SMT-Based Translation Validation for Machine Learning Compiler

Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo and Juneyoung Lee

August 10, 2022

SMT-based Translation Validation for Machine Learning Compiler

Seongwon Bang[1], Seunghyeon Nam[1], Inwhan Chun[1], Ho Young Jhoo[1], and Juneyoung Lee*[2][0000−0002−8152−9330]

[1] Seoul National University, Seoul, Korea
`{seongwon.bang, seunghyeon.nam, inwhan.chun, hoyoung.jhoo}@sf.snu.ac.kr`
[2] CryptoLab, Seoul, Korea
`aqjune@cryptolab.co.kr`

Abstract. Machine learning compilers are large software containing complex transformations for deep learning models, and any buggy transformation may cause a crash or silently bring a regression to the prediction accuracy and performance. This paper proposes an SMT-based translation validation framework for Multi-Level IR (MLIR), a compiler framework used by many deep learning compilers. It proposes an SMT encoding tailored for translation validation that is an over-approximation of the FP arithmetic and reduction operations. It performs abstraction refinement if validation fails. We also propose a new approach for encoding arithmetic properties of reductions in SMT. We found mismatches between the specification and implementation of MLIR, and validated high-level transformations for `SqueezeNet`, `MobileNet`, and `text_classification` with proper splitting.

1 Introduction

Machine learning compilers play a crucial role in the deep learning ecosystem. Their primary goal is to lower high-level tensor operations into fast machine instructions. To boost the speed of training and inference, they utilize several optimizations. Tensors' layouts may be changed for spatial locality, and loops lowered from tensor operations may be fused and offloaded into GPUs if beneficial. Any bug in the optimizations may cause a crash or silently bring a regression to the prediction accuracy and performance.

However, verifying machine learning compilers is a challenging goal. Open-source compilers like XLA, Glow, and MLIR are being updated daily. As their intermediate representations (IRs) are for internal uses, they are sometimes underspecified, making their formalization hard. Furthermore, programmers want to boost the performance at the expense of precision by allowing unsafe arithmetic properties such as associativity of addition.

Recently, SMT-based automatic translation validation has gained attention [33] because it fits well with fast-moving industrial compilers. Translation validation is an approach to checking whether a specific compilation is correct by inspecting the source (input) and target (output) programs. To cover a variety of compiler optimizations, it uses an SMT solver which is an automatic theorem prover for first-order logic. Using an SMT solver allows us to quickly explore possible semantics for IRs by implementing them and validating compilations of various programs.

A key challenge is how to make SMT solvers prove the verification condition in a reasonable time. To use an SMT solver, the given problem must not be too complex. Bit-vector and uninterpreted function (UF) theories are well-supported by the majority of solvers, whereas floating-point numbers are not [14]. This implies that finding an efficient encoding for tensors and their operations is important for practical validation of machine learning compilers.

In this paper, we propose an SMT-based translation validation framework for Multi-Level IR (MLIR). MLIR is a compiler framework for facilitating the modular development of domain-specific compilers by sharing IRs and relevant transformations. MLIR is primarily used by TensorFlow, TFLite, and IREE. More deep learning frameworks like PyTorch are adding supports for MLIR.

Our goal is to validate high-level, target-independent intraprocedural transformations in MLIR. These include lowering high-level tensor operations to loops, bufferizing tensors, simplifying tensor/buffer operations, and simple loop optimizations. Our tool does not receive hints about the ongoing transformation from the compiler.

The list of contributions of our paper is as follows:

– The first SMT-based translation validation for MLIR (Section 3).
– An abstract representation of FP arithmetic for translation validation (Section 4).
– An SMT encoding of tensor operations and loops as well as fast encoding of arithmetic properties of reduction operations (Section 5).
– Validation of compilation of three deep learning models as well as hundreds of unit tests in MLIR (Section 7).
– A discovery of several ambiguities in the semantics of MLIR (Section 7).

2 Multi-Level Intermediate Representation (MLIR)

The MLIR project is an open-source compiler infrastructure that facilitates the modular development of domain-specific compilers by sharing reusable parts. The reusable parts are dialects and relevant compiler transformations. A dialect is a subset of a compiler's intermediate representation language. An intermediate representation (IR) program in MLIR is expressed using one or more dialects. They are ultimately lowered into the input languages of low-level code generation frameworks such as LLVM IR or SPIR-V through first-class dialects. We will introduce several core dialects in MLIR, which are also our targets for validation.

Dialects for Tensors The `tensor` and `tosa` (Tensor Operator Set Architecture) dialects define the tensor type and operations. Pre-trained machine learning models can be lowered to them via importers.

A tensor type consists of an element type and dimensions. The tensor dimensions can be dynamic, which are retrievable in runtime. Its elements can be accessed through, e.g., `tensor.extract` with valid indices. Tensor registers do not alias each other and are in the static single assignment (SSA) form. `tosa` provides a set of operations commonly employed by deep neural networks, such as a convolution or pooling.

```
// f32 is the float type in C
func @calc(%img : tensor<2x64x64x3xf32>,
         %filter : tensor<16x3x6x3xf32>) {
%c0 = arith.constant -0.0 : f32
%bias = tensor.from_elements %c0, ... , %c0
                                : tensor<16xf32>
%res = tosa.conv2d(%img, %filter, %bias)
                 ...->tensor<2x62x59x16xf32>
return %res : tensor<2x62x59x16xf32>
}
```

(a) A convolution operation.

```
#col_major = affine_map<(d0, d1)->(d1*3+d0)>
func @example(%arg0 : memref<2x3xf32>,
    %arg1 : memref<2x3xf32, #col_major>) {
...
}
```

(b) Two `memref` arguments.

```
#map0 = affine_map<(d0, d1, d2)->(d0, d2)>
#map1 = affine_map<(d0, d1, d2)->(d2, d1)>
#map2 = affine_map<(d0, d1, d2)->(d0, d1)>
// i32 is the 32-bit int type in C
%output = linalg.generic {
// #map0, #map1: maps for %A, %B
// #map2:        a map for %C
indexing_maps = [#map0, #map1, #map2],
iterator_types = ["parallel", "parallel",
                  "reduction"]}
ins(%A, %B : tensor<16x8xi32>,
             tensor<8x32xi32>)
outs(%C : tensor<16x32xi32>) {
^bb0(%a: i32, %b: i32, %c: i32):
  %ab  = arith.muli %a, %b : i32
  %res = arith.addi %c, %ab : i32
  linalg.yield %res : i32
} -> tensor<16x32xi32>
```

(c) `%C`+`%A`×`%B` in `linalg`.

Fig. 1: Dialects for tensors and buffers in MLIR.

The `@calc` function in Figure 1(a) takes two tensor arguments, performs convolution (`tosa.conv2d`), and returns the result. The input bias and output tensor are stored at tensor-typed virtual registers `%bias` and `%res`. Note that different dialects – `tensor`, `tosa`, and `arith` (dialect for simple arithmetic operations) – can exist in one IR program.

MemRef Dialect The `memref` dialect has a type for memory references (which is also called `memref`) and relevant operations. The `memref` type is similar to a pointer type in C but has richer information than that. It has a layout map that maps multidimensional, logical indices into a one-dimensional, physical address[3]. It is used to create a view of a specific memory region in the form of a tensor. It supports arbitrary access patterns such as strided accesses or a transposed view. MLIR transformations assume that the layout map is injective.

