DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog

Abd-El-Aziz Zayed
McGill University
Montreal, Canada
abd-el-aziz.zayed@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
Montreal, Canada
christophe.dubach@mcgill.ca

Abstract

MLIR's ability to optimize programs at multiple levels of abstraction is key to enabling domain-specific optimizing compilers. However, expressing optimizations remains tedious. Optimizations can interact in unexpected ways, making it hard to unleash full performance.

Equality saturation promises to solve these challenges. First, it simplifies the expression of optimizations using rewrite rules. Secondly, it considers all possible optimization interactions, through saturation, selecting the best program variant. Despite these advantages, equality saturation remains absent from production compilers such as MLIR.

This paper proposes to integrate Egglog, a recent equality saturation engine, with MLIR, in a *dialect-agnostic* manner. This paper shows how the main MLIR constructs such as operations, types or attributes can be modeled in Egglog. It also presents DialEgg, a tool that pre-defines a large set of common MLIR constructs in Egglog and automatically translates between the MLIR and Egglog program representations. This paper uses a few use cases to demonstrate the potential for combining equality saturation and MLIR.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: MLIR, Equality Saturation, Egg, Egglog

ACM Reference Format:
Abd-El-Aziz Zayed and Christophe Dubach. 2025. DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708957

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708957

1 Introduction

Since the early days of Fortran, to today's high-level programming languages, compiler optimizations have driven the adoption of compiler technology [1]. In the C/C++ world, LLVM [4] has pushed compiler technology forward with its modular approach to optimization. This modularity stems from the design of the LLVM IR (Intermediate Representation) which offers a *single* abstraction to represent programs. While this single abstraction enables code reusability, it has become a limitation in the era of domain-specific hardware and language which requires domain-specific abstractions.

Building upon LLVM's success, MLIR [5] represents and transforms programs at multiple abstraction levels. MLIR allows high-level operations such as matrix multiplication or convolutions, mid-level loop structures, and low-level hardware-specific instructions to coexist in the same IR. This is achieved by introducing the concept of a dialect, a set of operations and types that are specific to a domain (*e.g.*, linear algebra, GPU (Graphics Processing Unit) programming).

However, optimizing MLIR programs is challenging. Optimization passes at different levels of abstraction may interact in unpredictable ways from a performance point of view. Optimization decisions are typically done locally with heuristics, without consideration for global optimally. Furthermore, determining the optimal sequence of optimizations — the phase ordering problem — is a well-known challenge.

Equality saturation [11] is a recent novel approach to optimization. It represents all equivalent programs simultaneously in an e-graph. Optimizations are expressed as rewrite rules and are applied to the e-graph until a fixed point is reached. The globally optimal program variant is extracted using a cost model, bypassing the phase ordering problem.

Equality saturation has recently matured with the availability of the high-performance Egg library [13] and the Egglog [16] DSL (Domain-Specific Language) and system. Egglog combines a high-performance equality saturation engine with the ability to express rewrites in a subset of Datalog. As we will see, Egglog makes it particularly suitable to express compiler optimizations in a declarative way.

This paper presents DialEgg, an MLIR optimizer that is dialect-agnostic and extensible. DialEgg combines MLIR with Egglog in a unified tool. DialEgg discovers globally optimal programs with equality saturation. Using DialEgg, analysis and optimizations are expressible concisely with Egglog and are applicable to any MLIR program in any dialect.

The paper describes how different MLIR concepts, such as operations, types, and attributes are expressed in Egglog. It also shows how to express customized cost models in Egglog, leveraging MLIR type information, such as tensor dimensions. Four use cases are presented, illustrating DialEgg's ability to express and apply optimizations that span multiple abstraction levels, dialects, and IR components within MLIR.

The paper evaluates DialEgg's performance on a small suite of benchmarks based on the use cases. It compares DialEgg-optimized programs against programs optimized by a traditional MLIR optimization pass. The evaluation shows that DialEgg handles a variety of use cases, with performance on par with the classical MLIR optimization pipeline.

To summarize the contributions, this paper:

1. Presents an open source tool that combines MLIR with equality saturation in a dialect-agnostic way;
2. Shows how MLIR constructs are expressed in Egglog;
3. Shows how optimizations and custom cost models are simply expressed with Egglog using MLIR operations;
4. Present experimental evidence that DialEgg is able to optimize, demonstrating the potential for this tool.

2 Background

2.1 MLIR

MLIR [5] represents programs at multiple levels of abstraction. MLIR allows high-level operations such as matrix multiplication, mid-level loop structures, and low-level hardware-specific instructions to coexist in the same IR. MLIR operations are grouped into domain-specific dialects with passes that optimize within a dialect or translate between dialects.

2.1.1 MLIR Operations. MLIR represents all constructs as operations, from high-level modules and functions to lower-level computations and control flow elements such as loops and if-then-else statements. All operations have a name, input operands, output results, attributes, and regions.

Inputs and outputs An operation takes operands as input and can produce several results as output. Both operands and results are typed values in SSA (Static Single-Assignment) form. MLIR uses a static strong type system, defining the type of all operation inputs and outputs. MLIR supports both built-in types (*e.g.,* integers, floats) and user-defined types.

Attributes Attributes are compile-time properties attached to operations. They are typed and represent constants, names, or other metadata that influence the behavior of operations.

Regions and blocks Operations can have multiple regions, each representing a list of blocks. Each block contains a list of operations. This recursive structure enables the representation of control flow and nested computations.

```
1  func.func @classic(%a: i64) -> i64 {
2    %c2 = arith.constant 2 : i64
3    %a2 = arith.muli %a, %c2 : i64
4    %a_2 = arith.divsi %a2, %c2 : i64
5    func.return %a_2 : i64 }
```

Listing 1. MLIR implementation of $(a \cdot 2)/2$

2.1.2 MLIR Dialects. MLIR's extensibility is primarily achieved through its dialects, collections of related operations, types, and attributes. The *builtin* dialect provides fundamental types (*e.g.,* integers, floats) and attributes (*e.g.,* int attributes, string attributes, array attributes); The *arith* dialect focuses on arithmetic operations such as addition, bitshifts, casts, and comparisons; *Math* focuses on mathematical operations such as trigonometric functions, logarithms, and exponentials; *Linalg* represents linear algebra operations and transformations; And *scf* represents structured control flow such as loops and if-then-else statements.

2.1.3 MLIR Passes. MLIR passes are applied to MLIR programs to perform analysis, optimization, or lowering. An important pass in MLIR is the canonicalization pass which transforms the IR into a standardized form. Canonicalization also simplifies the IR by applying algebraic simplifications, constant folding, and other dialect-specific normalizations.

2.1.4 MLIR Example. We now look at an example of how a function that computes $(a \cdot 2)/2$ is represented in MLIR using the *func* and *arith* dialects shown in listing 1. The `func.func` operation represents the function. It has two attributes, the function type and the symbol name, and one region with one block. The block has one argument, `a`, of type `i64`. Line 2 shows a constant, where the value 2 is actually represented as an attribute in MLIR attached to the operation. The rest of the listing should be self-explanatory.

2.2 Equality Saturation

Equality saturation [11], is a decade-old approach to program optimization that addresses the phase ordering problem.

