SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting

Jianyi Cheng[1], Samuel Coward[1,2], Lorenzo Chelini[1], Rafael Barbalho[1] and Theo Drane[1]
[1]*Intel Corporation, USA;* [2]*Imperial College London, UK*
jianyi.cheng@cl.cam.ac.uk, {samuel.coward, lorenzo.chelini, rafael.barbalho, theo.drane}@intel.com

Abstract

High-level synthesis (HLS) is a process that automatically translates a software program in a high-level language into a low-level hardware description. However, the hardware designs produced by HLS tools still suffer from a significant performance gap compared to manual implementations. This is because the input HLS programs must still be written using hardware design principles.

Existing techniques either leave the program source unchanged or perform a fixed sequence of source transformation passes, potentially missing opportunities to find the optimal design. We propose a super-optimization approach for HLS that automatically rewrites an arbitrary software program into efficient HLS code that can be used to generate an optimized hardware design. We developed a toolflow named SEER, based on the e-graph data structure, to efficiently explore equivalent implementations of a program at scale. SEER provides an extensible framework, orchestrating existing software compiler passes and hardware synthesis optimizers.

Our work is the first attempt to exploit e-graph rewriting for large software compiler frameworks, such as MLIR. Across a set of open-source benchmarks, we show that SEER achieves up to 38× the performance within 1.4× the area of the original program. Via an Intel-provided case study, SEER demonstrates the potential to outperform manually optimized designs produced by hardware experts.

1 Introduction

High-level synthesis (HLS) is a process that automatically translates a software program in a high-level language such as C/C++ into a hardware description in a low-level language such as Verilog/VHDL. This allows software engineers without any hardware background to customize their hardware accelerators. Today, HLS tools have been widely used and actively developed in both academia and industry, for example, Dynamatic [23] from EPFL, Bambu [5] from the Politecnico di Milano, Stratus HLS [48] from Cadence, Catapult HLS [6] from Siemens, Intel HLS [22] from Intel and Vitis HLS [56] from AMD Xilinx.

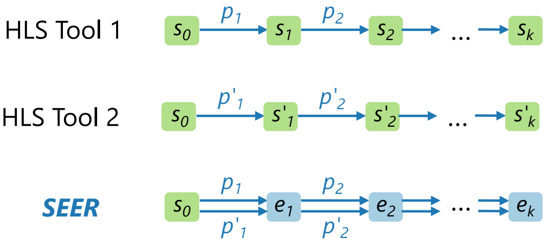


Figure 1. p_i denotes an optimization pass, and s_i denotes a representation of a program. HLS Tools 1 and 2 take the same input program s_0 and apply different sequences of optimization passes, p_i and p'_i respectively. The transformed programs s_k and s'_k may result in different hardware designs because of the difference in pass sequences. SEER efficiently explores all these possibilities in parallel using e-graphs, e_i.

Still, it remains the case that automatically synthesizing efficient hardware designs from arbitrary high-level software programs is challenging. A major reason is that each HLS tool only applies a fixed sequence of general source transformations for all input programs, as shown in Figure 1. This significantly restricts the optimization space for a particular program. This is known as the ***'phase-ordering problem'*** in compilers [34].

The phase-ordering problem for HLS tools is more challenging for two reasons. First, an HLS tool contains optimizations at ***different granularities***, such as higher-level control path optimizations and lower-level data path optimizations. These optimizations may interfere, resulting in a larger, more complex space of optimization orderings than in software compilers. Second, evaluating ***hardware metrics*** from an input software program is challenging in existing frameworks. This means that the optimizer needs to repeatedly call the downstream synthesis tool to evaluate which source representation is efficient when mapping into hardware.

Existing works on HLS source rewriting build an optimization sequence based on heuristics. This misses opportunities to perform program-specific optimizations for a given input program, potentially making the optimal hardware design unreachable. In practice, significant manual effort is spent on rewriting the program source for HLS tools to resolve the problem above. Both Stratus HLS [48] and Vitis HLS [56], provide coding guidelines to restrict users to a subset of C programs for better performance. A designer must write the

```
int x[200], y[200];

loop_1:
for (int i=0; i<100; i++)
  x[i+1] = f(x[i]);

loop_2:
for (int i=0; i<100; i++)
  y[i] = g(y[i]);

loop_3:
for (int i=0; i<100; i++)
  x[i+2] = h(x[i]);
```

Listing 1. Baseline code

=

```
int x[200], y[200];

// Fuse loop_1 and loop_2
loop_1_2:
for (int i=0; i<100; i++)
    {
  x[i+1] = f(x[i]);
  y[i]   = g(y[i]);
}


loop_3:
for (int i=0; i<100; i++)
  x[i+2] = h(x[i]);
```

Listing 2. Transform 1

=

```
int x[200], y[200];

loop_1:
for (int i=0; i<100; i++)
  x[i+1] = f(x[i]);


// Fuse loop_2 and loop_3
loop_2_3:
for (int i=0; i<100; i++)
    {
  y[i]   = g(y[i]);
  x[i+2] = h(x[i]);
}
```

Listing 3. Transform 2

Figure 2. A motivating example of loop fusion. `loop_1` and `loop_3` cannot be fused because of the memory dependence on array `x`. It is challenging to determine which representation is better, fusing `loop_1` and `loop_2` or fusing `loop_2` and `loop_3`?

Table 1. The performance of hardware generated from the representations in Figure 2 depends on the operation latencies. These could be affected by other transformation passes and are not evaluated in the existing flow. The best performance in each case is **highlighted**.

	f	g	h	Listing 1	Listing 2	Listing 3
Case 1	10	100	1	1308	**1196**	1205
Case 2	1	100	10	813	710	**701**

HLS program following these guidelines and using hardware design principles in order to produce efficient hardware.

In order to tackle the problems above, our work aims to solve the following challenges:

1) Efficiency: How should one efficiently explore the vast space of possible optimization sequences?
2) Hardware awareness: How should one pick a sequence of optimizations that can be mapped into efficient hardware based on the program source?

We propose an approach named SEER (**S**uper-optimization **E**xplorer using **E**-graph **R**ewriting) to resolve the challenges above. *Given a software program, SEER automatically determines a sequence of optimizations for efficient hardware synthesis.* SEER is the first approach to HLS ‘super-optimization’, since it explores different source-level optimization orderings in parallel, then customizes the sequence to the input program.

SEER enables super-optimization using an efficient data structure, known as an e(quivalence)-graph [7], which preserves a set of program representations to resolve the phase-ordering problem. As shown in Figure 1, SEER can explore alternative optimization sequences in a single e-graph at the same time. Our main research contributions include:

- a technique to determine an optimization order for efficient hardware synthesis by exploring equivalent representations of a program in an e-graph;
- an orchestration technique that explores an e-graph with existing optimization passes from large software frameworks, such as MLIR [33], and hardware synthesis optimizers, such as ROVER [10], to explore rewriting at scale;
- a hardware-aware evaluation model at the source level to evaluate the quality of hardware synthesized from a representation of a software program; and
- over a set of benchmarks, SEER achieves up to 38× the performance within 1.4× the area of the original program, and demonstrates the potential to outperform manually optimized designs by hardware experts.

The rest of the paper is organized as follows. Section 2 presents a motivating example to illustrate the challenge in automated source rewriting for HLS. Section 3 provides the necessary background. Section 4 explains the theoretical details of our work. Section 5 evaluates the effectiveness of our work by comparing it with a commercial HLS tool when passed the baseline pragma-free source code and when given human guidance via HLS pragmas. Finally, in Section 6 we discuss related work.

2 Motivating Example

Using an example, we present the challenge to conventional, fixed pass-order HLS flows and how our approach can overcome these challenges. Listing 1 presents a program with

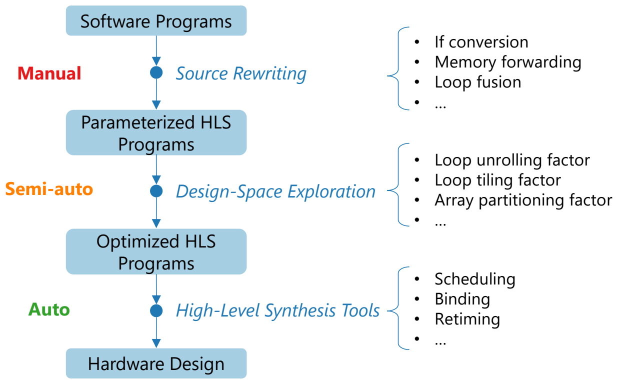


Figure 3. HLS development flow for hardware production. The right side provides examples of optimizations for each step. SEER aims to solve the challenge in efficient source rewriting for arbitrary programs (shown as **Manual**) for better hardware performance.

three sequential loops. Loop fusion is an optimization technique that combines multiple sequential loops into a single loop. In HLS, loop fusion avoids the area overhead of the loop control logic for separate loop instances and could exploit more data parallelism in the fused loop body. However, the throughput of a loop is restricted by the slowest data path in the loop body. Loop fusion can exhibit an area-performance tradeoff.

