, with support for tractable synthesis with lane-wise vector instructions. Ruler is effective at generating candidate rules, but perhaps *too* effective—realistic DSP instruction sets are so large that it generates hundreds of candidate rules, making equality saturation, and therefore vectorization, intractable. To mitigate this explosion of rules, Isaria groups them into *phases* by analyzing their effect on program performance using the abstract cost model. Importantly, it is not sufficient to only select rewrites that strictly decrease program cost; rules that (temporarily) increase cost are useful to reach different parts of the search space [28], but must be used judiciously to avoid fruitless or redundant exploration. Finally, at compile time, Isaria performs multiple iterations of equality saturation, scheduling rules into different iterations based on their phase. Between iterations, Isaria greedily prunes the search by extracting an optimal intermediate program from the saturated E-graph and discarding the rest. This pruning sacrifices completeness, but keeps the size of the E-graph in check, allowing Isaria to find interesting vectorizations while avoiding resource exhaustion.

We evaluate Isaria by generating a vectorizing compiler targeting Tensilica DSPs. We show that kernels compiled with this compiler outperform equivalent Tensilica SDK libraries by up to 6.9×, and the Tensilica clang-based auto-vectorizer by up to 25×. The kernels have similar performance to Diospyros's hand-crafted compiler—34% faster on average, although this average is skewed by a few large kernels—despite the Isaria-based compiler being automatically generated from the ISA specification. Isaria vectorizes most kernels in just a few minutes, although compilation is an average of 2.1× slower than Diospyros. We also demonstrate how Isaria helps DSP engineers experiment with customized instructions by exploring two new instructions to speed up a kernel without any manual compiler changes.
In summary, this paper makes the following contributions:

- The Isaria approach to automatically generating a vectorizing compiler for low-power architectures.
- A new phase-oriented approach to selecting and applying rewrite rules for equality saturation.
- An evaluation on an industrial DSP architecture showing that Isaria-based compilers outperform commercial tools and match the state of the art in manually crafted tools.

2 Rewriting for Vectorization

This section gives an overview of the Diopsyros [35] approach to vectorization on which Isaria builds. We show how equality saturation can produce efficient vectorized programs by searching with small, local rewrite rules. However, crafting these rewrite rules is a challenging task that makes it difficult for DSP engineers to realize the benefits of new architectures or custom ISA extensions. In this section we describe the key challenges in using rewrite rules for vectorization. Then in Section 3 we present the Isaria framework automatically building a vectorizing compiler for a DSP architecture, and show how it solves these challenges by specializing equality saturation to the vectorization domain.

2.1 Vectorization via Equality Saturation

Isaria follows the Diospyros [35] approach to vectorization using equality saturation. To see how this approach works, consider vectorizing this trivial program whose output is a 4-wide vector:

```
var r0 = x[0] + y[0];
var r1 = x[1] + y[1];
var r2 = x[2] + y[2];
var r3 = x[3];
return {r0, r1, r2, r3};
```

The Diospyros compiler starts by *lifting* this program to an expression using symbolic evaluation [33] to remove variables and control flow, leaving a program reflecting the output vector but written in the Diospyros vector DSL in Fig. 1:

```
(Vec
  (+ x[0] y[0])
  (+ x[1] y[1])
  (+ x[2] y[2])
  x[3])
```

Here, the Vec term represents a 4-wide vector value, where each lane can be a separate arbitrary expression. Such an instruction does not exist directly in the hardware for most DSPs; each vector lane must instead be constructed separately and moved into the vector register. Isaria uses Vec terms to abstract away the details of shuffle and data movement instructions of the DSP; these instead are handled by a lowering pass after equality saturation.

There are many potential vectorizations of this program. Traditional auto-vectorization approaches like superword-level parallelism [12] would analyze the program and apply a fixed sequence of program transformations to produce vectorized code. These techniques are often effective for regular kernels, but embedded DSP applications tend to be bottlenecked by smaller custom kernels like this irregular vector addition example. For these kernels, the best approaches to auto-vectorization [1, 7, 35] instead *search* the space of vectorizations to find an efficient one.

The Diospyros approach to vectorization starts with a set of small, local *rewrite rules* crafted by hand for a particular DSP architecture. A rewrite rule $(+\ a\ b) \rightsquigarrow (+\ b\ a)$ says that the term $(+\ a\ b)$ can be replaced with $(+\ b\ a)$ anywhere it appears in the program. Here, a and b are *wildcards* that match any expression, and so for example this rule allows rewriting the program $(+\ (-\ x\ y)\ z)$ to $(+\ z\ (-\ x\ y))$ by binding a to $(-\ x\ y)$ and b to z.

When combined in the right order, local rewrite rules can produce a vectorization of the program. For example, given two rewrite rules:

$$\begin{aligned}&(\text{Vec}\ (+\ a_0\ b_0)\ (+\ a_1\ b_1)\ (+\ a_2\ b_2)\ (+\ a_3\ b_3))\\ &\rightsquigarrow (\text{VecAdd}\ (\text{Vec}\ a_0\ a_1\ a_2\ a_3)\ (\text{Vec}\ b_0\ b_1\ b_2\ b_3))\end{aligned}$$

and

$$a_0 \rightsquigarrow (+\ a_0\ 0)$$

the example program can be vectorized by applying the second rule to x[3], which then makes the first rule applicable, yielding a vectorized program

```
(VecAdd (Vec x[0] x[1] x[2] x[3])
        (Vec y[0] y[1] y[2] 0))
```

that computes the result with a vector add instruction.

But how did we know to apply the rewrite rules in this order? A traditional compiler fixes an ordering to apply transformations, but Diospyros instead uses *equality saturation* [32] to search the space of rewrite orderings. Rather than applying rewrite rules destructively, equality saturation applies them to an *E-graph* data structure [19], which concisely represents a large set of programs and equivalences between them. Equality saturation repeatedly applies the rules to the E-graph until it stops changing, at which point we say the E-graph has *saturated* and contains all programs reachable by applying the rewrite rules in any order (including repeated applications) to the original program. The optimal solution can then be *extracted* from the E-graph by using a cost function that assigns a cost to each program in the graph, and selecting one with minimal cost. Diospyros showed that equality saturation is especially effective at vectorizing irregular DSP kernels, achieving an average of 3.1× better performance than hand-crafted linear algebra libraries.

2.2 Synthesizing Rewrite Rules

While Diospyros demonstrates the promise of equality saturation for vectorization, it places a heavy burden on the compiler writer. Crafting a set of rewrite rules for equality saturation requires striking a delicate balance between discovering novel vectorizations and being so general that saturation becomes intractable. For example, the rewrite rule $a_0 \rightsquigarrow (+\ a_0\ 0)$ above was essential to vectorize the example, but must be used carefully, as in principle it matches any term and so can be applied infinitely often. As a more complex example, the AC-matching problem [4] of checking whether two terms are equivalent up to associativity or commutativity is NP-complete, and so rewrite rules for associative or commutative identities can dramatically increase the size of the E-graph and prevent saturation [17].

$\langle prog \rangle$ ::= (List $\langle expr \rangle^{+}$) | $\langle expr \rangle$

$\langle expr \rangle$ ::= $\langle scalar \rangle$ | $\langle vector \rangle$

$\langle scalar \rangle$ ::= $\langle integer \rangle$ | $\langle variable \rangle$
| (+ $\langle scalar \rangle$ $\langle scalar \rangle$) | (- $\langle scalar \rangle$ $\langle scalar \rangle$)
| (* $\langle scalar \rangle$ $\langle scalar \rangle$) | (/ $\langle scalar \rangle$ $\langle scalar \rangle$)
| (sgn $\langle scalar \rangle$) | (sqrt $\langle scalar \rangle$) | (- $\langle scalar \rangle$)
| (Get $\langle variable \rangle$ $\langle integer \rangle$)

$\langle vector \rangle$ ::= (Vec $\langle scalar \rangle^{+}$) | (Concat $\langle vector \rangle$ $\langle vector \rangle$)
| (VecAdd $\langle vector \rangle$ $\langle vector \rangle$) | (VecMinus $\langle vector \rangle$ $\langle vector \rangle$)
| (VecMul $\langle vector \rangle$ $\langle vector \rangle$) | (VecDiv $\langle vector \rangle$ $\langle vector \rangle$)
| (VecMAC $\langle vector \rangle$ $\langle vector \rangle$ $\langle vector \rangle$)
| (VecSgn $\langle vector \rangle$) | (VecSqrt $\langle vector \rangle$)
| (VecNeg $\langle vector \rangle$)

Figure 1. The Diospyros vector DSL. This is the DSL that Isaria automatically learns rules over. A top-level program is a (possibly singleton) list of outputs. Expressions operate over both scalars and vectors.

