Fast Template-Based Code Generation for MLIR

Florian Drescher
Technical University of Munich
Munich, Germany
florian.drescher@tum.de

Alexis Engelke
Technical University of Munich
Munich, Germany
engelke@in.tum.de

Abstract

Fast compilation is essential for JIT-compilation use cases like dynamic languages or databases as well as development productivity when compiling static languages. Template-based compilation allows fast compilation times, but in existing approaches, templates are generally handwritten, limiting flexibility and causing substantial engineering effort.

In this paper, we introduce an approach based on MLIR that derives code templates for the instructions of any dialect automatically ahead-of-time. Template generation re-uses the existing compilation path present in the MLIR lowering of the instructions and thereby inherently supports code generation from different abstraction levels in a single step.

Our results on compiling database queries and standard C programs show a compile-time improvement of 10–30x compared to LLVM `-O0` with only moderate run-time slowdowns of 1–3x, resulting in an overall improvement of 2x in a JIT-compilation-based database setting.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Translator writing systems and compiler generators**.

Keywords: MLIR, JIT Compilation, Template-based Compilation, Fast Compilation, Binary Code Patching

ACM Reference Format:
Florian Drescher and Alexis Engelke. 2024. Fast Template-Based Code Generation for MLIR. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640537.3641567

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641567

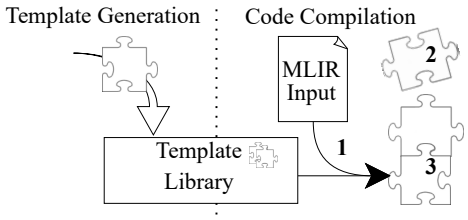

Figure 1. Templates are prepared from MLIR operations ahead-of-time; at compile-time, actual instructions are matched to templates (1), which are instantiated by filling in missing values (2), finally missing addresses are fixed up (3).

1 Introduction

Just-in-time compilation is commonly employed to improve the performance of programs, either by speeding up subsequent computations, as in compiling database query engines [14, 24, 28, 31], or by generating an executable for previously unseen code, such as client-side code execution of JavaScript or WebAssembly [5, 19, 34]. In either case, the time it takes to compile the input and generate an executable is counted towards the execution time of the program, directly affecting the user experience.

As a result, a key challenge for constructing JIT compilers is reducing the compilation time without ultimately sacrificing code quality. Template-based code generation, where precompiled code fragments are merely combined during compilation, allows for extremely low compile-times. This was previously demonstrated with VCode [16], focusing on quick encoding of machine code through a reduced set of operations with manually written templates for the machine code, and the Java virtual machine Maxine [41], whose baseline compiler combines templates for every bytecode instruction, which were written in Java and precompiled by the optimizing compiler. A more recent approach in this field [42] uses Clang/LLVM [25] to compile templates written in C++ and demonstrates the applicability for larger templates, which leads to shorter compile-times and better run-time performance due to more optimization during precompilation. However, all approaches so far require handwritten templates, which not only comes with substantial implementation effort, but the templates need to be maintained in addition to existing lowerings for optimized compilation.

We, therefore, propose a template-based compilation approach leveraging MLIR [26], where instructions usually provide a — possibly multi-step — lowering to LLVM-IR for native code generation. This approach allows for automatically

deriving templates using the existing lowerings, obviating the need for manual template development or maintenance. Moreover, as single MLIR instructions are not limited to being simple operations, this approach easily allows for templates with complex logic, enabling further optimizations during template pre-compilation and faster compile-times due to fewer templates. Fig. 1 visualizes this approach. Our results show that we can achieve a compilation time speedup of 10–30x over LLVM `-O0` with only moderate runtime slowdowns of 1–3x, which results in an overall improvement of up to 2x in a JIT-compilation-based database setting.

The main contributions of this paper are as follows:

- A framework for extracting the semantics of an arbitrary MLIR instruction from its defined lowering into a reusable template.
- A template-based compiler that generates native code from an MLIR program targeting x86-64 and AArch64.
- Optimizations for improved register usage and handling of constant values in a template-based compiler.

