DOI:10.1145/3802600

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3802603

Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors

By Samuel Thomas and James Bornholt

ABSTRACT

Embedded applications extract the best power–performance trade-off from digital signal processors (DSPs) by making extensive use of vectorized execution. Rather than handwriting 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 improves 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.

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 using simple but heavily vector-oriented architectures. Large DSP kernels can make easy use of these vector units by calling into optimized linear algebra libraries, but DSP applications are increasingly bottlenecked by a long tail of small, custom kernels not available off the shelf.[24] Optimizing these small kernels is critical for performance, but doing so often requires handwriting vectorized code that does not generalize well to new DSP architectures or make the best use of application-specific ISA customizations.

The original version of this paper was published in *Proceedings of the 29th ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*.

There are several promising approaches to automatically vectorizing these small kernels, including program synthesis,[1] pattern matching,[3] and autotuning.[20] In this paper, we focus on using *equality saturation*[21] for auto-vectorization as pioneered by Diospyros.[24] 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 by using an E-graph data structure to track program equivalences.[11,26] Diospyros then extracts the most efficient vectorized program from the E-graph using an abstract cost model.

While equality saturation achieves encouraging results—Diospyros kernels are up to 3.1× faster than off-the-shelf libraries—it places substantial burdens on the compiler writer. Crafting effective rewrite rules for equality saturation is a delicate balancing act:[10] Their congruence closure must reach interesting programs while avoiding rules that might loop infinitely or provoke combinatorial explosion. In practice, the best way to develop rewrite rules is through trial and error, but even this is difficult, as "error" means waiting for hours-long timeouts and manually examining E-graphs with millions of nodes. Because DSPs are often customized for specific applications, the rewrite rules must be tailored to the target instruction set, and so the compiler writer must repeat this tedious process for each new DSP architecture. 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,[24] 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,[22] 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.[21] But unlike existing compilers, Isaria discovers these phases automatically from an ISA specification and 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,[10] 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. This analysis includes retaining some rewrites that (temporarily) increase program cost to reach different parts of the search space.[18] 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×, the Tensilica clang-based auto-vectorizer by up to 25×, and have similar performance to Diospyros's hand-crafted compiler. Isaria vectorizes most kernels in just a few minutes, although compilation is a geometric mean 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.

2. REWRITING FOR VECTORIZATION

This section gives an overview of Diospyros's equality saturation approach to vectorization.[24] Consider vectorizing this trivial program, whose output is a four-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 first *lifts* this program to an expression using symbolic evaluation[22] to remove variables and control flow, leaving a program reflecting the output vector but written in a vector expression DSL (detailed in the original paper):

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

Here, the `Vec` term represents a four-wide vector value, where each lane can be a separate arbitrary expression. This instruction does not exist directly in the DSP hardware; each vector lane must instead be constructed separately and moved into a vector register. Isaria uses these abstract `Vec` terms to defer the details of shuffle and data movement instructions into a lowering pass after equality saturation.

There are many potential vectorizations of this program. Traditional auto-vectorization approaches like superword-level parallelism[6] would analyze the program and apply a fixed sequence of program transformations. These techniques are often effective for regular kernels but not for the smaller and irregular kernels that bottleneck embedded DSP applications. For these kernels, the best approaches to auto-vectorization[1,3,24] 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* hand-crafted 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; 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:

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

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*[21] to search the space of rewrite orderings. Rather than applying rewrite rules destructively, equality saturation applies them to an *E-graph* data structure[11] that 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 the E-graph is *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 selecting the program with the lowest cost according to an appropriate cost model.

2.1. Synthesizing rewrite rules.

Diospyros uses a hand-crafted set of rewrite rules for a specific DSP architecture. In principle, the manual effort of building Diospyros could be automated with a general-purpose tool for synthesizing rewrite rules such as Ruler.[10] However, crafting rewrite rules for compilation requires striking a delicate balance between discovering novel vectorizations and being so general that saturation becomes intractable. In practice, these synthesizers produce rules that cannot achieve this balance. For example, we applied Ruler directly to Diospyros's vector language and synthesized 300 rewrite rules. Trying to use these rules to vectorize a small 2D matrix convolution causes equality saturation to exhaust 64GB of memory without producing any results. Even after manual adjustments to rein in the size of the E-graph, these rules still fail to find any vectorized program after an hour of searching. In contrast, Isaria finds a fast vectorized solution in only three seconds while using 0.2GB of memory.