Diospyros works around these problems by first carefully hand-writing its rewrite rules to avoid them, and then equipping many of its rewrite rules with custom *scheduling* logic for when and how to apply them during saturation. This work does not generalize to different DSP architectures, which each need a new set of hand-crafted rules. The difficulty of producing rewrite rules also prevents DSP engineers from exploring potential customized instructions, as they need to extend the compiler with new rewrite rules and scheduling logic to support the new instructions.

One solution to this problem would be to use a general-purpose tool for synthesizing rewrite rules such as Ruler [18]. These tools take as input a language specification and an oracle for validating rewrite rules, and output a set of sound rewrite rules over the language. In principle, applying these rewrite rules using equality saturation would then yield a vectorizing compiler: saturation would find all possible ways to rewrite the scalar target program according to the rules, including some that are vectorized, and we could extract the fastest program from the saturated E-graph using the cost model. In practice, however, the synthesized sets of rewrite rules for DSP architectures are too general and large to be useful. For example, we applied Ruler directly to Diospyros's vector language, which synthesized 300 rewrite rules. Trying to use these rules to vectorize a small 2×2 by 2×2 2D matrix convolution causes equality saturation to exhaust 64 GiB of memory without producing any results. Some tweaking of equality saturation can reduce this memory usage, but even then it fails to find any vectorized program after an hour of searching. In contrast, Isaria finds a fast vectorized solution in only 3 seconds while using 0.2 GiB of memory.

2.3 E-graph Explosion

Why are rewrite rules synthesized with off-the-shelf tools not sufficient to build a high-quality vectorizing compiler? As an example, suppose the rewrite synthesis tool generates three rules about vector addition (using 2-wide vector instructions for clarity):

$$\text{(Vec (+ } a_0\ a_1\text{) (+ } b_0\ b_1\text{))} \rightsquigarrow \text{(VecAdd (Vec } a_0\ b_0\text{) (Vec } a_1\ b_1\text{))} \quad (1)$$

$$\text{(VecAdd (Vec } a_0\ a_1\text{) (Vec } b_0\ b_1\text{))} \rightsquigarrow \text{(Vec (+ } a_0\ b_0\text{) (+ } a_1\ b_1\text{))} \quad (2)$$

$$\text{(VecAdd } a\ b\text{)} \rightsquigarrow \text{(VecAdd } b\ a\text{)} \quad (3)$$

Consider applying these rules using equality saturation to this program that sums four elements of two arrays:

```
(Vec (+ (+ X[0] X[1])
        (+ X[2] X[3]))
     (+ (+ Y[0] Y[1])
        (+ Y[2] Y[3])))
```

One potential ordering of rule applications first vectorizes the outer additions using rule 1:

```
(VecAdd
 (Vec (+ X[0] X[1]) (+ Y[0] Y[1]))
 (Vec (+ X[2] X[3]) (+ Y[2] Y[3])))
```

Next we can apply rule 3 to commute the lanes of the vector addition:

```
(VecAdd
  (Vec (+ X[2] X[3]) (+ Y[2] Y[3]))
  (Vec (+ X[0] X[1]) (+ Y[0] Y[1])))
```

Finally, applying rule 2 undoes the vectorization, yielding a permutation of the original program:

```
  (Vec (+ (+ X[2] X[3])
          (+ X[0] X[1]))
       (+ (+ Y[2] Y[3])
          (+ Y[0] Y[1])))
```

Since this rule ordering is possible, equality saturation will explore it, along with many other orderings that result in similar permutations of the same unvectorized program. These redundant explorations will blow up the size of the E-graph, preventing saturation within reasonable time and therefore preventing vectorization.

This example highlights two problems with applying a general-purpose rule synthesizer to the vectorization problem. First, these rules can *go backwards* during search: because the tools do not understand the goal of vectorization (to produce a vectorized version of a scalar program), they synthesize rules that are cyclic and can undo vectorization, yielding a permutation of the unvectorized input program. In fact, equality saturation guarantees that the search will eventually explore every reachable permutation of the unvectorized input program. These variants are helpful in principle, as they might expose future vectorization opportunities. But the usefulness of these permutations is the second problem: the rules are *redundant*, with many paths to reach the same permutation, including by vectorizing and then unvectorizing the program. In the example, a simple scalar addition commutativity rule $(+\ a_0\ a_1) \rightsquigarrow (+\ a_1\ a_0)$ would have reached the same final program without round-tripping through a vectorization.

The key insight from this example is that to make synthesized rewrite rules for vectorization scale, we must avoid this E-graph explosion by carefully controlling which rules are applied and when. Indeed, Diospyros does this by hand, with manual heuristics to apply certain rewrite rules only at certain points during the search. With Isaria, we aim to automate this entire process, so that DSP engineers can experiment with new architectures and new custom instructions without first manually building a vectorizing compiler.

3 Phase-Oriented Rule Synthesis

The Isaria framework automates the development of rewrite rules for vectorizing compilers. By synthesizing rewrite rules automatically, Isaria makes it easier to build an effective compiler for a new DSP architecture, or to extend an existing architecture with application-specific custom instructions.

Figure 2 shows an overview of the Isaria workflow. It takes as input the semantics of the target instruction set (i.e., an interpreter) and an abstract cost model for programs in that instruction set (both inputs that the Diospyros compiler requires). Given these inputs, the Isaria workflow comprises three parts: *generation* of a set of candidate rewrite rules, *analysis and arrangement* of those rules into phases, and *application* of the rules by phase at compile time. This approach takes advantage of the insight that not all rewrite rules are equally useful for vectorization. Rather than trying to compute the congruence closure of a prohibitively large rule set, Isaria instead subsets the rules into distinct, coarse-grained phases and applies equality saturation to each phase independently.

3.1 Rewrite Rule Generation

To generate a candidate set of rewrite rules for vectorization, Isaria uses the Ruler synthesis engine [18]. Ruler takes as input an interpreter for the target language and aims to generate a small, sound set of useful rewrite rules over that language. It does so by enumerating terms in the target language up to equivalence, generating rewrite rules between those terms using a test-based filtering approach, and then shrinking that set of rules by using equality saturation to remove rules derivable from other candidates. To ensure soundness, Ruler augments test-based filtering with an SMT-based verifier that checks rule soundness under the semantics defined by the input interpreter. Nandi et al. [18] showed that Ruler can synthesize small, useful sets of rules for rewriters in an SMT solver [3] and a numerical analysis tool [23].

For Isaria, the target language is the instruction set of the DSP architecture. This language is not especially large—DSPs are simple processors by design—but includes both scalar and vector variants of many instructions, as well as a general Vec instruction (Section 2.1) that abstracts data movement. Ruler can synthesize a set of sound rewrite rules for this language, and its filtering is effective at reducing the size of the set. In the example in Section 2.2, Ruler considered 7,735 rewrite rules but its final synthesized rule set contained only 300 rules. However, even this filtered rule set is an order of magnitude larger than Diospyros's 28 hand-written rules for the same DSP, creating a scalability challenge we address in Section 3.2.

