Enhancing Translation Validation of Compiler Transformations with Large Language Models

Fei Xie
Portland State University

Yanzhao Wang
Portland State University, wyanzhao@pdx.edu

Enhancing Translation Validation of Compiler Transformations with Large Language Models

Yanzhao Wang and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97201, USA
{wyanzhao, xie}@pdx.edu

***Abstract*—This paper presents a framework that integrates Large Language Models (LLMs) into translation validation, targeting LLVM compiler transformations where formal verification tools fall short. Our framework first utilizes existing formal verification tools for translation validation. In this work, we use Alive2, a well-known tool in LLVM compiler verification, as an example. When formal verification tools are unable to confirm a transformation's soundness, our framework employs fine-tuned LLMs for prediction. It then applies fuzzing to transformations predicted as potentially unsound by the LLMs due to return values or memory inconsistencies, aiming to find counterexamples. In cases where transformations are unsound for other reasons or sound, or if no counterexamples emerge, the framework directly reports these outcomes without further fuzzing. This methodology has shown effectiveness in complex application such as deep-learning accelerator designs, where traditional formal verification tools struggle.**

I. Introduction

LLVM [1], a versatile open-source compiler, supports an extensive range of programming languages and hardware design languages. The core element of LLVM is its intermediate representation (IR). It serves as a key interface that streamlines interactions among frontends, backends, and transformation passes in the middle of the compiler, and therefore, is critical in integrating the varied components of the LLVM framework.

Translation validation, as detailed in [11], has evolved into a powerful method for verifying the correctness of compiler transformations. By validating the semantics of the source and transformed IRs, it establishes an efficient and robust mechanism for ensuring the reliability of compiler transformations. Alive2 [10] is such a translation validation tool specialized for LLVM IR. It leverages formal verification techniques, aiming to prevent that compiler optimizations do not introduce bugs into the resultant code.

Despite their advantages, translation validation tools such as Alive2 have significant limitations. They typically cannot manage unbounded loops and have limited ability in handling external function calls. Moreover, the SMT (Satisfiability modulo theories) solver [3], a fundamental component of these tools, is often incapable of dealing with complex computations.

Our framework introduces a novel integration of fine-tuned Large Language Models (LLMs) to overcome limitations in current compiler transformation validation flows. It can conduct predictive analyses of the correctness of transformations at which formal verification tools fall short. The process begins with inputting source and target Intermediate Representations (IR), the latter being a transformed version of the former. If the verification of this transformation is deemed unsolvable by the SMT solver of formal tools such as Alive2 (the example used in this paper), it is then passed to our `LLM-based transformation predictor`. This predictor assesses the transformation, categorizing it as likely sound or unsound. For transformations predicted as unsound, the predictor also provides the reasons underlying the unsoundness. Where transformations are unsound due to return values or memory inconsistencies, our framework employs fuzzing to find counterexamples. If the `fuzzer` finds counterexamples, it confirms the transformation's unsoundness. In cases where transformations are unsound for other reasons or sound, or if no counterexamples emerge, our framework reports their predicted soundness/unsoundness. This framework has been proven effective in practical applications, such as in Intel's deep-learning accelerator designs [2], addressing the challenges posed by unbounded loops and complex computations.

Our primary contributions are as follows:

- We pioneered the application LLMs in enhancing the reliability of compiler transformations.
- We seamlessly integrated Large Language Models (LLMs) with formal verification tools, exemplified by application to Alive2, to rapidly and efficiently assess LLVM transformation soundness. This integration proves particularly effective in scenarios where traditional formal verification tools struggle.

The remainder of this paper is structured as follows. Section 2 introduces the background. Section 3 presents an overview of our framework, while Section 4 dives into the specifics of our implementation. Section 5 reports our evaluation results. Section 6 discusses limitations and future work, and Section 7 concludes.

II. Background

Two primary methodologies are currently widely used in the domain of formal compiler verification:

1. **Theorem Proving**: This approach formally verifies if every transformation of a compiler preserves the semantics of the input program, e.g., CompCert [9] is a compiler for C that is formalized and verified in Coq. However, compiler certification using theorem proving is a highly complex and labor-intensive process, and every compiler revision requires reproofing. These drawbacks hinder future compiler improvements.

