Source Matching and Rewriting for MLIR Using String-Based Automata

VINICIUS ESPINDOLA, LUCIANO ZAGO, HERVÉ YVIQUEL, and GUIDO ARAUJO,
Institute of Computing - UNICAMP, Campinas, Brazil

A typical compiler flow relies on a uni-directional sequence of translation/optimization steps that *lower* the program abstract representation, making it hard to preserve higher-level program information across each transformation step. On the other hand, modern ISA extensions and hardware accelerators can benefit from the compiler's ability to detect and *raise* program idioms to acceleration instructions or optimized library calls. Although recent works based on Multi-Level IR (MLIR) have been proposed for code raising, they rely on specialized languages, compiler recompilation, or in-depth dialect knowledge. This article presents *Source Matching and Rewriting* (SMR), a user-oriented source-code-based approach for MLIR idiom matching and rewriting that does not require a compiler expert's intervention. SMR uses a two-phase automaton-based DAG-matching algorithm inspired by early work on tree-pattern matching. First, the idiom *Control-Dependency Graph* (CDG) is matched against the program's CDG to rule out code fragments that do not have a control-flow structure similar to the desired idiom. Second, candidate code fragments from the previous phase have their *Data-Dependency Graphs* (DDGs) constructed and matched against the idiom DDG. Experimental results show that SMR can effectively match idioms from Fortran (FIR) and C (CIL) programs while raising them as BLAS calls to improve performance. Additional experiments also show performance improvements when using SMR to enable code replacement in areas like approximate computing and hardware acceleration.

CCS Concepts: • **Theory of computation** → *Grammars and context-free languages*; **Pattern matching;** • **Hardware** → Emerging languages and compilers;

Additional Key Words and Phrases: Idiom recognition, automata, MLIR, rewriting, hardware accelerators

ACM Reference format:
Vinicius Espindola, Luciano Zago, Hervé Yviquel, and Guido Araujo. 2023. Source Matching and Rewriting for MLIR Using String-Based Automata. *ACM Trans. Arch. Code Optim.* 20, 2, Article 22 (March 2023), 26 pages.
https://doi.org/10.1145/3571283

1 INTRODUCTION

Idiom recognition is a well-known and -studied problem in computer science, which aims to identify program fragments [5, 7, 23, 24, 32, 40]. Although idiom recognition has found a niche in compiling technology, in areas like code generation (e.g., instruction selection) [2, 3, 20, 39], its broad application is considerably constrained by how modern compilers work. A typical compiler flow makes a series of translation passes that lowers the level of abstraction from source to machine code, with the goal of optimizing the code at each level. One such example is the Clang/LLVM

Authors' address: V. Espindola, L. Zago, H. Yviquel, and G. Araujo, UNICAMP Universidade Estadual de Campinas - Av. Albert Einstein, 1251 - Cidade Universitária, Campinas - SP, 13083-852; emails: {v188115, l182835}@dac.unicamp.br, {hyviquel, guido}@unicamp.br.

1544-3566/2023/03-ART22
https://doi.org/10.1145/3571283

compiler [30] which starts at a *high-level* language (e.g., C or Fortran) and gradually lowers the abstraction level to AST, LLVM IR, Machine IR, and finally binary code. After lowering from one level to another, program information is lost, thus restricting the possibility of simultaneously combining optimization passes from different abstraction levels.

Until recently, there was no demand for combining optimization passes from distinct abstraction levels, mainly due to two reasons. First, although many languages claim to implement *higher-level* representations, their abstraction levels are not that far above those found in machine code (C code is an example) [9]. Second, general-purpose processors implement very lowlevel computing primitives. However, the recent adoption of hardware accelerators is changing this scenario and pushing the need for optimizations and code generation at higher levels of abstractions (this also explains the renewed interest in Domain-Specific Languages).

To use such accelerators, a compiler should lower some parts of the program while raising others. For example, if an instruction in some ISA extension can perform a tiled multiplication (e.g., Intel AMX or IBM MMA), the compiler should be able to raise the code to this instruction (higher-level) representation while simultaneously lowering the remaining parts of the code. A similar situation also happens when generating code for **Machine Learning (ML)** engines. Thus, any robust approach for idiom recognition should be able to: (a) work on a representation that enables the co-existence of different levels of abstraction; (b) allow both raising and lowering of the program between different levels. Fortunately, recent research has started to address these two requirements. To deal with (a), MLIR [31] enables the interplay of different languages through a common Multi-Level IR. To address (b), Chelini et al. [9] and Lücke et al. [33] proposed techniques that raise the level of abstraction.

This article proposes a source-code-based approach for program matching and rewriting that leverages MLIR for lowering and raising. To achieve that, it makes two assumptions. First, it assumes the existence of MLIR implementations for both matching and replacement codes. Second, it considers source code a more approachable form of pattern description from the user's perspective, simplifying pattern exploration.

Given the discussion above, this article describes Source Matching and Rewriting **(SMR)**, a system that relies on a **macro definition (PAT)** to match and re-write program code fragments using MLIR as a supporting framework. This article aims to validate SMR as a viable tool for pattern rewriting in terms of usability, flexibility, efficiency, and scalability. Unlike other approaches, SMR intends to avoid custom idiom description languages and compilers to simplify its usage and enable idiom exploration, while also providing a sufficiently generic rewriting method to be used with a multitude of languages and goals. In SMR, the *input* and the *idiom* to be identified are lowered to their corresponding MLIR dialects (e.g., FIR [12] or CIL [48]). Then, an approach inspired by early work on tree-rewriting systems [2] is used to match the MLIR idiom pattern against the input MLIR to enable code lowering or raising. It should also be noted that some restrictions, such as disregarding regions with more than one MLIR basic block, were imposed for the initial development of the tool.

Consider, for example, the PAT description in Listing 1 that is divided into two code sections. The first section (lines 1–6) describes the idiom code. The letter "C" at the beginning of the first section (line 1) instructs SMR that the idiom to be detected was written in the C language. In a PAT description, the idiom is encapsulated as a function (lines 2–5) where the arguments are the inputs of the idiom (in this case, dot-product). The second section (lines 6–11) describes the replacement code. It is also declared as a function (lines 8–10) with the same signature as the idiom that it intends to replace (i.e., dot-product). As detailed in the following sections, a PAT description could potentially use any language that can be compiled to MLIR and integrated with SMR. Nevertheless, inter-language/dialect rewrites are not explored in this paper. As future work,

```
C {
 void dot(int N, float *x, int ix, float *y, int iy, float *out) {
  for (int i = 0; i  <  N;  i++)
   *out += x[i*ix] * y[i*iy];
 }
} = {
 #include <cblas.h>
 void dot(int M, float *p, int ip, float *q, int iq, float *out) {
  *out += cblas_sdot(M, p, ip, q, iq);
 }
}
```

Listing 1. Raising dot-product and replacing by a BLAS call using SMR. Idiom and replacement are encapsulated in wrapper functions.

we intend to research how to leverage MLIR inter-dialect functionalities to abstract inter-language rewrites in SMR.

This article is divided as follows: Section 2 reviews other works related to this article. Section 3 provides a background of the techniques used in the SMR design. Section 4 gives an overview of the SMR architecture, and Section 5 details its main algorithms. Section 6 shows the experimental results. Section 7 discusses other applications that can benefit from SMR, and Section 8 concludes the work.

2 RELATED WORK

Pattern-matching techniques have been used for decades to implement compiler optimizations. While early research mainly focused on IR pattern matching for code generation [2, 3, 20, 39], recent works explored more complex patterns that include program control flow [9, 15, 17, 24, 33, 52]. LLVM already provides a pattern matching mechanism, which has been used in [17] to replace computation kernels such as GEMM with optimized implementations. Although their approach results in good speedups, it is not very flexible. Every new *kernel* (i.e., idiom) they introduce needs to be hard-coded into the compiler. MLIR offers a more advanced and generic pattern matching and rewriting tool [31] based on a specific MLIR dialect known as the Pattern Descriptor Language **(PDL)** and a DAG Rewriter mechanism. PDL provides a declarative description of IR patterns and their replacements while also using the generic representation of MLIR to enable that to other MLIR dialects. PDL works quite well with simple patterns, but it is not yet possible to use it to describe more complex idioms such as matrix multiplications [45, 46]. In fact, the generic representation of MLIR is very low-level, and algorithms relying on advanced control structures are complex to express in it. On the other hand, dedicated MLIR dialects can be used to capture idioms for optimization. For example, Bondhugula [8] has matched GEMM idioms and replaced them with BLAS calls to improve performance. But similarly, as in [17], matching/replacement is hard-coded inside the compiler. It is also worth noting approaches like Gareev et al. [21] that, instead of replacing existing code with external optimized libraries, implements a compiler optimization in Polly that can reduce the performance gap with libraries like OpenBLAS.

Several works try to expand LLVM and MLIR pattern matching through domain-specific languages to ease the matching of complex idioms by using custom functional languages [9, 11, 33] or a constraint-based language [24]. For all those approaches, the idiom descriptions are synthesized as LLVM IR or MLIR passes that perform pattern matching. The fundamental difference between those works is that Chelini et al. [9] use pattern matching to raise the abstraction level of the intermediate representation of general-purpose languages enabling domain-specific optimizations, while Lucke et al. [33] mostly focus on simplifying pattern matching for a domain-specific IR.

On the performance side, Chelini et al. [9] already demonstrate their approach for linear algebra operations against standard compilation optimizations and polyhedral optimizations. All such approaches allow a more compact way to match and rewrite idioms. Nevertheless, they require re-compiling the compiler and/or are not very friendly to express idioms as source code is in SMR, two undesirable features when targeting idiom exploration.

Barthels et al. [6] explore the automatic rewriting of linear algebra problems from high-level representations (Julia, Eigen, or Matlab) using pattern matching and rewriting. Their work uses the mathematical properties of linear algebra operations and data structures to derive efficient implementations. Although they demonstrate the potential of raising, their approach only works when the target program uses high-level function calls to algebraic operations.

Similar to SMR, the XARK compiler [5] uses a two-phase approach to match idioms. However, SMR uses automaton string matching, which is more efficient than matching an IR graph representation directly, and starts by analyzing the CDG to quickly eliminate non-relevant idioms. Additionally, XARK reduces idioms' expressiveness more than SMR since it requires linearizing the accesses to multidimensional arrays.