Vector lane generalization. Ruler struggles to learn effective rewrite rules for vectorized lane-wise instructions like VecAdd. This is because it has no built-in knowledge that these instructions have the same effect on each lane, and that the lanes do not interact. As a result, Ruler spends most of its time rediscovering rules like associativity and commutativity for each lane and each combination of lanes, making little progress towards discovering interesting vectorization rules.

To avoid this redundant work, Isaria extends Ruler with the ability to *generalize* synthesized rules across vector lanes. At rule synthesis time, Isaria reduces the vector instructions in the ISA to a single lane. This allows Ruler to discover per-lane behavior only once and so make deeper progress into the search space. After synthesis, Isaria generalizes the vector instructions in the synthesized rules back to the architecture's actual vector width, replicating arguments from the first lane into the others but with fresh wildcards for each lane. This generalization risks creating unsound rules for vector instructions that have cross-lane interactions (e.g., reductions), but the DSP architectures we studied have few of these rules, and they are mostly data movement instructions that Isaria abstracts out of the language with Vec. To mitigate this risk, we use the executable ISA specification to formally verify the soundness of the expanded rules [15, 33], and drop rules that fail to verify (although there are no such rules in our experiment, owing to the simple nature of DSP architectures).

3.2 Discovering Rule Phases

While Ruler successfully synthesizes candidate rewrite rules, its rule sets are too large to be used for equality-saturation-based compilation directly (Section 2.3). To make vectorization with synthesized rewrite rules tractable, Isaria analyzes and categorizes the candidate rules produced by Ruler. This analysis builds on the observation that because our target language (the DSP instruction set) contains both scalar and vector instructions, a rewriting-based vectorization approach

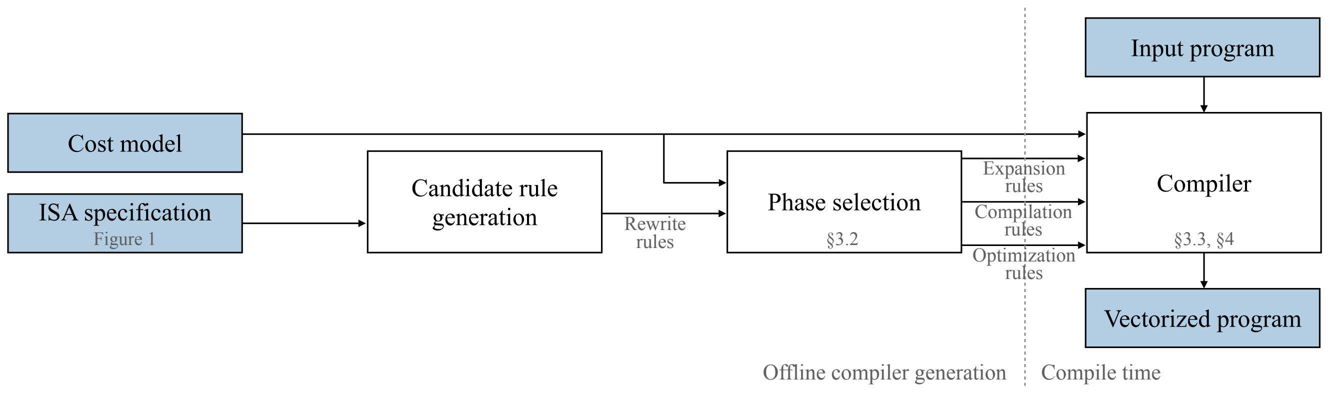

Figure 2. The Isaria workflow for automatically generating a vectorizing DSP compiler. Isaria takes as input an ISA specification and a cost model, and in an offline stage, synthesizes vectorization rewrite rules and organizes them into phases. At compile time, Isaria uses the synthesized rules and cost model to vectorize the program. Shaded boxes are inputs and outputs; unshaded boxes are provided by Isaria.

has natural *phases*, with different rules being useful during each phase. The intuition for these phases is that the program being vectorized starts out as a fully scalar term but ends (hopefully) as a fully vector term. An individual rewrite rule is small and so cannot vectorize the entire program in a single step. Therefore, the rewriting process must gradually transition (parts of) the program from scalar to vector form.

As these transitions happen, scalar-to-scalar rewrites become less useful, since there are fewer scalar parts of the program, while vector-to-vector rewrites become more useful. But a naive equality saturation approach has no way to take advantage of this progression, and will continue trying to apply (for example) scalar-to-scalar rules to terms that have already been profitably vectorized. This work occurs because a node in an E-graph represents *all* equivalent programs according to the equalities (rewrite rules) applied to the graph so far, and so the node for a vectorized term is still eligible to have more scalar rewrites applied to it. In theory, these additional rewrites are actually desirable, as futher scalar rewrites may expose a better vectorization later in the saturation process. But empirically we have observed that this outcome is rare—once a term has a vectorized form, it generally profits only from vector rewrites (i.e., optimizations). We can therefore make equality saturation more tractable by preventing this *interference* between scalar and vector rules.

Three rule phases. To make concrete this intuition about the effectiveness of rules, Isaria arranges candidate rewrite rules into three phases:

1. *Expansion* rules explore different ways of representing an unvectorized program.
2. *Compilation* rules lower unvectorized parts of a program onto vector instructions.
3. *Optimization* rules explore more efficient vectorizations of a vectorized program.

These phases roughly echo the workflow of a modern compiler, which applies independent optimizations at multiple levels of intermediate representations rather than trying to both lower and optimize the program at the same time.

The goal of these phases is to reduce the interference effect discussed above. For example, we can resolve the two problems in Section 2.3 by arranging for the three rules to be in different phases, so that they are not saturated together. Note that the names we give these phases have no semantic meaning but are just convenient labels for the types of rules typically involved. The important distinction between the phases is their effects on the *cost* of the program, as described next.

Cost-based phase assignment. Isaria assigns candidate rewrite rules to phases by analyzing them with the abstract cost model provided as input. The abstract cost model assigns a natural number cost to every program in the target language (the DSP instruction set).

Definition 1 (Cost function). *Let $\mathcal{L}$ be a target language. A* cost function *is a function $C : \mathcal{L} \to \mathbb{N}$ that assigns a cost to each program in $\mathcal{L}$.*

The cost model represents the expected performance of the program (e.g., cycle count). It need not be precise, but its faithfulness to the hardware affects the quality of the output of an Isaria compiler, which chooses a vectorized program that minimizes the cost function. Cost functions can be recursive. For example, the cost of a Vec term needs to be proportional to the cost of constructing the vector in hardware, and so needs to incorporate the cost of its subexpressions—a constant vector is cheaper to construct than one with distinct intermediate values, which in hardware must be loaded into a vector register one lane at a time.

As is common in equality-saturation-based compilers, we require the cost function to be strictly monotonic to avoid the need to consider zero-cost variations of a program when extracting the optimal solution from equality saturation (Section 2.1).

Definition 2 (Strict monotonicity)**.** *A cost function C is* strictly monotonic *if, whenever a program $P \in \mathcal{L}$ is a subexpression of another program $Q \in \mathcal{L}$, $C(P) < C(Q)$.*

Given a strictly monotonic cost function, Isaria's phase selection step assigns each candidate rewrite rule (Section 3.1) to one of the three rule phases by analyzing the cost of both sides of the rule. This analysis computes two metrics about a candidate rule: its *cost differential* and its *aggregate cost.*

Definition 3 (Cost differential)**.** *Let $P \rightsquigarrow Q$ be a rewrite rule, where $P, Q \in \mathcal{L}$. The* cost differential *of the rule $P \rightsquigarrow Q$, written $C_D(P \rightsquigarrow Q)$, is $C(P) - C(Q)$.*

Definition 4 (Aggregate cost)**.** *Let $P \rightsquigarrow Q$ be a rewrite rule, where $P, Q \in \mathcal{L}$. The* aggregate cost *of the rule $P \rightsquigarrow Q$, written $C_A(P \rightsquigarrow Q)$, is $C(P) + C(Q)$.*

Isaria uses the cost differential and aggregate cost to assign candidate rewrite rules to compilation phases using two parameters α and β that are part of the cost model. The assignment happens in two steps:

1. Rules with large cost differential $C_D(P \rightsquigarrow Q) > \alpha$ are *compilation* rules, because they are rules that significantly lower the cost of the program.
2. Otherwise, rules with large aggregate cost $C_A(P \rightsquigarrow Q) > \beta$ are *expansion* rules, while rules with small aggregate cost $C_A(P \rightsquigarrow Q) \leq \beta$ are *optimization* rules.