2 Background: MLIR

MLIR [26] is a compiler framework that aims to simplify the creation, transformation, and optimization of intermediate representations in SSA representation. It allows for the implementation of custom sets of instructions, called dialects, where each SSA instruction is made up of input and output values, can contain regions of other instruction sequences in their body (e.g., the body of a natural loop), and has constant attributes to further configure individual instructions (e.g., the value of a constant instruction). These attributes are further classified as inherent (also referred to as properties) or discardable, where discardable attributes may be omitted at any time and, therefore, do not contain essential information for the execution of an instruction.

The MLIR framework provides a consistent way for general optimizations (e.g., constant propagation and common sub-expression elimination) and conversions from a higher-level dialect to a lower-level one. For ease of use, MLIR comes with a set of upstream dialects, including the `scf` dialect for handling structured control flow, the `arith` and `math` dialect for mathematical expressions, and the `memref` dialect for interacting with memory. A common lowering target, optionally with different intermediate dialects, is LLVM-IR. For this purpose, MLIR provides a large part of LLVM-IR as MLIR dialect serving as the target for conversions, which can then be translated to the actual LLVM-IR easily.

Recent publications started applying the capabilities of MLIR to various topics going beyond primary machine learning use cases [10, 20, 27, 40] to discover new optimization opportunities when compiling static languages like Fortran, C and C++ [4, 29, 30] and different abstractions for data processing pipelines [13, 21].

3 Template Generation

Based on the input of sample programs, templates are derived and stored for all contained instructions. Further executions for programs from the same domain can now be executed based on the previously prepared templates.

3.1 Instruction Prerequisites

Our automatic template generation approach is generally applicable to all dialects independent of their abstraction level — we successfully applied it to low-level dialects, like LLVM IR, as well as very high-level ones, like the ONNX dialect [20]. The only prerequisite is that the lowering of each instruction does not depend on any information coming from outside the instruction itself. Otherwise, it is impossible to process each instruction in isolation and thus not feasible to automatically capture its semantics into a standalone template.

Most common instructions adhere to this rule. Nonetheless, there are some exceptions, e.g., the LLVM branching instructions, as they depend on external block labels, and the `alloca` instruction, as its lowering depends on the surrounding scope. Another set of conflicting instructions is the upstream OpenMP dialect, where many instructions are very tightly coupled to their surrounding instruction (e.g., instructions inside/outside a critical section).

3.2 Capturing Instruction Semantics

As we want to avoid handcrafting templates, our approach derives the native code templates from MLIR instructions without any manual implementation effort. The main challenge is that MLIR instructions themselves are opaque: their full semantics are only defined in the lowering. To capture the semantics of an instruction into a native code template, we isolate each instruction into a function, provide opaque inputs to the instruction and capture the output using `unrealized_conversion_casts`, which are typically not folded by any further conversions or transformations. Listing 1 shows an example.

```
1 func.func @add() -> ptr {
2     // inputs
3     %0, %1 = unrealized_conversion_cast to (i64, i64)
4     %2 = arith.addi %0, %1 : i64
5     // output(s) - return to keep it alive
6     %3 = unrealized_conversion_cast %2 : i64 to ptr
7     return %3 : ptr
8 }
```

Listing 1. Automatically derived abstraction of the `arith.addi` instruction in an internal intermediate state. Inputs and outputs are made opaque for the instruction using `unrealized_conversion_casts`. The symbolic value resulting from the cast applied on the outputs is returned from the function to keep the values and thus computations alive.

For instructions with regions, we furthermore place external function calls into every region to encapsulate the behavior. To keep track of region arguments, these are written to memory before the call, and the operands of the terminator instruction are loaded from memory.

The opaqueness of the instructions poses another challenge when matching an incoming instruction to a fitting binary template. The lowering of an instruction can be different depending on the operand or result types. For example, the behavior of most upstream instructions from the `math` or `arith` dialect depends on the type, which is not necessarily a scalar value but could be a tensor of values. Therefore, a unique template must be generated for each input and output type combination of an instruction, as the lowering may be different for each. The same applies to inherent attributes, which may affect the lowering but cannot be reflected by any of the instruction inputs.

For operations that do not adhere to the defined prerequisites, we provide two extension points to support them manually: (a) a custom abstraction can be provided during template generation time; and (b) a custom implementation can be inserted for the run-time compilation of the template.