E-graph Equality saturation builds a compact representation of all equivalent programs simultaneously, called an e-graph. An e-graph consists of a set of e-classes which contain e-nodes. All e-nodes in the same e-class are equivalent.

Saturation and rewrite rules The program e-graph is saturated using a fixed-point algorithm that applies rewrite rules. A rewrite rule specifies an equivalence between two patterns of e-nodes. For example, $a \cdot 2 \Leftrightarrow a \ll 1$ is a rewrite rule specifying an equivalence between an e-node representing a multiplication and an e-node representing a left-shift.

Extraction and cost model Once saturated, the best expression is extracted from the e-graph using a cost model, also called the profitability heuristic. The process starts from the root node in the e-graph. Each e-class selects the lower-cost e-node and proceeds recursively with the children.

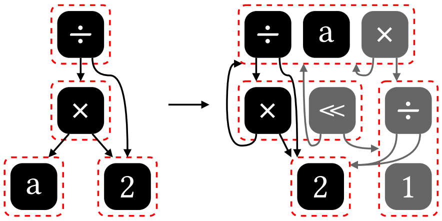

Figure 1. Example of e-graph before and after saturation. The lighter e-nodes are added after saturation.

Example We now look at an example of applying equality saturation to $(a \cdot 2)/2$ using the following rewrite rules:

$$x/x \Rightarrow 1 \qquad x \cdot 2 \Leftrightarrow x \ll 1$$

$$x \cdot 1 \Rightarrow x \qquad (x \cdot y)/z \Leftrightarrow x \cdot (y/z)$$

After saturation, the e-graph in fig. 1 shows the original expression is equivalent to a, but also to $(a \ll 1)/2$ among other alternatives. If the cost model minimizes the number of operations, the extracted "optimal" program would be a.

2.3 Egglog

Egglog [16] is a recent DSL combining the declarative nature of Datalog with equality saturation. It can be used to express e-node types and rewrite rules and also provides a rich set of built-in commands for defining and manipulating e-graphs.

2.3.1 Egglog Constructs.

Datatypes An e-node type is defined using the `sort` command. This defines a sum type, where each variant represents a different kind of e-node, expressed with the `function` command and the e-node type as the last argument (`Expr`). The cost can be set for each variant type, as seen below.

The datatypes for a simple arithmetic language consisting of constants, variables, and operations could be defined as:

```
(sort Expr)
(function Num (i64) Expr        :cost 1)
(function Var (String) Expr     :cost 1)
(function Add (Expr Expr) Expr :cost 1)
(function Mul (Expr Expr) Expr :cost 2)
(function Div (Expr Expr) Expr :cost 2)
(function Shl (Expr Expr) Expr :cost 1)
```

The expression $(a \cdot 2)/2$ seen earlier, would then be expressed in Egglog using a let-binding as:

```
(let expr (Div (Mul(Var "a")(Num 2)) (Num 2)))
```

Rewrite rules Rewrite rules express equivalence and are defined using the `rewrite` command. The rewrite rules seen earlier are expressed in Egglog as:

```
(rewrite (Div ?x ?x) (Num 1)) ; x / x => 1
(rewrite (Mul ?x (Num 1)) ?x) ; x * 1 => x
(rewrite ; x * 2 => x << 1
  (Mul ?x (Num 2)) (Shl ?x (Num 1)) )
(rewrite ; (x * y) / z => x * (y / z)
  (Div (Mul ?x ?y) ?z) (Mul ?x (Div ?y ?z)))
```

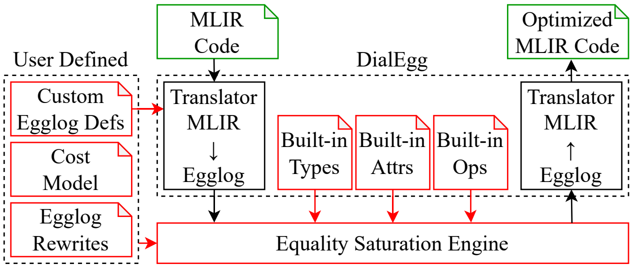

Figure 2. Overview of the DialEgg architecture.

The ? represents variables in Egglog, which can appear in the pattern to be matched (first argument of `rewrite`) or the newly rewritten term (second argument of `rewrite`).

Primitives Egglog has built-in support for types such as integers, floats, or strings. These are represented as `i64`, `f64`, and `String` respectively. Each type supports primitive operations, such as addition, division, and comparisons.

3 DialEgg Overview

DialEgg bridges the gap between MLIR's extensible, multi-level abstractions, and Egglog's expressive equality saturation engine. The contributions of this paper are twofold. First, it presents a dialect-agnostic methodology to represent and optimize MLIR constructs within the Egglog DSL. Secondly, DialEgg provides pre-defined built-in constructs and automatic translation between MLIR and Egglog representations.

The architecture of DialEgg is illustrated in fig. 2. DialEgg takes in MLIR code, translates it to Egglog, applies user-defined optimizations in Egglog, and then translates the optimized representation back to MLIR. DialEgg uses the expressiveness of Egglog for defining complex optimizations as rewrites while maintaining compatibility with MLIR.

Pre-defined constructs DialEgg comes with Egglog representations for commonly used MLIR types and attributes from the *builtin* dialect. These constructs can be combined with user-defined ones by the Egglog saturation engine.

User-defined constructs When integrating a new MLIR dialect with DialEgg, three major steps are required. First, the user needs to define the MLIR constructs needed in Egglog if not already provided. This information is used by both the translation process and the saturation engine. Next, the user can define a custom cost model in Egglog for every MLIR operation, which will guide the optimization process. Finally, the user defines optimizations as Egglog rewrite rules, specifying how MLIR constructs should be transformed.

Translation layer This layer handles the bidirectional translation between MLIR and Egglog representations. The translation preserves all relevant structures of the MLIR code in the Egglog representation. Operations not explicitly represented in Egglog are treated as black boxes. While these will not be rewritten, they will "survive" the optimization process and re-appear in the transformed MLIR code.

```
(sort Type)
(function F32  () Type) ; no argument
(function I64  () Type) ; no argument
(function None () Type) ; no argument
(function RankedTensor   (IntVec Type)   Type)
(function UnrankedTensor (Type)          Type)
(function OpaqueType     (String String) Type)
```

Listing 2. Example of MLIR Types expressed in Egglog.

By integrating all these components, DialEgg provides a framework for MLIR optimization that combines the flexibility of MLIR with the optimization capabilities of Egglog.

4 Representing MLIR Constructs in Egglog

When using DialEgg, the user first needs to create the Egglog data structures corresponding to the MLIR dialects of interest. This includes defining any types, attributes, and operations. Blocks and regions are automatically handled by DialEgg.

4.1 Types

Pre-defined types DialEgg comes with a representation of pre-defined MLIR types in Egglog. This includes most of the MLIR types from the MLIR's *builtin* dialect, as well as common types such as integer types, floating point types, tuple types, complex number types, and tensor types.

All the types are simply represented in Egglog as variants of the *Type* datatype, as illustrated in listing 2. Note that the type definitions only need to be done once per MLIR dialect.