Polly [26] focuses on polyhedral loop transformations, but also uses pattern matching in some cases. In [10], a polyhedral scheduler that can generate loop transformation recipes is presented, focusing on the traversal of large loop transformation spaces and reduced need for auto-tuning. Polygeist [37] is a C front-end for the MLIR Affine dialect that was not released in time to be used by this work. On the other hand, given its potential robustness, we intend to experiment with Polygeist as an alternative to CIL in the future.

Verified Lifting [29] uses a formal verification approach to perform pattern matching and higher-level rewriting. Their approach showed good results but it is specialized to stencil codes and, in some cases, needs to include user annotations to ensure the correct rewrite.

3 BACKGROUND

One of the central tasks in idiom recognition is the ability to pattern match the input program. Pattern matching has been extensively used to design instruction selection algorithms needed in the code generation phase of a compiler [2, 3, 20, 39].

The work proposed herein for idiom recognition (SMR) uses a similar approach to instruction selection but differs in two major aspects. First, instead of representing patterns and the input program in a low-level IR, SMR uses MLIR. This was motivated by the fact that MLIR enables a common IR structure [13] between distinct source languages. For example, in this article, both Fortran and C source codes were lowered to their respective MLIR dialects (FIR and CIL), from which matching and re-writing can be performed. Second, contrary to instruction selection, SMR uses DAG and not tree matching (Section 4) and applies it for both control and dataflow matching.

To better understand the algorithms proposed by SMR, this section covers background material on MLIR (Section 3.1) and tree-pattern matching (Section 3.2).

3.1 The Multi-Level Intermediate Representation

MLIR [31] is a *framework* with several tools for building complex and reusable compilers. Its key idea is a hybrid intermediate representation that can support different levels of abstraction. MLIR uses an interface that relies on a set of basic declarative elements [13]: *Operations, Attributes, Regions, Basic Blocks*, and re-writing rules. Such elements can be configured to form a *dialect* with a specific abstraction-level representation.

An MLIR dialect can be generated from source code to preserve high-level language information during the compilation flow. For example, C and Fortran can use MLIR to respectively translate Clang and Flang ASTs to the CIL [48] and FIR [12] dialects.

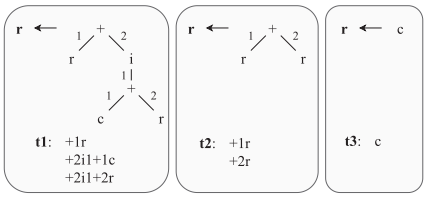


Fig. 1. Tree-patterns (t1, t2 and t3). Adapted from [2].

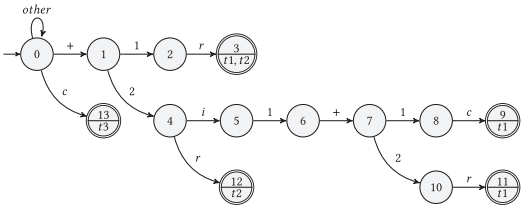


Fig. 2. Tree-pattern automaton. Adapted from [2].

Although each dialect is somewhat unique, all of them must follow a common set of rules imposed by the MLIR interface [13]. The different syntax concepts of the language are implemented by configuring this interface. For example, source code `if-else` clauses from distinct languages can be implemented by configuring MLIR concepts of *regions* and *operations*. As discussed in Section 4, SMR relies on the common rules/structures imposed by MLIR's interface [13] (while also allowing some configuration of its own on a dialect basis) to design a single algorithm that performs idiom matching and rewriting for both CIL and FIR inputs.

Some of the basic MLIR elements are described below:

- **Operations.** Operations in MLIR are similar to those in other IRs instructions: an operation may have input operands, generating an operation result. However, it can also contain attributes, a list of regions, and multiple operation results, among other aspects.
- **Attributes.** Attributes are used to identify specific features of operations. An operation that initializes a constant, for example, may have an attribute that defines its value, as is the case with the "`std.const`" operation, which uses the "`value`" attribute for this purpose.
- **Basic Blocks.** MLIR Basic Blocks **(MLIR BBs)** are defined similarly as in other IRs with a few differences. First, contrary to the classical definition in [4] and [14], MLIR BBs have operations that can contain MLIR regions. Another difference is the concept of *basic block arguments*. These arguments abstract control-flow dependent values indicating which SSA values are available in a block.
- **Regions.** MLIR regions are lists of basic blocks that may correspond to a classical *reducible* CFG region [4, 14], or not, depending on how the language generates its MLIR representation. For example, some languages (e.g., Fortran) implement classical reducible CFG regions (e.g., canonical loops) as an MLIR region with a single basic block, while *irreducible* parts of the CFG are implemented using a region with multiple basic blocks.

3.2 Automaton-Based Tree-Matching

Instruction selection is a well-studied area in compiling technology [4]. During instruction selection, expression trees from a low-level IR of the program (e.g., LLVM Machine IR) are covered using a library of *tree patterns*. Aho et al. [2] proposed a thorough solution based on a *tree-pattern matching* algorithm that leverages the Hoffman and O'Donnel [27] automaton string matching approach to reduce the size of the encoded tree-patterns and improve performance. Their solution was demonstrated in a tool called `Twig`. This article extends [2] to enable CDG and DDG pattern matching algorithms to be used for idiom recognition in MLIR.

Figures 1 and 2 show a quick overview on how `Twig`'s algorithms work. First, tree-patterns in Figure 1(`t1`)-(`t3`) are linearized by listing all paths from the root to a leaf. The result is a set of *path-strings* encoding each tree-pattern. These strings are then converted into an Automaton (Figure 2) where the transitions from states are annotated with the path-string elements, and the last state

```
1 langA {
2   fun (type1 arg1, type2 arg2,...) {
3     matchA
4   }
5 } = langB {
6   fun (type1 arg1, type2 arg2,...) {
7     rewriteB
8   }
9 }
```

Listing 2. Replacing matchA in langA with rewriteB in langB (optional if langB = langA).

associated with the end of a path-string is marked as final. Given that strings from different tree-patterns have common prefixes, they will traverse the same set of states in the automaton, thus reducing the automaton size. For example, consider the path-string (`+1r`) for patterns `t1` and `t2` of Figure 1. In Figure 2, they share the same sequence of states (`0,1,2,3`) finishing at state 3 which is marked final (double circled), and annotated as having recognized path string `+1r` from both tree-patterns `t1` and `t2`. A tree-pattern *matches* if all its path-strings end in a final state.

Aho's approach provides a unified automaton-based representation of all patterns with two note-worthy benefits. First, when merging common path-strings prefixes of each pattern, it preserves memory by reducing the automaton size. Second, the unified representation ensures that any given input string is automatically compared against each path-string of each pattern, greatly reducing the impact on the execution time as the size and number of patterns grow. Both aspects are beneficial when matching against a large number of patterns.

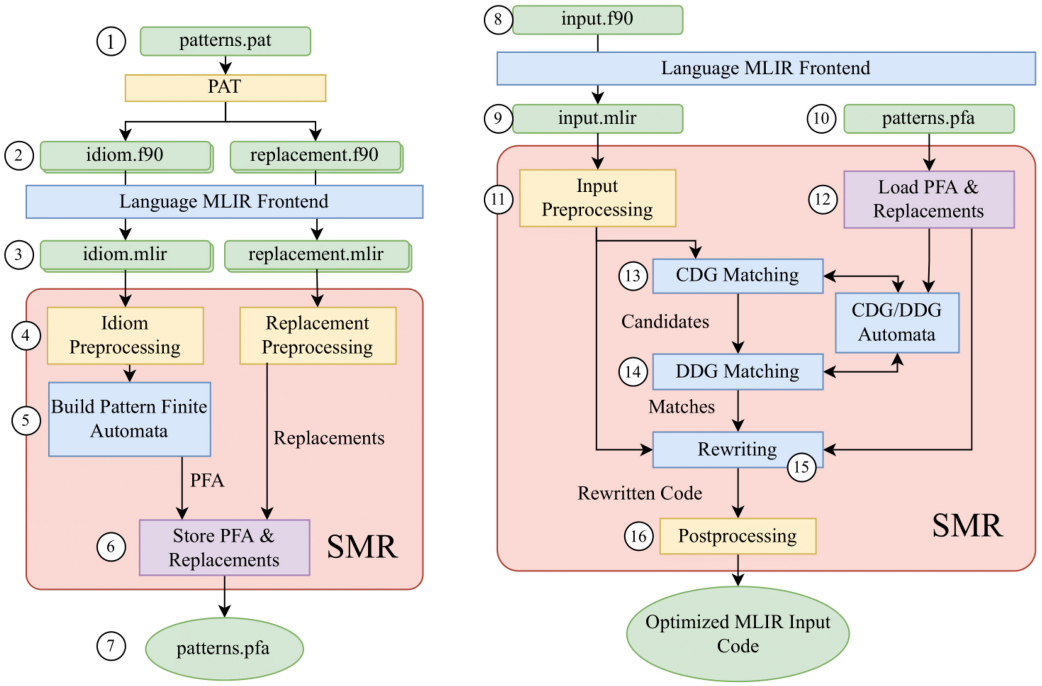
4 AN OVERVIEW OF SMR AND PAT

This section provides an overview of the SMR approach. It first describes PAT, a macro definition that enables the user (or compiler developer) to specify an idiom and its corresponding replacement code. Second, it shows how PAT and SMR are integrated into a Clang/LLVM compilation flow. The reader should be attentive to the following terminology, which will be used from now on: (a) *pattern* is the idiom code to be matched; (b) *input* is the program inside of which SMR will detect the idiom; (c) *replacement* is the code that will replace the matched idiom within the re-written input.

As shown in Listing 2, and anticipated in Section 1, a PAT[1] description is divided into two sections. In the first section (lines 1–5 of Listing 2), the language of the idiom to be matched (`langA`) is specified (line 1), followed by the declaration of the idiom's *wrapper function* (also written in `langA`) that declares the arguments of the idiom pattern and their corresponding types (line 2). The pattern itself is described in the body (`matchA`) of the wrapper function (line 3). The second section of a PAT description (lines 5–9) contains the replacement code that will rewrite the idiom when it matches some input code fragment. Similarly, as in the idiom section, the replacement code language (`langB`) is also specified (line 5). The wrapper function declaration (line 6, written in `langB`) matches the arguments of the idiom to the variables used by the matched input fragment. In the case of an idiom match, the matched input fragment is replaced by the code in `rewriteB`.