A pre-condition for loop fusion being valid is that the sequential loops must have no data dependence. For the example in Listing 1, `loop_1` and `loop_3` access array `x` at overlapping indices, preventing loop fusion. However, `loop_2` has no data dependence with either `loop_1` or `loop_3`, since it only accesses array `y`. This means that we can safely fuse `loop_1` and `loop_2` (Listing 2) or fuse `loop_2` and `loop_3` (Listing 3). Note that the user or automated tool must choose between these fusion passes, since Listing 2 is not reachable from Listing 3, and vice versa.

Without evaluating downstream hardware optimization passes, it is difficult to determine whether Listing 2 or 3 will generate better hardware. Table 1 shows the performance of the hardware generated from these representations for different latencies of the functions `f` and `h`. Such latency information is unpredictable at the source rewriting stage because later passes might alter these functions. This correlation could make a locally sub-optimal transformation globally optimal. SEER models hardware scheduling information in software and efficiently explores transformations of these representations in an e-graph instead of manipulating a single representation.

Problem Formalization

A key novelty of our work is that SEER explores the correlation among transformation passes, which opens up a larger design space. Let P be a set of available transformation passes, $P^{\mathbb{N}}$ be all possible sequences of an arbitrary length of the elements in P and let R be the set of functionally equivalent representations of a given program. As shown in Figure 1, an HLS tool uses a fixed sequence of passes $t = (p_0, p_1, ..., p_k) \in P^{\mathbb{N}}$. The transformation steps in t result in a set of representations R', where $R' \subseteq R$. SEER searches the space of pass sequences $P^{\mathbb{N}}$ and extracts a customized $t' \in P^{\mathbb{N}}$ for each input program. SEER can explore a potentially larger set of representations R'', where $R' \subseteq R'' \subseteq R$. This is because SEER searches for t' by exploring $P^{\mathbb{N}}$ in parallel. In the rest of the paper, we show how to construct R'' using an e-graph and how to determine t' for mapping an arbitrary program to efficient hardware.

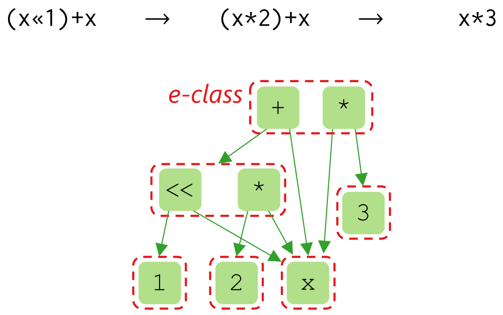


Figure 4. An e-graph grown from two rewriting steps to represent three equivalent expressions. Each green node is an e-node, and each red box is an e-class. Edges connect e-nodes to child e-classes.

3 Background

3.1 Phase-Ordering Challenges in High-Level Synthesis

HLS tools automatically map a high-level software program into a custom hardware design in a low-level hardware description, *e.g.* Verilog. A production HLS development flow comprises three steps, as shown in Figure 3. First, a high-level specification of an algorithm is *manually* rewritten following the recommended coding guidelines producing code that is amenable to optimization by the HLS tool. Second, the rewritten HLS program usually contains design constraints expressed via inline directives or pragmas to exploit hardware parallelism and resource sharing. The process of exploring these constraints is known as design-space exploration (DSE) [44] and is already semi-automated [17, 26, 35]. Finally, the optimized design constraints are sent with the HLS program to the HLS tool, which synthesizes a hardware design. The HLS tool automatically performs low-level hardware optimizations, such as hardware scheduling and binding, which maps the start times of operations into clock cycles with efficient hardware resource sharing [4, 20, 27, 61].

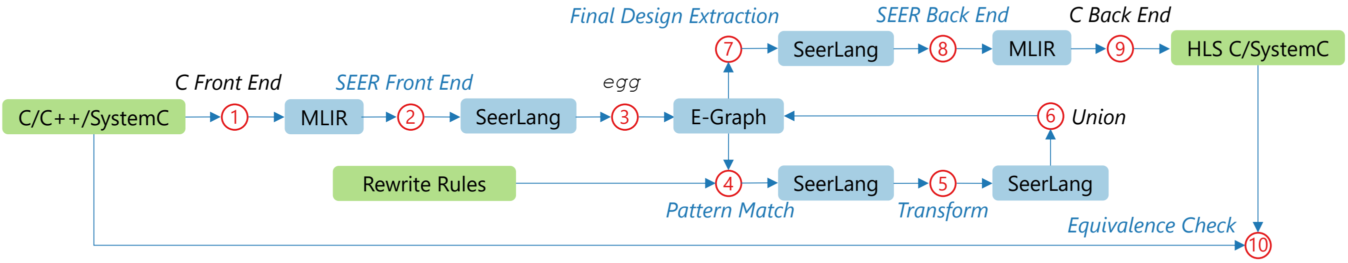


Figure 5. An overview of the SEER toolflow. Our contributions are *highlighted*.

The HLS tool also performs register retiming to achieve a high clock frequency [20, 27].

The phase-ordering problem refers to the challenge of determining the optimal order of optimization passes at compile time. It is challenging due to the destructive interaction of optimization passes, as discussed in Section 2. In HLS, the phase-ordering problem is more complex, because the benefit of software transformation passes, such as loop fusion and if conversion, can only be evaluated by analyzing the generated hardware. In this work, SEER orchestrates software transformations for hardware optimization using hardware modeling. To the best of our knowledge, SEER is the first attempt to resolve the general phase-ordering problem in HLS.

3.2 E-graph Representation

An e(quivalence)-graph, is a data structure used to represent a set of equivalent expressions [39, 50, 55] as shown in Figure 4. The e-graph organizes functionally equivalent expressions into equivalence classes, known as e-classes, drawn as red boxes in the figure. Nodes in an e-graph, known as e-nodes, represent either values or operators with edges connecting e-class children, illustrated as green nodes in the figure. E-classes are represented as groups of e-nodes. The e-graph is grown via constructive rewriting, meaning that the left-hand side of the rewrite is retained in the data structure. A minimal cost expression is typically extracted from the e-graph, based on a user-defined cost model.

A main benefit of the e-graph data structure is that it efficiently represents equivalent expressions by sharing and reusing common sub-expressions, such as sharing x in Figure 4. Operator e-nodes have edges connected to child e-classes. This captures the intuition that, for a given sub-expression, we can choose from a set of equivalent sub-expressions. The reduced redundancy in the e-graph enables more efficient analysis and optimization of a program.

E-graphs can be found in modern SMT solvers, such as Z3 [15, 16]. The recently developed egg library [55], provides an extensible e-graph implementation. One relevant work [12], identified improvements to program analysis capabilities by using the e-graph representation. We describe how SEER exploits this in Section 4.5. SEER is the first approach to apply e-graphs to program optimization using compiler frameworks such as MLIR.

In this work, we incorporate and extend an existing egg based data path optimization engine, named ROVER [10, 11]. ROVER takes a combinational hardware design and optimizes the data paths using optimizations for circuit area minimization. The existing ROVER implementation leaves the control path untouched, such as loops. SEER generalizes ROVER to a higher-level software abstraction for HLS tools and combines it with control path optimizations for pipelined designs.

3.3 Multi-Level Intermediate Representation

Multi-Level Intermediate Representation (MLIR) [33] is a compiler infrastructure framework developed within the Low-Level Virtual Machine (LLVM) project [32]. It aims to address the challenges of representing and optimizing programs at different levels of abstraction. Dialects can be seen as a namespace for operations, types and attributes modelling specific abstractions (i.e. control flow or affine loops). Primarily, SEER uses the `affine` and `scf` dialects. The `affine` dialect provides a program abstraction for affine operations, and the `scf` dialect provides a program abstraction for structured control flows. MLIR offers a comprehensive set of transformation and analysis passes that can be directly reused and explored in SEER.

4 Methodology

In this section, we describe the proposed source-to-source super-optimization tool for HLS. First, we provide an overview of the proposed SEER toolflow. We then introduce a new intermediate language, named SeerLang, that provides the first interface between MLIR and the egg e-graph library. Next, we explain the rewriting rules included in SEER and how to explore these rewriting rules in the e-graph, to construct R'', as defined in Section 2. Finally, we describe the cost functions used for representation extraction for determining t', as defined in Section 2.

4.1 SEER Overview

To maximize generality and avoid targeting a particular HLS tool, SEER performs source-to-source transformation on the input software program and generates an efficient representation for HLS tools. SEER accepts C, C++, SystemC code, and other software programming languages that can be translated to MLIR. Figure 5 illustrates a high-level overview of the SEER tool flow for HLS super-optimization.

① The input program in C/C++/SystemC is parsed by a C front end named Polygeist [37], a C (and C++) front end for MLIR, translating the program into the MLIR `affine` or `scf` dialects. We implemented MLIR transformation passes for converting a subset of SystemC.