3.3 Lowering & Compilation

Next, we apply the dialect-specific conversions on the derived abstraction to lower it to LLVM-IR. In the example, this converts the `arith.addi` into an `llvm.add` and converts the MLIR native pointer types to their LLVM pendants. Afterward, we need to provide implementations for the opaque inputs and outputs so that we can actually compile the code. A simple but nonetheless effective way is: inputs as memory loads and outputs as memory stores. The values are stored in a value storage, which will later be allocated on the stack and is passed as an argument to the template function. This method allows for efficient addressing with 32-bit offsets instead of arbitrary 64-bit memory addresses and also reuses the natural stack growth and shrinking. As used in [42], the actual locations and offsets are computed during run-time compilation and patched into the templates using addresses of symbols, which result in relocations and, therefore, can be patched during run-time compilation (cf. Sec. 4.2). As a refinement, we make these symbols *weak* to prevent LLVM from making any assumption about the value being non-zero and use the *absolute_symbol* attribute to restrict relocations to absolute 32-bit ones, which leads to more efficient code.

To allow the composition of the templates and to enable control flow between instructions, we leverage the continuation passing style (CPS) [38] concept: We enforce a tail call to the continuation function at the end of each template using the `musttail` annotation to transfer control flow to the next template. The continuation is an external symbol whose actual address is patched during run-time compilation. A resulting `jmp` instruction at the end of the template can easily be detected and omitted when concatenating templates.

However, this technique is not applicable for region calls, as those are regular non-tail calls, after which the execution continues inside the template. Instead, these result in regular calls with the value storage pointer passed as an argument. Operands for the region terminator are loaded from the value storage after the call. Listing 2 shows an example of the final LLVM-IR code, which is then compiled as the template.

```
; external symbol addresses as patchable constants
@off_0 = extern_weak global i8, align 1,
  !absolute_symbol !{i64 0, i64 INT32_MAX}
@off_1 = extern_weak global i8, align 1,
  !absolute_symbol !{i64 0, i64 INT32_MAX}
@off_2 = extern_weak global i8, align 1,
  !absolute_symbol !{i64 0, i64 INT32_MAX}
declare void @next(ptr %value_storage)
define void @add(ptr %value_storage) {
  ; load operands from memory at patched offsets
  %1 = getelementptr i8, ptr %value_storage,
    i64 ptrtoint (ptr @off_0 to i64)
  %2 = load i64, ptr %1, align 4
  %3 = getelementptr i8, ptr %value_storage,
    i64 ptrtoint (ptr @off_1 to i64)
  %4 = load i64, ptr %3, align 4
  %5 = add i64 %2, %4 ; the operation itself
  ; store result to memory at patched offset
  %6 = getelementptr i8, ptr %value_storage,
    i64 ptrtoint (ptr @off_2 to i64)
  store i64 %5, ptr %6, align 4
  ; call to continuation
  musttail call void @next(ptr %value_storage)
  ret void
}
```

Listing 2. Automatically derived abstraction of the `arith.addi` instruction in LLVM-IR. Operands are memory loads and the result is written back to memory. Offsets into the value storage are represented as addresses of external symbols and patched later during run-time compilation. Control flow is transferred by an enforced tail call.

3.4 Binary Format

To finally derive a binary code template, the abstracted function is compiled to a binary object — currently limited to the ELF format — using the LLVM optimization and code generation infrastructure at its highest optimization level.

```
add: ; rdi = pointer to value storage
  movq $off_0(%rdi), %rax
  addq $off_1(%rdi), %rax
  movq %rax, $off_2(%rdi)
  jmp $next
```

Listing 3. Compiled template of `arith.addi` from Listing 2. Offsets into the value storage and continuation address of the template result in relocations (highlighted).

We extract the templates by parsing the ELF file, using the text sections as binary code for the template and storing the data sections (e.g., `.data` and `.rodata`) to forward them

to the runtime system. Furthermore, we track the relocation entries and identify patchpoint symbols for value storage offsets and continuation calls by their name. We also track other relocations not originating from the framework, as we have to take over some tasks of a run-time linker as well (e.g., patching addresses to other symbols or data sections).

A simple template binary for the `arith.addi` operation is shown in Listing 3, which only consists of the binary code and some patchpoints, including the continuation address.