To illustrate the problem, suppose the rewrite synthesis tool generates three rules about vector addition (using 2-wide vector instructions for clarity):

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

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

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

Consider applying these rules to vectorize a small program that sums the four elements of each 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 blow up the size of the E-graph, preventing saturation within reasonable time and therefore preventing vectorization.

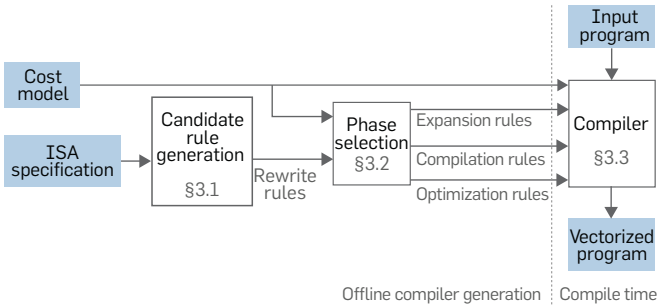
This example highlights two problems with applying a rule synthesizer to compilation. First, these rules can *go backward* during search: Because the tools do not understand the goal of the compiler (in this case, to produce a vectorized version of a scalar program), they synthesize rules like rule 2 that can undo vectorization, yielding a permutation of the unvectorized input program. In fact, equality saturation guarantees the search will eventually explore every reachable permutation of the unvectorized input program. These variants are helpful to expose future vectorization opportunities by, for example, commuting two terms as rules 2 and 3 do. 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 E-graph explosion by carefully controlling which rules are applied and when. Diospyros does this by hand, with manual heuristics to apply certain rewrite rules only at particular points during the search. With Isaria, we aim to automate this 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. Figure 1 shows an overview of the workflow. Isaria takes the same inputs as Diospyros: the semantics of the target instruction set (i.e., an interpreter) and an abstract cost model for programs in that instruction set. 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. The key insight to this workflow is that not all rewrite rules are equally useful for vectorization. Rather than trying to compute the congruence closure of a prohibitively

Figure 1. 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.



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.[10] 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.

For Isaria, the target language is derived from 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 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, from 7,735 to 300 rules in the example in Section 2. However, even this filtered rule set is an order of magnitude larger than Diospyros's 28 handwritten rules for the same DSP and so explodes the size of the E-graph when used as is.

Vector lane generalization. Ruler struggles to learn effective rewrite rules for vectorized lane-wise instructions like `VecAdd` because it does not know these instructions operate independently on each lane. As a result, Ruler spends most of its time rediscovering associativity and commutativity for each lane and each combination of lanes, making little progress toward discovering interesting vectorization rules. To avoid this redundant work, Isaria extends Ruler to *generalize* synthesized rules across vector lanes. At rule synthesis time, Isaria reduces vector instructions to a single lane, so that Ruler need only discover per-lane behavior a single time. 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 with fresh wildcards. To mitigate the risk of unsoundness from this generalization, Isaria uses Rosette and the ISA specification to verify the soundness of the generalized rules.

3.2. Discovering rule phases.

While Ruler successfully synthesizes candidate rewrite rules, its rule sets are too large to use directly in equality saturation. To make vectorization with these rules tractable, Isaria analyzes and categorizes the candidate rules produced by Ruler. The observation behind this analysis is that rewriting-based vectorization must *gradually* transform a fully scalar program into a fully vectorized one, as an individual rewrite rule is too small to vectorize the entire program in a single step. As these transformations progress, scalar-to-scalar rewrites become less useful, since there are fewer scalar parts of the program, while vector-to-vector rewrites become more useful.

Traditional equality saturation has no way to take advantage of this progression and will continue trying to apply 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 rewrite rules applied to the graph so far; so the node for a vectorized term is still eligible to have more scalar rewrites applied to it. In theory, these additional rewrites are desirable, as they may expose a better vectorization later in the saturation process. But empirically, this outcome is rare—once a term has a vectorized form, it generally profits only from vector rewrites. We can therefore make equality saturation more tractable by preventing this *interference* between scalar and vector rules.

Three rule phases. To realize 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. In the example in Section 2, we can arrange for the three synthesized rules to be in different phases, so they are not saturated together. Note that the names we give these phases have no semantic meaning but are just 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 is a function $C(P)$ that assigns a natural number cost to every program in the target language (the DSP instruction set), representing 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. 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.