The wrapper functions in the pattern and replacement sections of a PAT description serve two, and only two, purposes. First, they make the codes valid, as the front-end must be able to compile them. Second, they act as an interface between the input, pattern, and replacement codes. That said, the wrapper function is not a part of the pattern and will never be matched, only its arguments declarations are relevant. In the matching process, SMR considers the wrapper function arguments'

[1]The EBNF representation of PAT is trivial, and thus we opted to present it in a descriptive format.



(a) Pattern Finite Automaton Building flow (b) Input Matching/Rewriting flow

Fig. 3. Two-steps SMR Compilation Flow.

order and types, as well as the function's body, ignoring the function's and arguments' names. Keep in mind that this does not prevent SMR from matching function calls within the wrapper function, making it possible to match and rewrite function calls as well.

It is expected from the PAT file that each idiom pattern code is semantically equivalent to its respective replacement code. The author of the PAT file must ensure the correctness of such equivalence as SMR does not verify it.

PAT differs from other proposed matching languages in many ways. One remarkable difference should be highlighted, though. In contrast to approaches such as RISE [33], which use their own concepts and languages, PAT describes idiom and replacement codes using regular programming languages, thus considerably simplifying the description.

The SMR algorithm relies on a two-phase approach shown in Figure 3. In the first phase, a set of patterns and their corresponding replacements are designed and stored in a PAT file (`pattern.pat`). The PAT file is then converted to a **Pattern Finite Automaton (PFA)**, using the PFA Builder shown in Figure 3(a). Parsing a PAT file is simple: first, the language identifier is parsed, then the first opening brace is stacked, and the code block is read until the closing brace empties the stack. The process is repeated until all rewrites are parsed. Although this approach might be susceptible to failure in some languages, it did not present any issues with FIR or CIL. A more robust approach could easily be achieved by simply using delimiters composed of multiple characters instead of braces. The flow in Figure 3(a) is based on the FIR compiler. A similar flow can also be used for CIL or any other C-based dialect. Initially, the PAT Builder separates ① each pattern/replacement code available in `patterns.pat` into a pair of Fortran files: `idiom.f90` and `replacement.f90` ②. These files are then compiled using FIR's front-end to generate their corresponding MLIR (i.e., FIR) codes ③. Idiom and replacement FIR codes are then used by the

SMR algorithm to build a PFA automaton ⑤ that is stored into `patterns.pfa` ⑦. The reader should notice that converting a PAT file to a PFA is a task that needs to be done only once. A PAT file (and its corresponding PFA) can be viewed as a library of patterns/replacements that can be used many times by a compiler to detect idioms on arbitrary input programs. In the future, one can envisage specialized PFAs being designed to match idioms in several areas, from linear algebra to telecommunications. This approach can help the user to match patterns on its source code and replace them with instructions from newly designed accelerators.

The second phase of the SMR approach is shown in Figure 3(b). It resembles a regular MLIR compilation flow that takes an `input.f90` ⑧ source code and produces `input.mlir` ⑨ that can be lowered to LLVM IR. Contrary to a regular MLIR compilation flow, the SMR-modified flow can also take as input the PFA file `patterns.pfa` ⑩, (generated using the flow Figure 3(a)) if the user wants to match/replace code in `input.mlir` before lowering to LLVM IR. Both `input.mlir` and `patterns.pfa` are then used by the SMR algorithm (see Section 5) to match/replace the input program with the patterns from `patterns.pfa`. After that, regular MLIR compilation resumes with the lowering to LLVM IR.

5 THE SMR ALGORITHM

In this section, we provide an in-depth description of SMR's core algorithm, as its usage of MLIR for idiom recognition is the main contribution of this article. As mentioned before, SMR relies on automaton-based DAG isomorphism to match idioms against an input program. Two major issues need to be addressed to enable that. First, although idioms are fairly small code fragments and matching their DAGs is fast, input programs can contain thousands to millions of lines of code, making it unfeasible to pattern match idioms against a whole program. To address this, SMR uses a two-phase approach that narrows down the matching search space by (a) selecting a set of candidate fragments in the input program that has a control-flow structure similar to the one in a given idiom (Section 5.1); and (b) from the filtered set of candidates, identifying those which have the same data-dependencies as the desired idiom (Section 5.2). Moreover, although subgraph isomorphism is NP-Complete, the aforementioned filtering stage, together with the fact that idiom DDGs tend to be quite small and similar to trees, avoids potential combinatorial explosions in SMR execution time. Second, depending on the problem, the number of similar idioms that can be matched could be very large, thus increasing the DAG matching algorithm's execution time and memory usage. To optimize this task, SMR encodes idiom patterns as strings and uses automatons to compress them, similarly as proposed in Aho et al. [2]. Automatons can also be easily serialized and saved for reuse, thus removing the need to rebuild the patterns' CDG and DDG automatons. To deal with these tasks, SMR follows a compilation pipeline that implements the following sequence of operations (shown in Figure 3).

Pre/Post-processing: ④ ⑪ ⑯ SMR offers a language-wise pre-processing pipeline to ease the integration of new front-ends and dialects. The input and replacements may be optionally pre-processed in case they do not conform to the SMR constraints (Section 5.4). We call these tasks *normalizations*, as they aim to modify the code into an SMR-compliant structure without altering its computation. For now, we use normalization only to peel off the wrapper function from the idiom code, as only its body is of interest during matching. In the future, a language-wise pre/post-processing pipeline can also be defined to deal with other requirements that might arise with the addition of new compiler front-ends.

Build Pattern Finite Automaton ⑤. After pre-processing, the idiom's MLIR code is used to extract their CDG patterns and encode them as sets of strings. These sets are then used to build the CDG Automaton for matching. Similarly, the pass traverses the ud-chains of said idioms, encoding them

```
1  subroutine sum(test, val)
2    integer :: val, test
3
4    IF (test == 1) THEN
5      val = 1
6    ELSE
7      val = val - 1
8      IF (val == 1) THEN
9        test = 0
10     END IF
11   END IF
12
13 end subroutine
```

Listing 3. Fortran idiom code.

```
1  "fir.if" {
2    SEQ
3  } {
4    SEQ
5    "fir.if" {
6      SEQ
7    } {
8      SEQ
9    }
10   SEQ
11 }
```

Listing 4. Indented idiom control-string.

as strings to build the DDG Automaton. An in-depth description of this process can be found in Sections 5.1 and 5.2. After both automatons are built, they can be serialized.

Store/Load PFA and Replacements. The storing pass ⑥ serializes in a file the automata from ⑤ and the associated replacements so that they may be reused later without the need of re-building the CDG and DDG automatons. Once serialized, the file can be reused by the loading pass ⑫.

CDG Matching ⑬. After the input program has been compiled ⑨ and pre-processed ⑪, SMR traverses the input MLIR code and encodes the input CDG as a set of strings. This set is then fed to the CDG Automaton loaded ⑫ from the PFA file ⑩ to search for a control-flow match. Finally, CDG Matching outputs all candidates (input code fragments) that have a CDG identical to some idiom's pattern CDG.

DDG Matching ⑭. At this step, the candidate idioms resulting from CDG Matching are read by the DDG Matching pass, which builds a set of strings for each candidate based on their MLIR representations. Finally, it feeds the set of strings generated from the input candidates to the DDG Automaton, which is also loaded ⑫ from the PFA file ⑩, for matching.

Rewriting ⑮. After some idiom matches an input program fragment, its corresponding MLIR code is substituted by a function call to the replacement code associated with the matched idiom pattern. A definition of such a function is inserted into the input program to ensure that it is a callable function.[2]

Listing 3 shows an example of a Fortran idiom, and Listing 5, its corresponding MLIR representation. This idiom will be used in the following sections to better explain SMR, focusing particularly on the CDG/DDG Matching algorithms, as they are the two key tasks.

5.1 CDG Matching

As discussed before, the control structure of the generic MLIR representation is organized as a hierarchical representation of operations, regions, and basic blocks. A region in MLIR is defined by a control-flow operation called Region Defining Operation **(RDO)**, and each region may contain RDOs defining nested regions. In Listing 5, for example, the `fir.if` operation in line 7 is an RDO that defines two regions: (a) in the first region are the operations to be performed if the condition is true (lines 8–9); and (b) in the second, the operations performed if it is false (lines 11–22). Region (b) also has a nested RDO (`fir.if` in line 16) defining two other regions nested in (b).

Overall, the CDG matching algorithm works as shown in Algorithm 1. First, CDGMatch takes the MLIR code for the input and idiom patterns (line 1). It then traverses the MLIR code of each

[2]Function in-lining will eventually be used here.

```
"func"() ( {
^bb0(%arg0: !fir.ref<i32>, %arg1: !fir.ref<i32>):
  %c1_i32 = "std.constant"() {value = 1 : i32} : () -> i32
  %c0_i32 = "std.constant"() {value = 0 : i32} : () -> i32
  %0 = "fir.load"(%arg0) : (!fir.ref<i32>) -> i32
  %1 = "std.cmpi"(%0, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
  "fir.if"(%1) ( {
    "fir.store"(%c1_i32, %arg1) : (i32, !fir.ref<i32>) -> ()
    "fir.result"() : () -> ()
  },  {
    %2 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %3 = "std.subi"(%2, %c1_i32) : (i32, i32) -> i32
    "fir.store"(%3, %arg1) : (i32, !fir.ref<i32>) -> ()
    %4 = "fir.load"(%arg1) : (!fir.ref<i32>) -> i32
    %5 = "std.cmpi"(%4, %c1_i32) {predicate = 0 : i64} : (i32, i32) -> i1
    "fir.if"(%5) ( {
      "fir.store"(%c0_i32, %arg0) : (i32, !fir.ref<i32>) -> ()
      "fir.result"() : () -> ()
    },  {
      "fir.result"() : () -> ()
    }) : (i1) -> ()
    "fir.result"() : () -> ()
  }) : (i1) -> ()
  "std.return"() : () -> ()
}) {sym_name = "_QPsum", type = (!fir.ref<i32>, !fir.ref<i32>) -> ()} : () -> ()
```

Listing 5. MLIR FIR code of idiom in Listing 3.