Figure 1(b) shows two `memref` arguments with different layout maps. `%arg0` has a default row-major layout map. On the other hand, `%arg1` has a column-major layout map meaning that `%arg1[i][j]` is located at offset $(i + j \times 3)$ from the reference point. The values multiplied by offsets `d0`, `d1` (1 and 3 in `#col_major`) are called strides.

[3] Its domain can also be multidimensional in general, but we do not support the case.

Linalg Dialect The `linalg` dialect contains various loop-like operations on tensors and buffers. `linalg` operations are more primitive than `tosa`'s and can be performed on buffers.

In `linalg`, one can represent a generic loop in a structured form using the `linalg.generic` operation. Each loop explicitly takes input tensors or buffers as its operands. The loop's indexing maps describe which elements are chosen at each iteration. The elements chosen from the inputs at an iteration are represented as input arguments of the loop body region.

The loop body yields a value at each iteration, and the results constitute the output elements. A loop that takes an output buffer writes the resulting elements to the buffer. A loop that takes an output tensor stores the resulting tensor in a new tensor register, which can later be used as another input tensor.

Figure 1(c) shows how to represent `%C`+`%A`×`%B` for three matrices `%A`, `%B`, and `%C` in `linalg.generic`. `%C` and the resulting tensor (`%output`)'s shapes must be the same. The `linalg.generic` is a triple nested loop that has three induction variables `d0`, `d1`, `d2`. The `indexing_maps` describe which elements of the tensors are retrieved in each iteration. The retrieved elements are assigned into block arguments `%a`, `%b`, and `%c` of the loop body. The loop body performs integral multiplication (`arith.muli`) followed by addition (`arith.addi`) and yields it to the next iteration which again becomes `%c`. `iterator_type` shows that the third (innermost) loop is a reduction loop because it is doing summation, whereas the two outer loops can be parallelized.

`linalg` is the source and target dialect of several key transformations. First, `tosa`'s operations can be lowered into the combination of `linalg`'s operations on tensors. Second, bufferization on `linalg`'s operations changes their tensor operands into buffers. Third, the `linalg.generic` loops can be optimized into fused `linalg.generic` loops or simpler operations. Fourth, conversions from `linalg` to lower level dialects yield for loops (`affine`, `scf`) or control-flow graphs (`standard`).

Transformations in MLIR MLIR provides transformations that (1) convert the input programs written in high-level dialects into the low-level ones, or (2) optimize the input program into more efficient form. Except for those that intentionally change the input program's behavior, transformations must preserve the behavior.

3 Overview

In this section, we introduce `mlir-tv`, a translation validation framework for MLIR. Like other frameworks [22,33,36,41,42], `mlir-tv` takes two programs written in the IR and checks whether the transformation is correct. Since `mlir-tv` targets intraprocedural transformations, functions in the two programs with the same signature are checked pairwisely. `mlir-tv` relies on an SMT solver to automatically prove that the transformation is correct or find a counterexample if incorrect.

`mlir-tv` symbolically encodes each MLIR instruction in a function and emits its final state in a logical formula. After encoding the final states of the source and target functions f_{src} and f_{tgt}, `mlir-tv` checks a refinement predicate using an SMT solver. The predicate states that for any input state I consisting of an initial memory and argument values, $f_{\text{src}}(I) \sqsupseteq f_{\text{tgt}}(I)$ must hold where $\sqsupseteq$ is a refinement relation between two final states (Section 6.3). If the SMT solver finds an input that breaks the refinement, `mlir-tv` concludes that the compiler transformation is incorrect. If the SMT solver proves that such input does not exist, the transformation is correct.

3.1 Abstraction for Floating-Point Arithmetic

For practical validation of tensor transformations, it is crucial to efficiently represent floating-point (FP) arithmetic in SMT. SMT-LIB 2 formally supports IEEE-754 [10] under the name of the FPA theory [37]. SMT solvers supporting the FPA theory typically simulate the hardware implementation of the FP arithmetic by representing their bits as boolean variables and converting FP operations into boolean expressions (called *bit-blasting*) [13]. Then, the formula can be efficiently solved using their highly optimized SAT solvers.

However, there are two challenges in using the FPA theory to prove transformations on tensors. First, encoding FP arithmetic in SMT is expensive because solvers internally yield large expressions. Also, a significant portion of tensor transformations does not require such precise encoding. For example, bufferization is agnostic to the representation of the underlying values because its goal is *moving* the virtual registers to memory buffers correctly. Second, machine learning compilers want to support transformations that are incorrect under IEEE-754 for performance. We cannot simply rely on FPA in this case because it will invalidate the transformations.

To address these concerns, `mlir-tv` abstractly encodes the FP operations (Section 4). We find an abstract domain for FP numbers that is *specific* to the transformation to validate. It uses over-approximations meaning that a successful validation implies the correctness of the transformation. If it is not validated, `mlir-tv` refines the abstraction and try validation again (Section 6.3).

3.2 The Formal Semantics of Dialects

Since there is no official formal semantics for MLIR dialects yet, we read the textual specification of MLIR dialects and represented them in the encoding function. The function returns the final state in SMT expressions. Therefore, it implicitly defines the big-step formal semantics of the dialects in MLIR. Also, the function contains encoding rules for each instruction, which implicitly represent its small-step semantics.

Note that we are not proposing new formal semantics for unsafe FP arithmetic. We assume that there exists a valid FP semantics that satisfies certain arithmetic properties. The concrete semantics of FP operations is hidden under the uninterpreted functions used for the abstract encoding. The semantics of unsafe FP arithmetic is often explained using nondeterministic execution [11] and encoding it in SMT requires universal quantification which is expensive.

4 Encoding Floating-Point Numbers and Tensors

To overcome the challenges described in Section 3.1, we devise an abstract encoding of FP arithmetic tailored for translation validation. In this abstract encoding, an FP number is represented as a bit-vector that is typically smaller than its original bit width. The operations on FP numbers are represented as UFs satisfying arithmetic properties like commutativity. Our encoding does not miss bugs because it is an over-approximation of the FP arithmetic. On the other hand, validation failure does not always mean that the transformation is wrong.

4.1 Abstract Domain of Floating-Point Numbers

We begin with defining an abstract domain for FP numbers that is specific to the transformation to validate. We count the number of distinct FP numbers that are required to express at least one counterexample if the transformation is incorrect. As a result, if it is possible to prove that no counterexample is found in this abstract domain, no concrete counterexample can exist.

Consider a transformation that swaps the two operands of FP addition. An invocation of the source function (top) can observe at most three distinct FP numbers because it has three FP registers `%a`, `%b`, and `%c_src`. Similarly, the target function (below) can observe at most three different numbers. The number of distinct FP numbers required to validate the transformation is not greater than $4 = 3 + 3 - 2$ since two of those are shared as arguments.

```
// The source function
func @f(%a: f32, %b: f32) {
  %c_src = addf %a, %b: f32
  return %c_src: f32
}

// The target function
func @f(%a: f32, %b: f32) {
  %c_tgt = addf %b, %a: f32
  return %c_tgt: f32
}
```

After counting the number, we abstractly represent the values of FP registers and constants using bit-vectors. For the above example, 2 bits are enough in theory because $4 \leq 2^2$. We will use notation $[\![\texttt{\%a}]\!]$ to represent the abstract bit-vector value of `%a`. In SMT, two bit-vector variables are declared for `%a` and `%b` because they can be any value, and `%c_src` and `%c_tgt` are defined as expressions with respect to the variables.