② The SEER front end translates the MLIR into a new intermediate language, SeerLang, which provides an interface between MLIR and the e-graph library, egg [55]. SeerLang is described in Section 4.2.

③ From SeerLang an initial e-graph is constructed in egg, where each e-class contains a single e-node.

④ SEER provides a set of patterns to egg, which are used to search for rewriting opportunities in the e-graph, a process known as e-matching.

⑤ Once a pattern in the e-graph is matched, a validity condition is checked and a new equivalent SeerLang expression is constructed. Section 4.3 describes SEER's rewrites.

⑥ If the rewrite is valid, the new SeerLang expression is unioned into the e-graph, as shown in Figure 4. The e-graph continues to grow until reaching a user defined limit, or until no new equivalent representations can be found. Rewriting in SEER is explained in Section 4.4.

⑦ From the final e-graph, an extraction is performed to obtain an efficient implementation based on control path and data path hardware cost functions. The details of these cost functions are explained in Section 4.6.

⑧ The extracted SeerLang expression is translated back to the MLIR `affine` or `scf` dialects by the SEER back end, such that we can exploit existing MLIR back ends.

⑨ The generated MLIR is converted back to SystemC using emitC [36], such that the optimized program can be parsed by HLS tools. We extended the C back end to emit SystemC programs.

⑩ The equivalence between the original and transformed programs is proven by a formal equivalence checking tool, VC Formal from Synopsys [49], at SystemC level. The equivalence check steps are explained in Section 4.7

SEER is composed of two main components. MLIR passes translating MLIR dialects to and from SeerLang implemented in around 2600 lines of C++ and an e-graph optimizer built on-top of the egg library implemented in around 600 lines of Rust.

```
int x[8];
int i;
for (i=0;i<8;i++)
{
  int a = x[i];
  int b = a*2+1;
  x[i] = b;
}
```

Listing 4. C source

```
affine.for %i=0 to 8 step 1 {
  %a=affine.load %x[%i] : memref<8xi32>
  %b0=arith.muli %a,2 : i32
  %b1=arith.addi %b0,1 : i32
  affine.store %b1, %x[%i] : memref<8xi32>
}
```

Listing 5. MLIR code

```
(affine.for "affine.for_0" %i 0 8 1 none none none
(block
(seq
(affine.load i32 "%a" i32 (8) "%x" (%i))
(affine.store
i32 (+ i32 i32 (* i32 i32 "%a" i32 2) i32 1)
i32 (8) "%x" (%i))
)))
```

Listing 6. SeerLang expression

Figure 6. Example of SeerLang for expressing a `for` loop and memory operations.

4.2 SEER Intermediate Representation

A key challenge for enabling MLIR exploration via e-graph rewriting in egg is that these two frameworks do not share a common representation language, and re-implementing either would require significant engineering effort. We identified three potential solutions for orchestrating them in the same toolflow. First, we could keep each MLIR representation in memory but removing redundancy among these versions is challenging, making the memory size unscalable. Second, we could keep a single representation and pass traces for obtaining each new MLIR representation. This leads to unscalable compilation time for reproducing the required representation. Finally, we decided to propose a new language named SeerLang in egg for translation to and from MLIR.

In egg, users define an S-expression based language similar to Common Lisp [46] to represent expressions.

`term ::= (operator [term] [term]. . . [term])`

This language format allows users to concisely express rewrites. We defined a domain-specific representation, called SeerLang, that provides an interface between MLIR and egg. The semantics of SeerLang are similar to MLIR as the representation is for translation only. SeerLang supports a subset of MLIR operations including operations in the `affine`, `scf`, `memref` and `arith` dialects, but can be extended to support other MLIR operations. In addition, SeerLang supports a `seq`

Table 2. Example SEER rewriting rules implemented directly in egg. SEER contains 106 data path and gate-level rewrites [10]. All datapath rewrites are signage and bitwidth dependent.

Class	Pattern	Transformation
Control Path	`(seq` a `(seq` b c`))`	`(seq (seq` a b`)` c`)`
Data Path	$(a \times b) \ll c$	$(a \ll c) \times b$
	$a\ ?\ (b+c) : (d+e)$	$(a\ ?\ b : d) + (a\ ?\ c : e)$
	$(a \times b) + a$	$a \times (b+1)$
	$a \ll c$	$a \times 2^c$
	$(a \ll b) \ll c$	$a \ll (b+c)$
	$-a$	$\overline{a} + 1$
Gate Level	$(a\&b) \oplus (a\&c)$	$a\&(b \oplus c)$
	$a \oplus a$	0
	$\overline{a\&b}$	$\overline{a}\|\|\overline{b}$

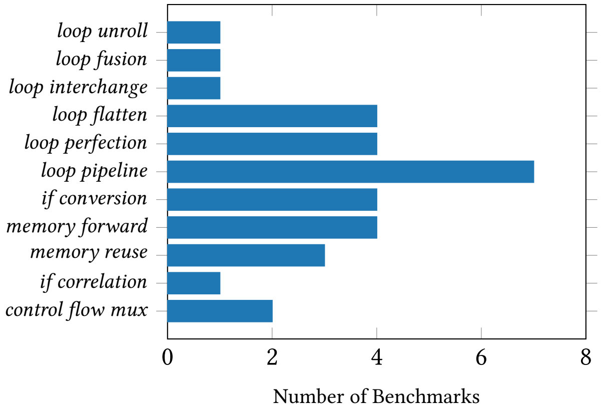


Figure 7. The number of benchmarks which each control flow transformation was successfully applied to out of the nine benchmarks evaluated.

operator to encode the original program ordering between two operations.

Here we introduce two key constructs in SeerLang, operation and block, inspired by MLIR. An operation takes a set of inputs and produces a set of results. An operation could be a data path operation like an `add` operation or a `mul` operation, or a control path operation like a function, a loop or an `if` statement. A block contains a set of operations. In each block, the SeerLang front end analyses the data dependence between operations in the same block and reconstructs expression trees. A `seq` operation is purely an annotation, preserving the original program order by keeping memory operations in the block and associating them using `seq` operations. This facilitates memory dependence analysis for the transformation pass.

An example of SeerLang is shown in Figure 6. Listing 4 shows a for loop that contains two memory operations. This is translated into Listing 5 in the MLIR `affine` dialect. The `for` loop in C is translated into an `affine.for` operation because the loop contains only affine memory accesses. The memory operations are translated into `affine` memory operations as the array index is a simple loop iterator and is in affine form. The arithmetic operations are translated to operations in the MLIR `arith` dialect. The equivalent SeerLang of Listing 5 is shown in Listing 6.

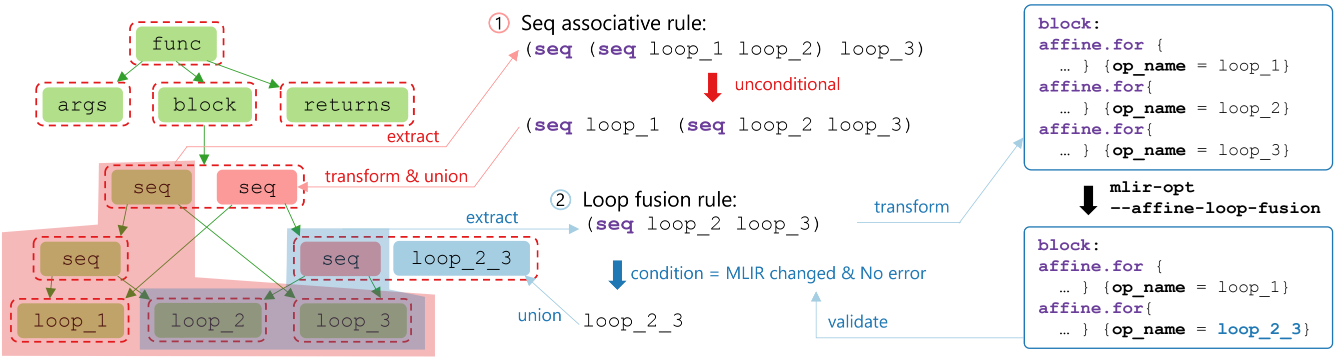
Translating into SeerLang from MLIR is nearly lossless since each operation in SeerLang keeps the type and operand information, except for the program order of independent data path operations. Independent operations can be safely executed out-of-order so SeerLang does not maintain their original program order during the translation. The data dependence is analyzed by the front end of SeerLang when translating from MLIR. For example, in Listing 6, the arithmetic operations for `%b0` and `%b1` are converted into a nested expression at line 6. This recovers the data flow graph of the block for data path optimization. The operations with a potential data dependence are connected using `seq` operations. In SEER, we assume there exists a data dependence between every two memory operations for simplicity. The memory operations are connected using `seq` operations, such as the load and store operations in Listing 6. This preserves the program order of memory access and ensures the correctness of memory transformation.