ALGORITHM 1: Control-Dependency Graph Matching

```
1:  function CDGMatch(Input,Patterns)
2:      ▷ Input: MLIR input code
3:      ▷ Patterns: set of MLIR pattern codes to match
4:
5:      ▷ Build automaton finite state machine from pattern's CDG strings
6:      patStrings ← []
7:      for each pat ∈ Patterns do
8:          ▷ pat: a single pattern code
9:          string ← StringyPatCDG(pat)
10:         patStrings.append(string)
11:     CDGAutomaton.build(patStrings)
12:     ▷ Generate the Input CDG string representation
13:     inStrings ← StringyInCDG(Input)
14:
15:     ▷ Find input code match candidates
16:     Candidates ← {}
17:     for each inString ∈ inStrings do
18:         patternIndexes ← CDGAutomaton.run(inString)
19:         rdo ← getRootRDO(inString)
20:         if patternIndexes ≠ [] then
21:             Candidates.insert(rdo)
22:
23:     ▷ Return RDO set with match candidates
24:     return Candidates
```

idiom pattern (`pat`), building a string representation of the sequence of RDOs that define the pattern's MLIR control-flow regions (lines 5–10). This way, the various idiom CDGs are stored as a set of strings (`patStrings`) where each string represents a sequence of control-flow regions for one idiom's MLIR code. We call these strings *control-strings*. Each control-string originates in an RDO operation inside the MLIR code. The resulting set of idiom control-strings (`patStrings`) is then used to build the CDGAutomaton (line 11). Similarly, the input CDG (line 13) is also generated resulting in `inStrings`. The set of input control-strings (`inStrings`) is then matched against CDGAutomaton (CDGAutomaton.Run) to search for input code fragments that have a similar

```
fir:
  cmpf:
    must-match-attr: "predicate"
  if:
    must-match-attr: "predicate"
```

Listing 6. FIR dialect configuration example.

```
cil:
  constant:
    must-match-attr: "value"
  global_address_of:
    must-match-attr: "global_name"
```

Listing 7. CIL dialect configuration example.

control-flow structure. An input code fragment is said to *control-match* an idiom if all elements of its control-string take the exact same sequence of states in the CDG automaton as the pattern idiom and end in a final state. Finally, CDGMatch returns all input fragments (`Candidates`) that control-match an idiom pattern.

To ease the understanding of Algorithm 1, please consider the Fortran idiom of Listing 3 and its corresponding CDG control-string (indented) representation (Listing 4). There, curly brackets delimit regions, `SEQ` are non-empty sequences of non-RDO MLIR operations, and strings (e.g., ``''fir.if''``) are RDOs.

Notice that the wrapper function in Listing 3 (lines 1–2) is not a part of the matching pattern. As previously mentioned, it acts only as an interface between input, pattern, and replacement. Therefore, the control-string starts on the first RDO of Listing 4 (line 1), which corresponds to the first `if-else` clause of Listing 3 (line 4). Such RDO is considered to be the RDO that defines the CDG string, which is retrieved at line 20 of Algorithm 1.

5.2 DDG Matching

Given the input candidates from the CDG match, SMR then proceeds to select from *Candidates* those that have a Data Dependency Graph **(DDG)** match to some idiom. If this happens, that idiom is said to be *matched*.

This task is performed by Algorithm 2 which takes as input the idiom *Patterns* and input *Candidates* (Algorithm 1 output). The goal of DDGMatch is to model the input code fragments in *Candidates* and the idioms in *Patterns* as DDGs (*Data Dependency Graphs*) and verify if they are isomorphic. Input and idiom DDGs are built from their generic MLIR representations using a combination of ud-chains and regions. Also, relevant particularities of a dialect are encoded in the strings through SMR's dialect-wise integration, allowing important attributes to be matched, and irrelevant ones, ignored.

Listings 6 and 7 exemplify the dialect-wise configuration. In FIR, for instance, we must ensure that the `predicate` attribute of the `cmpf` operation is matched because this operator identifies the comparison being used by the `if` clause. Similarly in CIL, the `constant` operation must take into consideration the `value` attribute, since this attribute identifies the constant being used. These configurations are then embedded into the DDG string representation so that it must be matched as well. For a `fir.if` operation to match with another, for example, the `predicate` attribute must also match. In other cases, we might wish to ignore attributes, such as the `bindc_name` from operation `fir.alloca`. This attribute is a string with the source code variable name. Embedding it into the DDG would cause the pattern to only be accepted if the variables' names matched as well, which is an undesired limitation. Currently, this configuration is hard-coded, but we plan to move it to a YAML that will be parsed by SMR.

Two methods are used to build DDGs: `BuildPatDDG` (line 9) and `BuildInDDG` (line 19). This separation is required due to a difference between the input and pattern DDGs. The wrapper function input arguments in a pattern idiom act as wildcards, not as regular IR values. A wildcard is a node that can match any operation, as long as this operation produces a result that has the same type as the function argument, allowing us to separate where the input ends and the pattern begins in the match.

ALGORITHM 2: Data-Dependency Graph Matching

```
1:  function DDGMatch(Candidates, Patterns)
2:      ▷ Candidates: MLIR operations filtered by CDG matching
3:      ▷ Patterns: List of MLIR code patterns to match
4:
5:      ▷ Generate the DDG stringset representation of each pattern
6:      patStrings ← []
7:      for pat ∈ Patterns do
8:          ▷ pat: single pattern code
9:          ddg ← BuildPatDDG(pat)
10:         dataStringSet ← StringfyPatDDG(ddg)
11:         patStringSets.append(dataStringSet)
12:
13:     ▷ Build the DDG automaton to resolve stringset matching
14:     DDGAutomaton.build(patStringSets)
15:
16:     ▷ Get which input candidates matches which patterns
17:     matches ← []
18:     for each rdo ∈ Candidates do
19:         ddg ← BuildInDDG(rdo)
20:         dataStringSet ← StringfyInDDG(ddg)
21:         patternIndexes ← DDGAutomaton.run(dataStringSet)
22:         for each i ∈ patternIndexes do
23:             matches[i].append(rdo)
24:
25:     ▷ Return a list with the matches of each pattern
26:     return matches
```

Although Algorithm 2 seems simple, it hides a complex set of tasks that considerably extends the approach originally proposed by Aho et al. [2]. These tasks are performed by two key functions that are described in detail below: (a) DDG Building (lines 9 and 19); and (b) DDG Stringfying (lines 10 and 20).

5.2.1 Building DDG. One of the advantages of operating SMR on top of a generic MLIR representation is that it can be interpreted directly as a Rooted Directed Acyclic Graph **(RDAG)** by using the MLIR regions and *ud-chains*.

The idiom in Listing 3 and its corresponding MLIR representation in Listing 5 are used here again to show the workings of Algorithm 2. To illustrate that, please refer to Figures 4 and 5, which shows the steps required to build the DDG of the MLIR code in Listing 5.

From the ud-chains shown in Listing 5, it is easy to build a DAG that serves as the basis for the final DDG. Each MLIR operation corresponds to a node in the graph, and it is labeled by a string composed of the name and relevant attributes of the operation. Note that this identification is not unique: there may be different nodes with the same string label. To connect these nodes, we convert the MLIR ud-chains to directed edges, where the source is the MLIR operation that uses a variable, and the destination is the operation that defines that variable. Since there are operations in which the order of arguments matters (subtraction, for example), we also number the outgoing edges of an operation according to the index of the input operand: the edge corresponding to the first operand is enumerated 1, the edge of the second is 2, and so on. This representation is illustrated by the graph in Figure 4. Observing said graph, one can notice two problems that prevent it from being used as a representation of the Listing 5 code: *control-flow incompleteness* and the absence of a root.

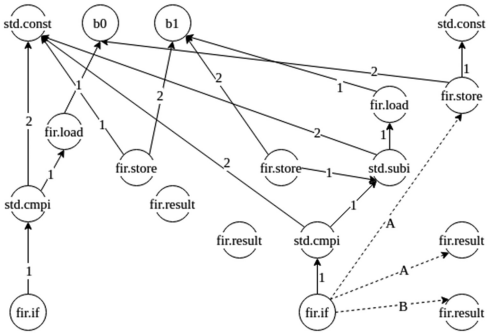


Fig. 4. Building ud-chains DDG.

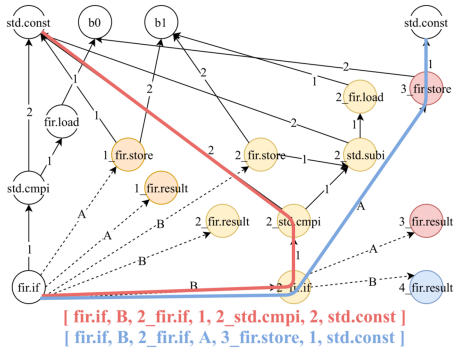


Fig. 5. Coloring MLIR regions, adding root, and extracting data-strings.

The MLIR control regions of Listing 5 are not represented in the graph of Figure 4. Therefore, there is a loss of information regarding the control structure of the code in Listing 5, making it incomplete. A possible solution for this is to assign a color to each region in Listing 5 so that all operations (graph nodes) are colored according to the region in which they are located. To represent this coloring, each color is identified with an integer ID prefixed on the string representation of the node. For example, if a node is in region 2 (yellow), e.g., `std_cmpi`, it will be labeled `2_std.cmpi`. This modification allows us to group MLIR operations according to their respective regions, preserving control-flow information. Such coloring is exemplified in Figure 5.

By analyzing Figure 4, one can notice that there are multiple nodes without incident edges, making the graph disjointed and rootless. Such nodes represent MLIR operations that do not define SSA variables and thus are not destinations of any edges created by the ud-chains in Figure 4. Examples of such MLIR operations are `fir.if`, `fir.store` and `fir.result`. From now on, these nodes will be called *potential roots*. As described below, two issues need to be addressed for the graph to be rooted.

Finding Region Edges. First, the disjoint DDG graph components must be connected. To address that, let us define *Region Edges*, which are labeled with capital letters in Figure 5. Regions of an MLIR operation are also ordered, and therefore these edges are created alphabetically: the first region has edge A, second, edge B, and so on. A region edge $X \xrightarrow{R} Y$ is created whenever there is a potential root node Y that lies within a region R defined by the RDO X. For example, in Figure 5, operation `fir.store` of line 8 of Listing 5 is in region A defined by operation `fir.if` (line 7 of Listing 5). Thus, in Figure 5, `fir.if` $\xrightarrow{A}$ `fir.store` is a region edge. All region edges in the figure are marked as dashed lines.