2. **Translation Validation**: The translation validation approach was first introduced by [11]; it entails a weaker formal technique that effectively certifies the conformity of a compiler's individual executions to the compiler's specification. This approach compares a compiler's input and output for its specific application. As illustrated in Figure 1, the workflow for translation validation sends both source and target programs to a validator. If the validator confirms that the target program refines the source program, it generates a proof. Conversely, it produces a counterexample if discrepancies are detected. Translation Validation is practical and adaptable, efficiently ensuring compiler execution aligns with specifications.

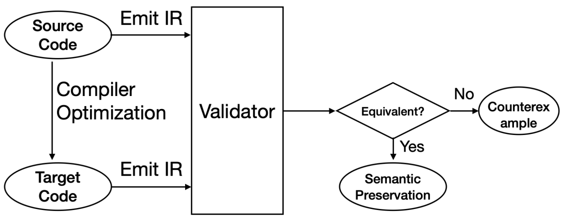


Fig. 1. Workflow of the Translation Validation Approach

III. Overview of The Framework

This section presents our novel translation validation framework, consisting of three major components: the `formal transformation checker`, `LLM-based transformation predictor`, and `fuzzer`. As shown in Figure 2, the framework aims to offer an efficient approach for evaluating the soundness of compiler transformations, particularly in scenarios where traditional formal verification tools struggle.

The workflow starts with a source IR and a target IR transformed from the source IR. In this paper, we use LLVM IR as an example, as demonstrated in Figure 3. The IRs are first processed by the `formal transformation checker`, utilizing the capabilities of established translation validation tools to analyze and verify IR transformations. In this paper, we use Alive2 as the example. The `formal transformation checker` compares the memory states and return values of the transformed code, ensuring no new undefined behavior arises. This involves encoding the source and target IR programs into SMT encodings and submitting these to the SMT solver [3] for verification.

If the SMT solver is able to verify the transformation, our framework reports the transformation validation results (indicating soundness or unsoundness) directly. However, if the SMT solver fails to verify the transformation, due to factors such as complex SMT expressions or unbounded loops, our framework forwards the IR program pair to the `transformation predictor` (Step 1 in Figure 2). After processing through the `prompt encode/decoder`, the input is sent to the fine-tuned LLM models, which generate the prediction results (Steps 2 and 3). Our `transformation predictor` currently supports the fine-tuned models: Mistral-7B [5], Llama2-7B [12], and GPT-3.5 [4]. The fine-tuning process of these LLM models is elaborated in a subsequent section.

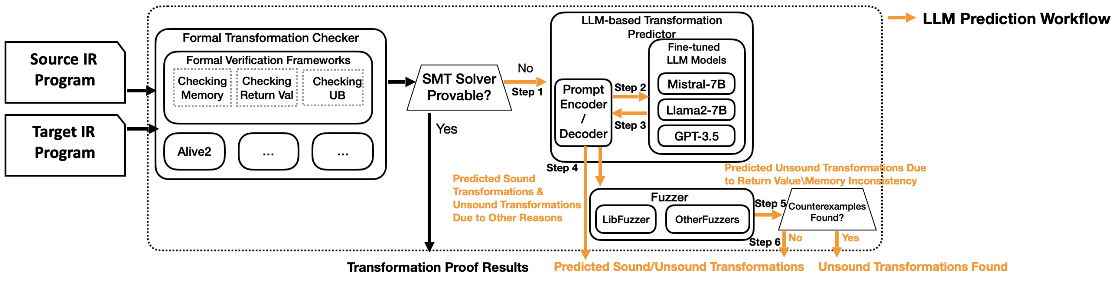
Then, for transformations predicted as unsound due to inconsistent memory or return values, our framework sends both the program pair and the reasons for unsoundness to the `fuzzer` to identify counterexamples (Step 4). If the `fuzzer` finds any counterexamples, the transformation is deemed unsound (Step 5 and 6). For transformations predicted as sound, and for unsound transformations where no counterexamples are found or those caused by other reasons, our framework directly reports their predicted soundness/unsoundness (Step 4).