4.3 Rewriting Rules

The rewriting rules in SEER enable the exploration of equivalent implementations of a program. SEER supports both internal rewrites, expressed directly in SeerLang, and external rewrites, expressed as MLIR passes. For internal rules, egg can directly apply them to add equivalent sub-expressions to the e-graph. A subset of these rules is shown in Table 2. For external rules, SEER leverages existing MLIR transformation passes, reducing the engineering effort required. SEER adopts existing rewrites in MLIR and orchestrates them in egg. Specifically, external rules are implemented as dynamic rewrites in egg, where SEER matches a SeerLang pattern and then construct an equivalent implementation using an external pass. Each dynamic rewrite calls a set of MLIR passes for analysis and transformation.

In this construction, SeerLang is translated into a compatible representation, modified by the external pass, and translated back to SeerLang. egg can then union this new sub-expression into the appropriate e-class. Such an approach makes it simple to implement new rules and enables the reuse of existing rules from other toolflows. SEER rewrites at different granularities, allowing it to simultaneously optimize at the control path-level, data path-level and gate-level.

First, the control path-level rewrites modify the control flow graph (CFG) of the original program. Particularly, we focus on the transformation of `for` loops and `if` statements.

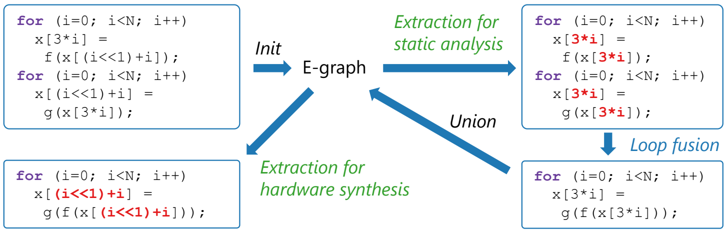


```
(seq (seq loop_1 loop_2) loop_3)
(seq loop_1 (seq loop_2 loop_3))
(seq loop_2 loop_3)
loop_2_3
```

```
block:
affine.for {
  … } {op_name = loop_1}
affine.for{
  … } {op_name = loop_2}
affine.for{
  … } {op_name = loop_3}
```

```
block:
affine.for {
  … } {op_name = loop_1}
affine.for{
  … } {op_name = loop_2_3}
```

Figure 8. E-graph exploration of the motivational example (Figure 2) using SEER. The e-graph is simplified by merging subgraphs of loops into single nodes. The green nodes represent the initial e-graph obtained from Listing 1. ① illustrates an example of an unconditional rewrite for a sequential association inside egg. For rewrite ①, the original sub-expression in the shaded red region is rewritten to the red node in the same e-class. ② illustrates an example of a conditional rewrite for loop fusion through MLIR. For rewrite ②, the original sub-expression in the shaded blue region is rewritten to the blue node in the same e-class.


Init
E-graph
Extraction for static analysis
Union
Extraction for hardware synthesis
Loop fusion

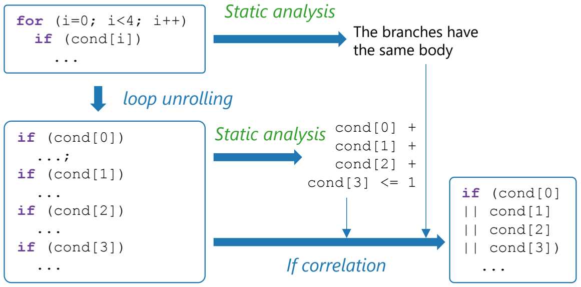
```
for (i=0; i<N; i++)
  x[3*i] =
    f(x[(i<<1)+i]);
for (i=0; i<N; i++)
  x[(i<<1)+i] =
    g(x[3*i]);
```

```
for (i=0; i<N; i++)
  x[3*i] =
    f(x[3*i]);
for (i=0; i<N; i++)
  x[3*i] =
    g(x[3*i]);
```

```
for (i=0; i<N; i++)
  x[3*i] =
    g(f(x[3*i]));
```

```
for (i=0; i<N; i++)
  x[(i<<1)+i] =
    g(f(x[(i<<1)+i]));
```

Figure 9. An example of extracting representations using cost functions for static analysis and hardware synthesis.



```
for (i=0; i<4; i++)
  if (cond[i])
    ...
```

```
if (cond[0])
  ...;
if (cond[1])
  ...
if (cond[2])
  ...
if (cond[3])
  ...
```

```
cond[0] +
cond[1] +
cond[2] +
cond[3] <= 1
```

```
if (cond[0]
|| cond[1]
|| cond[2]
|| cond[3])
  ...
```

Figure 10. An example of using program invariants from static analysis of one representation for transformation of another.

This includes ten MLIR passes for loop re-ordering, loop merging and if conversions. We maximize the reuse of available MLIR passes in upstream in SEER.

Most of the loop transformation passes are directly adopted from the MLIR/LLVM upstream and applied to SEER. The *loop unroll* pass completely unrolls a given loop. This enables potential loop body reduction by other passes such as the memory forward pass. We disable exploring loop unrolling with different unrolling factors by default to improve scalability. It is provided as a user option. The *loop fusion*, *loop interchange* and *loop flatten* passes are existing compiler transformations which are directly mapped to SEER. The *loop perfection* pass converts a loop nest that contains code in its outer loop body and outside its inner loop body to a perfect loop nest. This is done by moving the code outside the inner loop body into the inner loop body with predicates. Loop perfection opens up opportunities for more loop transformation, such as loop interchanging and loop flattening.

The *if conversion* pass is used to convert `if` statements to `select` operations, reducing the control flow complexity. This has been widely used in the HLS code transformation for maximizing a data path region for pipelining. The *memory forward* pass removes redundant load and store operations in the code to reduce memory accesses. The upstream pass only removes store operations. We extend it to remove redundant load operations as well.

Customized MLIR passes can also be easily extended to SEER with the same interface. For instance, the *if correlation* pass is a customized pass which detects correlation among conditions of several sequential `if` statements and merges them if the conditions are identical or disjoint. An example of `if` correlation is described in Section 4.5. The *memory reuse* pass moves a read-only memory access outside the loop. The *control flow mux* pass moves an operation in both branches of an `if` statement outside the `if` statement and select its args in the branches for resource sharing at the source level. The applicability of the MLIR passes to the nine benchmarks evaluated in Section 5 is shown in Figure 7.

The data path-level rewrites modify the program at a finer grain and are mostly re-used from the e-graph based ROVER tool [10, 11]. They include expression balancing, constant folding and manipulation, and strength reduction. Data path optimization is currently under-explored in existing commercial synthesis tools and recent synthesis-aware data path rewriting has been shown to reduce circuit area [10, 11]. Two rewrites from Table 2 are applied to the e-graph in Figure 4.

Finally, the gate-level rewrites also modify the program at the operator level but target bit-level hardware customization. Most gate-level rewrites are well exploited by the logic synthesis optimization in HLS tools. However, data path and bit-level rewriting can often interact providing a mutual benefit. SEER restricts the number of gate minimization techniques to improve scalability. We group these into the data path set.

4.4 E-graph Rewriting for Super-optimization

SEER alternates between iterations of control flow rewriting and data path rewriting. At each iteration all rules within the given rewrite set are applied, growing the e-graph. SEER interleaves the exploration of these rewrite sets since one might introduce more rewriting opportunities for the other. For instance, dead code elimination, a data path rewrite, can change the dependence constraints, enabling more control path rewrites. Loop fusion, a control path rewrite, can enable further rewrites for the fused loop body.

Figure 8 shows the e-graph exploration of the motivating example introduced in Figure 2. To the initial e-graph, represented by the green nodes, SEER applies an internal rewrite rule from Table 2, seq associativity. The general rule, shown in the top middle of the figure, matches the sub-expression covered by the red shading and returns an equivalent SeerLang expression. This new expression is unioned into the matched e-class, where the new nodes are shown in red.

Next SEER applies the external loop fusion MLIR transformation, that represents the transformation from Listing 1 to Listing 3. The loop fusion rule searches for two sequential loops and checks if they satisfy the particular dependency constraints. First, the sub-expression covered by the blue shading matches the pattern of the loop fusion rewrite. The matched SeerLang is translated into the equivalent MLIR. Then SEER calls the existing loop fusion pass in MLIR, generating a new MLIR implementation. The loop fusion pass performs the dependence check internally before the transformation. If the dependence constraints are unsatisfied or the transformation fails, the pass returns the original MLIR. The loop fusion rule in egg checks that there were no errors in the pass and that the returned MLIR differs from the input then converts this back to SeerLang and performs the union. In this example, the result from fusing `loop_2` and `loop_3` is added to the e-graph, as the blue `loop_2_3` node.