Identifying DDG Root. By combining ud-chains and region edges, we obtain the rooted graph of Figure 5. Since the wrapper function is not a part of the match, the outermost `fir.if` operation will not be linked to its parent wrapper function. Thus, as long as there are no sequential RDOs in the wrapper function's body (which is one of the constraints mentioned in Section 5.4), there will be only one root RDO after linking the ud-chains and region edges. In Figure 5, such an RDO is the `fir.if` root.

5.2.2 Building the DDG Automaton. To build the `DDGAutomaton` (line 14), the pattern idioms DDG must be represented as *data-strings*. This conversion is done by the `stringifyPatDDG` method (line 10), which takes a pattern DDG and returns a set of data-strings (i.e., `patStringSet`) that are stored into `patStringSets`. After the `DDGAutomaton` is built, the DDG of `Candidates`'

```
grammar ::= control_string

control_string ::= op_name '|' attributes '|'
      regions+

regions = '{' control_string '}'

attributes ::= attribute_entry ',' attributes
             | attribute_entry

// See MLIR documentation for the following:
//  - attributes
//  - attribute_entry
```

Listing 8. Control-strings EBNF grammar.

```
grammar :: = data_string

data_string ::= data_element ',' data_string
              | data_element

data_element ::= operation
               | argument_edge
               | region_edge

operation ::= region_color '|' (ret_type '|')?
      op_name ('{' attrs_list '}')?

attrs_list ::= attribute_entry ',' attrs_list
             | attribute_entry

ret_type ::= type
op_name ::= [^"\m\f\v\r]+
argument_index ::= [0-9]+
region_color ::= [0-9]+
region_edge ::= [A-Z]+

// See MLIR documentation for the following:
//  - type
//  - attribute_entry
```

Listing 9. Data-strings EBNF grammar.

RDOs are extracted (line 18–19) and are converted to a set of strings by function `stringifyInDDG`, producing `dataStringSet` (line 20), which is then fed to the automaton (line 21) for matching.

The conversion from a DDG to `dataStringSet` is trivial. Just list all possible paths from the root to the leaves of the DDG, so that each path is composed by the concatenation of the identifiers labeled at the nodes and edges. This process is illustrated in Figure 5.

The blue path data-string shows the data-dependency execution path that starts at the region defined by the root `fir.if` operation in line 7 of Listing 5 (line numbers from now on refer to that listing). From there, the path goes to the `fir.if` operation (line 16) that works as an RDO of an inner (yellow) region in Figure 5. It then proceeds to operation `std.cmpi` (line 15) which uses the constant `std.const %c1_i32` (line 3). On the other hand, the red path (i.e., data-string) starts at the same root `fir.if` (line 7), also continues to the inner region `fir.if` (line 16), but it then diverges through region edge A, reaching the `3_fir.store` operation (line 17) of the pink region, which uses `std.const %c0_i32` (line 4). Applying this process to every path of an idiom pattern DDG will result in a set of data-strings (`dataStringSet`) representing the pattern.

5.3 Idiom Re-writing

The result of an idiom match is a bijection between an idiom and fragments of the input code. This bijection makes the task of idiom re-writing trivial. SMR needs only to remove from the MLIR code all the operations that can be reached from the root of the DDG of Figure 5, and then replace the DDG root operation with a call to the wrapper function of the idiom replacement code in the PAT description. Since the removed code might use MLIR variables defined on the parent regions, the rewritten function receives these variables as arguments, ensuring the rewritten code will have access to them. Global variables behave a bit differently: instead of passing them as arguments, they are accessed by special operations and attributes. In Listing 1, the dot product C idiom is replaced by a call to its replacement code which eventually calls `cblas_sdot` from the CBLAS library (line 9).

5.4 SMR Limitations

This section is dedicated to discussing some of the limitations that were identified when developing SMR and its source-based pattern description approach.

5.4.1 MLIR Structure Restrictions on Pattern Idioms. SMR adopts a few restrictions regarding the pattern idiom MLIR representation, as modeling a complete and correct DDG for any MLIR code is complex. This first version of SMR was designed to handle only the cases of idioms that are reducible CFGs as defined in [4]. Moreover, a set of additional constraints have been adopted to accelerate the design of SMR: (a) idiom patterns may not have sequential RDOs in the wrapper function's body (only nested RDOs). For example, if an input idiom is composed of two sequential (or non-nested) loops, it cannot be matched using a single pattern idiom. This does not prevent the user from matching the input partially by writing two pattern idioms, one for each loop; (b) Regions must contain exactly one basic block; otherwise, the DDG would have to model the possible paths between these blocks; (c) Operations must define at most one result operand. To allow more than one result would require each edge to encode not only the input operand position but also the position of the operation result being used; (d) Variadic operations are not supported. Removing some constraints is possible, but to speed up the design of the initial version of SMR, this was left as future work.

5.4.2 Sensibility to Frontends and Dialects. A disadvantage of describing patterns in source code is the dependency on frontends that can lower these codes to MLIR, and their variations over time. However, by relying on MLIR common structure, SMR can also be updated with relative ease. If a new FIR version is released, for example, we would have to update dialect-wise configurations of SMR, and then recompile existing Fortran PAT files in order to reuse them. It is also worth noting that, since the PAT file idioms are described with source code, it can be recompiled for multiple frontends of the same language. That said, some limitations remain. For instance, as mentioned above, both pattern and replacement must be considered valid by the frontend being used in order to compile the PAT file to MLIR, which is partially the reason we adopted the concept of wrapper functions. Source code can also enforce types to every variable, meaning idioms are type specific.

5.4.3 Limited Pattern Generality. MLIR's flexibility acts as a double-edged sword in this aspect. While allowing us to easily adapt the algorithm to multiple dialects and, consequentially, languages, its high-level configurable representation makes it difficult to ensure correctness without using a strict representation. A lot of information must be inserted into the DDG, such as ordered regions, ordered operands, operation attributes & types, and so on. Each extra piece of information restricts the pattern, reducing its generality. When creating the DDG, it was difficult to find a correct yet generic model: therefore, patterns do not generalize well: small variations of commutative operations are enough to prevent matches due to the numbered operands; region-edges enforce an order among operations even if they do not depend on each other; the existence of type-specific operations makes it difficult to use the same match for different types. By using their dedicated idiom description methods, some approaches can better generalize patterns: PDL [45] can specify constraints for elements of its patterns to reduce generality only when necessary; IDL [24] extends the usage of constraints to describe and match idiom's semantics entirely as a constraint satisfaction problem; KernelFaRer [17] contributes with reusable extensions of the LLVM pattern-matching framework that can recognize complex idioms; MLT [11] uses a custom language to describe and match the semantics of tensor operations. These approaches are further discussed in Section 6. Most of SMR's problems regarding generality can be tackled, though the feasibility of the possible solutions must be properly evaluated. Hence, such evaluation was left as future work. For example, regarding commutative operations, the DDG could be generalized to remove numbered edges for input operands, as a tradeoff; this would complicate the matching process. A similar solution could be applied for region-edges, also complicating the match as a tradeoff. Type-specific operations, among other variations, could also be circumvented by generating multiple pattern

variations. Another possible workaround for generality is to use canonicalization passes [17] to reduce variations on both input and pattern.

5.4.4 FIR vs. CIL. As previously mentioned, the lowering process from source code to MLIR depends on the choices of the dialect designer. Hence, similar semantic clauses can be lowered in different ways. An example of such is the difference between CIL and FIR regarding the lowering of `if-else` clauses. While FIR directly handles the clause with regions, CIL uses the traditional basic blocks method with branching comparison operations, generating a lower-level MLIR representation than FIR. That said, FIR can still fall into the same scenario when unstructured code or branching operations are in the input code. If a Fortran `EXIT` statement is used within a loop, for example, the IR generated by such loop falls directly into an MLIR basic block representation. The same happens when dealing with `SELECT CASE` statements. In most other aspects, CIL and FIR are quite similar. However, while FIR is currently well supported by the community, CIL has not received any official contributions since it was presented to the LLVM community, rendering it a more crude MLIR dialect when compared to FIR. Regardless, at the writing of this article, and to the best of our knowledge, CIL is the only available MLIR dialect representation for C/C++.

6 EXPERIMENTAL RESULTS

This article aims to propose and validate SMR, a flexible programmer-friendly approach for idiom matching and rewriting. We do not seek to evaluate the performance of the rewritten programs for other established polyhedral-based techniques, as this has already been explored [9]. With this in mind, five sets of experiments have been designed to evaluate SMR. First, a set of Fortran programs was used to validate the correctness of the rewritten code by demonstrating the expected speedups. Fortran was selected because its corresponding MLIR dialect (FIR) is quite stable and well maintained by the community, while C's dialect (CIL) is still in a very brittle state. In the second set of experiments, we focused on efficiency by measuring the impact of SMR matching/replacement on the standard FIR compilation time. The third experiment focused on flexibility by evaluating SMR's ability to match another language dialect (C/CIL), and to measure the corresponding coverage. The fourth set of experiments focused on evaluating SMR's scalability as the size of the source input and the number of patterns increased. The last set of experiments measured the time taken by the flow in Figure 3(a) to build the PFA automaton for a set of patterns. Finally, to conclude the results, we present a brief qualitative comparison to other works illustrating which kernels we were able to match.

All experiments were performed using a dual Intel Xeon Silver 4208 CPU @ 2.10 GHz with 16 cores total and 191 GiB of RAM running Ubuntu 20.04. As for the software tool-chain, the following commits/versions have been used: (a) Flang (FIR) commit `8abd290` [43]; (b) CIL commit `195acc3` [42]; (c) LLVM/MLIR commit `1fdec59` [44]; (d) `OpenBLAS` version `0.3.20` [50]; and (e) GFortran version `9.4.0`. Experiments used programs from Fortran Polybench 1.0 benchmark [38], and a set of six large well-known C programs from different application domains (top labels of Table 1) that perform intense arithmetic computations, extracted using the Angha tool [16]. All experiments were executed following up the benchmark execution guidelines. They used standard reference inputs and were executed 5 times showing small execution time variations (<10%). SMR has detected no false-positive idioms in all experiments.