The intuition for this process is that rules with high cost differential are likely to be rules that transition from scalar to vector programs. For example, a rule like:

$$\begin{aligned}&(\text{Vec } (+\ a_0\ b_0)\ (+\ a_1\ b_1))\\ &\quad \rightsquigarrow (\text{VecAdd } (\text{Vec } a_0\ a_1)\ (\text{Vec } b_0\ b_1))\end{aligned}$$

has much higher cost on the left-hand side than the right-hand side, as the left-hand side does multiple scalar adds versus one vector add. Rules with low cost differential are likely scalar-to-scalar or vector-to-vector rules, and the cost aggregate allows Isaria to distinguish those two cases and assign them to the expansion or optimization phases, respectively.

The parameters α and β control how Isaria allocates rules to phases. In the limit, very high or low values of α and β reduce a Isaria compiler to a single phase, approximating the Diospyros approach. In practice, these parameters can be selected by inspecting the cost model. For example, β should be between the costs of a scalar and vector addition, and α should be at least as big as the difference in cost of any two scalar operations. We explore this further in Section 5.5.

An alternative strawman approach for assigning rules to phases would be purely syntactic: look at whether the root node of the AST on each side of the rule is scalar or vector and categorize accordingly. While this approach is simple, it fails in practice for two reasons. First, it struggles to deal with Vec literals, which abstract data movement. Some Vec rules are expansions, while others are compilation rules; separating the two is important for tractability as shown in Section 2.3. Second, it does not deal well with nested rules, which automated synthesis tools like Ruler can often generate. For example, a rule with VecAdd on both sides might in fact be optimizing a scalar addition nested inside an inner Vec literal, and so the rule is actually an expansion rule (transforming the scalar part of a partially vectorized program). Isaria's cost-based approach automatically handles these cases.

```
function COMPILE(P, R)
    $C_{\text{Old}} \leftarrow \text{COST}(P)$
    loop
        $E \leftarrow \text{EGRAPH}(P)$
        $E \leftarrow \text{EQSAT}(E, R_{\text{Expansion}})$
        $E \leftarrow \text{EQSAT}(E, R_{\text{Compilation}})$
        $P, C_{\text{New}} \leftarrow \text{EXTRACT}(E, C)$
        if $C_{\text{New}} = C_{\text{Old}}$ then
            break
        $C_{\text{Old}} \leftarrow C_{\text{New}}$
    $E \leftarrow \text{EGRAPH}(P)$
    $E \leftarrow \text{EQSAT}(E, R_{\text{Optimization}})$
    $P, \_ \leftarrow \text{EXTRACT}(E, C)$
    return $P$
```

Figure 3. The Isaria compilation algorithm takes as input a scalar program P and returns a compiled program. COMPILE is parameterized by a cost function C and a set of phased rewrite rules R generated offline. The EGRAPH, EQSAT, and EXTRACT procedures are provided by an E-graph library.

3.3 Scheduling Rule Phases at Compile Time

Once rule generation and phase discovery have been performed offline, Isaria emits a compiler equipped with the phased rule set. At compile time, this compiler takes as input a scalar program, and vectorizes it by applying the rules. This application process takes advantage of the phases inferred in Section 3.2 to make equality saturation tractable.

Figure 3 shows the Isaria compilation algorithm. An Isaria compiler is parameterized by a cost function C (Definition 1) and the set of rewrite rules R divided into phases using cost function C (Section 3.2), both of which are produced offline. The same cost function is used for dividing rules into phases and extraction. Given these parameters, COMPILE takes as input a scalar program P and applies multiple iterations of equality saturation using different phases of rules, rather than a single instance of equality saturation as in Diospyros.

The Compile algorithm has two key elements that solve the challenges in Section 2.3. First, it *separates phases* of compilation, preventing expensive interference between rules. Second, it *prunes* intermediate states of the E-graph to avoid redundant exploration of paths to the same program.

Phase scheduling. Compile applies the three phases of rewrite rules in separate calls to EqSat. First, at lines 5 and 6, it applies the expansion and then compilation rules in sequence. This approach is distinct from a single equality saturation on the union of the two rule sets, as it considers only rule orderings that first do expansion and then do compilation. In essence, separating the expansion and compilation phases creates an explore–exploit distinction, where we first explore as many ways to rewrite the scalar program as possible (ignoring vectorizations), and then try to exploit that search to find a vectorization without considering further scalar permutations. After these two phases complete looping (discussed next), we apply a single phase of optimization rules at line 12 to try to further improve the vectorized program.

Pruning. Even after separating equality saturation into multiple phases, compilation is still not tractable for the benchmarks we use in Section 5 because the E-graph grows too large to saturate in reasonable time. To address this explosion, we introduce a pruning loop in the Compile algorithm. The loop applies a timeout to the EqSat calls on lines 5 and 6, and then extracts (Section 2.1) a solution from the E-graph at line 7 that minimizes our cost-function C. If the extracted program improves, we repeat the loop, starting from a fresh E-graph containing only the extracted program. Eventually, this loop stops improving the program, at which point we break out and finish with optimization rules.

This loop has the effect of removing more expensive programs from the E-graph, focusing future loop iterations on a profitable path of rewrites. This pruning is greedy, sacrificing completeness by committing to a single rewrite path each time around the loop, and so loses the ability to consider other paths that might eventually reach cheaper solutions. However, this exploration can still happen *within* a single round of equality saturation. We show in Section 5.2 that pruning makes compilation dramatically more tractable with only minor impact on the optimality of compiled code.

4 Implementation

We implement the Isaria framework as an extension to the Diospyros compiler [35], targeting the Tensilica Fusion G3 digital signal processor [5]. Specifically, Isaria reuses Diospyros's front-end, which lifts imperative DSP kernels into the abstract expression language we use for rewrite rules, and its back-end, which lowers that expression language onto Fusion G3 intrinsics. Isaria replaces the manually written rewrite rules and custom rule application logic from Diospyros with the Compile algorithm in Fig. 3. Like Diospyros, Isaria uses the egg [37] library for E-graphs and equality saturation. We also use the Ruler tool [18], which also builds from egg, for synthesizing initial candidate rewrite rules.

Table 1. Lines of code for different components of Isaria, excluding comments and empty lines.

Component	LoC
ISA specification	73
Cost function	90
Offline framework	1113
Compile implementation	819
Total	2095

Table 1 shows a breakdown of the lines of code in the Isaria framework. We separate out the inputs to Isaria (the ISA specification and cost function) from the implementations of the offline (Sections 3.1 and 3.2) and compile-time (Section 3.3) parts. For our experiments in Section 5, we reuse the ISA specification (Fig. 1) and cost function for the Fusion G3 from Diospyros, but modify the cost function to more closely reflect the cost of Vec expressions when lowered to hardware.

5 Evaluation

We evaluate Isaria's effectiveness as a technique for automatically building vectorizing compilers by addressing five research questions:

1. How does code compiled by Isaria-based compilers compare to Diospyros and hand-written kernels, in terms of performance and compile times? (Section 5.1)
2. How do Isaria's rule phasing and pruning techniques affect the tractability of compilation and the performance of compiled code? (Section 5.2)
3. How does the amount of time spent in offline rule generation affect the quality of Isaria-compiled code? (Section 5.3)
4. How can Isaria help DSP engineers to explore potential ISA customizations? (Section 5.4)
5. How do values of α and β affect performance of Isaria?