A 32-bit float, for instance, would be represented as `(F32)`, while a ranked tensor of 64-bit integers with dimensions 2x3 would be expressed as follows using the Egglog `vec-of` primitive which produces a vector of integer (`IntVec`):

```
(RankedTensor (vec-of 2 3) (I64))
```

User-provided types Users can add new types in Egglog if required, by adding another type variant, using the same type name in Egglog and MLIR. This ensures that the DialEgg translation between MLIR and Egglog is able to associate each Egglog type with the corresponding MLIR type.

Opaque types In case an MLIR type does not have a corresponding Egglog type declared, it will be automatically translated to the `OpaqueType` variant. The first String is a serialized representation of the MLIR type, while the second String is its name. This information is used to rebuild the MLIR type during translation from Egglog to MLIR, although it should be used sparingly to maintain expressiveness.

The advantage of using pre-defined or user-defined types is that they can make the rewrites aware of the MLIR types. For instance, one could express the fact that only integer addition is commutative (as opposed to floating point):

```
(rewrite (Add (Add ?x ?y (I64)) ?z (I64))
         (Add ?x (Add ?y ?z (I64)) (I64)))
```

```
 1 (sort Attr)
 2 (function IntegerAttr (i64 Type) Attr)
 3 (function FloatAttr (f64 Type) Attr)
 4 (function DenseIntArrayAttr(i64 IntVec Type) Attr)
 5 (function DenseFPElementsAttr(FloatVec Type) Attr)
 6 (function SymbolRefAttr (String) Attr)
 7 (function OpaqueAttr (String String) Attr)
 8 (datatype AttrPair (NamedAttr String Attr))
 9 (datatype FastMathFlags (none) (reassoc) (nnan)
       (ninf) (nsz) (arcp) (contract) (afn) (fast))
10 (function arith_fastmath (FastMathFlags) Attr)
```

Listing 3. Example of MLIR Attributes encoded in Egglog.

```
(sort Op)
(function arith_constant (AttrPair Type) Op)
(function arith_addf (Op Op AttrPair Type) Op)
(function arith_addi (Op Op AttrPair Type) Op)
(function math_sin (Op AttrPair Type) Op)
(function math_powf (Op Op AttrPair Type) Op)
(function tensor_empty (Type) Op)
(function linalg_matmul (Op Op Op Type) Op)
(function func_call (AttrPair Type) Op)
```

Listing 4. Example of MLIR Operations in encoded in Egglog

4.2 Attributes

DialEgg already provides many of the attributes defined in the *builtin* dialect, with a few examples shown in listing 3. Opaque attributes are available as well. Similar to the types, MLIR attributes are represented as variants of the *Attr* datatype. Users can either use the pre-defined attributes or add new ones. For instance, an int attribute with constant 1 is represented as `(IntegerAttr 1 (I64))`.

Lines 9 and 10 of listing 3 present the `arith_fastmath` attribute, a custom attribute defined in the *arith* dialect. The `arith_fastmath` attribute takes a `FastMathFlags` variant as a parameter, which is a direct translation of MLIR enum `FastMathFlags` (sum types can also use the `datatype` keyword in Egglog). Depending on the value, more aggressive optimizations can be triggered for floating point operations.

Named attributes MLIR operations do not use attributes on their own, they store a list of named attributes, as discussed in section 2. Below is an instantiation of the named attribute `{value = 1}` expressed in Egglog:

```
(NamedAttr "value" (IntegerAttr 1 (I64)))
```

4.3 Operations and Values

MLIR operations are represented as variants of *Op* datatype. Listing 4 contains examples of some MLIR operations. DialEgg already comes equipped with many operations pre-defined. The user can extend this set by adding extra variants.

The name of each variant starts with the dialect name followed by the operation name. The list of parameters for each *Op* in Egglog must match the structure of corresponding MLIR operations to enable automatic translation.

Operations with variadic operands For operations with a variable number of operands, such as the MLIR *func.call*, DialEgg allows defining multiple variants with different numbers of parameters. The number of parameters is appended to the end of the variant name. For instance:

```
(function func_call_0(AttrPair Type) Op)
(function func_call_3(Op Op Op AttrPair Type) Op)
```

Values Values can be either the result of an operation, or a block argument. MLIR values in DialEgg are represented as the *Value* variant of the *Op* datatype, as illustrated below:

```
(function Value (i64 Type) Op) ; Value(id,type)
```

The *id* parameter is a unique identifier for the value, and the *type* parameter is the type of the value.

Opaque operations MLIR operations that are not defined in DialEgg become opaque operations. Opaque operations are represented automatically as a *Value* variant. This enables users of DialEgg to ignore irrelevant operations, while still allowing Egglog to optimize around them. This is a key feature of DialEgg which allows a user to only define the Egglog constructs related to the dialect of interest, with any undefined MLIR construct simply being mapped to an opaque representation.

4.4 Blocks and Regions

Blocks and regions are fundamental constructs in MLIR that represent hierarchical structure. DialEgg provides a representation for these in Egglog that maintains their semantics.

In Egglog, regions are ordered lists of blocks, and blocks are ordered lists of operations, as seen below. Block arguments are represented using the *Value* variant, as seen earlier.

```
(sort OpVec (Vec Op)) ; Vector of Operations
(datatype Block (Blk OpVec)) ; Block
(sort BlockVec (Vec Block)) ; Vector of Blocks
(datatype Region (Reg BlockVec)) ; Region
```

5 Translation Between MLIR and Egglog

The translation layer in DialEgg is a core component that enables the integration of MLIR and Egglog. This section details the bidirectional translation process, explaining how MLIR constructs are represented in Egglog and vice versa.

5.1 Preparation Phase

DialEgg first processes the pre-defined and user-defined Egglog files where it identifies all MLIR construct declarations present in the file. For each construct, DialEgg registers it and records essential information such as the expected number of operands, attributes, regions, and associated cost.

5.2 Support for Custom Types and Attributes

DialEgg provides a flexible, semi-automatic, mechanism for handling the rare cases where custom attributes and types are needed. At the time of writing, the user must implement two small C++ functions for each custom type or attribute: an eggifier and a de-eggifier. The eggifier function is responsible for printing the Egglog representation of an MLIR type or attribute. Conversely, the de-eggifier function parses the Egglog representation and builds the corresponding MLIR constructs. In the near future, DialEgg will be improved to support the automatic generation of the eggifier and de-eggifier function.

5.3 Translation Between MLIR and Egglog

Translating MLIR operations to Egglog involves traversing the MLIR program structure recursively, depth-first, to create the Egglog counterparts. MLIR constructs that do not have any declared Egglog counterparts are automatically treated as opaque constructs as explained earlier. Custom types and attributes make use of the user-provided eggifier function.

The semantics of the MLIR program is preserved by the translation. Thanks to the SSA form of MLIR, any definition of an SSA value will be translated into a let-binding in Egglog. This is the classical way to model SSA values in functional languages. Once the e-graph is constructed, equivalent expressions are unified into the same e-nodes, and let-bindings simply disappear. For opaque operations, DialEgg assigned a unique identifier to each of them as shown in section 4.3. The identifier is exposed in Egglog, ensuring every opaque expression is treated as distinct e-nodes in the e-graph.

After saturation, DialEgg translates the optimized Egglog representation back into valid MLIR code. The translator parses the Egglog code, systematically converting Egglog constructs back into their MLIR equivalents. The process heavily relies on the information gathered during the initial preparation phase and the de-eggifier functions for the custom types and attributes.