3.5 Case Study: Templates for the LLVM Dialect

As one example, we look at the application of our approach to the LLVM dialect, which is required to run the benchmarks used during evaluation. For most instructions, we can generate templates automatically without any manual interaction. However, to fully cover the LLVM IR instructions required for the benchmarks, we had to provide some custom implementations:

- Branching instructions (`Br`, `Condbr` and `Switch`) are hand-assembled and run-time compilation additionally deals with SSA destruction.
- `AddressOf` requires a custom abstraction, as memory locations are only determined at run-time compilation.
- Regions and value attributes of globals are evaluated during run-time compilation and placed into memory, where they can be referenced with `AddressOf`.
- Stack allocations (`Alloca`) retrieve memory from the value storage (fixed size) or the heap (var sized).

Nonetheless, this is a comparatively small amount of effort to spend on such an extensive instruction set.

4 Run-time Compilation of Templates

The run-time compilation phase stitches together the precompiled templates to produce code for a previously unseen input program. This corresponds to the compilation part of a JIT compiler and is therefore further referred to as compilation. In contrast to the previous stage, it is time-critical as the compilation contributes to the overall execution time.

4.1 Selection

As a first step, the input program must be covered with existing templates. To facilitate that, we walk the input program starting at the top-level region in a depth-first manner and find a matching template from our template library for each instruction individually. As described in Sec. 3.2, an instruction can occur with different configurations (e.g., input types or property values) and therefore, for a template to match, it must match the full *signature* consisting of the operation name, input and output types, the number of regions and the property values. To make looking up signatures as efficient as possible, we store them in a hash map. The used hash is constructed over the operation type and the properties, as the combination of those provides most of the entropy of a signature. Furthermore, an MLIR context ensures that each registered operation type is unique, allowing us to compare the operation type identifier, which is just a pointer, instead of the string representations.

4.2 Instantiation

Once the matching template is found, it is instantiated. The corresponding binary code is copied to the designated memory location and the identified patchpoints — mainly offsets into the value storage and continuation addresses — are adjusted to their according values. During copying, we can omit unnecessary jumps between two neighboring templates.

All values defined by the current instruction (results and region arguments) are assigned to a slot in the value storage. To keep compile-times low, we allocate the slots during the same pass that generates the native code. In order to reduce the memory usage of the value storage, we track the liveness of the slots and reuse them once they become free. For performance, however, we do not perform a dedicated liveness analysis but generously over-approximate the lifetime intervals: The end is defined as the end of the region unless the instruction has only a single use in the same basic block, where the lifetime ends at that instruction.

4.3 Fixup and Wrapper Function

As we generate code in a single pass over the input, some addresses or symbols are not known upon their first reference. This mainly happens due to forward references to yet undefined functions, global symbols, or basic block labels. Those locations are tracked and updated as soon as the referenced data becomes available. For all address references — also during instantiation — we take advantage of compile-time information, which can lead to further linking optimizations, e.g., we replace loads from the GOT by directly computing the desired address if in range, saving the space of the GOT entry and the load from memory at run-time.

Once code generation finishes, control flow has to be transitioned into the newly generated assembly. Transitioning from our host C++ program to the generated assembly is possible by looking up the address of the generated main function and calling it with the default C calling convention. The function template, which was generated to embody the function declaration, takes care of allocating the initial value storage, saving registers (and restoring them upon return), preparing the value storage pointer argument and ultimately invoking its body. From this point on, our framework does not provide any runtime components during execution.

5 Optimizations

5.1 Constant Evaluation

In contrast to LLVM, where constants can be used as values arbitrarily, MLIR conceptually does not distinguish constants and models them as constant instructions, for example,

`arith.constant` for constant numbers. Because the actual values for those constants are stored as attributes, each of those instructions would be recognized as its own template. To avoid a huge number of different constant templates only differing by their value, operations that have no side effects, no regions, no input operands, and no references to any dynamic addresses, can be executed during template generation. During run-time compilation, the results can then be injected on demand using a custom template. This optimization considerably reduces the number of generated templates as all constant-evaluated instructions share a single dedicated template.

5.2 Template Calling Convention

So far, every value resides in the value storage in memory and is passed into a template by patching its offset into the instantiated binary. For constants, this causes a store operation of the value to memory, which the template loads again immediately afterward. To improve this, we adjust the handling of inputs and outputs of our templates.