Benchmarks. The benchmarks used in our evaluation are the same as those used by Diospyros [35]. They are a collection of kernels inspired by use cases in computer vision and machine perception. For 2D convolution (2DConv), matrix multiplication (Matrix Mul), and QR decomposition (QrD) benchmarks, we compile a specialized kernel for each of a range of input sizes, as most high-performance linear algebra libraries will choose among several available implementations based on size. For the quarternion product (QP) benchmark we include only a single size commonly used in pose estimation.

Methodology. To measure performance of compiled code, we report cycle counts from Tensilica's cycle-level simulator for the Fusion G3 DSP, version 2021.8, in its default memory configuration. We run rule synthesis and compilation experiments on an AWS `r5.8xlarge` EC2 instance (Intel Xeon Platinum 8259CL CPU, 16 cores/32 threads, 256GiB RAM) running Ubuntu 22.04 with Linux 5.15.

Equality saturation tools are prone to either timing out or running out of memory. In our default configuration we run Isaria's offline phase with a one day timeout, and a 220GiB memory limit. Section 5.3 evaluates the impact of these timeouts on the quality of the compiler. We also apply a 180 second timeout to individual EqSat calls at compile time; increasing this timeout by 10× had minimal effect on our results.

5.1 How does Isaria-compiled code compare to existing compilers and hand-written kernels?

Figure 4 shows the performance of code compiled by the Isaria-generated compiler for the Tensilica Fusion G3 across our benchmark suite. We report speedup over an unvectorized C++ baseline compiled by the `xt-clang` Clang compiler provided with version 2021.8 of the Xtensa toolchain for the Fusion G3. We compare Isaria to three existing tools: the same `xt-clang` compiler with auto-vectorization enabled, the hand-written *Nature* kernels provided with the Tensilica SDK, and the output of the Diospyros [35] equality-saturation-based compiler. Not all kernels have a Nature comparison as the library omits some smaller irregular sizes.

Code compiled by Isaria performs comparably to the output of Diospyros: Isaria kernels are an average of 34% faster, but this average is skewed by large kernels where Diospyros runs out of memory before finding a good vectorization. Isaria kernels are 1.0–6.9× faster than expert-written Nature kernels (mean of 3.5× and median of 4.8×). These results show that Isaria produces similar results to state-of-the-art DSP vectorization approaches despite the compiler being generated automatically from an ISA specification. High-quality automatic compiler generation makes it easy for developers to experiment with ISA extensions and quickly get an accurate picture of the potential performance improvements, as we evaluate in Section 5.4.

Since the results reported by the Diospyros paper [35], the Tensilica toolchain's Clang-based auto-vectorization appears to have improved significantly, and so we report those results separately to the Clang baseline in Figure 4. The evaluation shows that it performs well for the regular kernels of matrix multiplication and quarterion product, but is stymied by more complex kernels. For large kernels, both Diospyros and Isaria fall behind the hand-written Nature kernels, which are able to use loops rather than unrolling the kernel.

Scalability to larger kernels. To probe the limits of Isaria's scalability, we tested compiling larger 2DConv and MatMul kernels. For 2DConv we were able to compile kernels up to 30 × 30 with a 5 × 5 filter, and for MatMul up to 22 × 22, before the compiler ran out of memory. However, although Isaria could compile these largest kernels, the Tensilica cycle-level simulator ran out of memory and so we could not extract performance results. The compiled kernel was for 2DConv was 77K lines of C in a single function, and MatMul was 32K lines, because Isaria fully unrolls loops to expose parallelization opportunities. These results show that making Isaria scale to larger kernels would require the ability to reason about and emit loops without unrolling.

Compile times. Figure 5 reports the time to compile our benchmarks using both Isaria and Diospyros (we omit compile times for the other baselines as they are mostly trivial). Isaria's automation comes with an average 2.1× slowdown in compilation time. This slowdown is due to the larger size of the Isaria-generated rewrite rules compared to Diospyros. Although phasing and pruning reduce the impact of this size difference (Section 5.2), Isaria compilation time is often dominated by a small number of the calls to our equality saturation engine at lines 5 and 6 in Figure 3. On average, a compilation makes 6 calls to EqSat, but 1 or 2 of these calls take most of the compile time. Overall, while Isaria compilation times are worse than the manually written Diospyros compiler, we believe the productivity gain from an automated compiler framework is worth the trade-off.

5.2 Are rule phasing and pruning effective at making compilation tractable?

Isaria uses rule phases (Section 3.2) and compile-time pruning (Section 3.3) to make equality saturation tractable with an automatically generated set of rewrite rules. To understand the significance of these techniques, we experimented with disabling them in the Isaria workflow.

Rule phases. We removed rule phases from Isaria by replacing Compile (Fig. 3) with just a single call to equality saturation on the entire set of rules generated by Ruler. In this configuration, even our smallest benchmark quickly runs out of memory, and no benchmark successfully saturates. It is possible to use intermediate states of the E-graph to extract a solution during the search before running out of memory. However, none of these solutions for any benchmark used any vector instructions. Phases are therefore essential to making Isaria practical on even the smallest benchmarks.

Pruning. We removed E-graph pruning from Isaria by modifying Compile (Fig. 3) to retain the E-graph E across iterations of the loop at line 4, rather than creating a fresh E-graph each time. This approach effectively alternates between expansion and compilation phases without discarding the intermediate results.

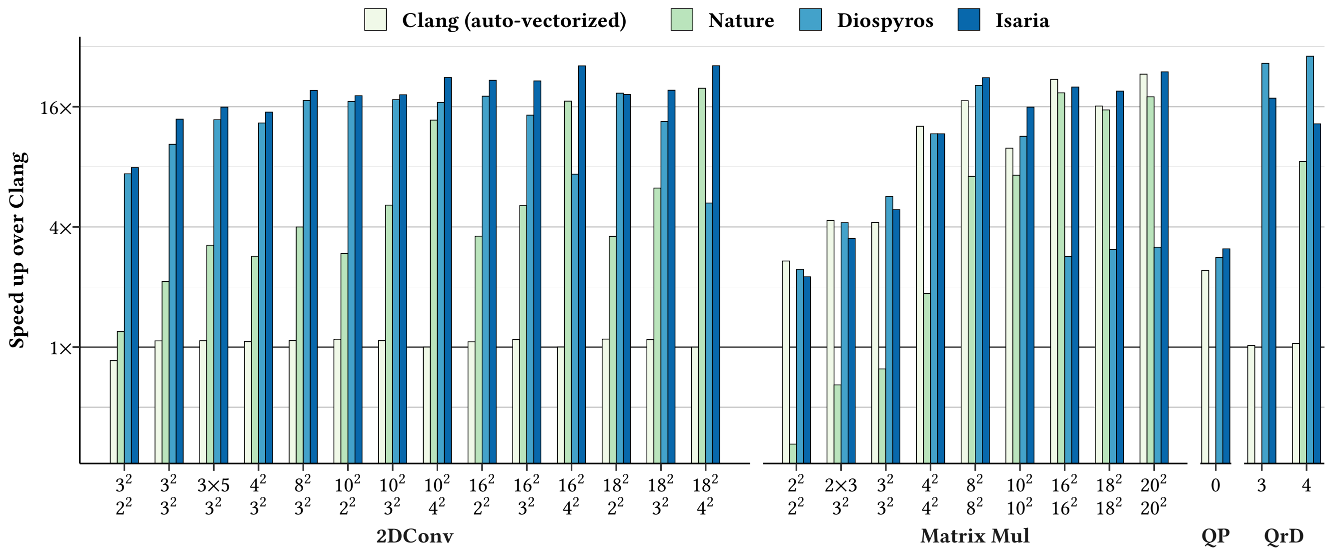

Figure 4. Performance of DSP kernels compiled by Isaria, compared to Clang auto-vectorization, the hand-written Nature kernels provided with the Tensilica SDK, and the Diospyros [35] manually written compiler. We measure performance with a cycle-level simulator, and normalize to a naive C++ loop nest implementation compiled by the Tensilica Clang-based toolchain with auto-vectorization disabled.