IV. Implementations

A. Formal Transformation Checker

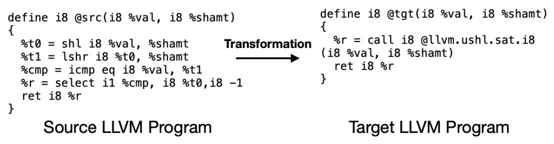
The formal transformation checker is crucial for analyzing compiler transformations. It focuses on verifying memory states, return values, and preventing the introduction of new undefined behaviors (UBs), thereby ensuring transformation correctness and reliability.

1) Checking Memory States: The checker uses existing formal verification tools for a detailed comparison of memory states between the source IR P_{src} and the target IR P_{tgt}. It verifies that memory operations in P_{src} and P_{tgt} are equivalent, denoted as:



Source IR Program
Target IR Program
Formal Transformation Checker
Formal Verification Frameworks
Checking Memory
Checking Return Val
Checking UB
Alive2
...
...
SMT Solver Provable?
No
Step 1
Yes
LLM-based Transformation Predictor
Prompt Encoder / Decoder
Step 2
Step 3
Fine-tuned LLM Models
Mistral-7B
Llama2-7B
GPT-3.5
Step 4
Predicted Sound Transformations & Unsound Transformations Due to Other Reasons
Predicted Unsound Transformations Due to Return Value\Memory Inconsistency
Fuzzer
LibFuzzer
OtherFuzzers
Step 5
Counterexamples Found?
Step 6
No
Yes
LLM Prediction Workflow
Transformation Proof Results
Predicted Sound/Unsound Transformations
Unsound Transformations Found

Fig. 2. Schematic Diagram of the Translation Validation Framework



```
define i8 @src(i8 %val, i8 %shamt)
{
  %t0 = shl i8 %val, %shamt
  %t1 = lshr i8 %t0, %shamt
  %cmp = icmp eq i8 %val, %t1
  %r = select i1 %cmp, i8 %t0,i8 -1
  ret i8 %r
}
```

```
define i8 @tgt(i8 %val, i8 %shamt)
{
  %r = call i8 @llvm.ushl.sat.i8
(i8 %val, i8 %shamt)
  ret i8 %r
}
```

Fig. 3. Example of LLVM programs

$$\forall \sigma_{\text{src}}, \sigma_{\text{tgt}} : \text{Memory}(P_{\text{src}}, \sigma_{\text{src}}) \equiv \text{Memory}(P_{\text{tgt}}, \sigma_{\text{tgt}})$$

2) Checking Return Values: Similar to memory state verification, the checker ensures consistent return values post-transformation, preserving the program's functional behavior. It encodes return values from both source and target programs and employs a SMT solver, such as Z3 [3], to compare these SMT encodings, flagging any discrepancies.

3) Detecting Introduction of New Undefined Behaviors: The checker detects new undefined behaviors (UBs) in the transformed program, ensuring no new UBs are introduced. This involves checking against division by zero, null pointer dereferencing, and other unpredictable or unsafe operations, thereby maintaining the program's correctness and reliability.

B. LLM-Based Transformation Predictor

This subsection introduces our primary innovation: fine-tuning Large Language Models (LLMs) in enhancing the robustness of the compiler transformations.

Currently, our training data originates from the verification outcomes of llvm-project [7] transformations and unit tests conducted by Alive2, while similar data from other tools could also be applied. After eliminating duplicate records, our dataset comprised 32,850 sound transformations and 405 unsound transformations. We randomly selected 40 pairs from these to form our test dataset and allocated the remainder to the training set. For each unsound transformation, we included reasons for the unsoundness, such as memory issues, return values, or the introduction of new undefined behaviors (UB). We then formatted this data into prompts for LLM fine-tuning, utilizing the `prompt encode/decoder` within the `transformation predictor`. We defined the fine-tuning prompt structure as follows:

- **System Content:** "Your task involves analyzing the given IR transformations. On receiving a transformation in the "Transformation: `X` → `Y`" format, analyze and respond in the "Status: `A` Reason: `B`" format. "`A`" denotes the transformation's correctness as CORRECT or UNSOUND. For UNSOUND transformations, list reasons "`B`". Your analysis involves a two-step approach:
  - **Special Value Injection**: Inject specific values into both original and transformed IR to observe and compare behaviors.
  - **Step-by-Step Computation and Analysis**: Execute detailed computations for both IR versions to pinpoint discrepancies.