Defining Operations To abstractly define `addf`, we declare a UF for addition. If the arithmetic properties of addition are ignored, $[\![\texttt{addf(\%a, \%b)}]\!]$ may be defined as $\texttt{addf}_{\texttt{SMT}}([\![\texttt{\%a}]\!], [\![\texttt{\%b}]\!])$ where the definition of UF $\texttt{addf}_{\texttt{SMT}}$ is arbitrarily determined by the SMT solver. Since the solver's goal is to find a counterexample, it will try to find a definition of $\texttt{addf}_{\texttt{SMT}}$ that breaks the transformation. If the solver couldn't find one, the transformation is correct under any definition of FP addition.

Note that validating the above example requires encoding commutativity '$\texttt{addf}_{\texttt{SMT}}(\llbracket\texttt{\%a}\rrbracket, \llbracket\texttt{\%b}\rrbracket) = \texttt{addf}_{\texttt{SMT}}(\llbracket\texttt{\%b}\rrbracket, \llbracket\texttt{\%a}\rrbracket)$'. Instead of using an expensive universal quantification, we encode addition as '$\texttt{addf}'_{\texttt{SMT}}(x, y)$ `&` $\texttt{addf}'_{\texttt{SMT}}(y, x)$' where `&` is the bitwise and operation and $\texttt{addf}'_{\texttt{SMT}}$ is another UF. Without loss of generality, it encodes all possible commutative functions[4].

To encode the result of operations on $\pm 0, \pm 1, \pm\text{fMAX}$ (finite max), $\pm\infty$ and NaN, we use the ite (if-then-else) expression in SMT. For example, to encode 'NaN $+ y =$ NaN', the expression is wrapped with an ite that checks if one of the inputs is NaN. Combined with the commutativity encoding, the expression for $x + y$ becomes as follows. 'x is NaN' is the SMT formula checking x is NaN by inspecting x's abstract representation which will be described later.

$$\mathsf{ite}(x \text{ is NaN} \lor y \text{ is NaN},\ \ \text{NaN},\ \ \texttt{addf}'_{\texttt{SMT}}(x, y)\ \texttt{\&}\ \texttt{addf}'_{\texttt{SMT}}(y, x))$$

Using UFs and ites, we abstractly encode $+$, $-$, $\times$, $/$ and x^y. Subtraction is defined as an addition of the negated second operand. Division is not equivalent to multiplication of the inversed operand due to the existence of subnormal values. Therefore, it is encoded using a separate UF.

Comparisons, $|x|$ and $-x$ are precisely encoded because our bit-vector representation natively supports them. Their representation will be described below.

Bit-vector Structure A bit-vector for FP consists of a sign bit (SB) at its most significant bit and magnitude bits (MB) at the entire lesser significant bits. They represent the sign and the order of absolute value of the original number, respectively. Therefore, comparing the magnitudes of two finite FP numbers is equivalent to simply comparing their MBs. If $\text{MB}[1 \ldots |\text{MB}| - 1]$ are all set to 1, the original value is ∞ ($\text{MB}[0] = 0$) or NaN value (1). Unlike IEEE-754 [10] which have multiple NaN values per sign, we have one representation per sign[5].

The bit-vector representation of an FP constant number is a concatenation of the sign bit and magnitude bits which is a bit-vector variable in SMT. The bit-vector variables are given preconditions so that a constant with a larger absolute value is guaranteed to have larger MB.

Supporting Floating Point Casts To support FP casts, MB is further split into three parts: limit bits (LB), truncated bits (TB), and precision bits (PB) in descending significance order. These parts determine the result of casting the value into a smaller FP type. LB represents the overflow condition. If LB is 0, a cast to the smaller size yields a finite value. If not, it yields $\pm\infty$. TB represents the magnitude floored to the target type. Its bit width is equivalent to the bit width of MB of the smaller type. PB represents the offset from the floored value. If PB is 0, the value is truncated to the exact value without loss of precision. Otherwise, the value must be rounded, and the direction is determined by a UF

[4] We describe its formal proof in our online supplementary material [5].

[5] We chose this policy because respecting the bits invalidates several transformations in LLVM and the behavior of processors canonicalizing NaN values [33].

returning boolean. Extension is done by copying MB to TB and filling LB and PB with 0[6].

4.2 Encoding Tensors

In SMT, a tensor is represented as an array expression from the address space-sized bit-vector to the element type. A multidimensional tensor is encoded as a one-dimensional array in row-major order. The dimension sizes of dynamically shaped tensor arguments are encoded as bit-vector variables. The number of elements of a tensor cannot exceed the size of the address space.

For each tensor argument in MLIR function, a new SMT array variable is assigned because its value can be fully arbitrary. The results of tensor operations are encoded as lambda expressions in SMT which is described in Section 5.1.

Uninitialized Tensors A tensor may contain uninitialized elements. In SMT, a tensor carries another boolean array that indicates uninitialized elements.

We define accessing uninitialized elements as an undefined behavior (UB) for the following reason. During bufferization, `linalg.init_tensor` operation that returns an uninitialized tensor is lowered into `memref.alloc`. The `memref.alloc` operation is then converted into a `malloc` call in LLVM IR, reading uninitialized bytes of which and using them may raise UB.

Tensor arguments in MLIR are assumed to be fully initialized. `linalg`'s `init_tensor` is the only operation that creates an uninitialized tensor. Operations like `tensor.insert` can create a partially initialized tensor.

4.3 Calculating the Bit Width

The bit width of the abstract representation of FP numbers is decided by the number of float registers and constants. Since all FP registers can store distinct FP numbers, the number of different FP numbers that may appear during the source and target program execution is bounded by the number of FP registers and distinct constants.

However, an operation that does not return an FP number can internally observe an unseen number. For example, suppose `is_int(x)` that returns true if float `x` is an integral value. Given an UF $\texttt{floor}_{\texttt{SMT}}(x)$ that returns an abstract float with its decimal truncated, this operation can be encoded as '$\texttt{x} == \texttt{floor}_{\texttt{SMT}}(x)$', which hides an unseen number in $\texttt{floor}_{\texttt{SMT}}(x)$.

Therefore, we count the number of UFs applied to abstract FP numbers while encoding the source and target instructions. The size of the BV field is $\lceil \log_2 N \rceil$ where N is the number of applied UFs added by the number of FP arguments as well as distinct constants of the source and target functions. From the above example, `is_int(x)` must increment N even if it returns boolean because $\texttt{floor}_{\texttt{SMT}}(x)$ can return an unseen FP value.