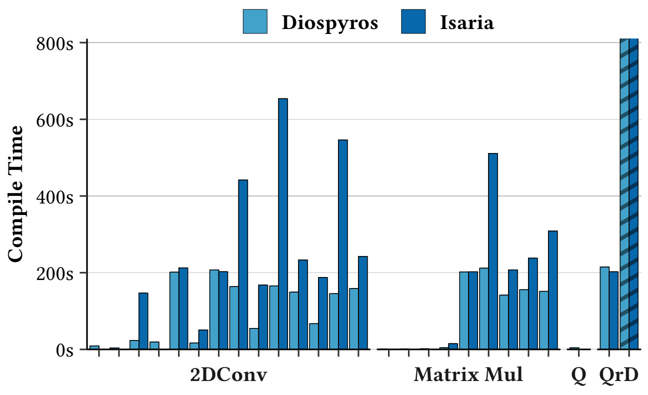

Figure 5. Compilation time for benchmarks using Diospyros and Isaria. Benchmarks are in the same order as Fig. 4. Striped bars indicate benchmarks that took longer than 800s to complete. Isaria-based compilers are slower, but the process of developing a Isaria compiler is far more automated, a worthwhile productivity trade-off.

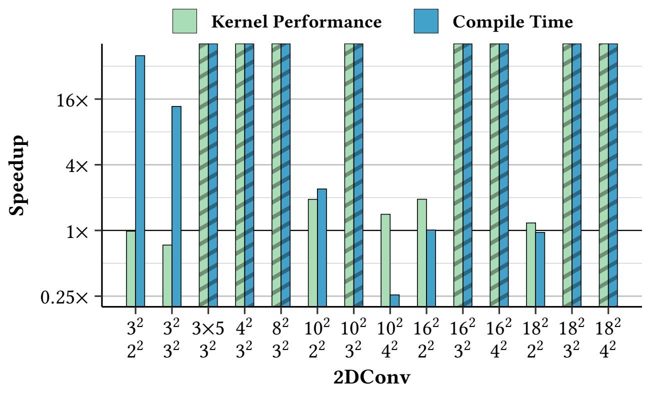

Figure 6. Improvement in kernel performance and compile time with pruning enabled (Section 3.3). Striped bars ran out of memory at compile time when pruning was disabled (no benchmarks ran out of memory with pruning enabled). Pruning gives up completeness on smaller kernels, finding slightly worse results, but enables compiling much larger kernels without running out of memory.

Figure 6 shows the improvement of enabling pruning in both kernel performance and compile time across our 2D convolution kernel benchmarks. In a majority of cases, disabling pruning causes compilation to run out of memory (indicated by stripes across the bars). For the smaller benchmarks that succeed, disabling pruning can find a slightly better kernel at the cost of more compile time. This is expected because, as described in Section 3.3, pruning gives up some completeness in exchange for much better scalability. For the larger benchmarks, pruning allows Isaria to go deeper into the search to find a faster program, because the E-graph stays smaller.

This experiment highlights an interesting difference between synthesized and hand-written rewrite rules. When inspecting these results, we found that the synthesized rules were often "shortcut" rules that combine what would have been several rules if naturally hand-written. Shortcuts are a double-edged sword: they can cause the E-graph to grow much more quickly by adding larger terms (risking running

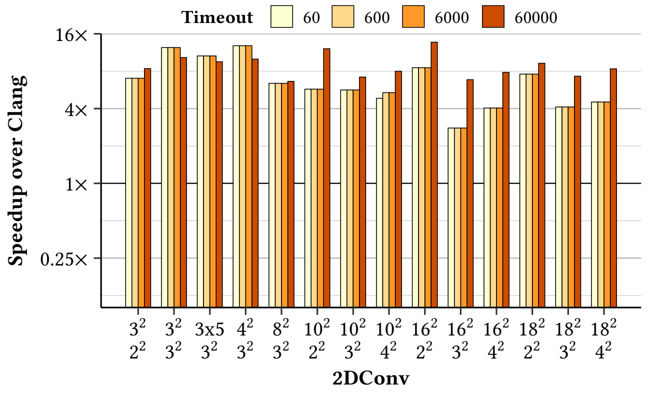

Figure 7. Impact of the timeout for rule generation on performance of kernels compiled by Isaria. Investing more time into rule generation has little impact for small kernels, although larger kernels benefit from finding more vectorization rules.

out of memory), but also allow the search to explore further in fewer iterations. Pruning helps control the downside of these rules while still enabling the upside of deeper exploration.

5.3 How does time spent in offline rule generation affect the quality of an Isaria compiler?

The Isaria workflow comprises an offline part, which synthesizes and analyzes vectorization rewrite rules (Sections 3.1 and 3.2), and a compile-time part that applies those rules (Section 3.3). Investing larger amounts of compute into the offline part can improve the quality of the generated rewrite rules by giving rule synthesis longer to explore. Since this step runs only once per instruction set, rather than once per compiled program, the cost of this additional compute can be amortized over many compiled programs.

Figure 7 shows the performance of the 2D convolution benchmarks as a function of the timeout for offline rule generation (other benchmarks follow a similar trend). In general, investing more compute in rule generation has little benefit, with up to 2.5× speedup (mean of 1.4×) by going from 60 seconds to 60,000 seconds. In fact, in a few small benchmark cases, spending more time actually reduces the quality of the compiler. These cases are dominated by VLIW scheduling effects that are not captured by our abstract cost model, and so a more accurate cost model would likely eliminate this effect. At large benchmark sizes, investing more rule generation time improves compiler quality because the exploration finds better vectorization rules for the compilation phase. We have experimented with timeouts beyond one day but did not see further improvements.

5.4 How can Isaria help DSP engineers to explore potential ISA customizations?

Isaria's automated compiler development intends to help DSP engineers explore potential customizations and changes to their instruction set. This customization is often necessary to fit embedded applications within a power and performance budget. Today, such customization is difficult, as it requires co-designing a compiler that can make effective use of the extra functionality. Isaria makes this process easier by automatically incorporating the new instructions into the compiler with high-quality vectorization.

We explored adding two new instructions to the Fusion G3 DSP as a way to illustrate this exploration process. Our focus was on accelerating the QR decomposition benchmark. First, we manually inspected the compiled code to identify potential instruction patterns that might benefit from hardening. We decided to test two new customized instructions:

1. A vectorized multiply-subtract instruction VecMulSub, like a multiply-accumuate but subtracting instead of adding.
2. A vectorized square-root-sign-product instruction VecSqrtSgn that computes $\sqrt{a} \times \text{sign}(-b)$.

These unusual operations were common in QR decomposition, and so might benefit from vectorized variants.

To add new instructions to an Isaria compiler, the DSP engineer need only add them to the ISA specification and the cost model (Fig. 2). The ISA specification changes are just a few lines of code in a Rosette interpreter; for example, for VecSqrtSgn we add scalar and vector semantics:

```
(match inst
  ...
  [(sqrt-sgn e1 e2) (* (sqrt e1) (sgn (- e2)))]
  [(vector-sqrt-sgn e1 e2)
     (for/list ([e1 v1] [e2 v2])
       (sqrt-sgn e1 e2))]
  ...)
```

The cost model changes are similarly simple, just adding a new case to the function C (Definition 1).

With these changes made, we re-ran the offline part of Isaria to synthesize a new compiler. To isolate the benefits of each instruction, we synthesized compilers for all four combinations of the two new instructions (both added, neither added, etc). This synthesis process is expensive—about a day in Isaria's default configuration—although Section 5.3 showed it can be sped up to minutes for exploration purposes with little effect on the final results.