Given a cost function, Isaria's phase-selection step assigns each synthesized rewrite rule to one of the three rule phases by analyzing the cost of both sides of the rule. This analysis uses the cost model $C(P)$ to compute two metrics about a candidate rewrite rule $P \rightsquigarrow Q$:

- The *cost differential* $C_D(P \rightsquigarrow Q)$ is $C(P) - C(Q)$.
- The *aggregate cost* $C_A(P \rightsquigarrow Q)$ is $C(P) + C(Q)$.

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

1. Rules with large cost differential $C_D(P \rightsquigarrow Q) > \alpha$ are *compilation* rules, because they 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:

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

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. The aggregate cost allows Isaria to distinguish those two cases—scalar programs are more expensive and so have higher aggregate cost—and assign them to the expansion or optimization phases, respectively.

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. The compiler takes as input a scalar program and vectorizes it by applying the rules using equality saturation. This application process takes advantage of the phases inferred in Section 3.2 to make equality saturation tractable.

a The original paper evaluates the impact of selecting appropriate values for these hyperparameters.

Figure 2. The Isaria compilation algorithm takes as input a scalar program *P* and returns a compiled program.

```
1:  function Compile(P, R)
2:    C_Old ← Cost(P)
3:    loop
4:      E ← EGraph(P)
5:      E ← EqSat(E, R_Expansion)
6:      E ← EqSat(E, R_Compilation)
7:      P, C_New ← Extract(E, Cost)
8:      if C_New = C_Old then
9:        break
10:     C_Old ← C_New
11:     E ← EGraph(P)
12:     E ← EqSat(E, R_Optimization)
13:     P, _ ← Extract(E, Cost)
14:   return P
```

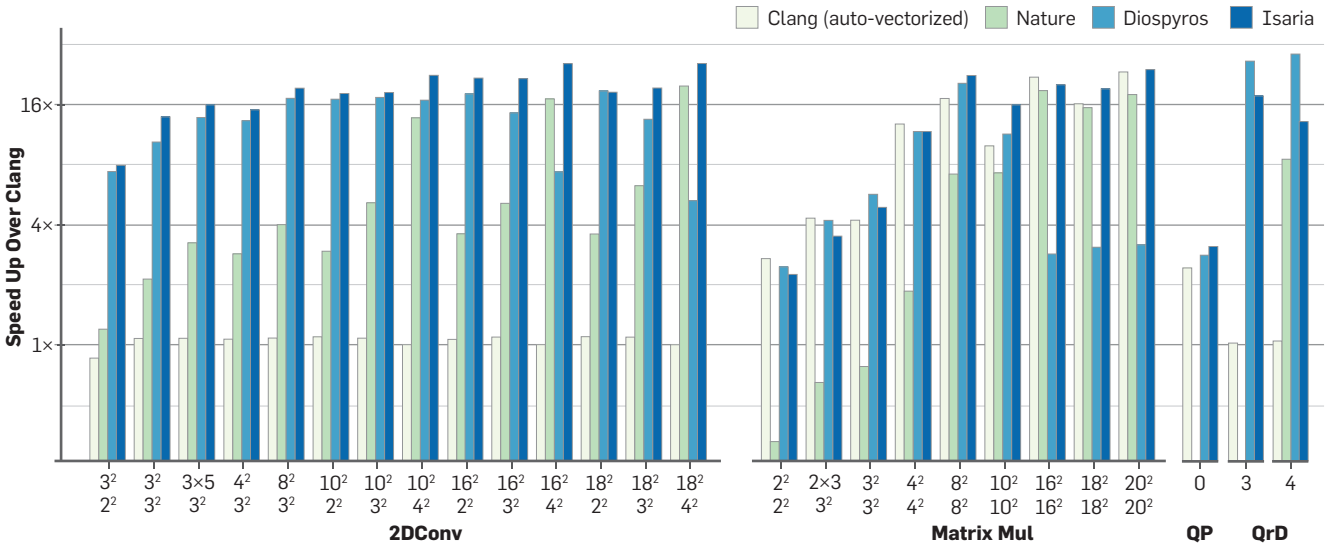
Figure 2 shows the Isaria compilation algorithm. An Isaria compiler is parameterized by a cost function C and the set of rewrite rules R divided into phases, both of which are produced offline. The same cost function is used for both dividing rules into phases and extraction. Given these parameters, 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. 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 not the same as 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. Separating the expansion and compilation phases first explores as many ways to rewrite the scalar program as possible without any vectorization, and then exploits 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 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 4 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 a solution from the E-graph at line 7 that minimizes the 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, sacri-

Figure 3. 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 manually written compiler.



ficing 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 4.2 that pruning makes compilation dramatically more tractable with only minor impact on the optimality of compiled code.