[6] We describe the full encoding of constants of different types and other details in our online supplementary material [5].

```
func @f(%x, %y: tensor<8xf32>) {
  %z_src = tosa.add %x, %y
  return %z_src
}


func @f(%x, %y: tensor<8xf32>) {
  %z_tgt = tosa.add %y, %x
  return %z_tgt
}
```

(a)

```
func @f(%x, %y: tensor<8xf32>,
        %i: index) -> f32 {
  %z_src = tosa.add %x, %y
  %z_src_i = %z_src[%i]
  return %z_src_i
}

func @f(%x, %y: tensor<8xf32>,
        %i: index) -> f32 {
  %z_tgt = tosa.add %y, %x
  %z_tgt_i = %z_tgt[%i]
  return %z_tgt
}
```

(b)

```
func @f(...) -> f32 {
  %x_i, %y_i = %x[%i], %y[%i]
  %z_src_i = addf %x_i, %y_i
  return %z_src_i
}


func @f(...) -> f32 {
  %x_i, %y_i = %x[%i], %y[%i]
  %z_tgt_i = addf %y_i, %x_i
  return %z_tgt
}
```

(c)

Fig. 2: Reducing elementwise tensor operations into scalar operations.

Considering Tensors and Memory In general, a tensor with M elements must increase N by M because it can have M different floats. To reduce the bound, we again rely on the fact that finding only *one* counter-example is enough. If that counter-example is a tensor, one mismatched element is sufficient.

If all tensor operations in functions are elementwise, we can simply ignore tensors' dimensions and count them as FP numbers when evaluating N. Consider the example in Figure 2(a). To validate that transforming the upper `f` to the lower `f` is correct, we must check whether `%z_src[i]` and `%z_tgt[i]` are equal for any `i`. Therefore, we can rewrite the functions into the form in Figure 2(b) without affecting the correctness of the transformation. Note that the return types of two functions are changed from tensor to float. Since `tosa.add` is an instruction that performs `addf` elementwisely, choosing `i` from `tosa.add` only requires `i`'th elements from its input tensors. Therefore, the functions can again be rewritten as in Figure 2(c). Since only the `i`'th elements of tensors `%x` and `%y` are used, the functions can again be rewritten to take `%x_i` and `%y_i` as function arguments instead, which is not depicted in the figure. Therefore, validating the initial pair is equivalent to validating two functions taking and adding two FP numbers.

Given a `memref` value, one can only access in-bounds locations. Thus, its size is added into N. If all tensor operations are elementwise, it is counted as one.

5 Supporting Tensor Operations and Loops

In this section, we introduce the SMT encoding of tensor operations and loops.

5.1 Encoding Tensor Operations

The result of a tensor operation is encoded as a lambda expression in SMT. For example, a negation of tensor `t` is encoded as '$\mathsf{lambda}\ i, \mathtt{negate}(\mathsf{select}(\mathtt{t}, i))$' where i is a 32 bit-vector variable, '$\mathsf{select}(\mathtt{t}, i)$' selects the i-th element from the SMT array of `t`, and '$\mathtt{negate}(bv)$' is an alias for an SMT expression extracting the sign bit of bv and concatenating its negation with its BV bits. Note that it does not check whether i is within the bound of the tensor. It is because the values at out-of-bounds indices cannot affect the program's behavior.

For operations returning a multidimensional tensor, the lambda chooses and returns the element in row-major order. For example, transpose of `t` whose size is $N \times N$ is encoded as '$\mathsf{lambda}\ i, \mathsf{select}(\mathtt{t}, i\%N \times N + i/N)$'.

Encoding Reduction Operations In general, reduction operations like summation of an array cannot be precisely encoded in SMT-LIB 2. To support them, we abstractly encode the reduction operations using UFs. For example, we declare `sum` which is a UF taking an array and returning a float number. Since this is an over-approximation, the validation may fail. In this case, we perform abstraction refinement, which will be described in Section 6.3.

The out-of-bounds elements of an array are wiped out before applying to UF because they must not affect the result. This is done by wrapping the input array with lambda and select. The select returns the value that do not affect the result of the reduction (e.g., -0.0 for a summation) if the index is out of bounds.

Tensor Operations and Undefined Behavior The documentation was not clear about the behavior of a program violating the assumptions that tensor operations expect at runtime. The violations include out-of-bounds access, size mismatch of the dynamic-shaped tensors, and reading an uninitialized element. If it is defined as having well-defined side effects such as calling `exit`, dead tensor operations cannot be freely removed and lowering to LLVM IR whose behavior may be undefined cannot be explained. Therefore, we define them as UB.

5.2 Encoding Loops

In MLIR, `linalg` loops are typically generated from high-level tensor operations. Compared to loops in general programs, they are simple and syntactically provide rich information. The loop consists of instructions without side-effect (modulo UB), and `linalg` loops explicitly state input/output tensors' index mappings as well as parallelizable induction variables. Therefore, we can construct the output tensor or buffer without synthesizing loop invariants.

```
#id = affine_map<(d0, d1) -> (d0, d1)>
#transposed = affine_map<(d0, d1) -> (d1, d0)>

// %C = %A + %B^T, %C's shape = %out's shape
%C = linalg.generic {indexing_maps = [#id, #transposed, #id],
                     iterator_types = ["parallel", "parallel"]}
    ins(%A, %B : tensor<?x?xf32>) outs(%out : tensor<?x?xf32>) {
^bb0(%a: f32, %b: f32, %unused: f32):
  %c = arith.addf %a, %b: f32
  linalg.yield %c : f32
} -> tensor<?x?xf32>
```

Consider the above loop that adds tensors `%A` and `%B`^{T}. Indexing maps (`#id`, `#transposed`, `#id`) are mappings from two induction variables (hence a doubly nested loop) to the indices of input (`%A`, `%B`) and output (`%out`) tensors. The loop body shows that the initial value of `%out` is not used. Since iterations over each dimension have no dependency because they are parallel (`iterator_types`), we can conclude that `%out[i][j] = %A[i][j] + %B[j][i]`.

In this section, we propose an encoding of loops in `linalg` using the lambda theory and a universal quantification. Encoding a loop in `linalg` starts with finding loop bounds. Loop bounds are determined by matching the ranges of the indexing maps with the tensor (buffer) sizes. Then, the loop body which yields the element of the resulting tensor is encoded. If the output type is tensor, the resulting tensor is encoded in lambda in row-major order. If the output type is buffer, the memory locations are accordingly updated.

For the above example, the yielded result at each iteration is described as a lambda expression with two parameters: '`lambda` (d_0, d_1), `add(%A`$[d_0, d_1]$`, %B`$[d_1, d_0]$`)`'. Then, the output tensor `%C` is encoded as a lambda with a single parameter i. It selects $(i \,/\, N, i \,\%\, N)$ from the first lambda where N is `%out`'s width.

Determining Loop Bounds If the sizes of `%A` and `%B` are larger than that of `%out`, should the `linalg.generic` raise UB or add parts of the inputs?

To find its valid semantics, the first transformation to consider is `linalg`'s conversion from `linalg.generic` to a canonical `for` loop in another dialect. The conversion generates a `for` loop with the upper bounds of induction variables explicitly given. The conversion sequentially visits the indexing maps, and finds the first dimension that exactly matches. Exact matching means that the range of the indexing map must be identity, not e.g., `d0 + 1`. If such dimension cannot be found, the `linalg.generic` is considered syntactically invalid.