6.1 Usability

In the first experiment, a set of idioms targeting Fortran double-precision BLAS kernels have been designed using the PAT language and applied to Fortran Polybench programs for matching. Idioms were substituted by the corresponding Fortran BLAS calls (Listing 10, for example), and execution

```
f90 {
subroutine p3mm_double(a, b, e, ni, nj, nk)
  double precision, dimension(nj, nk) :: b
  double precision, dimension(nj, ni) :: e
  double precision, dimension(nk, ni) :: a
  integer :: ni, nj, nk

  do i = 1, ni
    do j = 1, nj
      e(j,i) = 0.0
      do k = 1, nk
        e(j,i) = e(j,i) + a(k,i) * b(j,k)
      end do
    end do
  end do
end subroutine
}={
subroutine p3mm_double(a, b, e, ni, nj, nk)
  double precision, dimension(nj, nk) :: b
  double precision, dimension(nj, ni) :: e
  double precision, dimension(nk, ni) :: a
  integer :: ni, nj, nk

  external :: dgemm

  call dgemm('N', 'N', nj, ni, nk, 1.0D0,
             b, nk, a, nj, 0.0D0, e, nj)
end subroutine
}
```

Listing 10. PAT for Polybench's 3-mm kernel.

```
f90 {
subroutine atax_double(a, x, y, tmp, nx, ny)
  double precision, dimension(ny, nx) :: a
  double precision, dimension(ny) :: x
  double precision, dimension(ny) :: y
  double precision, dimension(nx) :: tmp
  integer :: nx, ny

  do i = 1, nx
    tmp(i) = 0.0D0
    do j = 1, ny
      tmp(i) = tmp(i) + (a(j, i) * x(j))
    end do
    do j = 1, ny
      y(j) = y(j) + a(j, i) * tmp(i)
    end do
  end do
end subroutine
}={
subroutine atax_double(a, x, y, tmp, nx, ny)
  double precision, dimension(ny, nx) :: a
  double precision, dimension(ny) :: x
  double precision, dimension(ny) :: y
  double precision, dimension(nx) :: tmp
  integer :: nx, ny

  external :: dgemv

  call dgemv('T', nx, ny, 1.0D0, a,
             ny, x, 1, 0.0D0, tmp, 1)
  call dgemv('N', ny, nx, 1.0D0, a,
             ny, tmp, 1, 0.0D0, y, 1)
end subroutine
}
```

Listing 11. PAT for Polybench's atax kernel.

Table 1. Matching with CIL and CBLAS Idioms

Idiom	Darknet [41]	Cello [28]	Exploitdb [47]	Ffmpeg [18]	Hpgmg [1]	Nekrs [19]	**Total**
saxpy	1						1
scopy	1						1
sdot	1			1			2
sgemm	4						4
scall	2						2
ddot		1			1	2	4
dgemm			1			3	4
dgemv						1	1
dscal						3	3
Total	9	1	1	1	1	9	22

times were measured with the LARGE_DATASET option.[3] As shown in Figure 6, when using the FIR front-end together with SMR idiom detection and Fortran BLAS replacement (SMR+BLAS, blue bar), all programs showed speed-ups, ranging from 3× to 518×. Listings 10 and 11 illustrate the PAT language straightforward representation for describing Polybench's kernel rewrites, reiterating SMR's usability. For some programs, SMR could not replace the kernel with a single BLAS call, either because there is no matching BLAS call for the kernel or because it does not meet SMR

[3]Due to the differences in execution times, y-axes were broken, and programs separated into two groups.

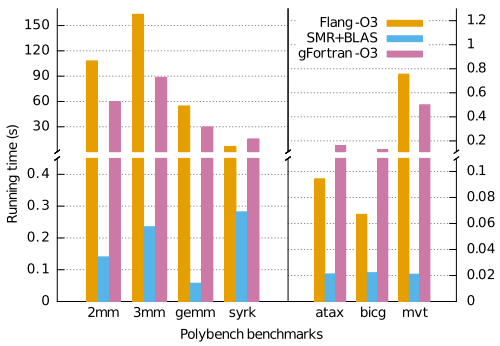


Fig. 6. Polybench running time after BLAS replacement.

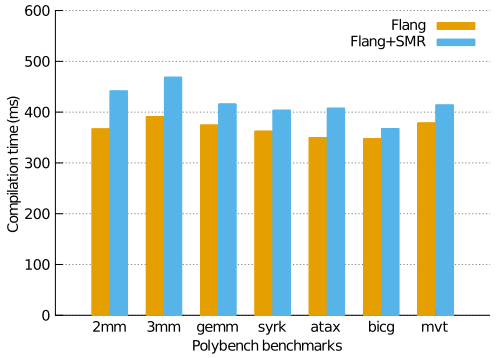


Fig. 7. FIR compilation time with/without SMR+BLAS.

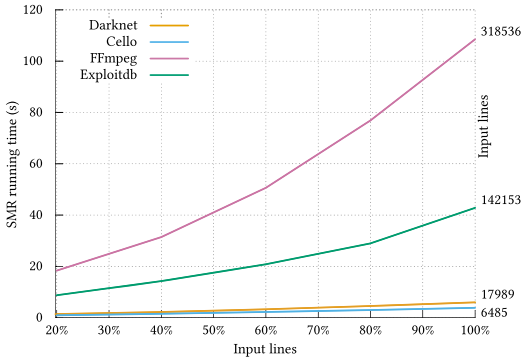
restrictions. In such cases, partial replacements by multiple BLAS calls occurred. This resulted in speed-ups, as in the case of `2mm` (replaced by two GEMMs), `atax` (partially replaced by two GEMVs, as shown in Listing 11) and `bicg` (also partially replaced by GEMVs). Other programs from the benchmark were neither fully nor partially rewritten (e.g., `syr2k`), given they are composed of sequences of RDOs, an SMR restriction, and could not be separated into multiple BLAS calls. As discussed before, this restriction is a solvable issue, and work is underway to address it.

In the second set of experiments, compilation times for Fortran Polybench programs were measured using FIR compilation with and without SMR+BLAS (Figure 7). As shown in the figure, the compilation overhead ranged from 20 ms to 78 ms.

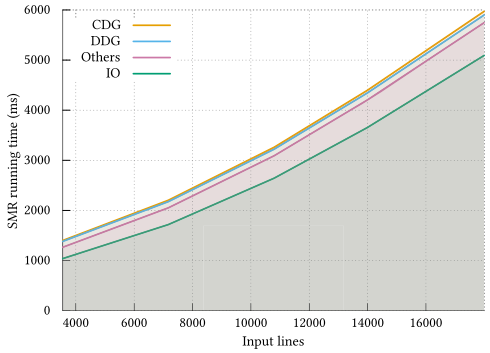
In the third set of experiments, nine idioms associated with CBLAS calls were used with the CIL front-end to perform idiom matching on six C programs. The idioms used ranged from level 1 routines, such as dot products, to level 3 routines, such as GEMMs. Program re-writing was skipped due to the unstable state of the CIL front-end. Table 1 shows the number of idiom occurrences detected per input program. Overall, 22 idiom occurrences were detected in all six programs, with Darknet and Nekrs matching nine occurrences each, respectively, using five and four distinct idioms.

6.2 Scalability

In the fourth set of experiments, four C programs used in Table 1 were selected to evaluate scalability. The source code of each program in Figure 8(a) was divided into groups ranging from 20% to 100% of the total program's size. The number at the end of each curve indicates the total number of lines (100%) of each program. For all curves, we observed a quasi-linear increase in the run time. In Figure 8(b), a more detailed breakout of the execution time was performed for Darknet, the program with the most matches. As shown, the bulk of the computation lies in the IO operations (reading source code files and retrieving their compiled code). The CDG and DDG matching portion of the execution time is where the SMR algorithm complexity lies. In Figure 8(b), the performance of the CDG matching algorithm is noteworthy: despite generating the CDG for the whole program, its execution time represents a tiny fraction of SMR's total run time, corroborating that program control detection is a fast and lightweight task. Meanwhile, DDG matching is a much heavier process: even after CDG matching filters out the majority of the input code (≈90%), the candidates elected to proceed to DDG matching compose a substantial part of the execution time. This reiterates the need for a two-step matching, as running DDG matching on the entire input code would be heavily detrimental to SMR's performance.



(a) Evaluation for 4 programs.



(b) Darknet breakout.

Fig. 8. SMR input scalability, using 95 patterns.

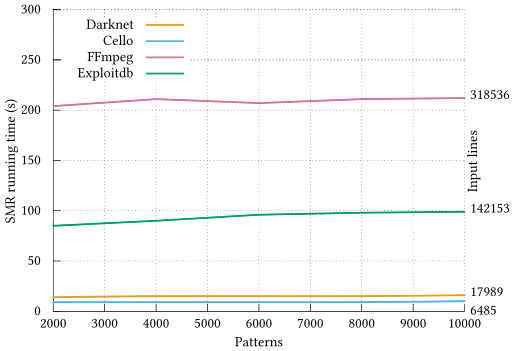


Fig. 9. SMR pattern scalability for four programs using up to 10,000 patterns.

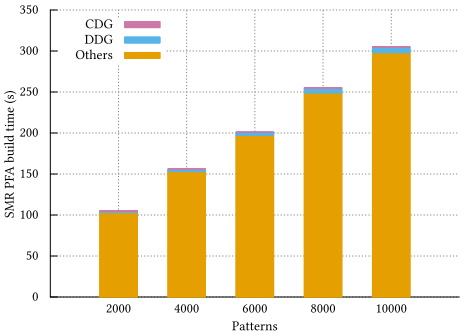


Fig. 10. SMR PFA build time.

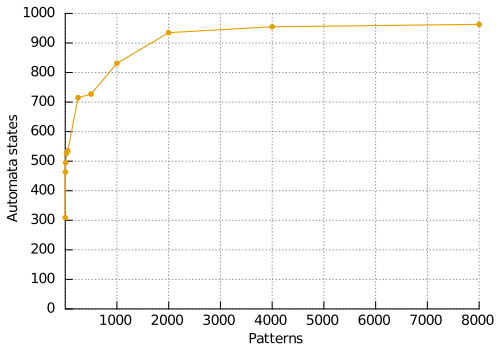


Fig. 11. SMR's automaton prefix merging.