The semantics is preserved by the translation. E-nodes that appear multiple times in the extracted expression are turned into a single SSA value definition with multiple uses.

5.4 Example

We now show an example of computing $\sqrt{|x|}$ in MLIR and its Egglog translation. The MLIR code mixes four dialects: *func*, *arith*, *scf*, and *math*, and uses values, operations, attributes, types, and regions:

```
func.func @sqrt_abs(%x: f32) -> f32 {
  %zero = arith.constant 0.0 : f32
  %cond = arith.cmpf oge, %x, %zero : f32
  %sqrt = scf.if %cond -> (f32) {
    %sqrt = math.sqrt %x fastmath<fast> : f32
    scf.yield %sqrt : f32
  } else {
    %neg = arith.negf %x : f32
    %sqrt = math.sqrt %neg : f32
    scf.yield %sqrt : f32
  }
  func.return %sqrt : f32 }
```

The corresponding translation to Egglog is:

```
1  (let fmnone (NamedAttr "fastmath"
       (arith_fastmath (none))))
2  (let fmfast (NamedAttr "fastmath"
       (arith_fastmath (fast))))
3  (let op0 (Value 0 (F32))) ; argument 'x'
4  (let op1 (arith_constant (NamedAttr "value"
       (FloatAttr 0.0 (F32))) (F32)))
5  (let op2 (arith_cmpf op0 op1 fmnone (NamedAttr
       "predicate" (IntegerAttr 3 (I64))) (I1)))
6  (let op3 (math_sqrt op0 fmfast (F32)))
7  (let op4 (scf_yield op3))
8  (let op5 (arith_negf op0 fmnone (F32)))
9  (let op6 (math_sqrt op4 fmnone (F32)))
10 (let op7 (scf_yield op6))
11 (let op8 (scf_if op2
12   (Reg (vec-of ; then branch
13     (Blk (vec-of op3 op4))))
14   (Reg (vec-of ; else branch
15     (Blk (vec-of op5 op6 op7))))
16   (F32)))
17 (let op9 (func_return op8))
```

Lines 1–2 define variants of the attribute `arith_fastmath`. Line 3 corresponds to the function argument `%x`: argument 0. Line 4 corresponds to the `arith_constant` op on line 2 of the MLIR code. The value of the constant is a named attribute, translated just as discussed is section 4.2. Line 5 corresponds to the `arith_cmpf` op on line 2 of the MLIR code. This operation takes as input 2 operands and 2 named attributes. The second named attribute defines the comparison predicate, 3 stands for the ordinary greater than (oge) predicate.

Line 6 corresponds to the `math_sqrt` op on line 5 of the MLIR code. Line 7 corresponds to the `scf_yield` op on line 6 of the MLIR code. Line 8 corresponds to the `arith_negf` op on line 8 of the MLIR code. Line 9 corresponds to `math_sqrt` op on line 9 of the MLIR code.

Lines 11–16 corresponds to the `scf_if` op on lines 4–11 of the MLIR code. This operation has 2 regions, one for the `if` and another for the `else`. Each region has a single block, which in turn has a list of its operations. Line 17 corresponds to the return on the last line of the MLIR code.

6 Cost Model in Egglog for MLIR Dialects

Having seen how the MLIR constructs are represented and translated into Egglog, we now turn our attention to how a user specifies a cost model. The cost model plays a crucial role in guiding the optimization process with equality saturation.

6.1 Fixed Cost

Egglog makes it easy to define a direct fixed cost to specific MLIR operations. The following example assigns different costs to division and right shift operations:

```
(function arith_divsi (Op Op Type) Op :cost 2)
(function arith_shrsi (Op Op Type) Op :cost 1)
```

```
(rule
((linalg_matmul ?x ?y ?xy (RankedTensor ?d ?t))
 (= ?a (nrows (type-of ?x)))
 (= ?b (ncols (type-of ?x)))
 (= ?c (ncols (type-of ?y))))
((unstable-cost
  (linalg_matmul ?x ?y ?xy (RankedTensor ?d ?t))
  (* (* ?a ?b) ?c))))
```

Listing 5. Matrix multiplication cost model

```
(function nrows (Type) i64)
(function ncols (Type) i64)
(rule ; if a tensor, define nrows and ncols
  (= ?t (RankedTensor ?shape ?))
  ((set (nrows ?t) (vec-get ?shape 0))
   (set (ncols ?t) (vec-get ?shape 1))))
```

Listing 6. Dimension analysis helper functions

This example defines that an MLIR signed integer division operation (`arith.divsi`) has a cost of 2, while a right shift operation (`arith.shrsi`) has a cost of 1. By assigning these costs, Egglog will prefer to extract expressions from the e-graph with right shifts rather than division whenever possible. If no cost is assigned, a default cost of 1 is assumed.

6.2 Variable Cost

Sometimes, the cost of an operation might depend on the input data shape. For instance, the cost of matrix multiplication depends on the number of rows and columns.

Listing 5 defines in Egglog the cost of matrix multiplication based on the number of scalar multiplications performed. Two helper functions, `nrows` and `ncols`, extract the row and column count and are explained later. The `type-of` helper function returns the operation's type.

The cost in listing 5 is computed with a rule matching on a `linalg_matmul` e-node in the e-graph, which then sets the cost of the e-node with the `unstable-cost` command, depending on some parameters. `unstable-cost` is a new command specifically added into Egglog to support the use-case presented here. During extraction, Egglog will use this information to establish whether this `matmul` operation has a lower cost than any other in the equivalence class.

Dimension analysis Listing 6 shows two helper functions, `nrows` and `ncols` to return the dimensions of tensor types. The rule here matches for all `RankedTensor` types and defines `nrows` and `ncols` as the number of rows and columns in the tensor shape, respectively.

By using these flexible cost modeling capabilities, DialEgg enables sophisticated optimization strategies that can adapt to different optimization goals, hardware targets, and program characteristics. This cost-aware approach allows DialEgg to make informed decisions throughout the optimization process, potentially leading to more efficient code compared to traditional, fixed-sequence optimization passes.

```
1  (rule ((= ?lhs (arith.divsi ?x (arith.constant
       (NamedAttr "value"(IntegerAttr ?n ?t)) ?t)
       ?t)) ; match x / n
2     (= ?k (log2 ?n ))  ; set k = log2(n)
3     (= ?n (<< 1 ?k ))) ; check n = 2^k
4    ((union ?lhs (arith.shrsi ?x (arith.constant
         (NamedAttr "value" (IntegerAttr ?k ?t))
         ?t) ?t))))
```

Listing 7. Division by a power of 2 is equivalent to a right shift by the `log2` of the divisor.

7 Case Study of Optimizations with Egglog

Having seen how a user expresses custom cost models, this section now looks at expressing optimizations for MLIR using rewrite rules in Egglog. It presents one toy examples and four case studies in increasing order of complexity.