The second transformation is the canonicalization of `linalg.generic`. If a `linalg.generic` loop iterates over the input tensors and simply returns the elements, its output is replaced with the input tensors regardless of the input/output tensors' shapes. However, if we determine the loop bounds only by the shape of the first matched tensor, this transformation cannot be justified when input tensors have different sizes.

Therefore, we encode the loop bounds of `linalg.generic` as follows. First, we find loop bounds according to the algorithm of the first transformation (`generic` to `for`). For the above example, the upper bounds of `d0` and `d1` are the the dimension's sizes of `%A` because the first indexing map is for `%A`. Second, all input tensors' shapes must match the determined loop bounds, otherwise UB. In the case of the above example, `%A`, `%B` and `%out`'s shapes must be equal.

Encoding Loops on Buffers If inputs/outputs are buffers, tensors are loaded from the inputs, the loop is performed on the tensors, and the resulting tensor is stored into the output buffer. The input and output buffers of `linalg.generic` must be disjoint (Section 6.2). If the output buffer's layout map is identity, the output memory block is updated using lambda. If not, a fresh SMT array for the updated block is created, and the equalities between old/new elements of the block and the output tensor are encoded using forall quantifications.

Encoding Reduction Loops Induction variables which have "`parallel`" in the `iterator_types` attribute must appear as the parameters of the SMT lambda expression. Other variables, however, must be accordingly encoded. To encode reduction loops, we syntactically match the operand of the last `yield` and use the corresponding UF for the reduction (Section 5.1). This worked well in practice because the reduction loops in MLIR had common patterns.

5.3 Supporting Arithmetic Properties of Reductions

Floating-point addition and multiplication are not associative, but programmers sometimes want to boost performance at the expense of precision by allowing compiler optimizations that rely on the property. To encode the property, the definition of addition and multiplication must be different from IEEE-754 because using it causes inconsistency in the underlying logic.

Then, what is the semantics of $x + y + z$? One possible solution is that its evaluation nondeterministically yields either $(x+y)+z$ or $x+(y+z)$ [11]. However, encoding the semantics in SMT requires introducing quantified variables.

Therefore, as described in Section 5.1, we start from abstractly encoding reduction operations in UFs. For example, UF `sum` takes an array $[x, y, z]$ and returns its summation. A question is how to encode their arithmetic properties like $\texttt{sum}([\texttt{sum}([x, y]), z]) = \texttt{sum}([x, \texttt{sum}([y, z])])$. We introduce a new technique that works when the length of the input array is constant. This technique is not specific to a summation but can be applied to any reduction.

Encoding Commutativity The first arithmetic property to consider is commutativity: '$\texttt{sum}([..., x, ..., y, ...]) = \texttt{sum}([..., y, ..., x, ...])$'.

A straightforward solution is to use the multiset theory. Two `sum`s are considered equal if the multisets converted from input arrays are equal. For the solvers that do not support the multiset theory, a multiset can be simulated using an array taking an element and returning its count. However, this multiset-based approach does not scale well (Section 7.3). We conjecture that existing algorithms in the solvers are not good at checking the equality of two multisets (cvc5) / counter arrays (Z3).

We suggest a hash-based approach for encoding the multiset equality. Our approach begins with defining a hash function F on an array. If two arrays are equal, their hash values must be equal. The inverse holds when the range of F is sufficiently large. It only uses the theory of UF and BV, which are cheap.

To define F, we define another hash function f on floating-point numbers. $F(A)$ is defined as a summation of hash values of its elements $\sum_{x \in A} f(x)$. By the arithmetic property of bit-vector addition, $F(A) = F(A')$ if A' is a permutation of A. The inverse direction also holds. We prove that if $F(A) = F(A')$ for any f, A' is a permutation of A.

Theorem 1. *Given A and A' that are arrays of type T, if $\forall f \, . \, \sum_{x \in A} f(x) = \sum_{x \in A'} f(x)$ where $f \in T \to BV(\lceil \log_2 max(|A|, |A'|) \rceil)$, A' is a permutation of A.*

Proof. Let's assume that $count(S, x)$ is the number of x in multiset S. For example, $count(\{1, 1, 3\}, 1)$ is 2. We first prove the following lemma.

Lemma 1. *Given two multisets S and S', $S = S'$ holds if*

$$\forall g, \left(\sum_{x \in S} count(S, x) \times g(x) \right) = \left(\sum_{x \in S'} count(S', x) \times g(x) \right)$$

where $g \in T \to BV(\lceil \log_2 max(|S|, |S'|) \rceil)$.

Proof. Assume that $g_k(x)$ is a function that returns 1 if $x = k$ and 0 otherwise. By picking each element of S as k and $g = g_k$, $S = S'$ holds. $\square$

Assume that S is a multiset from array A and S' from A'. From the assumption $\forall f \, . \, \sum_{x \in A} f(x) = \sum_{x \in A'} f(x)$, we can derive $\forall g, \left(\sum_{x \in S} count(S, x) \times g(x) \right) = \left(\sum_{x \in S'} count(S', x) \times g(x) \right)$. Then, we can apply the lemma. By the conclusion of the lemma, the two multisets are equal, hence A is a permutation of A'. $\square$

For each pair of two sum function calls appearing in the source and target, their equality is encoded as a constraint. Since $P \implies Q$ iff $\neg Q \implies \neg P$, the universal quantification in the Theorem 1 can be converted into an existential form '$\texttt{sum}(A) \neq \texttt{sum}(A') \implies \exists f \, . \, \sum_{x \in A} f(x) \neq \sum_{x \in A'} f(x)$'. Since $\exists f$ can be moved out, the precondition is quantifier-free.

Encoding Flattening of a Nested Reduction By expanding the hash function based approach, we can encode the equality between nested reductions. Consider this equality: '$\texttt{sum}([\texttt{sum}(A), \texttt{sum}(B)]) = \texttt{sum}(A \mathbin{+\!\!+} B)$'.

Since the array $[\texttt{sum}(A), \texttt{sum}(B)]$ is not a permutation of $A \mathbin{+\!\!+} B$, the previous encoding does guarantee that the two summations are equivalent. To support this case, given a hash function F and summation $\texttt{sum}(A)$, we add a precondition $F(\texttt{sum}(A)) = \sum_{x \in A} F(x)$. That is, the hash value of $\texttt{sum}(A)$ is equivalent to the summation of hash values of $x \in A$.

Note that the hash function is individually defined per a pair of summations in the programs. This causes additional preconditions for each hash pair to relate inner and outer summation. We reduce the number of preconditions by unifying hash functions into one[7].

6 Encoding Memory and Refinement

MLIR has several dialects providing memory operations, such as `memref`, `affine`, and `bufferization`. We propose a memory model for these dialects. Also, we illustrate our SMT encoding for the model.

[7] Due to the limited space, we prove that the unified hash function's range must not be smaller than $[0, p^2 n)$ where p is the number of summation pairs and n is the maximum size of an array in our online supplementary material [5].

6.1 Memory Model

Memory Block A memory is made up of smaller memory blocks in our memory model. A memory block is a unit of a memory allocation, and is either created by `memref.alloc`, `memref.alloca`, clone-like operations of `bufferization`, or defining a global variable. `memref.alloca` allocates a block at stack whereas `memref.alloc` has no such constraint. `memref.dealloc` frees the block.