Reasonably small input values (up to two registers wide) are passed directly in registers, while larger values remain in memory and are addressed via patched offsets. We achieve this by passing such values as parameters, which are passed in registers by the underlying calling convention, to the function template. Similarly, small output operands are passed as arguments to the continuation function. For larger data types, the handling remains unchanged, as only a small subset of its values is typically used. These are, therefore, better suited for in-memory passing, as the template can specifically access the required elements, avoiding large loads and stores. Listing 4 gives the binary code for the `arith.addi` example with this optimization.

```
1 add: ; params: %rdi = value store, %rsi = a, %rdx = b
2   addq %rdx, %rsi
3   jmp $next ; params: %rdi = value store, %rsi = res
```

Listing 4. Compiled template of `arith.addi` using the optimized calling convention: Input and output values in registers; Continuation address to be patched (highlighted).

During run-time compilation, we additionally emit code for loading the values from memory where required and materializing constants directly into registers. Results are written back to memory after each template. Due to the explicit separation of operand loading and computation, memory operands can no longer be fused into arithmetic operations (as happened in Listing 3; loading the second operand is fused into the `addq` instruction), but this had no measurable performance impact, as most modern x86-64 CPUs split load-arith instruction into multiple micro-ops anyway. As a side effect, this also reduces the template size significantly, as loading/storing values is no longer part of every template.

5.3 Register Caching

Even with the previous optimization, the resulting code excessively loads values from memory into registers and stores results back into memory. To reduce the number of memory loads, we cache the result values in registers — in addition to writing them back to the value storage — thus, in many cases, replacing the load from memory with a copy from a register. We can even eliminate the store to memory if the value has its single use in the immediately following instruction.

While it is possible to rely solely on callee-saved registers, which are guaranteed to be unmodified by the template, many templates use only very few registers, so other caller-saved registers can serve as additional cache space. During template generation, we, therefore, analyze which registers are clobbered. We obtained this information during template compilation from the LLVM by analyzing the instructions of the final Machine IR for the function.

During code generation, we additionally track the cached values in registers and generate code to move result values into and out of such registers. While the additional moves increase the number of instructions, this pays off during execution as we save on loads from memory.

We evaluated two different strategies to assign the cache registers. When caching a register and none is available, we either (1) do not cache the value at all or (2) override one of the cached values in a round-robin manner. The round-robin approach was chosen to account for SSA values usually used rather locally and losing importance once the code progresses. But there is no significant difference between them — both save up to 30% of the memory loads. Therefore, our evaluation uses strategy (1) due to its lower compilation time. Cache registers become free if a value reaches the end of its lifetime (cf. Sec. 4.2) or a template clobbers them — in which case we fall back to loading from the value storage.

5.4 Higher-Level Optimizations

Further common optimizations for interpreters and template-base compilers include supernode generation [11, 37] and template specialization on certain inputs [22]. However, with a flexible framework like MLIR, we believe that there is no need for such techniques. Instead, one can leverage the multi-level approach of MLIR and apply the code generation on a higher-level, domain-specific dialect. This implicitly creates supernodes, as higher-level instructions are typically more complex and often lowered to several lower-level instructions. A simple example is the `scf` dialect for structured control flow, which already provides explicit operations for common constructs like `while` and `for` loops instead of using plain branch instructions found in the lower-level dialects `cf` and `llvm`. Due to the reduced number of instructions and a simpler control flow, the compilation time is also reduced by using higher dialects as a starting point for code generation.