7.1 Constant Folding

Constant folding is a simple optimization that is standard in most compilers. Consider the following MLIR code which exhibits a constant addition that can be folded:

```
%c2  = arith.constant 2    : i32
%c3  = arith.constant 3    : i32
%sum = arith.addi %c2, %c3 : i32
```

The following Egglog rewrite rule expresses constant folding:

```
(rewrite
 (arith_addi ; x + y
  (arith_constant ; x
   (NamedAttr "value"(IntegerAttr ?x ?t))?t) ?t)
  (arith_constant ; y
   (NamedAttr "value"(IntegerAttr ?y ?t))?t) ?t)
 (arith_constant ; evaluate x+y into a constant
  (NamedAttr "value" (IntegerAttr (+ ?x ?y) ?t))
  ?t))
```

This rule matches any addition of two integer constants, whose value are stored as an attribute in MLIR, and replaces it with a new constant representing their sum (using the highlighted + Egglog primitive). After applying this rule, the optimized MLIR code would look like this:

```
%sum = arith.constant 5 : i32
```

While this example is straightforward, it demonstrates the ability to perform local optimizations through rewriting.

7.2 Conditional Rewrite Rules

Conditional rewrite rules in DialEgg allow for context-sensitive optimizations, enabling more precise and targeted optimizations. Using the cost model defined in section 6.1, we now show a conditional rewrite rule that optimizes integer division by a power of 2 into a more efficient bitwise right shift operation. This rewrite rule is shown in listing 7.

This rewrite is more complex than others since it needs to be conditionally applied and perform computations. Line 1 matches for a signed integer division of x by a constant n.

```
1 (let fattr
2  (NamedAttr "fastmath" (arith.fastmath (fast))))
3 (rewrite
4  (arith.divf ; 1.0 / sqrt(x)
5   (arith.constant  ; cst 1.0
6    (NamedAttr "value" (FloatAttr 1.0(F32)))(F32))
7   (math.sqrt ?x fattr (F32)) ; sqrt(x)
8   fattr (F32))
9  (func_call ?x (NamedAttr "callee"
      (SymbolRefAttr "fast_inv_sqrt")) (F32)))
```

Listing 8. Asserting equivalence of an inverse square root operation and the fast inverse square root function

Line 2 computes the log2 of the divisor $k = log_2 x$ and assigns it to k. Line 3 checks if n is a power of 2 and the rule will only be applied if this is the case. Line 4 declares the expression on line 1 is equivalent to a right shift (`arith.shrsi`) by k by unionizing them in the e-graph. A union is an Egglog builtin assigning both expressions into the same equivalence class.

This rule demonstrates both conditional application (only for powers of 2) and computation within the rule (calculating log2). The following MLIR code

```
%c256 = arith.constant 256 : i64
%result = arith.divsi %x, %c256 : i64
```

would be transformed as follows thanks to this rule.

```
%c8 = arith.constant 8 : i64
%result = arith.shrsi %x, %c8 : i64
```

7.3 Attribute-Based Matching and Rewriting

Attribute-based matching is a feature that enables optimization decisions based on the attributes of MLIR operations that are explicitly exposed in Egglog by DialEgg. This is particularly useful for applying specialized optimizations when certain conditions or flags are set. This case study demonstrates how DialEgg uses attribute matching to apply the fast inverse square root approximation, a technique famously used in the Quake III Arena game engine.

The rewrite rule expressing this optimization is shown in listing 8. Lines 1–2 define the *fastmath<fast>* attribute that we will look for in the e-graph. Lines 3–8 look for the pattern $1/\sqrt{x}$ where both the division and the square root operations use the *fastmath<fast>* attribute. Line 9 rewrites $1/\sqrt{x}$ as a call to the fast inverse square root function.

Given the following MLIR code:

```
%c1 = arith.constant 1.0 : f32
%dist = math.sqrt %x fastmath<fast> : f32
%inv_dist = arith.divf %c1,%dist fastmath<fast>
```

DialEgg would then transform it to:

```
%inv_dist = func.call @fast_inv_sqrt(%x)
```

This optimization demonstrates how attributes can be used to apply domain-specific optimizations.

```
1  (rule
2   ((= ?lhs (linalg_matmul
3               (linalg_matmul ?x ?y ?xy ?xy_t)
4               ?z ?xy_z ?xyz_t))
5    (= ?b (nrows (type-of ?y)))
6    (= ?d (ncols (type-of ?z)))
7    (= ?xyz_t (RankedTensor ?d1 ?t)))
8   ((let yz_t (RankedTensor (vec-of ?b ?d) ?t))
9    (union ?lhs
10    (linalg_matmul ?x (linalg_matmul ?y ?z
11      (tensor_empty yz_t) yz_t) ?xy_z ?xyz_t))))
```

Listing 9. Associativity of matrix multiplication

7.4 Type-Based Cost Model

Type-based cost models allow DialEgg to make optimization decisions based on the operand's type. This is useful for operations where the cost depends on the data shape. This case study shows how DialEgg can take advantage of the associativity of matrix multiplication. The cost of performing a matrix multiplication can be expressed in the cost model using type information as shown earlier in listing 5.

Listing 9 encodes of matrix multiplication associativity in Egglog. Lines 2–4 matches for the form $(XY)Z$. Line 8 creates the output type using the result of the two helper functions `nrows` and `ncols` seen earlier. Lines 9–11 unionize the expressions $(XY)Z$ and $X(YZ)$ in the e-graph.

The computational cost of matrix multiplications varies depending on associativity. Consider the matrices X with size $a \times b$, Y with size $b \times c$, and Z with size $c \times d$. The algorithmic complexity of multiplying X and Y is $O(a \cdot b \cdot c)$ number of scalar multiplications, as seen in listing 5. The algorithmic complexity of $(XY)Z$ is $O(a \cdot b \cdot c + b \cdot c \cdot d)$ while the algorithmic complexity of $X(YZ)$ is $O(b \cdot c \cdot d + a \cdot b \cdot d)$.

Using the cost model for matrix multiplication, and the associativity rule above, Egglog is able to find the best way to group matrix multiplication to lower the algorithmic complexity, depending on the shapes. Given the following MLIR code which performs 270, 000 multiplications:

```
%xy = linalg.matmul ins(%x, %y:
      tensor<100x10xi64>, tensor<10x150xi64>)
      outs(%xy_init) -> tensor<100x150xi64>
%xyz = linalg.matmul ins(%xy, %z:
      tensor<100x150xi64>, tensor<150x8xi64>)
      outs(%xyz_init) -> tensor<100x8xi64>
```

DialEgg will produce the following transformed program:

```
%yz = linalg.matmul ins(%y, %z:
      tensor<10x150xi64>, tensor<150x8xi64>)
      outs(%yz_init) -> tensor<10x8xi64>
%xyz = linalg.matmul ins(%x, %yz:
      tensor<100x10xi64>, tensor<10x8xi64>)
      outs(%xyz_init) -> tensor<100x8xi64>
```

which only requires 20, 000 multiplications! This is achieved thanks to the ability to encode in Egglog a single associativity rule, and a cost model for matrix multiplication operations.

7.5 Multiple Rules and Recursive Rewrite Rules

The real power of equality saturation lies in its ability to consider all possible transformations at once. In addition, rules can be made recursive, enabling the handling of nested structures. In this case study, we demonstrate a set of rules, including one recursive, for optimizing polynomial evaluation with Horner's method.