Finally, with the new compilers, we re-compiled the QR decomposition benchmark and pushed the generated code through the Fusion G3 cycle-level simulator. To make this experiment easier, we did not add the new instructions to the (closed-source) simulator, but instead replaced each instance of them with an instruction that we expect would have the

Table 2. Speedup of QR decomposition when the Fusion G3 is customized with new VecMulSub and VecSqrtSgn instructions, normalized to the base instruction set. Each case reflects a new compiler synthesized with Isaria after adding the instruction(s) to the ISA specification and cost model.

	VecMulSub	No VecMulSub
VecSqrtSgn	2.0%	1.7%
No VecSqrtSgn	0.5%	—

same cycle latency. Table 2 shows the speedup of the kernels compiled by each of the four compilers. We see that the VecSqrtSgn instruction improves QR decomposition performance by 1.7%, while VecMulSub improves performance by only 0.5%. Adding both instructions to the DSP improves performance by 2%. Overall, these results demonstrate how Isaria helps DSP engineers experiment with architecture customizations without manually crafting compiler changes.

5.5 How sensitive is Isaria to the phase assignment of rules?

Isaria allocates automatically synthesized rewrite rules into three phases based on an analysis of their costs that depends on two parameters α and β (Section 3.2). We chose the values for these parameters manually in our experiments. Fig. 8 shows all 294 rules that Isaria generated for the Diospyros language, plotted by their aggregate cost and cost differential, along with our manually chosen values of α and β. This graph shows that the rules fall into clear clusters. Nonetheless, we would prefer to generate the phase boundaries automatically if possible.

To understand how sensitive Isaria is to these parameters and how easy they would be to choose automatically, we ran an ablation study over the parameter space for α and β. Fig. 9 shows estimated cycle count for the 2D-CONV $16^2 \times 4^2$ benchmark over a range of values for α and β. The large dark space in this graph (darker is better) suggests that although the α and β boundaries are important, they are not especially difficult to choose, with a wide space of high-quality parameters. However, there are still regions of the space to avoid. For example, in the top right corner of Fig. 9, Isaria assigns all rules to the optimization phase, which reduces to a single equality saturation that times out before finding a good vectorized program.

6 Related Work

Vectorizing compilers. The Halide scheduling language [26] has been extended to target DSPs [36], giving precise (manual) control over the scheduling of loop nests. Franchetti and Püschel [9] describe a term rewriting technique (not using equality saturation) that vectorizes small matrix kernels

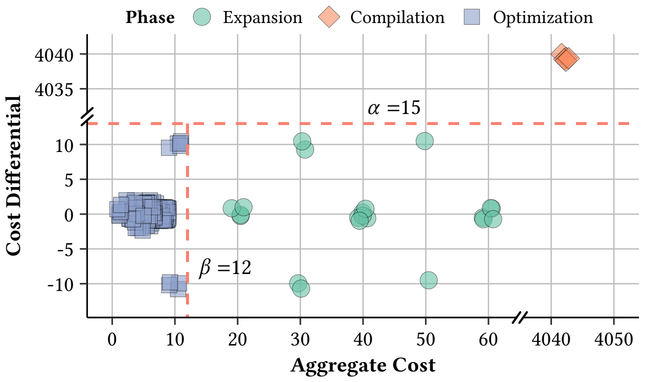

Figure 8. The 294 rules synthesized by Isaria for the Diospyros language, plotted by their aggregate cost and cost differential (Section 3.2). Points are colored by the phase Isaria assigns them to, and are jittered by 1 unit to better show their distribution.

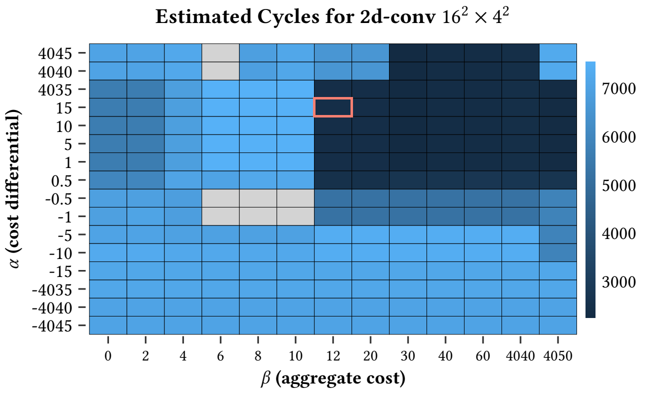

Figure 9. Estimated cycle count for a range of α, β values on the 2D-CONV $16^2 \times 4^2$ benchmark. Compilation for the light gray squares timed out after 180 seconds. The red square highlights $\alpha = 15$, $\beta = 12$.

as part of the SPIRAL project [25]. Neither of these techniques deal well with irregular kernels like the ones we target here. SLinGen [30], also part of the SPIRAL project, does focus on small kernels, using hand-crafted templates and autotuning. Diospyros [35] pioneered the technique of using equality saturation to efficiently vectorize DSP code and deal with irregular data patterns. It produces results better than expert-written specialized kernels and vendor-supplied libraries. However, as we have previously discussed, tools like SLinGen and Diospyros still place a large burden on the DSP engineer to come up with a good set of rewrite rules. Isaria extends Diospyros with automatic rule synthesis and phasing to automate the construction of DSP compilers.

Auto-vectorization techniques for general-purpose compilers perform well in certain situations. Techniques range from vectorizing loops by automatically detecting dependencies between iterations [2], to superword-level parallelism that finds parallelism at the level of basic blocks [11, 16, 22]. What unites most of these techniques is that they focus on regular kernels, and so benefit most from techniques such as polyhedral loop nest analysis [34]. In contrast, Isaria (like Diospyros) focuses on irregular or small kernels that are not well supported by off-the-shelf libraries or general-purpose auto-vectorization.

Rule synthesis. Isaria synthesizes a collection of rewrite rules to automatically discover vectorization approaches. Because term rewriting systems are ubiquitous, rewrite rule synthesis is a well studied problem. Nötzli et al. [21] synthesize rewrite rules for an SMT solver's simplification pass using enumerative syntax-guided synthesis. SWAPPER [29] synthesizes simplification rules for logic formulas from a corpus of examples. Newcomb et al. [20] synthesize rewrite rules for the Halide scheduling language [26] using a specialized synthesis pipeline. Ruler [18] is a more general-purpose tool for synthesizing rewrite rules given a specification of a term language. It uses equality saturation to improve the speed and quality of rule synthesis. Isaria uses an extended version of Ruler to generate an initial set of candidate rules, but using this set off the shelf makes equality saturation intractable, so we also introduce new techniques for mitigating this explosion. We expect these techniques would generalize to other applications of Ruler for compilation.

Rewrite-based optimizers. Outside the vectorization domain, a number of compiler optimization techniques use rewrite rules in a similar style to Isaria. STOKE [28] uses a Monte-Carlo Markov Chain (MCMC) sampler to search the space of straight-line assembly code for faster implementations of a given code fragment. It uses local rewrites like adding or deleting instructions as the proposal distribution for the MCMC search. Peephole optimizers [14, 15] are widely used in compilers, and are essentially large corpuses of rewrite rules. Souper [27] is a tool that automatically synthesizes peephole optimizations by mining LLVM code for common patterns.

7 Conclusion

Isaria is a framework for automatically generating a high-quality vectorizing compiler for digital signal processors. At the core of this automation are new insights into how to automatically synthesize rewrite rules for a new architecture, and how to schedule the application of those rules at compile time to ensure tractability. We showed that Isaria-based compilers generate high-quality code, often better than hand-written examples and competitive with state-of-the-art manually crafted auto-vectorizers. By automating compiler generation, Isaria makes it easier for DSP engineers to experiment with new instruction sets and application-specific customizations.