4. EVALUATION

We evaluate Isaria's effectiveness for building vectorizing compilers by addressing three 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 4.1)
2. How do Isaria's rule phasing and pruning techniques affect the tractability of compilation and the performance of compiled code? (Section 4.2)
3. How can Isaria help DSP engineers to explore potential ISA customizations? (Section 4.3)

We implement the Isaria framework as an extension to the Diospyros compiler,[24] targeting the Cadence Tensilica Fusion G3 DSP. Like Diospyros, Isaria uses the egg[26] library for E-graphs and equality saturation. We also use Ruler[10] for synthesizing initial candidate rewrite rules.

Benchmarks and methodology. The benchmarks in our evaluation are a collection of kernels from 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.

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. Equality saturation tools are prone to either timing out or running out of memory. We run Isaria's offline phase with a one-day timeout and a 220GB memory limit. In the original paper, we also evaluate with much shorter timeouts and see little impact on the quality of the generated compiler.

4.1. Compiler performance.

Figure 3 shows the performance of code compiled by the Isaria-generated compiler across our benchmark suite. We report speed-up over an unvectorized C++ baseline compiled with the vendor-supplied Clang compiler for the Fusion G3. We compare Isaria to three existing tools: the same Clang compiler with auto-vectorization enabled, the hand-written Nature kernels provided with the Tensilica SDK, and the output of the Diospyros[24] 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 a geometric mean of 34% faster, although this is skewed by large kernels where Diospyros compilation runs out of memory. Isaria kernels are 1.0–6.9× faster than expert-written Nature kernels (geometric mean of 3.5× and median of 4.8×). These results show that Isaria produces results similar to state-of-the-art DSP vectorization approaches and better-than-optimized off-the-shelf linear algebra libraries.

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, 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. Making Isaria scale to

larger kernels would require the ability to reason about and emit loops without unrolling.

Compile times. Isaria's automation slows compilation by a geometric mean of 2.1× compared to Diospyros. This slowdown is due to the larger size of the Isaria-generated rewrite rule set. Although phasing and pruning reduce the impact of this size difference (Section 4.2), Isaria makes an average of six calls to equality saturation per compilation (Figure 2), compared to Diospyros's single call.

4.2. Rule phasing and pruning.

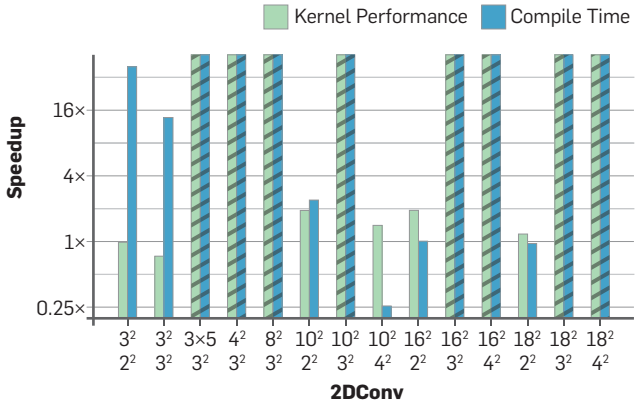
To understand the significance of Isaria's rule phasing and pruning techniques, we experiment with disabling each of them.

We removed rule phases from Isaria by replacing COMPILE (Fig. 2) 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, but none of these solutions for any benchmark used any vector instructions.

We removed E-graph pruning from Isaria by modifying COMPILE to retain the E-graph E across iterations of the loop at line 4, rather than creating a fresh E-graph each time. Figure 4 shows the impact of disabling pruning on 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. 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 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.

When inspecting these results, we found that synthesized rules were often *shortcut* rules that combine what would have been several rules if naturally handwritten. Shortcuts are a double-edged sword: They can cause the E-graph to grow much more quickly by adding larger terms (risking running 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.

Figure 4. 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.



4.3. Case study: Exploring new instructions.

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. We explored adding two new instructions to the Fusion G3 DSP to illustrate this exploration process. Our focus was on accelerating the QR decomposition benchmark. First, we manually inspected the compiled code to identify common 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-accumulate but subtracting instead of adding.
2. A vectorized square-root-sign-product instruction `VecSqrtSgn` that computes $\sqrt{a} \times \text{sign}(-b)$.

To add new instructions to an Isaria compiler, the DSP engineer adds them to the ISA specification and the cost model (Figure 1). 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. 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).