  Base your analysis on the following to assess soundness:
  - **Undefined Behavior Consistency**: The target should only trigger UB if the source does. New UB in the target renders the transformation unsound.
  - **Return Domain Consistency**: The target's return domain must align with the source's, except when the source triggers UB. A mismatched return domain without source UB suggests unsoundness.
  - **Poison Value Propagation**: The target's return value should indicate poison only if the source's does. Any additional poison in the target signals unsoundness.
  - **Undefined Value Handling**: The target's return value should be Undefined only if the source's is Undefined or poison. Introduction of Undefined values by the target without source justification is unsound.
  - **Return Value Consistency**: The return values

of both the source and target should match when the source is clear of Undefined or poison. Variances under a well-defined source indicate unsoundness.
  - **Memory State Refinement**: Verify that the memory state after target execution refines that of the source's. Memory state inconsistencies suggest unsoundness.
- **User Content:** "$\{$*Source IR*$\} \rightarrow \{$*Target IR*$\}$"
- **Assistant Content:** "Transformation status: $\{$*SOUND/UNSOUND*$\}$ Reason: $\{$*UNSOUND REASON*$\}$"

Choosing the appropriate training data is crucial. Our dataset is imbalanced, with far more sound than unsound transformations. Using the dataset directly could make the model biased. To alleviate the potential bias issue, for sound transformations, we removed 1,874 instances where the source and target code were exactly the same. Then we sampled the sound and unsound data at ratios of 1:1, 2:1, 4:1, and 8:1 to evaluate the performance.

After fine-tuning, when our framework identifies transformations that the SMT solver cannot prove, it forwards the transformation pair to the `transformation predictor`. First, this data is structured using the `prompt encoder`, adhering to the fine-tuning prompt structure. Next, the formatted data is sent to the LLMs. The LLMs then predict the soundness/unsoundness of the transformations. Finally, the `prompt decoder` converts the models' output into a human-readable text format, completing the transformation prediction process.

C. Fuzzer

The `fuzzer` module is a key part of our framework. It aims to find concrete counterexamples that confirm the unsoundness due to inconsistent memory or return values predicted by the `transformation predictor`. For LLVM transformations, our framework uses LibFuzzer [7].

The `fuzzer` takes program pairs and their LLM-predicted unsound reasons given by the framework as inputs. The input format is:

$$\text{Input: } \{\text{Source IR}, \text{Target IR}, \text{Unsoundness Reasons}\}$$

Based on the predicted reasons for unsoundness, the `fuzzer` employs different strategies to fuzz the predictions:

- **Handling Return Value Inconsistencies:** If the unsoundness is caused by return value issues, the `fuzzer` focuses on the return logic of the program pair. It tests both programs under various inputs to spot any output differences.
- **Handling Memory Inconsistencies:** For unsoundness caused by memory, the `fuzzer` examines memory operations in both source and target IRs. It aims to find any differences in memory state after execution, checking that the target IR does not introduce new memory behaviors or access issues not in the source IR.

V. Evaluation Results

In this section, we present the evaluation results of our framework, focusing on assessing the accuracy of LLMs in determining the soundness of compiler transformations. We selected 80 transformations from the `llvm-project` and `Alive2`'s unit tests, maintaining a balanced ratio of sound to unsound results. For locally deployed models such as `Llama2-7B` and `Mistral-7B`, we conducted our runs on an Apple M3 Max 128GB platform. For `GPT-3.5`, we accessed the model via the OpenAI API. Additionally, we assessed the baseline performance of the original `GPT-4` and `GPT-3.5` models without fine-tuning.

We conducted comparison of the fine-tuned `Llama2-7B`, `Mistral-7B`, and `GPT-3.5` models, as well as the `GPT-4` and `GPT-3.5` models without fine-tuning, using the same test data. These models were evaluated based on their accuracy, considering various ratios of SOUND to UNSOUND data in the training set, specifically 1:1, 2:1, 4:1, and 8:1.