A memory block is uniquely identified with a block id. Its properties consist of the number of elements, block type, writability, liveness, and the list of elements with the list of booleans indicating whether each element is initialized. The block type is a boolean value which shows whether it is created by `memref.alloc`. Allocating instruction creates a new memory block which is initially alive, writable, and fully uninitialized. The clone-like operation marks the source block with permanent read-only. The behavior of accessing a dead block is undefined, and also accessing an uninitialized element is undefined behavior. This decision is described in Section 4.2 as well.

Memory Reference The `memref` type is a reference to a specific memory area. It consists of the pointing block's id, block offset, layout map, dimension sizes of the pointing area, and a flag indicating whether it is a view reference. A block offset may be non-zero because `memref` allows creating an aliased reference via `memref.view`, which may not point to the head of the block. `memref` may point to an out-of-bounds area of the block, and accessing that area is UB.

Loading a tensor from `memref` is well-defined if (1) the referenced area is within the bounds of the memory block, (2) the block is alive (i.e. not deallocated yet), and (3) the visited offsets are fully initialized. Writing a tensor is well-defined if the area is in-bounds and the block is alive and writable.

6.2 Encoding the Memory Model

The properties of memory blocks are encoded as SMT variables `size, writable, liveness, block_type, elements, initialized`. By default, all properties are defined as SMT variables because we cannot make any assumption on how and when a block is created in general. If the block's definition is visible (e.g., it is a global variable), they are initialized with literals in SMT. `elements` and `initialized` are encoded as SMT arrays from the offset to the value and boolean.

The number of blocks necessary to validate the transformation is determined via static analysis, which is described in [30]. The number is bounded because we do not support loops containing allocating operations. This works in practice because allocations are usually located outside of the loops. After the analysis, each block and property declares one SMT variable. The blocks for global variables and allocating operations are assigned constant block ids.

Local and Non-Local Blocks We adopt the notion of local and non-local blocks from [30]. Local blocks are created by the allocating instructions that belong to the validated function, whereas non-local blocks are not. Only the non-local blocks are checked at the refinement of final states. We do not consider escaped local blocks because (1) `memref` cannot have `memref` as its element type, and (2) we do not support call instructions.

Encoding Memory Access The SMT encodings of memory load/store operations follow the encodings described in [30]. The result of loading a value from `memref %m` is encoded as $\mathtt{ite}(\%\mathtt{m}.\mathrm{bid} = 0, \mathrm{arr}_0[\%\mathtt{m}.\mathrm{ofs}], \mathtt{ite}(\%\mathtt{m}.\mathrm{bid} = 1, \ldots))$ where arr_0 has the elements of memory block 0. Storing a value to `memref` updates the elements of possibly aliased blocks with `ite`s.

Encoding disjointnesses of two `memref` accesses – which is required by several buffer operations – is hard in general because a memref can point to non-contiguous locations in arbitrary patterns. Therefore, we support encoding a disjointness of `memref`s with trivial, row-major layout maps only, raising an error otherwise.

6.3 Compiler Correctness and Abstraction Refinement

Finally, we compare the final states of the source and target functions. A final state is defined as (ub, m, v) where ub is UB, m is the memory, and v is the return value. A final state refines another, or $(ub, m, v) \sqsupseteq (ub', m', v')$, if (1) ub is true, or (2) $ub = ub' \wedge v = v'$[8] $\wedge\ m$ refines m'. A memory m refines m' if for non-local blocks (b, b') with same id in the source and target, if (1) reading b at offset o is successful, so does the access to o at b', and (2) if b is writable, so does b'. For any input state I consisting of an initial memory and argument values, $f_{\mathrm{src}}(I) \sqsupseteq f_{\mathrm{tgt}}(I)$ must hold where $f(I)$ denotes the final state of function f. In SMT, the formula is inverted to remove the outermost quantification.

Abstraction Refinement To make validations cheap on average, we progressively refine the abstraction scheme that describes the abstraction level of encodings. Abstraction refinement happens when a validation fails or timeouts.

In the first round, the integer and FP dot operations are encoded using independent UFs which are not related to a summation. Also, FP numbers of different types are independently encoded and casts are defined as full UFs. If validation fails, the dot operations are encoded as a composition of a summation and multiplications, and the encoding for casts described in Section 4.1 is used. If this also fails, summations of arrays having small constant numbers of elements are unrolled into a sequence of additions and validated again. This validates, for example, folding $\mathtt{sum}([1.0, 2.0])$ into $1.0 + 2.0$.

Our abstraction cannot validate the constant folding optimization in general. To address this, `mlir-tv` provides a command-line option for using IEEE-754. It disables the unsafe properties on reductions because they are not compatible.

[8] For floats, NaNs of different signs are considered equivalent. For memrefs, we do not support references to local blocks because it needs universal quantifications [30]. Validating functions returning such values may result in false alarms.

7 Implementation and Evaluation

`mlir-tv` consists of 8,900 lines of C++ codes. It supports 25 `tosa` ops, 11 `memref` ops, 13 `linalg` ops, 10 `tensor` ops, 29 `arith` ops, 3 `bufferization` ops, and 8 other ops. `mlir-tv` uses Z3 4.8.13 and cvc5 0.0.3 as a solver, with 30 seconds timeout. The experiments in this section are performed using Z3 because Z3 showed better performance than cvc5 in `mlir-tv`'s sanity tests. We used the Apple M1 CPU and 16GB RAM with a fixed version of MLIR (`b5a0f0f`, 26/Dec).

We wrote 57 function pairs to check that it validates correct transformations and finds counterexamples for wrong pairs. From these tests, we observed that using the abstract encoding was 13.6x faster on average than the concrete IEEE-754 encoding. Shrinking the bit width of abstract FP (Section 4.3) was important because it brought 2.2x speedup compared to simply using 32 bits.

7.1 Validating MLIR Unit Tests

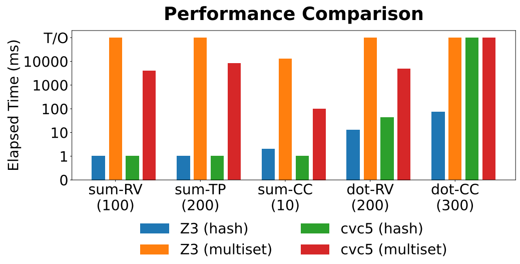
We validated the unit tests in the official MLIR project using `mlir-tv`. The unit tests (1) apply specific transformations to small, pre-defined MLIR programs, and (2) check whether the output programs syntactically match the test patterns. Using `mlir-tv`, we validated that the outputs of the transformations preserved the semantics of the inputs as well. We bounded the size of dynamic-shaped tensors to 100 to avoid timeouts. Bugs in tests with such tensors may have been missed.

Among the MLIR's unit tests, which consist of 2,467 function pairs in total, `mlir-tv` validated 433 tests, raised timeout for 8 tests, and failed for 8 tests. Validating the tests did not require encoding the unsafe arithmetic properties for reductions, but we are aware of uncovered transformations that require them.

We could find several issues in the semantics of MLIR dialects.