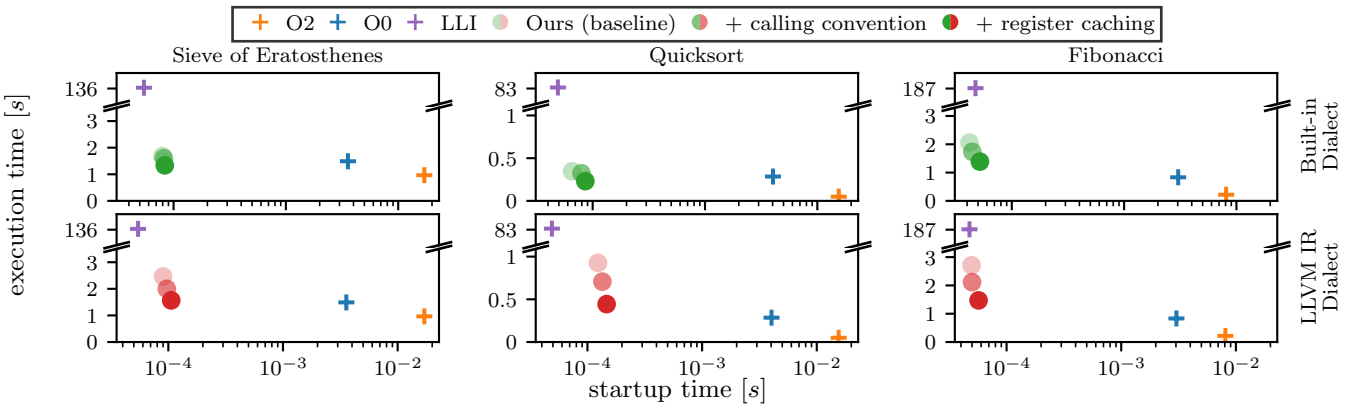

Figure 2. Comparison of our approach and optimizations with LLVM `-O0` and `-O2` and the LLVM Interpreter on three micro-benchmarks on x86-64. The first row shows results when compiling from upstream dialect operations (`scf`, `arith` and `memref`); the second row starts from the `llvm` dialect.

6 Target Architecture Considerations

Although the examples so far targeted the x86-64 architectures, our approach does not require a specific architecture. The template generation solely relies on LLVM and thus is capable of generating templates for various architectures. Even internal code generation (e.g., storing register to memory or vice versa) reuses the template compilation approach. Thus, porting the approach to a new architecture requires only moderate effort (e.g., architecture-specific relocations).

In addition to x86-64, we currently also support AArch64. A key difference is the fixed-size and, therefore, less flexible instruction set. In particular, constants are often composed through multiple instructions and applying relocations often involves bit-level adjustments to code — in contrast to x86-64, where relocations are generally byte-aligned and continuous. In turn, the fixed length instruction set allows for more straightforward modification of the binary code, thus simplifying optimization of instructions, like replacing GOT entry loads with direct address computation.

7 Evaluation

We evaluate our approach on a range of micro-benchmarks and benchmark suites. We assume all required templates are generated and prepared for usage for all benchmarks. This assumption is generally feasible, as the number of potential instructions is inherently limited — as shown by [42].

We compare our approach against different LLVM back-ends. As LLVM back-ends do not operate on MLIR directly but instead use LLVM-IR, we first lower the MLIR input to LLVM-IR. This step is not included in the measurements, as it is already higher than the compilation time with our approach altogether. We then use LLVM ORC JIT, typically with the small code model; only SPEC requires the medium code model. For `-O0`, we used FastISel as the instruction selector; for optimized compilation, we use `-O2`, as there are no significant differences to the other back-end optimization levels. Where possible, we compared compiling with our approach from the higher-level upstream dialects and the lower-level LLVM dialect. The MLIR upstream dialect input was derived using the C frontend of Polygeist [29] with optimizations turned off. For the LLVM dialect, we first apply Clang with `-O0` to derive LLVM-IR and, afterward, use the MLIRTranslate tool for importing to MLIR.

Our x86-64 benchmark platform is an Intel Xeon Platinum 8260 CPU equipped with 160 GiB RAM; our AArch64 platform is an Apple M1 core equipped with 16 GiB RAM; all machines are running Linux and an LLVM development snapshot (commit `5d492766a8`). Besides the expensive SPEC benchmarks, all diagrams report the median of ten runs.

7.1 Impact of Optimizations

To analyze the impact of the optimizations described in Sec. 5, we compare our approach with LLVM back-end optimization levels `-O0` and `-O2` as well as the LLVM interpreter on a set of micro-benchmarks. The benchmarks were designed to stress the impact of our optimization on compute-intensive tasks. The Eratosthenes sieve runs in a single function and benchmarks control flow, memory and arithmetic operations. The quicksort benchmark extends on this idea, slightly shifting focus from control flow inside a single function to recursive calls and memory operations. Fibonacci is finally used to show the negligible startup overhead for minimal programs while also indicating the limitations of our approach in programs exclusively bound by function calls. Figure 2 shows the results. The standard deviation of the result remains below 10% for our approach and below 5% for the LLVM levels in the compilation time dimension and below 3% for all approaches regarding execution time.