Many other SEER rewrites can be applied to grow a larger e-graph than the one presented in Figure 8. Thanks to the e-graph representation, despite fusing `loop_2` and `loop_3`, the fusion of `loop_1` and `loop_2` can also still be triggered. This would not be possible in a traditional compiler. Note that the fusion of `loop_1` and `loop_2_3` will be attempted but will fail due to the validity checks. The e-graph grown after several rewriting iterations, represents the explored design space of equivalent implementations. From this e-graph SEER must now select an efficient HLS implementation.

4.5 Deeper Optimization Opportunities

Prior work observed how retaining multiple representations in an e-graph can improve program analysis [12]. Here we observe a practical benefit of this, allowing SEER to discover implementations that are unreachable with existing compiler passes. SEER can learn program invariants from one representation which it can use to rewrite any equivalent representation.

Firstly, we shall describe how SEER can resolve loop dependence analysis limitations. Existing loop optimization passes use polyhedral analysis to detect any loop dependency issues. Such tools are unable to analyze memory access patterns that are not obviously affine. This introduces a tension, as a representation for efficient hardware synthesis could be complex for static analysis. For example in Figure 9, the memory access index `(i«1)+i` is area-efficient in hardware because the shift operation is area-free in ASIC design, and only one adder is used. Polyhedral analysis tools will fail to interpret such a non-affine access pattern and conservatively may prevent subsequent loop optimizations.

Applying the data path rewrites to an e-graph containing `(i«1)+i` as shown in Figure 4, SEER discovers the equivalent *affine* expression `3*i`, which is interpretable by the static analyzer. Such an expression may not be area-efficient since it uses a multiplier.

Starting from an initial e-graph containing the input program in Figure 9, SEER applies data path rewrites, discovering a representation where both memory indices are in affine form (top right). When SEER calls its loop fusion pass, it is presented with a choice of many equivalent loop implementations which it could pass to the external compiler pass. SEER aims to pass on analysis-friendly implementations, namely those with affine memory accesses. To achieve this SEER includes an analysis-friendly cost function, that assigns low cost to multiplications and additions, making affine expressions lower cost than alternative logic expressions. When a loop optimization triggers, SEER runs a local extraction process on the matched e-class, using the analysis-friendly cost function, extracting affine memory access patterns where possible. In Figure 9, SEER can successfully fuse the two loops (bottom right) and generate a final HLS program using the hardware-efficient but non-affine memory access pattern (bottom left).

The extraction process deployed during the rewrite process uses the built-in greedy egg extraction method [55], where the best node is selected from each e-class without taking into account common sub-expressions. The greedy

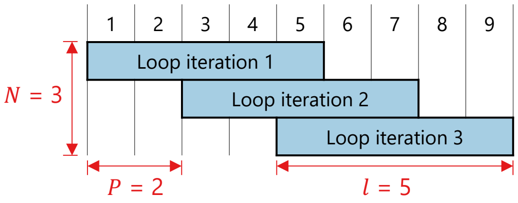


Figure 11. A schedule of a pipelined loop in HLS

method is fast. This is important as SEER may use this extraction process many times during e-graph rewriting. In Section 4.6 we describe a more computationally expensive extraction process that provides an accurate model of hardware implementation cost. It is run only once on the final e-graph.

The advantage is also seen in other programming constructs. The code in the top left of Figure 10 shows an `if` statement in a `for` loop. A possible transformation is loop unrolling, leading to straight-line code with four `if` statements (bottom left). Assuming that all conditions are independent and at most one of them is true, it is possible to merge the conditions into a single block. However, existing compiler passes struggle to compare the source of the true branches for the `if` statements, particularly when the code size is large. Fortunately, these invariants can be easily obtained from the original representation (top left), which is still present in the e-graph. With the invariants in place, the transformation is successful.

4.6 Cost Function Specifications

Once the e-graph rewriting process terminates, due to saturation or reaching a computational limit, SEER extracts an efficient HLS implementation. This extraction process is run only once on the final e-graph and provides an accurate model of hardware implementation cost. SEER combines a pair of theoretical cost functions to rank the equivalent implementations. We separate the extraction into a two-phase problem, first extract the control flow nodes that maximizes performance, then from the fixed control flow minimize the data path circuit area. The control flow nodes are the subset of SeerLang operations, for example `for` and `if` statements, that describe the program's control flow. SEER has pre-defined patterns for the cost function, allowing the extractor to identify and extract these control flow nodes.

The control path is usually parallelized, such that the latency of each data path is hidden by the pipeline. Pipelining is usually beneficial because it improves performance at a slightly higher area cost. SEER assumes all the loops are pipelined by default to achieve better performance.

The control flow cost function evaluates the latency of pipelined loops in terms of clock cycles. A pipelined loop has three scheduling constraints: the initiation interval P, iteration latency l, and loop trip count N. Figure 11 shows an example of a simple pipelined loop. The initiation interval is the number of clock cycles between two consecutive loop iterations. The iteration latency is the latency of a single iteration in clock cycles. For this example, $P = 2$ and $l = 5$. These are typically constants because most HLS tools use static scheduling [4, 61]. The loop trip count represents the number of iterations. For this example, $N = 3$. The total latency L of a pipelined loop can be obtained based on the formula shown in Constraint 1 [4]. SEER obtains the scheduling constraints of each loops in the original representation of the program by calling the HLS tool to schedule the original representation.

$$L = (N - 1) \times P + l \qquad (1)$$

In order to improve scalability, we approximate the schedule constraints of the newly generated loops during the exploration from the existing scheduling constraints of the original loops. This approximation facilitates exploration at scale, avoiding calls to the HLS scheduler for each new representation.

For each loop in the initial representation, SEER obtains (P, l, N, A) from the initial HLS run, where A is the set of memory accesses in the loop. A is a resource constraint used for estimating the upper bound of throughput based on the memory bandwidth at run time. For instance, each loop in Listing 1 has $|A| = 2$.

Here we provide three key examples of loop transformations at different levels, loop fusion, loop flattening and loop unrolling. First, let (P_1, l_1, N_1, A_1) and (P_2, l_2, N_2, A_2) be the scheduling constraints for two sequential loops to be fused. Let $M(A)$ be the maximum number of memory accesses to a single array, and assume all the BRAM blocks are single-port. The fused loop will have constraints (P', l', N', A') as follows.

$$\begin{aligned} l' &= max(l_1, l_2) & N' &= max(N_1, N_2) \\ A' &= A_1 \cup A_2 & P' &= max(P_1, P_2, M(A')) \end{aligned}$$

Second, if the outer loop and the inner loop of a perfect loop nest have scheduling constraints $(P_{outer}, l_{outer}, N_{outer}, A_{outer})$ and $(P_{inner}, l_{inner}, N_{inner}, A_{inner})$ respectively, the flattened loop then has scheduling constraints $(P_{inner}, l_{inner}, N_{inner} \times N_{outer}, A_{inner})$. Finally, if a loop with scheduling constraints (P, l, N, A) is unrolled, the scheduling constraints after the transformation are $(1, N \times l, 1, N \times A)$.

To formulate the extraction, let E denote the set of all program representations in the e-graph. SEER assigns a latency, $L(n)$, to each e-node, n:

$$L(n) = \begin{cases} (N_n - 1) \times P_n + l_n, & \text{if } n \text{ is a loop} \\ 0, & \text{otherwise,} \end{cases} \qquad (2)$$

where we denote the scheduling constraints of a loop node n by (P_n, l_n, N_n, A_n). A completely unrolled loop is still considered a loop with an iteration count of 1 to avoid zero cost during the extraction. The if statements are extracted based on the data path cost function. The objective for control flow extraction is then to minimize the sum of loop latencies.

$$\begin{aligned} \min_{e} \quad & \sum_{n \in e} L(n) \\ \text{s.t.} \quad & e \in E \end{aligned} \tag{3}$$

We use a greedy extraction method for control flow, selecting the lowest cost loop implementation in each control flow e-class. Since data path nodes are considered zero cost at this stage, control flow extraction returns a subset of representations $E' \subseteq E$, that all share the same optimized control flow. SEER must now select an efficient data path implementation from these remaining implementations.

For the data path, SEER minimizes area rather than latency, as operation latencies are often hidden by loop pipelining. SEER leverages an existing cost function from ROVER to extract the minimal circuit area expression in each block [10]. ROVER assigns an area cost, $A(n)$, to each e-node, n, based on a bitwidth-dependent gate count. The objective for data path extraction is then:

$$\begin{aligned} \min_{e} \quad & \sum_{n \in e} A(n) \\ \text{s.t.} \quad & e \in E' \end{aligned} \tag{4}$$

In ROVER, data path extraction is formulated as an integer linear programming (ILP) problem [10, 54], solved using the Coin-Or CBC solver [19]. The ILP returns a single SeerLang representation, which is passed to the SEER back end.