Signed Zero, NaN and $-\infty$ The `tosa.conv2d` and `tosa.depthwise_conv2d` operations are lowered to `linalg.pad_tensor` with the input tensors padded with +0.0. Also, we found that MLIR was folding $x+(+0.0)$ into x. However, this is incorrect since (1) $x+(+0.0) \neq x$ if $x = -0.0$ and (2) the `tosa` specification [6] states that an FP type must support signed zero. This problem was also found from `tosa.fully_connected` and `tosa.reduce_sum` operations whose lowered loops fill the initial tensors with +0.0. We reported these issues to the LLVM community. After the report, $x + (+0.0) \to x$ folding has been fixed [1]. We also found that lowering `tosa.clamp` and `tosa.max_pool2d` does not preserve their outputs if the inputs contain a NaN and $-\infty$ value.

`memref` Operations and Read-Only Blocks We couldn't find a good semantics for `linalg.fill` with a `memref` reference to a read-only memory block given as its operand. If `linalg.fill` with a read-only `memref` raises UB, it cannot explain the `linalg-bufferize` transformation because it creates `linalg.fill`



Name	Transformation
sum-RV	$\Sigma\mathrm{rev}(X) \to \Sigma X$
sum-TP	$\Sigma X^T \to \Sigma X$
sum-CC	$\Sigma(\Sigma X_1, \Sigma X_2) \to \Sigma(X_1 +\!\!+ X_2)$
dot-RV	$\mathrm{rev}(X_1) \cdot \mathrm{rev}(X_2) \to X_1 \cdot X_2$
dot-CC	$X_1 \cdot X_2 + X_3 \cdot X_4 \to (X_1 +\!\!+ X_3) \cdot (X_2 +\!\!+ X_4)$

Fig. 3: (a) A graph showing the effectiveness of our encoding of unsafe arithmetic properties of reduction operations. The numbers below the labels indicate the sizes of input tensors. The Y-axis shows the running time of `mlir-tv`. Timeout is 30,000ms. (b) Descriptions of the test cases. X is a 1D or 2D tensor, Σ is a summation, rev is a reverse, $\cdot$ is a dot product, and $+\!\!+$ is a concatenation.

with its `memref` operand pointing to a read-only block. If it is well-defined, it cannot explain the `linalg-generalize-named-ops` transformation because this converts `linalg.fill` into a loop storing a value to the pointer.

We found that `buffer-deallocation` transformation was introducing UB. It inserts `memref.dealloc` to free the unused result of `memref.clone`. But, mutating the result of `clone` is UB according to the specification.

Also, it was not clearly stated in the document when `memref.clone` makes the referenced location read-only. We discussed this issue in the online LLVM Discussion Forums, and the document was fixed to clearly state that it is immediately after the operation that the block becomes read-only [2].

7.2 Validating Compilation of Deep Learning Models

We compiled the TFLite models of `text_classification_v2`, `SqueezeNet` [25] and `MobileNet` [24], taken from the official TensorFlow website [7,8], by running them through `tosa-to-linalg`, `tosa-to-standard`, `canonicalize`, `fuse-elementwise-ops`, `tensor-constant-bufferize`, `linalg-bufferize`, `tensor-bufferize` transformations. To address validation failures of `tosa-to-linalg` due to the problem in `tosa`'s ± 0.0 handling, we tweaked `mlir-tv` so that it recognizes $+0.0$ used by certain operations as -0.0 instead.

To validate them in a reasonable time, we split the source and target programs into smaller functions. Since the networks did not have complex control flows other than loops, splitting was not very hard. The split functions contain an average 9.5 to 11.6 instructions. All transformations were validated correctly in `text_classification_v2`, but the last two transformations were failed in the other models since they have unsupported operations.

7.3 Performance Evaluation of Hash-Based Encoding

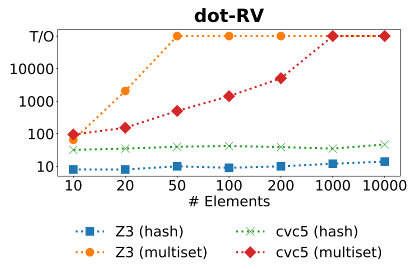


Fig. 4: Running times of dot-RV by tensor size. Timeout is 30,000ms.

We compared the performance of our hash-based encoding to the multiset-based encoding. In the latter encoding, two reductions are assumed to be equal if the multisets converted from the input arrays are equal. For cvc5, we used its native multiset theory. For Z3, we simulated multisets by defining an array that counts the numbers of elements. We set `QF_AUFBV` logic to Z3 by default and used `ALL` logic only when the solver failed. For cvc5, we used `HO_AUFBV` and `HO_ALL` logic respectively. We ran tests 10 times and calculated their average execution times. The timeout was set to 30 seconds.

Our hash-based encoding was faster than multiset-based encoding in overall cases (Figures 3 and 4). cvc5's multiset theory performed better than the Z3's array encoding, but was still slower than the hash-based encoding. The hash-based encoding showed consistent running time regardless of tensor size.

8 Related Work

Verifying Programs with Floating-Points Strategies for verifying programs using FP arithmetic (FPA) vary with their goals and background theories. Several works using abstract interpretation [29], SMT solvers [23,40] or computer algebra systems [31] target checking round-off errors of FP operations automatically. Axiomatizing and verifying FPA in theorem provers [12] enable us to make analysis sound and complete, but they require significant efforts.

To realize bit-precise FP reasoning in SMT, one can use a bit-vector representation of FP numbers (bit-blasting). Since bit-blasting can generate large and complex formulae, researchers have tried to find better FP abstraction. [15] presents an abstraction technique using either large or reduced precision of FPA. UppSAT [44] proposes an abstraction framework including fixed-point and real arithmetic. SymFPU [13] gives an effective yet correct bit-vector encoding of FPA considering various types and special cases of IEEE 754.

Verifying Programs using Arrays Several works have proposed their approaches to embedding the theory of arrays [34] into SMT solvers. [21,35] consider array read and write terms as uninterpreted functions, and regard the theory of array as axioms. FreqHorn [20] and SPACER [26] utilize constrained Horn clauses (CHC) engines.[27] analyzes array programs with broader theories by translating the axioms of the theory of array into the CHC format.

Yet another approach uses mathematical induction-based techniques to reason about array-manipulating programs with loops. [16] verify the validity of a given parameterized Hoare triple where the length of array N is used as a parameter of the pre- and post-condition.

Machine Learning Compilers Optimizing deep learning specific workloads has been a major working field for both hardware vendors [3,4] and software developers. [9,19] These frameworks translate neural-net representations in several frameworks into high-level computation graphs. Then, they optimize the graphs via well-known optimizations such as operator fusion or data layout transformation. Recent works allow optimization of dynamic workloads [38,45] and supports optimizations for heterogeneous systems [43].

[39] surveyed the bugs in DL compilers. They reported that the high-level IR transformations are the most buggy ones and stated that finding wrong code generation is challenging and should receive more attention.

Compiler Verification [11] relaxes FPA semantics since a compiler can ignore strict IEEE-754 behavior like fast-math optimizations in LLVM. They propose Icing which is a language allowing IEEE 754-unsafe FPA optimizations, and CakeML [28] which is a verified compiler with the optimizations. [32] proposes a verified tensor optimizer whose optimizations can be explored via Coq's tactics.