Compared to compiling with LLVM, the compilation times of our approach are an order of magnitude faster compared to `-O0`, in the range of 32–72x. Run-time performance on the

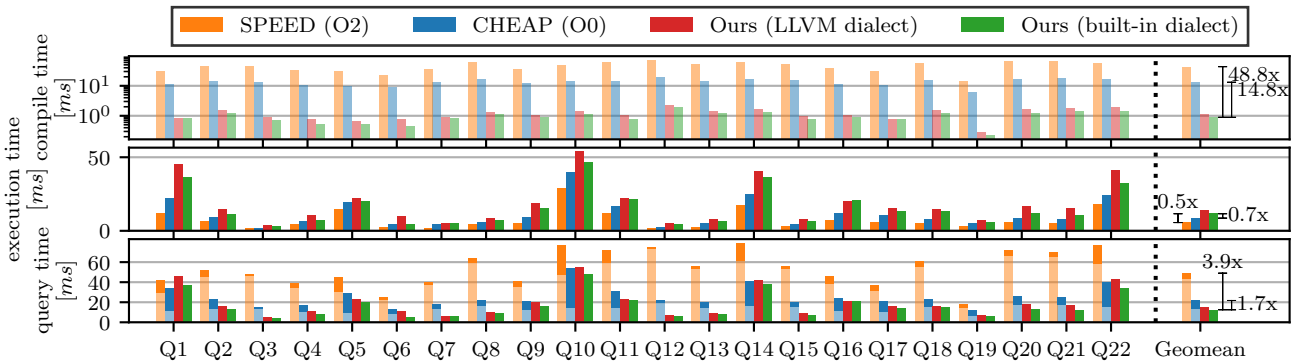

Figure 3. Performance of compiling and executing the TPC-H queries (sf=0.1) on x86-64 comparing LLVM `-O2`/`-O0` with our approach on the LLVM dialect and the `scf`/`arith`/`util` dialects. Query time is the sum of compilation and execution time.

micro-benchmarks is generally comparable and between 2x slower to 20% faster than LLVM `-O0`. The LLVM interpreter is around 100x slower than all other approaches. Start-up times are generally lower, but even these are outperformed by our approach once. The significant run-time overhead in the Fibonacci benchmark is due to the impact of recursive calls, which prevents register caching as calls conceptually clobber all registers. Our optimization of adjusting templates to primarily use registers and materializing constants reduces execution time by 15%. However, the cost of separately inserting loads/stores increases compilation times slightly by 8%. Extending it with our register caching strategies significantly reduces execution time by another 26% (37% over baseline). Nonetheless, generating the additional instructions and tracking registers has a compile-time cost of 10% (19% over baseline).

Within our approach, starting from a higher abstraction level (first row) is clearly favorable as such programs generally contain fewer and more complex operations and, therefore, cause less work during compilation and allow for better code inside the templates.

7.2 LingoDB — an MLIR-based Database Engine

Compiling database engines is an important application area of JIT compilation, as the compilation time fully counts toward the overall processing time of a query. LingoDB [21] is an MLIR-based query execution engine that lowers SQL queries through a declarative top-level dialect, on which it also performs query plan optimization, towards the LLVM dialect to compile queries to native code.

We replaced the last lowering stages to generate native code with our approach directly. This technique is applied on two levels: the lowest LLVM-IR level and one above, consisting of upstream dialects and a LingoDB-specific utility dialect. We use TPC-H [39] (scaling factor 0.1) as a typical small-sized workload. As our code generation approach is intended as an unoptimized tier, an adaptive execution pipeline could switch to optimizing back-ends for larger data sets. Fig. 3 shows the results comparing the existing LingoDB modes *speed* (`-O2`) and *cheap* (`-O0`) with our approach. The standard deviation for the LLVM execution stages is about 5% for execution and compilation, while our approach deviates by 20% on the compilation time dimension due to the significantly lower absolute values, and 10% (upstream dialects) to 20% (LLVM IR dialect) on execution time.

When compiling from the higher-level dialects, our approach generates code an order of magnitude faster than LLVM `-O0`, taking about one millisecond per query. As trade-off, the mean execution time increases by 40% (3% (Q14) to 76% (Q17)) compared to LLVM `-O0`. Nonetheless, when accounting for both stages, the time to compile and execute a TPC-H query is reduced by 43% with our approach compared to LingoDB *cheap* and by around 4x compared to LingoDB *speed*. Considering that query execution is usually multi-threaded, speedup further increases, as parallel execution can only start after single-threaded compilation finishes.