4.7 Verification

Our work benefits from SEER orchestrating existing transformation passes for super-optimization exploration. However, these passes may be unverified and could introduce non-equivalent representations. In hardware design formal verification increases trust in the correctness of an implementation. We adopt a translation validation approach based on the egg proof production feature [18]. SEER traces back the intermediate forms to the original program from the extracted representation, generating SystemC for each step that differs from the previous step by a single rewrite. SEER then generates a sequence of equivalence checks, constructing a sound chain of reasoning that the original and generated programs are functionally equivalent. Each intermediate check is proven using a commercial equivalence checker. By decomposing the verification problem into a sequence of simpler sub-problems, SEER provides a robust verification flow. In this paper, we focus on the optimization capabilities of SEER, but a detailed discussion on verification using e-graphs is provided in Section 6.3.

5 Experiments

We evaluate SEER on a set of benchmarks. We compare SEER with a vanilla commercial HLS tool for ASICs and the data path optimizer ROVER [10], as a hardware optimizer that also uses e-graphs. We do not compare against the related works [59, 62] mentioned in Section 3.3 since they target FPGAs and we target ASICs.

We assume that the designer has no hardware knowledge. To ensure fairness, we synthesize the original, ROVER-generated, and SEER-generated programs using the same HLS configuration. We evaluate the impact of SEER on circuit area, performance in wall clock time and power. The total clock cycles were obtained from the HLS co-simulation. The area and power results were obtained from the Post & Route report from the HLS tool. We targeted a 45nm technology library.

5.1 Benchmarks

Finding suitable benchmarks is a perennial problem for papers that push the limits of HLS, in part because existing benchmarks tend to be tailored to what HLS tools can already comfortably handle. In this work, we combine artificially constructed, Intel provided and open-source benchmarks from the MachSuite [43] set. SEER is amenable to programs with complex data path blocks, control flow, or memory access patterns. In this work, we include the subset of MachSuite benchmarks (8 out of 19) for which current HLS tools are unable to achieve the optimal results. For the remaining benchmarks in MachSuite, HLS tools are already able to match expert human designers. The selected benchmarks implement algorithms as low-level kernels suitable for hardware acceleration. We aim to evaluate SEER on super-optimization for 1) different application programs and 2) different implementations of the same application program using different algorithms. We use the following benchmarks:

`seq_loops` represents the sequential loop example shown in Figure 9, amenable to loop fusion.

`byte_enable_calc` pre-processes and combines multiple messages into one. Widely used in computer architectures.

`kmp` is an implementation of the Knuth-Morris-Pratt algorithm [24] for string matching.

`gemm (ncubed/blocked)` is a naive/blocked implementation of dense matrix multiplication. The ncubed algorithm is unoptimized and has a complexity of $O(n^3)$. The blocked algorithm [31] provides better locality.

`md (grid/knn)` simulates molecular dynamics using N-body methods to compute local forces. The grid implementation uses spatial decomposition from polyhedral transformations. The knn implementation was originally from the SHOC benchmark suite [14] and uses K-nearest neighbors.

`sort (merge/radix)` sorts an integer array. Merge uses the merge sort algorithm [9], and the radix implementation compares 4-bits blocks at a time.

`seq_loops` is made artificially to demonstrate a simple example, and `byte_enable_calc` is production code provided by Intel. The rest of the benchmarks are directly obtained from the MachSuite [43] benchmark set.

```
 1 for (j=1; j<4; j++) {
 2   for (i=0; i<4; i++) { ①
 3     if (list[j][i]
 4      && enable[i]②)③
 5     { ④
 6       sum++;
 7       enable[0] = false;
 8       enable[1] = false;
 9       enable[2] = false;
10       ennable[3] = false; ⑥
11     }
12     if (list[j][i]) ⑤
13       enable[i] = true;
14   }
15 }
```

Figure 12. `byte_enable_calc`

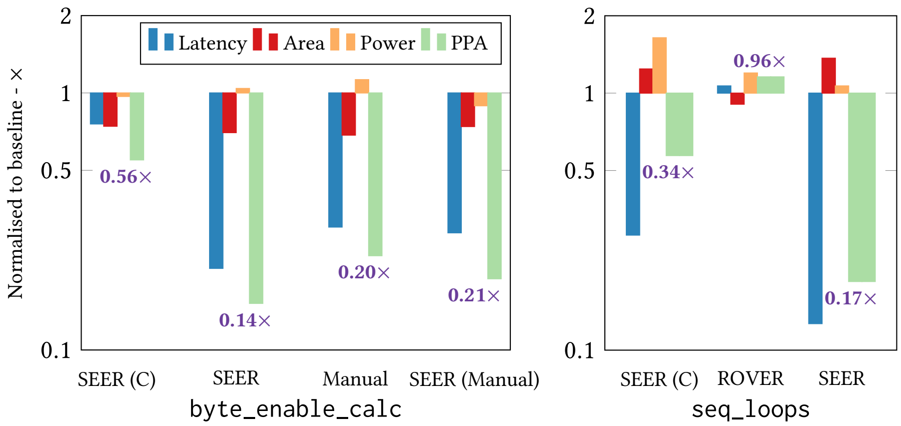


Figure 13. Normalized results of hardware designs for `byte_enable_calc` and `seq_loops` using different approaches compared to the baseline results of hardware designs from original programs. The results by ROVER-only on `byte_enable_calc` are the same as the baseline. The **text** represents the normalized area-delay products.

5.2 Intel Production Code Case Study

We first provide a case study of applying SEER to a snippet of unoptimized Intel production code, shown in Figure 12. The source code implements a resource combining algorithm. It is commonly used as write combination logic, as part of a larger state machine aiming to maximize bus traffic. The example uses the minimization of bus accesses as a criteria and uses the bits to track these resources. Due to the time critical nature of these kind of resource management state machines the logic needs to be quick so that any decisions on dispatch can be made in few clock cycles.

The HLS tool cannot synthesize efficient hardware because of the data dependence on `enable` across loop iterations. The control path shown as `if` statements are also unnecessary and are challenging for the tool to interpret. The following are potential optimization opportunities:

① *Loop Unroll:* The iteration counts of both loops are small. It may be beneficial to unroll the loops for more data parallelism at low area overhead.

② *Memory Forward:* There are multiple load and store operations to `enable`. These operations could be folded to reduce data dependence on `enable`.

③ *If Correlation:* When the loop is unrolled, the if conditions in different loop iterations may be correlated as shown in Figure 10.

④ *If Conversion:* The true branch of the `if` statement at line 13 is a single line and could be rewritten as a multiplexer.

⑤ *Mux Reduction:* The same true branch updates a single bit using a constant, which could be directly fetched from the `if` condition. The same applies to lines 7-10.

⑥ *Gate Reduction:* The logic expressions in the loop body could be simplified once folded into a single data path using the transformation steps above.

Table 3. Case study on the `byte_enable_calc` benchmark. SEER (C) only explores control path optimizations. SEER (Manual) explores the manually optimized source code. CP = Critical Path. ET = Execution Time. PPA = Performance Power Area product.

Approaches	Area (μm^2)	Cycles	CP (ns)	ET (ns)	Power (mW)	PPA ($yW \cdot m^2 \cdot s$)
Baseline	1.94	119	0.976	116	0.49	110
ROVER	1.94	119	0.976	116	0.49	110
SEER (C)	1.44	81	1.09	87.9	0.475	60.4
SEER	1.36	**29**	**0.831**	**24.1**	0.51	**16.7**
Manual	**1.33**	42	**0.831**	34.9	0.552	25.6
SEER (Manual)	1.44	34	0.976	33.2	**0.44**	20.8

The results for the code in Figure 12 are shown on the left of Figure 13. In the figure, smaller values indicate better results. In addition to the results obtained from the original program and SEER, we also obtained the results from the manually optimized design by Intel hardware experts. Furthermore, we gave the manually optimized design to SEER. We made the following observations:

- ROVER could not optimize the input program because the data paths are separated by control operations.
- Exploring MLIR passes (SEER (C)) improves performance and area, due to the conversion of control path operations to data path operations and memory forwarding.
- By combining ROVER rewrites and MLIR passes SEER achieves significantly better performance improvements and area reduction. The performance of SEER-generated hardware even outperforms the manually optimized hardware design with a small area overhead.

Table 4. Evaluation of SEER over a set of benchmarks. Area in μm^2. Total Cycles in 1000's. Critical Path in ns. Power in mW.