7.1 Future Work

We anticipate future work to improve Isaria in three ways.

Continuing to scale. Today, Isaria's scalability is bottlenecked by aggressive inlining and unrolling, which produce massive kernels. Even setting aside the effect of kernel size on equality saturation, the Diospyros backend and Xtensa toolchain struggle with our largest kernels. Two promising directions to look for scalability are to allow equality saturation to optimize looping code, and to automatically detect common sub-routines that can be factored out and optimized independently. Cranelift, an optimizing backend for WebAssembly, uses E-graphs to implement a general compiler optimization framework [8]. Extending their encoding of control flow to the vectorizing compiler domain would be an interesting direction for future work. E-graphs have also been used for library learning [6], which could be integrated into Isaria to automatically identify common patterns that would benefit from being factored out. Finally, Isaria throws away the entire E-graph for every pruning operation (line 7 in Fig. 3). We expect that we could save computation and therefore improve scalability by throwing away only "unimportant" parts of the E-graph at pruning time.

CPU architectures. We would also like to extend Isaria beyond specialized DSP processors. CPUs have long used specialized vector units to accelerate bulk computations, and in principle Isaria could generate kernels for these architectures as well. We are particularly interested in exploring compilation for ARM's Scalable Vector Extension [31] where the vector width is not fixed. We think that Isaria could help automatically explore an optimal vector width configuration for a particular program.

Better rule generation. Isaria can only generate rules that operate on vector lanes uniformly. We would like to explore extending the rule synthesis process so that we can scalably support learning more complicated rules that move data between lanes or compute across lanes. This would both expand Isaria's ability to find novel vectorizations, akin to Swizzle Inventor's ability to invent novel shuffles [24], and reduce the complexity of the Diospyros backend.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Chandrakana Nandi, for their helpful feedback on this paper. This work was supported in part by the National Science Foundation under grant 2124044 and by gifts from Amazon and Google.

References

[1] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 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*, ASPLOS '22, page 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507714. URL https://doi.org/10.1145/3503222.3507714.

[2] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4): 491–542, oct 1987. ISSN 0164-0925. doi: 10.1145/29873.29875. URL https://doi.org/10.1145/29873.29875.

[3] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14.

[4] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1):203–216, 1987.

[5] Cadence Design Systems, Inc. Tensilica customizable cores, 2020. https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[6] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, jan 2023. doi: 10.1145/3571207. URL https://doi.org/10.1145%2F3571207.

[7] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 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*, ASPLOS '21, page 902–914, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446692. URL https://doi.org/10.1145/3445814.3446692.

[8] Chris Fallin. Cranelift: Using e-graphs for verified, cooperating middle-end optimizations, 2022. https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-egraph.md.

[9] Franz Franchetti and Markus Püschel. Generating simd vectorized permutations. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, page 116–131, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787909.

[10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010. http://eigen.tuxfamily.org.

[11] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5): 145–156, may 2000. ISSN 0362-1340. doi: 10.1145/358438.349320. URL https://doi.org/10.1145/358438.349320.

[12] Samuel Larsen and Saman P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 145–156. ACM, 2000. URL https://doi.org/10.1145/349299.349320.

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.*, 49(6):216–226, jun 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594334. URL https://doi.org/10.1145/2666356.2594334.

[14] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. *SIGPLAN Not.*, 50(6):22–32, jun 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737965. URL https://doi.org/10.1145/2813885.2737965.

[15] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454030. URL https://doi.org/10.1145/3453483.3454030.

[16] Charith Mendis and Saman Amarasinghe. Goslp: Globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276480. URL https://doi.org/10.1145/3276480.

[17] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44. ACM, 2020. doi: 10.1145/3385412.3386012. URL https://doi.org/10.1145/3385412.3386012.

[18] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485496. URL https://doi.org/10.1145/3485496.

[19] Greg Nelson. *Techniques for program verification.* PhD thesis, Stanford University, 1980.

[20] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodík, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA): 166:1–166:28, 2020. doi: 10.1145/3428234. URL https://doi.org/10.1145/3428234.

[21] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. Syntax-guided rewrite rule enumeration for SMT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 279–297, 2019. doi: 10.1007/978-3-030-24258-9_20. URL https://doi.org/10.1007/978-3-030-24258-9_20.

[22] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 132–143, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933204. doi: 10.1145/1133981.1133997. URL https://doi.org/10.1145/1133981.1133997.

[23] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11, 2015. doi: 10.1145/2737924.2737959. URL https://doi.org/10.1145/2737924.2737959.

[24] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: Data movement synthesis for gpu kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 65–78, New York, NY, USA, 2019. doi: 10.1145/3297858.3304059.

[25] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in

image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530, 2013. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

[27] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017. URL http://arxiv.org/abs/1711.04422.

[28] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318709. doi: 10.1145/2451116.2451150. URL https://doi.org/10.1145/2451116.2451150.

[29] Rohit Singh and Armando Solar-Lezama. SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 185–192, 2016. doi: 10.1109/FMCAD.2016.7886678. URL https://doi.org/10.1109/FMCAD.2016.7886678.

[30] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 327–339, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168812. URL https://doi.org/10.1145/3168812.

[31] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The Arm Scalable Vector Extension. *IEEE Micro*, 37(2): 26–39, March 2017. doi: 10.1109/MM.2017.35.

[32] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *SIGPLAN Not.*, 44(1): 264–276, jan 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480915. URL https://doi.org/10.1145/1594834.1480915.

[33] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509586. URL https://doi.org/10.1145/2509578.2509586.

[34] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 327–337. IEEE Computer Society, 2009. doi: 10.1109/PACT.2009.18. URL https://doi.org/10.1109/PACT.2009.18.

[35] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 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*, ASPLOS 2021, page 874–886, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446707. URL https://doi.org/10.1145/3445814.3446707.

[36] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. Extending halide to improve software development for imaging dsps. *ACM Trans. Archit. Code Optim.*, 14(3), aug 2017. ISSN 1544-3566. doi: 10.1145/3106343. URL https://doi.org/10.1145/3106343.

[37] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434304. URL https://doi.org/10.1145/3434304.

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, jun 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL https://doi.org/10.1145/1993316.1993532.

A Artifact Appendix

A.1 Abstract

Our artifact packages the required materials to reproduce our results. There are two main components, each packaged as a container. The first is the client, available at ghcr.io/sgpthomas/isaria-aec-client:latest. This packages scripts to generate jobs, and code to make figures. The second is the experiment server, available here: ghcr.io/sgpthomas/isaria-aec:latest. The experiment server can generate new rulesets, run equality saturation compilation on a provided program, and use the XTENSA toolchain to perform cycle estimations. The containers can also be found here: https://zenodo.org/records/10058900.

A note on proprietary tools. We use the XTENSA toolchain to perform cycle estimations. This is a proprietary tool that we are not allowed to redistribute. However, they do offer free educational licenses, and there are instructions on how to setup the tools with our experiment server on our Github repository.

A.2 Artifact check-list (meta-information)

- **Program:** We use the linear algebra benchmarks distributed with Diospyros.
- **Binary:** All binaries included except the proprietary tools that we use for cycle estimation.
- **Metrics:** Cycle counts
- **How much disk space required (approximately)?:** 16 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** About an hour for the short version of experiments, and several days for the complete set of experiments.
- **Publicly available?:** Yes

A.3 Description and Installation

A.3.1 How to access. The artifact is provided as a container image available on ghcr.io. Instructions for using the container, as well as installing tools from scratch, can be found on our Github: https://github.com/sgpthomas/comp-gen/blob/main/artifact-evaluation.org.

A.3.2 Software and Hardware dependencies. We will provide the hardware and software licenses to artifact reviewers so that they can reproduce our results. Reviewers only need a Linux or Mac machine that can run DOCKER or PODMAN.

A.4 Evaluation and expected results

The evaluation process aims to reproduce all of the figures presented in our paper.

A.5 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html