Finally, with the new compilers, we re-compiled the QR decomposition benchmark. The table shows the speed-up of the kernels compiled by each of the four compilers. 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%. These results demonstrate how Isaria helps DSP engineers experiment with architecture customizations without manually crafting compiler changes.

Table. Speed-up of QR decomposition when the Fusion G3 is customized with new `VecMulSub` and `VecSqrtSgn` instructions, normalized to the base instruction set.

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

5. RELATED WORK

Vectorizing compilers. The Halide scheduling language[16] has been extended to target DSPs,[25] giving precise (manual) control over the scheduling of loop nests. Franchetti and Püschel[4] describe a term-rewriting technique (not using equality saturation) that vectorizes small matrix kernels as part of the SPIRAL project.[15] Neither of these techniques deal well with irregular kernels like the ones we target here. SLinGen,[20] also part of the SPIRAL project, does focus on small kernels, using handcrafted templates and autotuning. Diospyros[24] pioneered the technique of using equality saturation to efficiently vectorize DSP code and deal with irregular data patterns. It produces better results 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.[5,9,14] 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.[23] 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.[13] synthesize rewrite rules for an SMT solver's simplification pass using enumerative syntax-guided synthesis. SWAPPER[19] synthesizes simplification rules for logic formulas from a corpus of examples. Newcomb et al.[12] synthesize rewrite rules for the Halide scheduling language using a specialized synthesis pipeline. Ruler[10] 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[18] 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[7,8] are widely used in compilers and are essentially large corpuses of rewrite rules. Souper[17] is a tool that automatically synthesizes peephole optimizations by mining LLVM code for common patterns.

6. CONCLUSION

Isaria is a framework for automatically generating a high-quality vectorizing compiler for DSPs. 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 handwritten 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.

ACKNOWLEDGMENTS

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, Cadence, and Google. C

References

1. Ahmad, M.B.S. et al. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery (2022), 1004–1016.
2. Allen, R. and Kennedy, K. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst. 9*, 4 (Oct. 1987), 491–542.
3. Chen, Y., Mendis, C., Carbin, M., and Amarasinghe, S. Vegen: A vectorizer generator for SIMD and beyond. In *Proceedings of the 26th ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery (2021), 902–914.
4. Franchetti, F. and Püschel, M. Generating simd vectorized permutations. Generating SIMD vectorized permutations. In *Proceedings of the Joint European Confs. on Theory and Practice of Software 17th Intern. Conf. on Compiler Construction*, Springer-Verlag (2008), 116–131.
5. Larsen, S. and Amarasinghe, S. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not. 35*, 5 (May 2000), 145–156.
6. Larsen, S. and Amarasinghe, S.P. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM (June 2000), 145–156.
7. Lopes, N.P. et al. Alive2: Bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN Intern. Conf. on Programming Language Design and Implementation*, Association for Computing Machinery (2021), 65–79.
8. Lopes, N.P., Menendez, D., Nagarakatte, S., and Regehr, J. Provably correct peephole optimizations with Alive. *SIGPLAN Not. 50*, 6 (June 2015), 22–32.
9. Mendis, C. and Amarasinghe, S. Goslp: Globally optimized superword level parallelism framework. In *Proceedings of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications 2*, (Oct. 2018).
10. Nandi, C. et al. Rewrite rule inference using equality saturation. In *Proceedings of the ACM on Programming Languages 5* (Oct. 2021).
11. Nelson, G. *Techniques for Program Verification*. Ph.D. thesis, Stanford University (1980).
12. Newcomb, J.L. et al. Verifying and improving Halide's term rewriting system with program synthesis. In *Proceedings of the Conf. on Object-Oriented Programming, Systems, Languages, and Application. 4*, (2020), 166:1–166:28.
13. Nötzli, A. et al. Syntax-guided rewrite rule enumeration for SMT solvers. In *Proceedings of the 22nd Intern. Conf on Theory and Applications of Satisfiability Testing–Vol. 11628 of Lecture Notes in Computer Science* (2019), 279–297.
14. Nuzman, D., Rosen, I., and Zaks, A. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Association for Computing Machinery (2006), 132–143.
15. Puschel, M. et al. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE 93*, 2 (2005), 232–275.
16. Ragan-Kelley, J. et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*