Benchmarks	Baseline				ROVER				SEER			
	Area	Total Cycles	Critical Path	Power	Area	Total Cycles	Critical Path	Power	Area	Total Cycles	Critical Path	Power
`seq_loops`	1.77	1.5	0.943	**0.224**	**1.61**	1.6	0.943	0.268	2.42	**0.203**	**0.883**	0.238
`kmp`	8.09	357	1.53	0.581	8.09	357	1.53	0.581	**7.72**	**292**	**1.42**	**0.546**
`gemm (blocked)`	**14.4**	4620	1.16	**0.735**	**14.4**	4620	1.16	**0.735**	16.3	**537**	**1.11**	3.44
`gemm (ncubed)`	**11.8**	3410	0.971	**0.972**	**11.8**	3410	0.971	**0.972**	12.7	**535**	0.971	5.83
`md (grid)`	**132**	1480	1.55	**2.31**	**132**	1480	1.55	**2.31**	180	**346**	**1.54**	5.07
`md (knn)`	**107**	303	1.19	2.34	**107**	303	1.19	2.27	127	**8.25**	**1.14**	**1.62**
`sort (merge)`	26.4	238	1.42	1.56	26.4	238	1.42	1.56	**14.8**	**153**	**1.24**	**1.36**
`sort (radix)`	10.8	223	1.39	0.262	10.8	223	1.39	1.06	**9.05**	**136**	**1.28**	**1.02**
Norm. Geom. Mean.	1×	1×	1×	1×	**0.99×**	1.01×	1×	**1.4×**	1.06×	**0.34×**	**0.95×**	2.54×

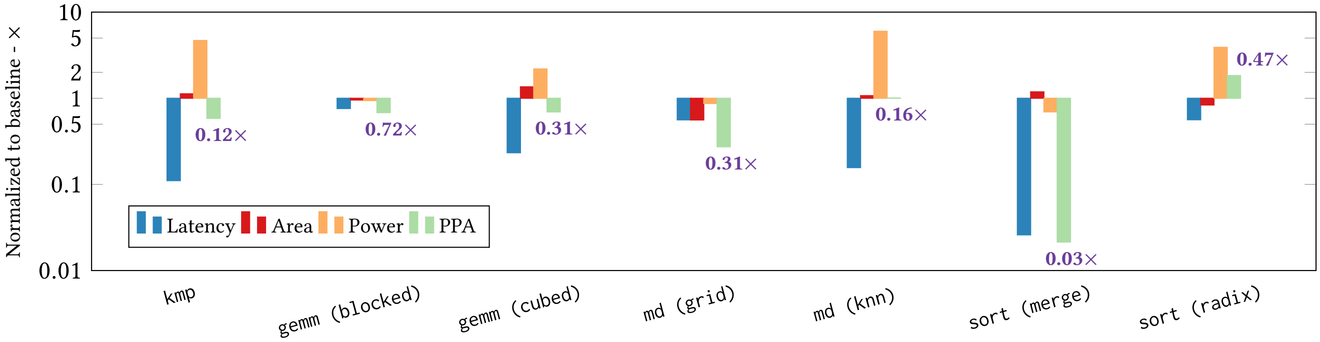


Figure 14. Normalized results for the SEER generated programs compared to the vanilla HLS programs, across the key hardware efficiency metrics for the benchmarks evaluated. The **text** represents the normalized area-delay products.

- SEER achieves the best hardware results among automatically generated designs, and SEER even optimizes the manually optimized hardware design by hardware experts, pushing the limit in both performance and area.

The detailed results for `byte_enable_calc` are shown in Table 3. We also observed similar results for benchmark `seq_loops`, in the left of Figure 13:

- Exploring ROVER optimization only achieves area reduction in the data path.
- Exploring MLIR passes only (SEER (C)) achieves performance improvements with area and power overhead.
- Exploring both ROVER and MLIR passes with SEER achieves the best performance with less power overhead due to the interaction between ROVER rewrites and MLIR passes, which enables more optimization opportunities.

5.3 Overall Results

The improvements on other benchmarks are shown in Figure 14, and the detailed results are shown in Table 4. SEER has achieved better performance for all the benchmarks by enabling automatic loop pipelining. The HLS tool cannot auto-pipeline loops without human guidance. Loop pipelining also causes additional area and power overhead. Overall, the PPA for most benchmarks is improved. In the case of the `sort (radix)` benchmark, the loop has a small trip count, so loop pipelining only provides marginal performance improvements at significant power overhead. Since our cost functions do not consider power, SEER only focuses on performance and area. This potentially introduces inefficient power in the final hardware design. In Figure 14 we highlight the normalized area-delay products to show the effectiveness of SEER on the design metrics explored. On average, SEER achieved a speedup of 2.9× with 0.06× area overhead, and a PPA of 8.9× the original design.

Figure 15 compares the hardware produced by SEER and the hardware obtained by manually inserting pragmas into the unmodified source code. The commercial HLS tool has a set of pragmas to direct some of the transformations explored by SEER, that were described in Section 4.3. There are pragmas to instruct the HLS tool where to apply the *loop pipeline* and the *loop fusion* transformations. The HLS tool also has a pragma to direct the *loop coalesce* transformation, which is a more capable version of the *loop flatten* transformation described in Section 4.3. The *loop flatten* in SEER can only flatten perfect loop nests, while the *loop coalesce* in the HLS tool can flatten more complex loop nests. Finally, *if conversion* and *memory forwarding* are automatically performed

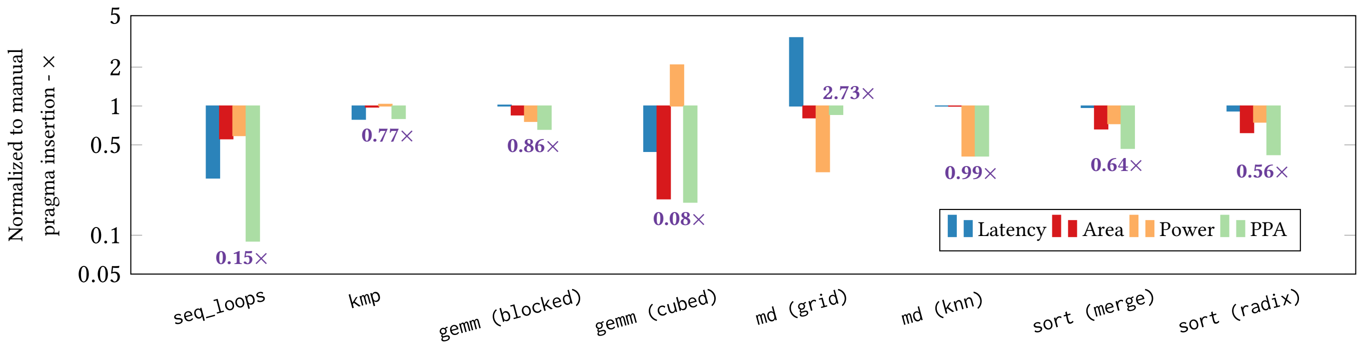


Figure 15. Normalized results for the SEER generated programs compared to HLS programs with manual pragma insertion but no source code modifications, across the key hardware efficiency metrics for the benchmarks evaluated. The **text** represents the normalized area-delay products.

Table 5. The size of the e-graphs and total search times for the evaluated benchmarks.

Benchmarks	Nodes	Time in MLIR (s)	Time in egg (s)	Total Time (s)
`byte_enable_calc`	31769	0.65	160.35	161
`seq_loops`	11906	0.29	0.01	0.3
`kmp`	504	0.54	2.05	2.59
`gemm (blocked)`	10947	3.96	16.24	20.2
`gemm (ncubed)`	114	0.88	0.42	1.3
`md (grid)`	44328	5.03	56.37	61.4
`md (knn)`	43580	0.37	51.83	52.2
`sort (merge)`	310	0.52	2.07	2.59
`sort (radix)`	273	1.38	1.12	2.5

by default. Users with deeper hardware knowledge can use pragmas to instruct the commercial HLS tool to perform logic synthesis level optimizations. These pragmas were not explored in these experiments as such optimizations are beyond the scope of SEER which targets only control path and data path optimization.

In Figure 15, SEER achieves better results than manually inserted pragmas for most of the benchmarks, because it contains transformations that cannot be expressed by pragmas, such as *memory reuse* and *if correlation*. The pragmas provided by the HLS tool cover certain transformations that are currently not covered by SEER, such as *loop coalesce*. For example, the HLS tool coalesces all loops in `md (grid)` into a single loop for efficient hardware pipelining, while the MLIR passes in SEER lack the necessary steps to coalesce all loops, leading to a worse latency. We expect that SEER will match or even outperform manual pragma insertion once it has the same set of transformations as the HLS tool.

The size of e-graph and exploration time for each benchmark are shown in Table 5. The MLIR runtimes are generally smaller than the time spent in egg, because the space of equivalent data path implementations can be large and the final extraction can be computationally expensive. Benchmarks that have complex loop structures, such as benchmarks `gemm (blocked)` and `md (grid)`, contain more patterns for MLIR transformations, leading to a larger exploration space and longer MLIR runtime.

The scalability problem is an open challenge in both superoptimization and equality saturation. In hardware design, the modular design principles greatly help to limit the size and scope of optimization. In addition, the bar for compile times is set low for hardware compilation, meaning users are willing to wait for quality results.