Pattern scalability was evaluated in Figures 9, 10, and 11. The intention here is to evaluate how increasing the number of patterns in the automaton representation affects SMR's performance. The reader should notice that, in these experiments, the PFA file has been already produced and thus the time to build the patterns' CDG and DDG automatons (Figure 10) is not taken into consideration. As shown in Figure 9, little or no variation in SMR execution time was detected while increasing the number of patterns. In Figure 11, we observe the automaton prefix merging in action. While

Table 2. Different Kernels Matched by Each Tool

	Tools			
Patterns	MLT	KernelFaRer	IDL	SMR
atax	X			X
bicg	X			X
mvt	X			X
gemver	X			
gesummv	X			X
2mm	X	X	X	X
3mm	X	X	X	X
gemm	X	X	X	X
syrk				X
symm				X
syr2k		X		
trmm				X
jacobi-3d			X	X
SpMV			X	X
Polyhedral contractions*	X			X

*ab-acd-dbc, abc-acd-db, abc-ad-bdc, ab-cad-dcb, abc-bda-dc, abcd-aebf-dfce, abcd-aebf-fdec.

the initial patterns add a lot of states to the automaton, subsequent patterns mostly reuse states for the already existing patterns, drastically limiting the automaton's size increase per pattern. These findings highlight the main benefits that led us to choose an automaton-based approach in the very beginning: the fact that the input is matched against all patterns simultaneously and the re-usability of existing states. Although the DDG matching complexity also depends on the number of strings present at each pattern, and on the number of input candidates, both of these quantities are relatively small. Despite matching against several patterns, SMR still maintains good performance.

In the last set of experiments, the time to build the PFA automaton using the flow of Figure 3(a) was measured as the number of patterns increases up to 10,000. To generate such patterns, we used an in-house template generation language that creates combinations of GEMM patterns through commutativity, associativity, variations on the array index expressions, and element access order. This language is still under development and outside of this article's scope, so we won't delve into further details. As discussed before, although the core of the SMR algorithm is dominated by the CDG and DDG matching, most of the PFA building time is taken by other tasks not related to SMR (e.g., IO) which are represented as `Others` in the graph. It takes about 5 minutes to build a 10,000-pattern PFA (Figure 11) mostly due to unoptimized IO operations and parsing, the actual DDG and CDG operations take less than 20 seconds.

6.3 Comparisons

To conclude the results section, we present a qualitative comparison (Table 2) of SMR with other similar tools. PDL [45] was disconsidered since it cannot, at the time of writing, match MLIR regions [46].

KernelFaRer [17] extends LLVM pattern matching tools to implement complex idiom recognition directly onto LLVM IR through custom optimization passes, making integration with existing LLVM-based compilers trivial and with little compile-time overhead. However, this also complicates idiom description since LLVM IR is rather low-level affecting usability and control-flow

matching. As a consequence, adding variations and entirely new kernels is a complex task that also required recompilation, which is why pattern evaluation was limited to kernels that contain a `gemm` idiom (Table 2). Despite the small variety of kernels, KernelFaRer properly evaluates generality by matching a single `gemm` pattern against 16 handmade naive variations and 3 partially optimized variations [35], exceeding the generality of other approaches such as IDL [24].

IDL [24] describes the idiom's semantics as a constraint satisfaction problem to increase the generality of each idiom description at the cost of complexity. The IDL language is aimed at compiler experts since it is difficult to develop for and to debug, moreover, it considerably increases compile times by an average of 86% due to its compilation pipeline which uses a solver for the constraint satisfaction problem. Despite IDL's pattern generality claims, this is not empirically demonstrated in this article. Even so, they manage to match a considerable variety of patterns as seen in Table 2.

MLT [11] uses a custom **tensor-based language (TDL)** to describe its patterns. TDL is seemingly simpler than the other mentioned pattern description methods and manages to describe multiple kernels, as seen in Table 2. It should be noted, however, that it matches only MLIR's affine dialect, implying that its usage requires the code to be lowered to said dialect. Their approach demonstrates some generality by using the same `gemm` pattern to match slightly different `gemm` implementations in Polybench's `gemm`, `2mm`/`3mm` kernels. Although it fails to cover Darknet's GEMM occurrences, it mentions that this can be solved by using MLIR transformation passes to canonicalize said occurrences, which is something we also aim to explore with SMR.

The best evaluation we found regarding coverage was done by FACC [49], a tool to replace **fast Fourier transforms (FFTs)** with hardware accelerators' API calls. This was not included in the comparison because it matches *only* FFTs and not any idiom. Regardless of it being a niche tool, the authors manage to capture an astonishing variety of implementations and present a high-level description of how to apply the same methodology with other idioms.

Overall, similarly to SMR, most tools are somehow restricted regarding pattern generality. KernelFaRer [17] manages to capture a wide variety of `gemm`s, only two distinct kernels (`gemm` and `syr2k`). While MLT [11] manages to capture a wide variety of kernels, but only two distinct `gemm`s. Still, a sufficiently generic description of patterns seems to be essential for a viable tool. Although SMR is limited in this area, it has the benefit of being able to tackle this issue on both fronts: by generalizing the DDG model to improve pattern generality, and by simplifying pattern descriptions to ease the addition of new patterns.

7 OTHER APPLICATIONS

As the reader can imagine, the SMR approach has several potential applications that go beyond only replacing code fragments for optimized library calls. This is by design, as we aim for SMR to be as flexible as possible. Some additional applications that we are considering are described below.

7.1 Approximate Computing

Approximate computing is a set of techniques that can be used to redesign programs to decrease the computation time at the expense of the precision of the output result [51]. It can only be applied to algorithms for which there is some tolerance on the precision of the final output. For example, in some applications, a reduction of the image quality is an acceptable price to pay for a much faster execution time.

Loop perforation is an approximate computing technique that allows skipping some iterations of a loop to speed up the computation. There are many ways to implement loop perforation [36]. The adoption of a skip factor is a very common one. In such a case, only sections of the loop trip count will be executed. Consider, for example, the pattern (lines 2-10) of the PAT description of Listing 12, which uses a loop to accumulate a vector (`sum`). Assume now that the programmer wants

```
1  f90 {
2  subroutine sum_double(n, x, sum)
3    integer :: n, i
4    double precision, dimension(n) :: x
5    double precision :: sum
6
7    do i = 1, n
8      sum = sum + x(i)
9    end do
10 end subroutine
11 }={
12 # define INCR 2
13 subroutine sum_double(n, x, sum)
14   integer :: n, i
15   double precision, dimension(n) :: x
16   double precision :: sum
17
18   do i = 1, n, INCR
19     sum = sum + x(i)
20   end do
21   sum = sum * INCR
22 end subroutine
23 }
```

Listing 12. Replacing a loop by an approximated loop perforation pattern.

```
1  f90 {
2  subroutine gemm_single(m, n, k, alpha
       , A, B, beta, C)
3    integer :: m, n, k
4    real, dimension(m, n) :: C
5    real, dimension(m, k) :: A
6    real, dimension(k, n) :: B
7    real :: alpha, beta
8    do nn = 1, n
9      do mm = 1, m
10       c(mm, nn) = c(mm, nn) * beta
11       do i  = 1, k
12         c(mm, nn) = c(mm, nn) + (
       alpha * b(i, nn) * a(mm, i))
13       end do
14     end do
15   end do
16 end subroutine
17 }={
18 subroutine gemm_single(m, n, k, alpha
       , A, B, beta, C)
19   integer :: m, n, k
20   real, dimension(m, n) :: C
21   real, dimension(m, k) :: A
22   real, dimension(k, n) :: B
23   real :: alpha, beta
24   external :: sgemm
25   call sgemm('N', 'N', m, n, k, alpha
       , A, m, B, k, beta, C, m)
26 end subroutine
27 }
```

Listing 13. Replacing GEMM by a BLAS call accelerated with the GEMMINI hardware.

to replace all occurrences of such pattern in a larger program with an approximate version. It could use the PAT description of Listing 12 and replace it for the approximate version in lines 12-23, with skip factor INCR. To evaluate the impact of the approximate code, an experiment was performed on an Intel Core i5-4210U CPU that used a vector dimension n of 16,000,000 and executed the approximate code 100 times. The execution time of the original pattern reduced from 5.55s to 2.84s (≈49%), and the energy consumption, from 33.1 Joules to 17.8 Joules (≈46%).

Modern perforation optimization tools, such as ALONA [34], can leverage polyhedral frameworks to perforate nested loops and rank them using Barvinok's score. We could envision something with SMR where a ranked PAT file could hold a myriad of perforation patterns to be applied to programs without the need for polyhedral tools.

7.2 Hardware Acceleration

Modern hardware accelerators and CPU ISA extensions are gradually becoming a major trend for improving performance. Such accelerators, however, are difficult to use due to a mismatch between the diversity of user code and accelerator APIs. An example of such accelerators is GEMMINI [22], an open-source hardware engine focused on speeding up vector/matrix operations like those found in ML models. GEMM is the most relevant ML operation that GEMMINI can execute. To ease the task of using the GEMMINI hardware, the calls to GEMMINI were hidden behind a BLAS-like API to compose a library named GEMMINI BLAS.

Like FACC [49], which replaces fast Fourier transforms with accelerated hardware API calls, SMR seeks to bridge this gap by aiming for a more generic form of rewriting, albeit not as robust, that may be able to replace multiple kernels by custom hardware accelerators. SMR can be used as an approach to search programs for variations of Matrix Multiplication patterns which can then be replaced by GEMMINI BLAS calls. Consider, for example, the GEMM f90 pattern from lines 2–16 of the PAT description in Listing 13. SMR allows programmers to find patterns like this (or similar) and replace them for an accelerated call to GEMMINI BLAS. A small experiment was performed to validate that. The experiment uses a RISCV 4-wide superscalar BOOM processor [53] combined with the GEMMINI accelerator in a single SoC operating at 1GHz. The RTL Verilator simulator was used to extract performance data. A SiFive BOOM model was configured with a 32 KB 8-way L1 Cache, and 512 KB 8-way L2 Cache with 64-byte blocks. For the GEMMINI accelerator, an 8×8 systolic array was instantiated, with each PE configured to support 32-bit floating-point instructions, combined with a 256 KB Scratchpad and a 64 KB Accumulator.