(June 2013), 519–530.
17. Sasnauskas, R. et al. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422 (2017).
18. Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *Proceedings of the 18th Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery (2013), 305–316.
19. Singh, R. and Solar-Lezama, A. SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *Proceedings of the 2016 Conf. on Formal Methods in Computer-Aided Design*, R. Piskac and M. Talupur, eds. (Oct. 2016), 185–192.
20. Spampinato, D.G., Fabregat-Traver, D., Bientinesi, P., and Püschel, M. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 Intern. Symp. on Code Generation and Optimization*, Association for Computing Machinery (2018), 327–339.
21. Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. Equality saturation: A new approach to optimization. *SIGPLAN Not. 44*, 1 (Jan. 2009), 264–276.
22. Torlak, E. and Bodik, R. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM Intern. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software*, Association for Computing Machinery (2013), 135–152.
23. Trifunovic, K. et al. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 18th Intern. Conf. on Parallel Architectures and Compilation Techniques*, IEEE Computer Society (Sept. 2009), 327–337.
24. VanHattum, A. et al. Vectorization for digital signal processors via equality saturation. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM Intern. Conf. on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery (2021), 874–886.
25. Vocke, S. et al. Extending Halide to improve software development for imaging DSPs. *ACM Trans. Archit. Code Optim. 14*, 3 (Aug. 2017).
26. Willsey, M. et al. Egg: Fast and extensible equality saturation. In *Proceedings of the Symp. on Principles of Programming Languages 5* (Jan. 2021).

Samuel Thomas is a graduate research assistant at the University of Texas at Austin, Austin, TX, USA.

James Bornholt (bornholt@cs.utexas.edu) is a senior principal engineer on the S3 team at Amazon Web Services, Seattle, WA, USA.

This work is licensed under a Creative Commons Attribution International 4.0 License.
© 2026 Copyright held by the owner/author(s).

Advertise with ACM!

Reach the innovators and thought leaders working at the cutting edge of computing and information technology through ACM's magazines, websites and newsletters.

Request a media kit with specifications and pricing:

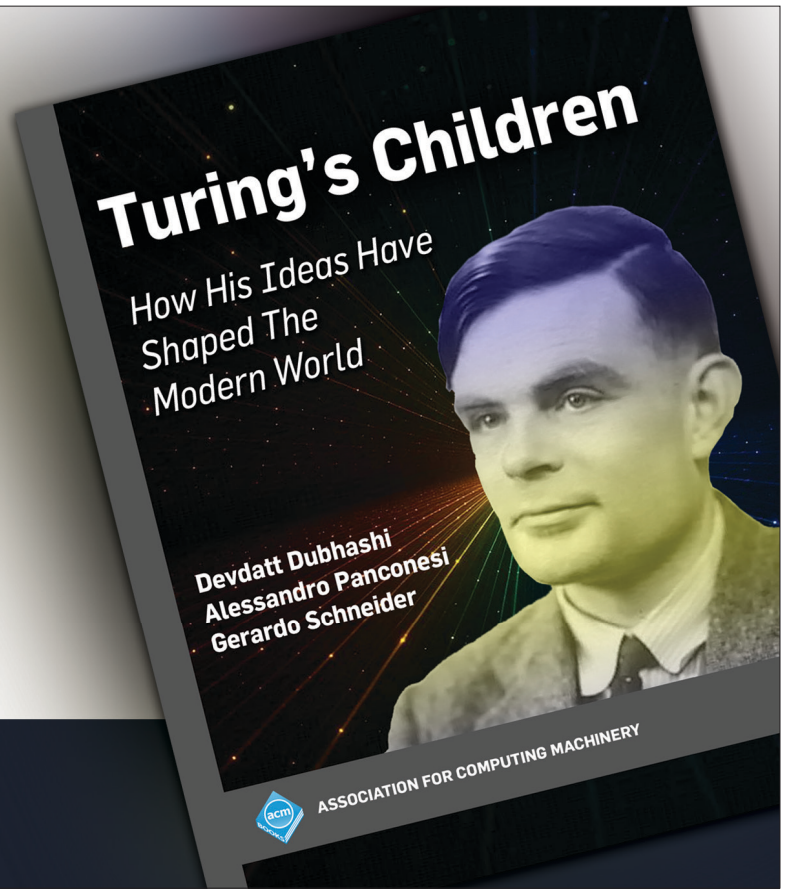
Ilia Rodriguez
+1 212-626-0686
acmmediasales@acm.org

Turing's Children
How His Ideas Have Shaped the Modern World

Devdatt Dubhashi
Alessandro Panconesi
Gerardo Schneider

ISBN: 979-8-4007-3177-8
DOI: 10.1145/3731975

http://books.acm.org



ACM BOOKS
Collection III