Horner's method is an algorithm for efficient polynomial evaluation. For a polynomial of degree n: $P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$. Horner's method rewrites it as: $P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)))$. This requires only n multiplications and n additions, compared to $n(n+1)/2$ multiplications and n additions in the naive approach.

Rules To implement Horner's method in Egglog, we need a set of rules that work together to transform polynomial expressions. These rules are presented in listings 10 to 12. Listing 12 implements the commutative, associative, and distributive properties of addition and multiplication. These rules allow DialEgg to rearrange terms, which is crucial for grouping like terms in the polynomial. The distributive rule in particular factors out common terms, creating the nested structure of Horner's form. Listing 10 defines exponentiation recursively. This rule is crucial for breaking down high-degree terms in polynomials into lower-degree ones. Finally, the identity rules (listing 11) will allow equality saturation to simplify the resulting expressions.

Cost model Listing 13 sets the cost of exponentiation much larger than the cost of multiplication, which is in turn much larger than the cost of addition. This cost function plays a crucial role in guiding the optimization process. Egglog is incentivized to prefer solutions that minimize the use of these expensive operations. This aligns perfectly with the goal of Horner's method, which reduces the number of multiplications and eliminates exponentiation operations.

MLIR code example Equipped with all these rules, DialEgg will turn the following MLIR code, which represents the polynomial $c + bx + ax^2$

```
%c2 = arith.constant 2.0  : f64
%x2 = math.powf %x, %c2   : f64
%t1 = arith.mulf %b, %x   : f64 // bx
%t2 = arith.mulf %a, %x2  : f64 // ax^2
%t3 = arith.addf %t1, %t2 : f64 // bx+ax^2
%t4 = arith.addf %c, %t3  : f64 // c+bx+ax^2
```

into this more efficient MLIR code:

```
%t0 = arith.mulf %x, %a  : f64 // ax
%t1 = arith.addf %b, %t0 : f64 // b + ax
%t2 = arith.mulf %x, %t1 : f64 // x(b + ax)
%t3 = arith.addf %c, %t2 : f64 // c+x(b+ax)
```

A compiler would be hard-pressed to implement such an optimization in MLIR, given the non-trivial number of rules that need to interact with one another. In contrast, while a bit verbose (due to MLIR's level of detail), Egglog and DialEgg offer a natural way to express this type of optimization.

```
(rule ; x^n = x * x^(n - 1)
  ((= ?lhs (math_powf ?x (arith_constant (NamedAttr "value" (FloatAttr ?n ?t)) ?t) ?t))
   (>= ?n 1.0))
  ((union ?lhs (arith_mulf ?x
      (math_powf ?x (arith_constant (NamedAttr "value" (FloatAttr (- ?n 1.0) ?t)) ?t) ?t) ?a ?t))))
```

Listing 10. Recursive definition of exponentiation.

```
(rewrite (arith_mulf ?x (arith_constant (NamedAttr "value" (FloatAttr 1.0 ?t)) ?t) ?t) ?x) ; x*1=x
(rewrite (math_powf ?x (arith_constant (NamedAttr "value" (FloatAttr 0.0 ?t)) ?t) ?t)      ; x^0=1
         (arith_constant (NamedAttr "value" (FloatAttr 1.0 ?t)) ?t))
```

Listing 11. Identity rules for floating point multiplication and power.

```
(rewrite (arith_addf ?x ?y ?a ?t) ; x+y = y+x
         (arith_addf ?y ?x ?a ?t))
(rewrite (arith_mulf ?x ?y ?a ?t) ; x*y = y*x
         (arith_mulf ?y ?x ?a ?t))
(rewrite ; (x+y)+z = x+(y+z)
  (arith_addf (arith_addf ?x ?y ?a ?t) ?z ?a ?t)
  (arith_addf ?x (arith_addf ?y ?z ?a ?t) ?a ?t))
(rewrite ; (x*y)*z = x*(y*z)
  (arith_mulf (arith_mulf ?x ?y ?a ?t) ?z ?a ?t)
  (arith_mulf ?x (arith_mulf ?y ?z ?a ?t) ?a ?t))
(rewrite ; mx + nx = x(m + n)
  (arith_addf (arith_mulf ?m ?x ?a ?t)
              (arith_mulf ?n ?x ?a ?t) ?a ?t)
  (arith_mulf ?x (arith_addf ?m ?n ?a ?t) ?a ?t))
```

Listing 12. Commutativity, associativity, and distributivity rules for floating point addition and multiplication

```
(function arith_constant   (AttrPair Type) Op)
(function arith_mulf (Op Op AttrPair Type) Op
    :cost 100)
(function arith_addf (Op Op AttrPair Type) Op)
(function math_powf  (Op Op AttrPair Type) Op
    :cost 100000)
```

Listing 13. Cost function for the pow and mult operations

Equality saturation will take care of the complex interactions required between these different transformations to generate the most cost-effective program in the end.

8 Evaluation

The evaluation aims to show that DialEgg applies to various use cases, alone or with other optimizers.

8.1 Experimental Setup

All benchmarks are run on a MacBook Pro with an Apple M1 Pro chip, running MacOS 14.5. They are expressed in MLIR, from LLVM version 18.1.4. When using DialEgg, the benchmarks are translated automatically to Egglog, optimized with equality saturation, and transformed back into MLIR code. They are then lowered to the LLVM dialect and turned into the LLVM IR. A binary is produced, using the -O3 LLVM optimization level. For each experiment, the median of eleven runs is reported, and the output is verified.

8.2 Benchmarks and MLIR Dialects Used

DialEgg is evaluated on five benchmarks which combine different MLIR dialects as shown in table 1. All benchmarks make use of *scf* for loops, *func* for functions, and *tensor* to create tensors. Some use *arith* to express arithmetic expressions and *math* for functions such as sqrt. *Linalg* is used by two benchmarks to express operations such as matrix multiplication. Using multiple dialects shows DialEgg is able to compose dialects, a key feature of MLIR. The MLIR code for each benchmark can be found in the artifact.

Converting a 4K RGB image to grayscale For each pixel, a weighted sum approximates the human eye's sensitivity to different colors: $(77 \cdot R + 150 \cdot G + 29 \cdot B)/256$. This benchmark is likely to benefit from the rewrite rule presented in listing 7 to optimize the division by a power of 2.

Vector normalization The benchmark computes the inverse of the norm of 1 million 3D vectors. It is compiled with fast-math mode, which benefits from the rewrite rule presented in listing 8 to optimize the square root.

Evaluating polynomials Evaluating polynomials can be optimized using Horner's method, as discussed in section 7.5. This benchmark iterates over $1,000,000$ 3rd-degree polynomials and evaluates each at a given point. The rewrite rules presented in listings 10 to 12 are expected to be beneficial.

Matrix multiplications (2MM and 3MM) 2MM computes $(A \cdot B) \cdot C$ while 3MM computes $((A \cdot B) \cdot C) \cdot D$. These benchmarks showcase DialEgg's ability to leverage algebraic properties expressed as rewrite rules in listing 9.

8.3 Performance Results

For each benchmark, three optimization approaches are compared against a baseline without MLIR optimizations: (1) DialEgg-optimized code, (2) MLIR canonicalization, and (3) both DialEgg and canonicalization. An extra optimization setup is used for 2MM and 3MM which will be discussed

Table 1. Benchmarks and their properties. The rightmost columns show the number of operations used from each dialect.