The experiment used as input a 128×128 matrix. It aimed at matching the GEMM code pattern from lines 2–16 of Listing 13, and replacing it with two optimized library calls: (a) RISCV BOOM BLAS library which maximizes the utilization of the cache hierarchy using Goto's algorithm [25]; and (b) the GEMMINI BLAS hardware-accelerated library. Execution cycles for the GEMM pattern running on RISCV, and for the two optimized library call replacements were measured using Verilator. The resulting number of cycles are: 26,210,880 cycles (GEMM pattern using RISCV CPU), 5,049,802 cycles (BOOM BLAS replacement), and 766,655 cycles (GEMMINI BLAS hardware-accelerated replacement). This corresponds to GEMMINI BLAS speedups of 6.5× when compared to BOOM BLAS and 34.7× when compared to the RISCV CPU code pattern.

8 CONCLUSIONS

This article proposes SMR, a source code and MLIR-based idiom matching and re-writing mechanism. By using MLIR generality, SMR could match both Fortran and C programs through their corresponding dialects (FIR and CIL). In FIR's case, it was also able to optimize programs by replacing patterns for optimized library calls. By using CIL, we were able to demonstrate the scalability of SMR as the size of input source code and the number of patterns increase, as well as its ability to integrate multiple MLIR frontends. Future extensions will address the SMR constraints listed above, optimize the IO overhead, enable inter-language replacement, match novel accelerated ISA extensions, and propose new applications.

REFERENCES

[1] Mark F. Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Samuel Williams. 2014. HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems. University of California, Lawrence Berkeley National Laboratory, 11 pages. https://escholarship.org/uc/item/00r9w79m.

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. 1989. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 4 (1989), 491–516. DOI: https://doi.org/10.1145/69558.75700

[3] A. V. Aho and S. C. Johnson. 1976. Optimal code generation for expression trees. *J. ACM* 23, 3 (July 1976), 488–501. DOI: https://doi.org/10.1145/321958.321970

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison Wesley, Boston, MA.

[5] Manuel Arenaz, Juan Touriño, and Ramon Doallo. 2008. XARK: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 32 (Oct. 2008), 56 pages. DOI: https://doi.org/10.1145/1391956.1391959

[6] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. 2020. Automatic generation of efficient linear algebra programs. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Geneva, Switzerland) (PASC'20). ACM, New York, Article 1, 11 pages. DOI: https://doi.org/10.1145/3394277.3401836

[7] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David A. Padua, Paul Petersen, William M. Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. 1994. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC'94)*. Springer-Verlag, Berlin, 141–154.

[8] Uday Bondhugula. 2020. High performance code generation in MLIR: An early case study with GEMM. *CoRR* abs/2003.00532 (2020), 23. https://arxiv.org/abs/2003.00532

[9] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive raising in multi-level IR. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) (CGO'21). IEEE Press, 15–26. DOI: https://doi.org/10.1109/CGO51591.2021.9370332

[10] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event (PACT'20). ACM New York, 83–96. DOI: https://doi.org/10.1145/3410463.3414635

[11] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. 2019. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (Dec. 2019), 25 pages. DOI: https://doi.org/10.1145/3372266

[12] Flang Compiler. 2019. F18 LLVM Project (fir-dev branch). https://github.com/flang-compiler/f18-llvm-project/tree/fir-dev.

[13] MLIR contributors. 2020. MLIR Documentation. https://mlir.llvm.org/docs/.

[14] Keith Cooper and Linda Torczon. 2012. *Engineering a Compiler*. Morgan-Kaufmann, Boston, MA. DOI: https://doi.org/10.1016/c2009-0-27982-7

[15] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F. P. O'Boyle, and Hugh Leather. 2021. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *Proceedings of the 38th International Conference on Machine Learning* (Proceedings of Machine Learning Research, Vol. 139), Marina Meila and Tong Zhang (Eds.). *PMLR, A Virtual Conference*, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html.

[16] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event) (CGO'21). IEEE Press, 378–390. DOI: https://doi.org/10.1109/CGO51591.2021.9370322

[17] João P. L. De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing native-code idioms with high-performance library calls. *ACM Trans. Archit. Code Optim.* 18, 3, Article 38 (June 2021), 22 pages. DOI: https://doi.org/10.1145/3459010

[18] FFmpeg Developers. 2021. FFmpeg tool. https://ffmpeg.org/.

[19] Paul Fischer, Stefan Kerkemeier, Misun Min, Yu-Hsiang Lan, Malachi Phillips, Thilina Rathnayake, Elia Merzari, Ananias Tomboulides, Ali Karakus, Noel Chalmers, and Tim Warburton. 2022. NekRS, a GPU-accelerated spectral element navier-stokes solver. *Parallel Computing* 114 (2022), 102982. https://www.sciencedirect.com/science/article/pii/S0167819122000710.

[20] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. 1992. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 3 (1992), 213–226. DOI: https://doi.org/10.1145/151640.151642

[21] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (Sept. 2018), 27 pages. DOI: https://doi.org/10.1145/3235029

[22] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA). IEEE Press, 769–774. DOI: https://doi.org/10.1109/DAC18074.2021.9586216

[23] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. 2020. Automatically harnessing sparse acceleration. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA) (CC 2020). ACM, New York, 179–190. DOI: https://doi.org/10.1145/3377555.3377893

[24] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA) (ASPLOS'18). ACM, New York, 139–153. DOI: https://doi.org/10.1145/3173162.3173182

[25] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3 (May 2008), 1–25. DOI: https://doi.org/10.1145/1356052.1356053

[26] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. POLLY - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 4 (Dec. 2012), 1250010. DOI: https://doi.org/10.1142/S0129626412500107

[27] Christoph M. Hoffmann and Michael J. O'Donnell. 1982. Pattern matching in trees. *Journal of the ACM (JACM)* 29, 1 (1982), 68–95. DOI: https://doi.org/10.1145/322290.322295

[28] Daniel Holden. 2015. Cello: Higher level programming in C. https://libcello.org/.

[29] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA) (PLDI'16). ACM, New York, 711–726. DOI: https://doi.org/10.1145/2908080.2908117

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) (CGO'04). IEEE Computer Society, 75.

[31] 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 *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event) (CGO'21). IEEE Press, 2–14. DOI: https://doi.org/10.1109/CGO51591.2021.9370308

[32] Sang Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. 2004. Cetus - An extensible compiler infrastructure for source-to-source transformation. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2958 (2004), 539–553. DOI: https://doi.org/10.1007/978-3-540-24644-2_35

[33] Martin Lücke, Michel Steuwer, and Aaron Smith. 2021. Integrating a functional pattern-based IR into MLIR. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual) (CC 2021). ACM New York, 12–22. DOI: https://doi.org/10.1145/3446804.3446844

[34] Daniel Maier, Biagio Cosenza, and Ben Juurlink. 2021. ALONA: Automatic loop nest approximation with reconstruction & space pruning. In *Euro-Par 2021: Parallel Processing: Proceedings of the 27th International Conference on Parallel and Distributed Computing (Lisbon, Portugal, September 1–3)*. Springer-Verlag, Berlin, 3–18. DOI: https://doi.org/10.1007/978-3-030-85665-6_1

[35] Vijay Menon and Keshav Pingali. 1999. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) (ICS'99). ACM, New York, 434–443. DOI: https://doi.org/10.1145/305138.305230

[36] Sasa Misailovic, Daniel M. Roy, and Martin Rinard. 2011. Probabilistic and statistical analysis of perforated patterns, Massachusetts Institute of Technology, Computer Science & Artificial Intelligence Laboratory. http://hdl.handle.net/1721.1/60675.

[37] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *Proceedings of the 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, Los Alamitos, CAA, 45–59. DOI: https://doi.org/10.1109/PACT52795.2021.00011

[38] Mohanish Narayan and Louis-Noel Pouchet. 2012. PolyBench/Fortran 1.0. https://www.cs.colostate.edu/pouchet/software/polybench/polybench-fortran.html.

[39] E. Pelegrí-Llopart and S. L. Graham. 1988. Optimal code generation for expression trees: An application BURS theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California), (POPL'88). ACM, New York, 294–308. DOI: https://doi.org/10.1145/73560.73586

[40] Bill Pottenger and Rudolf Eigenmann. 1995. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing* (Barcelona, Spain) (ICS'95). Association for Computing Machinery, New York, NY, USA, 444–448. DOI: https://doi.org/10.1145/224538.224655

[41] Joseph Redmon. 2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[42] CIL repository. 2021. Commit. https://github.com/compiler-tree-technologies/cil/commit/195acc33e14715da9f5a1746b489814d56c015f7.

[43] FIR repository. 2021. Commit. https://github.com/flang-compiler/f18-llvm-project/commit/8abd290c2c791c26cd1237b218def1b85998d403.

[44] LLVM/MLIR repository. 2021. Commit. https://github.com/llvm/llvm-project/commit/1fdec59bffc11ae37eb51a1b9869f0696bfd5312.

[45] River Riddle. 2021. Pattern Descriptor Language. https://drive.google.com/file/d/17WYUvlmCzNTiqLaxWf_uz4GiLm3QVoEV/view.
[46] River Riddle and Jeff Niu. 2021. Public communication at LLVM/MLIR Discourse. https://llvm.discourse.group/t/how-can-i-use-pdl-to-match-affine-code/4837.
[47] Offensive Security. 2021. The official Exploit Database repository. https://github.com/offensive-security/exploitdb.
[48] Compiler Tree Technologies. 2021. CIL. https://github.com/compiler-tree-technologies/cil.
[49] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. 2022. Bind the gap: Compiling real software to hardware FFT accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA) (PLDI 2022). ACM, New York, 687–702. DOI: https://doi.org/10.1145/3519939.3523439
[50] Zhang Xianyi and Martin Kroeker. 2020. OpenBLAS: An optimized BLAS library. https://www.openblas.net/.
[51] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate computing: A survey. *IEEE Design Test* 33, 1 (2016), 8–22. DOI: https://doi.org/10.1109/MDAT.2015.2505723
[52] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy G. Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. 2020. MISIM: An end-to-end neural code similarity system. *CoRR* abs/2006.05265 (2020), 23. arXiv:2006.05265 https://arxiv.org/abs/2006.05265.
[53] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. http://people.eecs.berkeley.edu/krste/papers/SonicBOOM-CARRV2020.pdf.

Received 26 May 2022; revised 23 September 2022; accepted 27 October 2022