As depicted in Figure 4, the fine-tuned `GPT-3.5` model achieved an accuracy of up to 88%, while locally deployable models, like the fine-tuned `Llama2-7B`, reached an accuracy of up to 86%, and the `Mistral-7B` reached an accuracy of up to 79%. The `GPT-4` model, without fine-tuning, attained only an accuracy of 64%, and the `GPT-3.5` without fine-tuning achieved only 50% accuracy. These results demonstrate that LLMs, when fine-tuned with an appropriate dataset, can effectively provide rapid assessments of compiler transformations. Moreover, it shows that properly fine-tuned smaller models, such as `Llama2-7B` and `GPT-3.5`, can outperform larger models like `GPT-4`. This evaluation highlights the potential of LLMs in assessing the soundness of compiler transformations.

A. Evaluation of Deep-learning Accelerator Designs in LLVM IR

We employed our framework to assess Intel deep-learning accelerator designs [2] specified using HeteroCL [6], which features a Python-based domain-specific language (DSL) for specifying hardware designs at a high abstraction level. It then compiles the

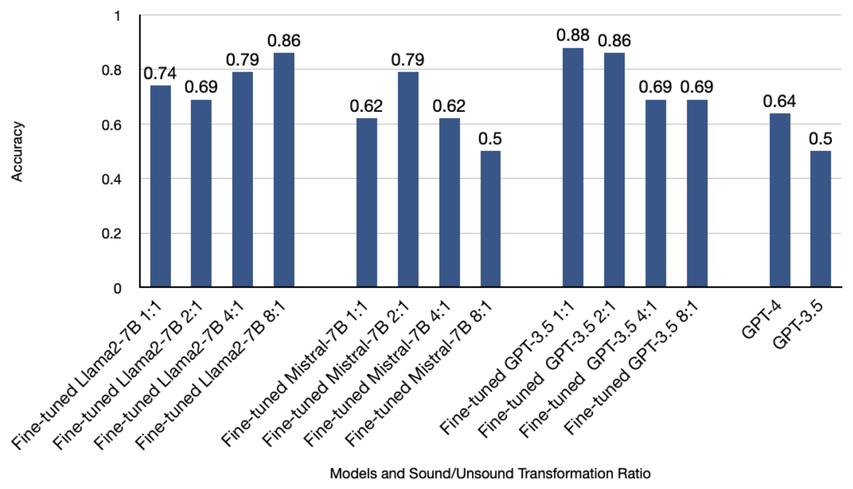


Fig. 4. Evaluation Results on LLVM Transformations

```
def store_2d(sram_base, dram_base, x_size, y_size, x_stride, sram,
dram):
    '''sub-block of store'''
    _, nrows, ncols = sram.shape
    with hcl.Stage("store"):
        def fmutate(y, x):
            tile = sram[sram_base + y*x_size + x]
            dram_idx = (dram_base + y*x_stride + x) * nrows * ncols
            def move_tile(row, col):
                dram[dram_idx + row*ncols + col] = tile[row][col]
            hcl.mutate((nrows, ncols), move_tile)
        hcl.mutate((y_size, x_size), fmutate, 'store_data')
```

Fig. 5. load_2d module from hVTA

hardware from this DSL to a lower-level design language such as LLVM, and that of high-level synthesis. In this evaluation, we utilized one source deep-learning design (named hVTA), and two transformed designs from the source design (named sVTA and uVTA) as examples. These designs adhere to the VTA open-source accelerator specifications[1] and consist of four modules: `load`, `ALU`, `GEMM`, and `store`.

We discovered that individual modules generated LLVM code exceeding the LLMs' 4096 token limit. Therefore, we further divided the modules into smaller, functionally discrete units such as store_2d, load_data, load_uop, and others, totaling 19 modules. Figure 5 shows an example of the store_2d module's code from hVTA, illustrating how it performs tiling according to VTA's SRAM shape and writes data into DRAM. Out of these 19 modules, 12 were not successfully processed by Alive2. For each of these 12 modules from hVTA, sVTA, and uVTA, we used hVTA's modules' LLVM IR as the source program and the respective modules' LLVM IR from sVTA and uVTA as the targets. We employed a fine-tuned GPT-3.5 model, with a SOUND: UNSOUND data ratio of 2:1, to predict the soundness of their LLVM IR. The model indicated UNSOUND results for the pad_top modules in hVTA and uVTA with inconsistent memory. The fuzzer generated a counterexample with `is_min_pad_value` set to `true`. As shown in Figure 6, we found that uVTA altered hVTA's behavior. While the hVTA module directly wrote 0 to SRAM, uVTA's value written to SRAM depended on pad_val.