Benchmark	**Input size**	*scf*	*func*	*tensor*	*arith*	*math*	*linalg*
Image conversion	$3840 \times 2160 \times 3$	4	3	5	17	0	0
Vector norm	$1,000,000 \times 3$	2	7	7	27	1	0
Polynomial	$1,000,000 \times 4$	2	3	6	13	2	0
2MM	$A = 100 \times 10, B = 10 \times 150, C = 150 \times 8$	0	2	2	0	0	2
3MM	$A = 200 \times 175, B = 175 \times 250, C = 250 \times 150, D = 250 \times 10$	0	2	3	0	0	3

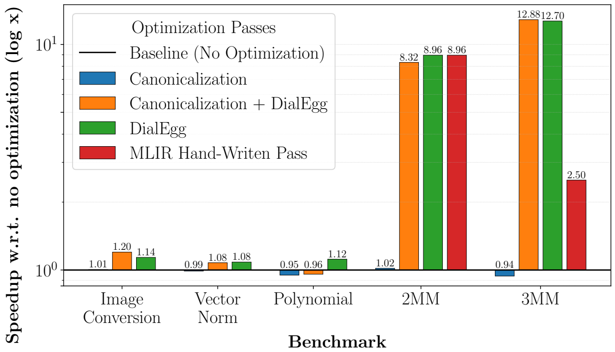

Figure 3. Speedup of DialEgg and the default canonicalization pass from MLIR. Higher is Better.

later. Canonicalization refers to the default existing optimizations applied by each dialect in the MLIR codebase. DialEgg produces results that are at least as good as those achieved by traditional optimizers, as evident from the results in fig. 3.

For image conversion, we can see that the default optimizations available (*Canonicalization*) in MLIR do not achieve any speedup. DialEgg achieves a speedup between 1.14× and 1.20× over the baseline, thanks to the optimization of the division by a power of 2. Vector norm behaves in a similar way, although with a more modest speedup of around 1.08× for DialEgg. The speedup is due to the optimization of the inverse-squared root applied by DialEgg.

For Polynomial, DialEgg achieves a speedup of 1.12× when combined with canonicalization. This is thanks to the use of Horner's method which is encoded in DialEgg.

Finally, 2MM and 3MM exhibit the largest speedup among all. The reason is that the rewrite changes the algorithmic complexity of the problems, depending on which order the matrices are multiplied. The number itself is not so meaningful since it could be made arbitrarily large by choosing more extreme matrix sizes. Nonetheless, this example illustrates precisely why having the ability to apply equality saturation on MLIR programs offers much potential for future work.

8.4 MLIR Optimization Versus Egglog Optimization

We now turn our attention to the complexity of implementing an optimization discussed in this paper without Egglog: matrix associativity. When implemented in MLIR, this optimization, expressed in 12 lines of Egglog in listing 9 requires over 120 lines of C++ code. This includes creating a new pass with an operation rewrite pattern and registering it with an existing optimization driver. This paper argues that, despite the relative verbosity of the rules, it is much simpler to express such optimization with DialEgg than in MLIR.

Furthermore, the hand-written MLIR optimization might actually lead to sub-optimal performance. For 2MM, this hand-written MLIR optimization matches the performance of DialEgg as can be seen in fig. 3. However, when used with 3MM, the optimization fails to achieve the same level of performance as DialEgg. The reason is that the hand-written MLIR optimization only operates at a *local* level in a greedy manner. It only ever considers the case of three matrices at a time when deciding how to associate the operations. In contrast, equality saturation is able to consider *all* possible associations of matrix multiplication operations, resulting in a global optimum and higher performance.

This final result demonstrates the advantage and potential for using equality saturation to express optimizations operating on MLIR operations. DialEgg offers a solution to the problem of integrating MLIR and equality saturation which will open up potential future research in this field.

8.5 Compilation Time and Scalability Study

Table 2 shows a breakdown of DialEgg's compilation time. This includes the conversion between MLIR and Egglog (through a file), the time spent in Egglog, including the time spent saturating within Egglog, and the time back from Egglog to MLIR. As can be seen, the majority of the time is spent in Egglog. The table also includes the canonicalization time in MLIR and the time taken by the hand-written MLIR pass in C++ which optimizes the matrix operations order.

A limited scalability study is shown using 3MM as a starting point, by chaining more matrix operations. As expected, saturation time increases exponentially with the number of operations. In contrast, the time to run the greedy MLIR C++ pass is linear with the number of operations as expected.

The scalability of the system is dependent on the scalability of equality saturation. In some cases, a long compile time might be acceptable, in others not. This is a well-known trade-off of equality saturation, independent of this work.

Table 2. Benchmark compilation and e-graph saturation times.

Benchmark	#Rules	#Ops	MLIR →Egglog	Egglog	Saturat.	Egglog →MLIR	Canon.	C++ MLIR Pass
Img Conv	1	29	0.3ms	14.6ms	<0.1ms	0.1ms	0.2ms	N/A
Vec Norm	1	44	0.4ms	21.6ms	<0.1ms	0.1ms	0.2ms	N/A
Poly	8	26	0.3ms	18.9ms	2ms	0.2ms	0.2ms	N/A
2MM	5	6	0.2ms	8.6ms	<0.1ms	0.1ms	0.1ms	0.1ms
3MM	5	8	0.2ms	8.7ms	1ms	0.1ms	0.1ms	0.1ms
10MM	5	22	0.2ms	14.4ms	4ms	0.3ms	0.1ms	0.2ms
20MM	5	42	0.3ms	41.3ms	23ms	0.7ms	0.2ms	0.3ms
40MM	5	82	0.4ms	296.2ms	235ms	1.4ms	0.3ms	0.6ms
80MM	5	162	0.5ms	4939.3ms	3732ms	6.8ms	0.6ms	1.3ms

9 Limitations and Discussions

MLIR interfaces support DialEgg does not currently support MLIR interfaces. However, they could be supported using a similar way attributes are handled in DialEgg.

Correctness It is the user's responsibility to define rules that do not break the program semantics. This could be delicate in the presence of opaque operations with side effects (*e.g.,* load/store). One could use a conditionally enabled rule if the opaque operation is side-effect-free using the MemoryEffectsOpInterface when support is added to DialEgg.

Cost model and analysis complexity As seen, the cost model can make use of typing information. Capturing more complex behavior, *e.g.,* cache-related performance for loop optimization, might be harder and perhaps best left to the classical MLIR pipeline using the *affine* dialect for now. However, an exciting direction could be to use the lattice operations supported by Egglog. The original Egglog paper [16] features an implementation of "points-to analysis". A similar approach could be used to express other complex analyses.

10 Related Work

The closest related work, SEER [2], automatically synthesizes efficient HLS (High-Level Synthesis) code with MLIR and the Egg [13] equality saturation library. While SEER and DialEgg both use e-graphs for optimization, SEER achieves this using the *affine* and *scf* dialects. In contrast, DialEgg is more general and applicable to any dialect. Furthermore, to the best of the authors' knowledge, SEER is unavailable publicly while DialEgg is an open-source tool publicly available.