6 Related Work

6.1 Phase-Ordering Challenges in Compilers

The phase-ordering problem in compilers has been addressed using two main approaches. First, there are works that use machine learning [2, 3, 21, 29, 40] for inferring a productive sequence of optimization steps. These approaches only work for domain-specific programs, while our approach works for arbitrary programs. Second, there are works that use heuristic-based or iterative approaches for efficient searching for the optimization steps [28, 41, 60]. The intermediate traces during the iterations are not efficiently preserved, while our work carries it in the e-graph during the exploration. All these approaches only target software optimization, while our approach targets hardware optimization.

6.2 MLIR HLS Frameworks

In addition to SEER, there have been several attempts to build hardware design tools using the MLIR framework. CIRCT [8] is an MLIR-based hardware compiler framework under LLVM, which lowers MLIR to register-transfer level (RTL) code as an open-sourced HLS tool. Xu *et al.* propose a specific MLIR dialect named HECTOR for hardware synthesis [57], which can be translated into RTL code. There are also source transformation tools that transform MLIR into optimized HLS code in C [30, 59] or LLVM IR [1, 62]. Both these works and SEER have an end-to-end HLS flow in MLIR. Prior work suffers from the phase-ordering problem illustrated in Figure 1 because they use a fixed sequence of transformation passes. SEER overcomes the phase-ordering challenge using e-graphs, customizing the MLIR pass order for each input program and optimization objective.

6.3 Optimization and Verification using E-Graphs

Lastly, the egg library has fueled a new wave of e-graph research. E-graphs were first used to explore programs containing loops in [50], where the authors introduced θ nodes to represent values that vary inside of a loop within an e-graph. This representation was used to implement a Java bytecode optimizer [50]. In addition to performance optimization, e-graphs have also been applied to automate numerical stability improvement [42], and much more [38, 45, 53, 58]. Beyond applications, there has been some work to address scalability issues via sketch-guiding [25]. In the hardware domain, there is growing interest, with Ustun, Yu and Zhang advocating e-graph rewriting [52] and applying it to multiplier design for FPGAs [51].

The set of representations maintained by an e-graph has also been leveraged to improve verification. Specifically, it is possible to extract rewrite sequences that justify the equivalence of two representations found in the same equivalence class. This technique is known as proof production [18]. These rewrite sequences can then be checked. In the software domain, an e-graph optimization tool [50] was adapted to perform translation validation of LLVM optimizers [47]. More recently, egg was used to develop an RTL datapth equivalence checking assistant [13], where the problem decomposition approach described in Section 4.7, facilitated proof convergence and reduced verification runtimes.

7 Conclusion

This paper described an approach to resolving the phase-ordering problem for HLS. By simultaneously exploring optimizations at different granularities in an e-graph, our approach opens up a larger optimization space for an arbitrary program than existing HLS works. We demonstrated how high-level control optimizations and low-level data path optimizations can mutually benefit, enabling further optimization opportunities. We model the hardware performance for the control path at a software abstraction level and determine efficient HLS code for high throughput and area efficiency.

We implemented a toolflow, SEER, that uses an e-graph to orchestrate high-level software optimizations in MLIR and low-level hardware optimizations in ROVER. We introduced a new intermediate language, SeerLang, that interfaces the egg library and MLIR. We evaluated SEER on open-source benchmarks and an Intel-provided case study, demonstrating an average speedup of 2.9× with minimal area overhead.

Our future work will involve several directions. First, from the programming language point of view, we plan to improve SeerLang for deeper integration with MLIR and egg for efficient translation. For instance, we plan to investigate how complex constructs, such as function calls and global variables can be optimized in SEER. This would also resolve some engineering challenges since most MLIR passes target entire functions rather than local transformations. Second, we plan to extend the exploration space and granularities by integrating optimization techniques from other MLIR projects, such as CIRCT [8] and POLSCA [62]. We will investigate parallel e-graph exploration using multiple threads to improve scalability. Further efficiency gains could be made by partitioning the e-graph and exploring different sub-graphs independently. Lastly, we will evaluate SEER on larger benchmarks to understand the practical limitations of the approach.

References

[1] Nicolas Bohm Agostini, Serena Curzel, David Kaeli, and Antonino Tumeo. Soda-opt an mlir based flow for co-design and high-level synthesis. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 201–202, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–28, 2017.

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[4] Andrew Canis, Stephen D Brown, and Jason H Anderson. Modulo sdc scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[5] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. High-level synthesis of memory bound and irregular parallel applications with Bambu. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–1, Cupertino, CA, Aug 2014. IEEE.

[6] Catapult High-Level Synthesis, 2023.

[7] Chong-Yun Chao and Earl Glen Whitehead. On chromatic equivalence of graphs. In *Theory and Applications of Graphs: Proceedings, Michigan May 11–15, 1976*, pages 121–131. Springer, 1978.

[8] Circuit IR Compilers and Tools, 2023.

[9] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[10] Samuel Coward, George A. Constantinides, and Theo Drane. Automatic Datapath Optimization using E-Graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 43–50, 2022.

[11] Samuel Coward, George A Constantinides, and Theo Drane. Automating constraint-aware datapath optimization using e-graphs. *arXiv preprint arXiv:2303.01839*, 2023.

[12] Samuel Coward, George A Constantinides, and Theo Drane. Combining e-graphs with abstract interpretation. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 1–7, 2023.

[13] Samuel Coward, Emiliano Morini, Bryan Tan, Theo Drane, and George Constantinides. Datapath verification via word-level e-graph rewriting. *arXiv preprint arXiv:2308.00431*, 2023.

[14] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter.

The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 63–74, 2010.

[15] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*, pages 183–198. Springer, 2007.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4963 LNCS, 2008.

[17] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P Carloni, and Laura Pozzi. Leveraging prior knowledge for effective design-space exploration in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3736–3747, 2020.

[18] O. Flatt, S. Coward, M. Willsey, Z. Tatlock, and P. Panchekha. Small Proofs from Congruence Closure. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design, FMCAD 2022*, 2022.

[19] John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

[20] Yuko Hara-Azumi, Toshinobu Matsuba, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Selective resource sharing with rt-level retiming for clock enhancement in high-level synthesis. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1534–1540. IEEE, 2012.

[21] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308. IEEE, 2019.

[22] Intel HLS Compiler, 2023.

[23] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 127–136, Monterey, CA, 2018. ACM.

[24] Knuth-Morris-Pratt algorithm, 2023.

[25] Thomas Koehler, Phil Trinder, and Michel Steuwer. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations of Functional Programs. 11 2021.

[26] Vyas Krishnan and Srinivas Katkoori. A genetic algorithm for the design space exploration of datapaths during high-level synthesis. *IEEE Transactions on Evolutionary Computation*, 10(3):213–229, 2006.

[27] PN Krishnapriya and B Bala Tripura Sundari. High level synthesis for retiming stochastic vlsi signal processing architectures. *Procedia computer science*, 143:10–19, 2018.

[28] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1–36, 2009.

[29] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 147–162, 2012.

[30] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 242–251, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGPLAN Not.*, 26(4):63–74, apr 1991.

[32] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[34] Bruce W Leverett, Roderic Geoffrey Galton Cattell, Steven O Hobbs, Joseph M Newcomer, Andrew Henry Reiner, Bruce R Schatz, and William A Wulf. An overview of the production quality compiler-compiler project. *Computer*, 13(8):38–49, 1980.

[35] Hung-Yi Liu and Luca P Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th annual design automation conference*, pages 1–7, 2013.

[36] MLIR EmitC, 2023.

[37] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59. IEEE, 2021.

[38] 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. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

[39] Charles Gregory Nelson. *Techniques for program verification.* PhD thesis, Stanford University, 1980.

[40] Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. Flowtune: End-to-end automatic logic optimization exploration via domain-specific multi-armed bandit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[41] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. *ACM SIGPLAN Notices*, 51(5):21–30, 2016.

[42] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[43] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[44] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.

[45] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–31, 2021.

[46] Guy Steele. *Common LISP: the language.* Elsevier, 1990.

[47] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *Proceedings of the 23rd international conference on Computer Aided Verification*, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] Stratus High-Level Synthesis, 2023.

[49] Synopsys HECTOR, 2023.

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

[51] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. Impress: Large integer multiplication expression rewriting for fpga hls. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–10. IEEE, 2022.

[52] Ecenur Ustun, Cunxi Yu, and Zhiru Zhang. Equality Saturation for Datapath Synthesis: A Pathway to Pareto Optimality.

[53] 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*, pages 874–886, 2021.

[54] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proceedings of the VLDB Endowment*, 13(11), 2020.

[55] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[56] Xilinx Vitis HLS, 2023.

[57] Ruifan Xu, Youwei Xiao, Jin Luo, and Yun Liang. HECTOR: A Multi-Level Intermediate Representation for Hardware Synthesis Methodologies. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, 2021.

[59] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755, 2022.

[60] Qian Zhang, Jiyuan Wang, Guoqing Harry Xu, and Miryung Kim. Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1017–1029, 2022.

[61] Zhiru Zhang and Bin Liu. Sdc-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 211–218. IEEE, 2013.

[62] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A. Constantinides. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 235–242, 2022.