VI. Limitations and Future Work

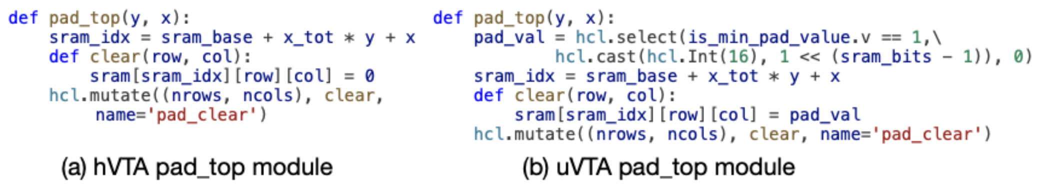
This section discusses current limitations of our approach and outlines potential directions for future work.

Limitations: Our framework, despite promising results, has limitations. One primary constraint is the limited diversity of our dataset. We managed to collect only 405 unsound transformations, which may not sufficiently represent the wide spectrum of potential unsound transformations in the wild. This limitation could affect the generalizability of our model's predictions.

Moreover, our fine-tuning efforts were constrained by platform capabilities. We only conducted the fine-tuning of the Llama2-7B model and Mistral-7B locally. Larger models may yield better performance.

Additionally, the inherent limitations of the models' context window restrict our framework's ability to handle larger programs. This limitation can prevent the processing of more extensive and complex program structures, posing a challenge to the scope of transformations that our framework can accurately evaluate.

[1]github.com/apache/tvm/blob/v0.6/vta/include/vta/hw_spec.h



```
def pad_top(y, x):
    sram_idx = sram_base + x_tot * y + x
    def clear(row, col):
        sram[sram_idx][row][col] = 0
    hcl.mutate((nrows, ncols), clear,
        name='pad_clear')
```

(a) hVTA pad_top module

```
def pad_top(y, x):
    pad_val = hcl.select(is_min_pad_value.v == 1,\
            hcl.cast(hcl.Int(16), 1 << (sram_bits - 1)), 0)
    sram_idx = sram_base + x_tot * y + x
    def clear(row, col):
        sram[sram_idx][row][col] = pad_val
    hcl.mutate((nrows, ncols), clear, name='pad_clear')
```

(b) uVTA pad_top module

Fig. 6. load_2d module from hVTA-uVTA

Future Work: To address these limitations and enhance the robustness of our framework, we plan to undertake the following initiatives in our future work:

- **Synthetic Data Generation**: We are considering developing rules for synthetic data generation. This approach could potentially enable us to generate a larger number of training samples, thereby enhancing the training process and potentially improving the model's predictive accuracy. A larger and more diverse dataset will likely improve the model's ability to generalize and accurately predict the correctness of LLVM transformations.
- **Exploring Larger Models**: We intend to experiment with fine-tuning larger LLM models. Larger models have a greater capacity for learning and might demonstrate superior performance in predicting transformation correctness.
- **Exploring Other IRs**: While our method is currently using LLVM IR and Alive2 as examples, it is not inherently limited to them. In our future work, we also aim to explore its applicability to other IRs, such as MLIR [8], broadening the scope and versatility of our framework.

VII. Summary

In this paper, we have presented a Translation Validation framework integrated with LLMs. We utilized verification data from an existing formal verification tool's processing of LLVM transformations as the training set to fine-tune models such as Llama2-7B, Mistral-7B, and GPT-3.5. This approach aimed to provide a predictive mechanism for transformations that prove challenging for traditional formal verification tools. The evaluation results underscore the potential of LLMs in enhancing the robustness of compiler transformations.

References

[1] The llvm compiler infrastructure. llvm.org. I

[2] Vta implementations in heterocl. I, V-A

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. I, III, IV-A2

[4] L. Floridi and M. Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020. III

[5] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023. III

[6] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. V-A

[7] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004. IV-B, IV-C

[8] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. VI

[9] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. II

[10] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. I

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. I, II

[12] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. III