As for translation validation (TV), [18] proposes a practical TV framework for Halide which is a language for processing arrays. To support fast-math optimizations, it mainly uses Z3's type for real numbers. For general-purpose compilers, many different tools have been developed [36]. Alive2 [33], LLVM-MD [42] and Peggy [41] validate the transformations in LLVM using various techniques. The SMT memory model for Alive2 [30] uses a technique that is similar to our approach in order to bound the number of memory blocks. Some TV tools [22,17] split the original programs and validate the smaller pairs.

9 Conclusion

We propose `mlir-tv`, an SMT-based translation validation framework for MLIR. It abstractly encodes the FP arithmetic and reduction operations in SMT. Since the abstraction is an over-approximation, `mlir-tv` does not miss bugs unless a flag for bounding the size of dynamically shaped tensors is given. If validation fails, `mlir-tv` tries again with refined abstractions. We also propose a hash-based approach for encoding arithmetic properties of reductions, which outperformed a multiset-based one. `mlir-tv` found several mismatches between the specification and implementation of MLIR from the unit tests. Finally, `mlir-tv` validated high-level transformations for three pretrained DL models.

Acknowledgements The authors thank Chung-Kil Hur, Nuno P. Lopes and anonymous reviewers for their feedbacks on previous versions of this paper. We thank Jiun Kim for helping our experiments by writing useful scripts. We also thank Hong-Seok Kim, Jieung Kim, Hyunchul Park, Inho Seo and engineers from Google for their precious comments. Seunghyeon Nam, Inwhan Chun and Ho Young Jhoo were supported by the Basic Science Research Program through the National Research Foundation of Korea under Project Number 2020R1A2C2011947.

References

1. https://reviews.llvm.org/D114127
2. https://reviews.llvm.org/D106258
3. Arm NN SDK. https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn
4. NVIDIA TensorRT^{TM}. https://developer.nvidia.com/tensorrt
5. Supplementary material. https://doi.org/10.5281/zenodo.6615676
6. Tensor operator set architecture (TOSA) v0.23.0. https://developer.mlplatform.org/w/tosa/?v=19
7. TensorFlow Lite Examples: Text classification. https://www.tensorflow.org/lite/examples/text_classification/overview
8. Tensorflow lite: Hosted models. TensorFlowLite:Hostedmodels
9. XLA: Optimizing compiler for machine learning. https://www.tensorflow.org/xla
10. Ieee standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (2008). https://doi.org/10.1109/IEEESTD.2008.4610935
11. Becker, H., et al.: Icing: Supporting fast-math style optimizations in a verified compiler. In: Computer Aided Verification. vol. 11562, pp. 155–173. Springer (2019)
12. Boldo, S., et al.: Flocq: A unified library for proving floating-point algorithms in coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic. pp. 243–252 (2011)
13. Brain, M., et al.: Building better bit-blasting for floating-point problems. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 79–98. Springer International Publishing, Cham (2019)
14. Brain, M., et al.: Invertibility conditions for floating-point formulas. In: CAV (2019)
15. Brillout, A., et al.: Mixed abstractions for floating-point arithmetic. In: 2009 Formal Methods in Computer-Aided Design. pp. 69–76 (2009)
16. Chakraborty, S., et al.: Diffy: Inductive reasoning of array programs using difference invariants. In: Computer Aided Verification. pp. 911–935. Springer International Publishing, Cham (2021)
17. Churchill, B., et al.: Semantic program alignment for equivalence checking. In: PLDI (2019). https://doi.org/10.1145/3314221.3314596
18. Clément, B., et al.: End-to-end translation validation for the halide language. Proc. ACM Program. Lang. **6**(OOPSLA1) (apr 2022). https://doi.org/10.1145/3527328, https://doi.org/10.1145/3527328
19. Cyphers, D.S., et al.: Intel ngraph: An intermediate representation, compiler, and executor for deep learning. arXiv **abs/1801.08058** (2018)
20. Fedyukovich, G., et al.: Quantified invariants via syntax-guided synthesis. In: Computer Aided Verification. pp. 259–277. Springer International Publishing, Cham (2019)
21. Ganesh, V., et al.: A decision procedure for bit-vectors and arrays. In: Computer Aided Verification. pp. 519–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
22. Gupta, S., et al.: Counterexample-guided correlation algorithm for translation validation. In: OOPSLA (2020). https://doi.org/10.1145/3428289
23. Haller, L., et al.: Deciding floating-point logic with systematic abstraction. In: 2012 Formal Methods in Computer-Aided Design. pp. 131–140 (2012)
24. Howard, A.G., et al.: Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR **abs/1704.04861** (2017)
25. Iandola, F.N., et al.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. ArXiv **abs/1602.07360** (2016)

26. Komuravelli, A., et al.: Smt-based model checking for recursive programs. In: Computer Aided Verification. pp. 17–34. Springer International Publishing, Cham (2014)
27. Komuravelli, A., et al.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. p. 89–96. FMCAD Inc, Austin, Texas (2015)
28. Kumar, R., et al.: CakeML: A verified implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 179–191. Association for Computing Machinery, New York, NY, USA (2014)
29. Kästner, D., et al.: Astrée: Proving the absence of runtime errors. Embedded Real Time Software and Systems (2010)
30. Lee, J., et al.: An smt encoding of llvm's memory model for bounded translation validation. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 752–776. Springer International Publishing, Cham (2021)
31. Lee, W., et al.: On automatically proving the correctness of math.h implementations. Proceedings of the ACM on Programming Languages **2**(POPL) (2017)
32. Liu, A., et al.: Verified tensor-program optimization via high-level scheduling rewrites. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498717, https://doi.org/10.1145/3498717
33. Lopes, N.P., et al.: Alive2: Bounded translation validation for LLVM. In: PLDI (2021). https://doi.org/10.1145/3453483.3454030
34. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress (1962)
35. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: 2009 Formal Methods in Computer-Aided Design. pp. 45–52 (2009)
36. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000). https://doi.org/10.1145/349299.349314
37. Rümmer, P., Wahl, T.: An smt-lib theory of binary floating-point arithmetic. In: SMT 2010 Workshop (2010)
38. Shen, H., et al.: Nimble: Efficiently compiling dynamic neural networks for model inference. In: Smola, A., Dimakis, A., Stoica, I. (eds.) Proceedings of Machine Learning and Systems. vol. 3, pp. 208–222 (2021)
39. Shen, Q., et al.: A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ESEC/FSE. p. 968–980. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021)
40. Solovyev, A., et al.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Transactions on Programming Languages and Systems **41**(1) (2018)
41. Stepp, M., et al.: Equality-based translation validator for LLVM. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-159
42. Tristan, J.B., et al.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011). https://doi.org/10.1145/1993316.1993533
43. Yadav, R., Aiken, A., Kjolstad, F.: Distal: The distributed tensor algebra compiler (2022)
44. Zeljic, A., et al.: Exploring approximations for floating-point arithmetic using UppSAT. In: International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 10900, pp. 246–262. Springer (2018)
45. Zhu, K., et al.: DISC: A dynamic shape compiler for machine learning workloads. arXiv **abs/2103.05288** (2021)