PDL (Pattern Definition Language) is a pattern detection dialect in MLIR where patterns are represented as MLIR operations. The *transform* dialect [6] exposes fine-grained MLIR transformations, giving the user the ability to drive the optimization process. In contrast, DialEgg uses a more declarative approach where users provide equivalences in the form of rewrite rules. Equality saturation takes care of how to apply transformations automatically.

Equality saturation has also been used before with LLVM, to validate the correctness of the code produced [10]. DialEgg is a superior approach since the optimized code is guaranteed to be valid by construction — assuming no bug in the rewrites, which is the same limitation for prior work [10].

Ruler [8] uses equality saturation to infer rewrite rules, which is orthogonal to the work presented in this paper.

Szalinski [7] uses equality saturation to optimize CAD models expressed with constructive solid geometry. Several approaches [9, 14] have used equality saturation to optimize tensor programs. Diospyros [12] finds efficient vectorized code with irregular structure using equality saturation and symbolic execution. LIAR [3] uses a minimalist functional IR and equality saturation to find code idioms. All these prior works use specialized compilers with custom integrations of equality saturation. This is the type of work that could have benefited from a tool like DialEgg, which combines, in a dialect-agnostic way, MLIR and equality saturation.

11 Conclusion

This paper has presented DialEgg, a dialect-agnostic approach to express MLIR constructs in Egglog. DialEgg is able to automatically convert MLIR code to Egglog, apply various optimizations expressed as rewrite in a subset of the Datalog language, and transform the code back to MLIR.

This paper has shown that it is straightforward to encode optimizations with DialEgg. Equality saturation can combine multiple transformations and can find a globally optimal solution that is beyond the reach of classical MLIR passes.

DialEgg lowers the barrier to entry for researchers to experiment with equality saturation in the context of MLIR. It offers full integration with a modern equality saturation library, Egglog, which the authors hope will unleash the power of equality saturation to the MLIR community.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

A Artifact Appendix

A.1 Abstract

This artifact [15] contains a source tree and a Dockerfile. Docker will create and build a container that includes DialEgg and its dependencies, LLVM, MLIR, and Egglog. This container is designed to reproduce this paper's experimental results (fig. 3 and table 2).

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** An equality saturation-based MLIR optimizer translating back and forth from MLIR to Egglog.
- **Compilation:** Docker builds and loads the artifact. We used Docker version 27.3.1.
- **Data set:** All data sets are handwritten for DialEgg and included in the artifact.
- **Run-time environment:** This artifact has only been tested on MacOS 14.5 with an ARM SoC, and in a docker container based on the image packaged with this artifact.
- **Hardware:** Any system that supports Docker, or an ARM SoC if you wish to reproduce the runtime results in the paper as closely as possible. 16 GB of RAM is preferable for building LLVM and MLIR to avoid linking issues. We ran the experiments on a 10-core M1 Pro SoC with 16 GB of RAM.
- **Metrics and Output:** For each benchmark, we share the total number of operations from each dialect, the number of rules, runtimes of optimization passes, compilation times, and saturation times.
- **Experiments:** Resulting runtime and compilation times of various optimization passes, including using DialEgg.
- **How much disk space required (approximately)?:** 15GB
- **How much time is needed to prepare workflow (approximately)?:** 5 hours, varies depending on the CPU/SoC.
- **How much time is needed to complete experiments (approximately)?:** 1 hour, varies depending on the CPU/SoC.
- **Publicly available?:** Yes
- **Code licenses?:** Apache-2.0 license
- **Archived?:** DOI: 10.6084/m9.figshare.27668247.v3

A.3 Description

A.3.1 How Delivered.

The artifact is available on both figshare (DOI: 10.6084/m9.figshare.27668247.v3) and the *AzizZayed/dialegg-cgo-artifact* GitHub repository.

A.3.2 Hardware Dependencies.

A system with at least 16 GB of RAM. An ARM SoC if you wish to reproduce the runtime results in the paper as closely as possible.

A.3.3 Software Dependencies.

A Docker installation.

A.4 Installation

Build the Docker image and run a Docker container from the artifact root directory using the following commands. Run the first command below to build the image and the second to run and enter a container with the image.

```
$ docker build -t dialegg-img .
$ docker run --name dialegg --rm -i -t dialegg-img bash
```

All the following commands assume you are inside the running Docker container. From within the `/dialegg` directory, run the build script to build DialEgg and all its dependencies. Afterward, run the `source` commands to prepare the environment to run the experiments.

```
$ ./build.sh
$ source .env
$ source venv/bin/activate
```

A.5 Experiment Workflow

A.5.1 Runtime Performance.

To gather the data from fig. 3, run the following commands from within the `/dialegg` directory in the container. The first command runs the optimizers for each benchmark. The second command collects the runtime performance data and stores it. The third command plots the performance data and generates fig. 3.

```
$ python bench/opt.py
$ python bench/bench.py
$ python bench/plot.py
```

A.5.2 Compilation Time.

The following command collects and presents the compilation time data in table 2.

```
$ python test/timer.py
```

A.6 Evaluation and Expected Result

A.6.1 Number of Rules and Operations.

Each benchmark has its own directory in the `bench` directory. The data from table 1 is gathered by counting the number of relevant operations in the file `bench/<benchmark>/<benchmark>.mlir`. The number of rules in table 2 is gathered by counting the number of rules in the egg file: `bench/<benchmark>/<benchmark>.egg`.

A.6.2 Runtime Performance.

The generated plot for fig. 3 can be found in the `bench` directory named `speedup.pdf`.

A.6.3 Compilation Time.

The data from table 2 will be output by the command given above in appendix A.5.2.

A.7 Reusability

DialEgg can be used to optimize any MLIR file with equality saturation, as long as each operation has a single result. The core of the artifact is `egg-opt`, the binary produced under the `build` directory. This is an `mlir-opt` tool that can take any supported MLIR code, and a corresponding egg file and perform equality saturation-based optimization.

A.8 Known Discrepancies

DialEgg was originally evaluated on Mac OS 14.5 running on an ARM CPU, but for ease of reproducibility, this artifact is designed to run in a Docker container running Ubuntu Linux. This change has led to known differences in the results reported in the paper. The size of the Vector Norm and Polynomial benchmarks was increased from $1,000,000$ to $100,000,000$, due to noise introduced by the virtualization.

References

[1] John Backus. 1978. *The history of Fortran I, II, and III.* Association for Computing Machinery, New York, NY, USA, 25–74. https://doi.org/10.1145/800025.1198345

[2] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1029–1044. https://doi.org/10.1145/3620665.3640392

[3] J. Van Der Cruysse and C. Dubach. 2024. Latent Idiom Recognition for a Minimalist Functional Array Language Using Equality Saturation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 270–282. https://doi.org/10.1109/CGO57630.2024.10444879

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

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

[6] Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. 2024. The MLIR Transform Dialect. Your compiler is more powerful than you think. arXiv:2409.03864 [cs.PL] https://arxiv.org/abs/2409.03864

[7] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[8] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

[9] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

[10] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742.

[11] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[12] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

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

[14] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[15] Abd-El-Aziz Zayed. 2024. dialegg-cgo-artifact. (11 2024). https://doi.org/10.6084/m9.figshare.27668247.v3

[16] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (jun 2023), 25 pages. https://doi.org/10.1145/3591239

Received 2024-09-12; accepted 2024-11-04