When comparing with LLVM `-O2`, the execution time is 108% higher (35% (Q15) to 200% (Q17)). However, the time spent generating optimized code does not amortize at such small data sets. Similar to the micro-benchmarks, starting from the higher-level dialects improves performance in both compilation and execution time.

Although the template-based compilation approach of [42] also supports a subset of the TPC-H queries, it is difficult to compare the two directly, as LingoDB provides more optimized operator implementations and an improved query planner. Even for TPC-H Q6, where the query plans are identical, our approach executes around 2x faster — both run on our machine. For other queries, where query optimization is important, the run-time differences are orders of magnitude. The compilation time of our approach, however, is 10x larger; we elaborate on the details later in Sec. 7.5. Nonetheless, a code generation time in the order of milliseconds is sufficiently small, as execution times of the remaining stages in the query execution pipeline — query optimization and prior MLIR lowerings — take multiple milliseconds anyway.

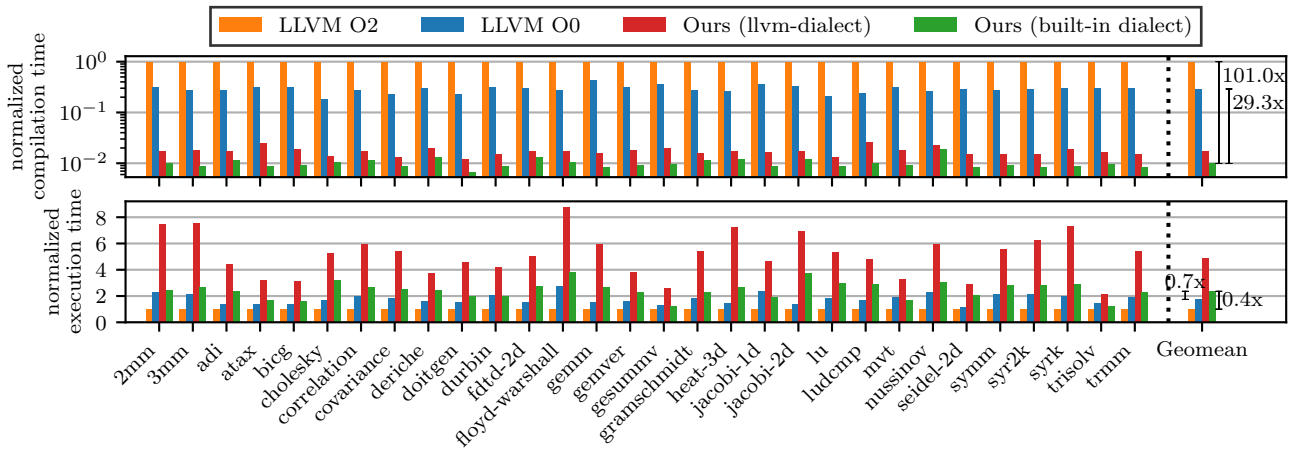

Figure 4. Performance of the PolyBenchC suite normalized to LLVM `-O2` on x86-64.

7.3 PolyBenchC

In addition to JIT compilation, our approach can also be used for the static compilation of languages like Fortran and C, where short compile-times are essential during development.

We first evaluate our approach on the PolyBenchC [36] benchmark suite, a widespread example of polyhedral optimization techniques. Figure 4 shows the results. The standard deviation of the metric remains below 10% for execution and compilation time with all approaches.

Compiling from the upstream dialect input not only provides faster compilation than starting from the LLVM dialect but also the execution time difference is very apparent for these programs, as the canonicalized representation using MLIR upstream dialects is more concise and expressive when it comes to numeric kernel computations. Compilation is one to two orders of magnitude faster with our approach on both abstraction levels compared to LLVM; the execution speed ranges from a slowdown of 165% (*jacobi-2d*) to a 20% speedup (*jacobi-1d*) compared to LLVM `-O0` (median slowdown: 34%).

A comparison with the approach of [42] is only partially possible. Although they run the same benchmark suite, their evaluation targets compilation from optimized WebAssembly code, in contrast to a less optimized and more high-level representation as we do. The closest we can get is instead of starting from unoptimized input code to use Clang optimization level `-O3` to derive an optimized input representation similar to how the WebAssembly input was already optimized once. However, as our input still remains comparably high-level, we miss any possible back-end optimizations that might be applied during WebAssembly emission. Again, we reevaluated their results on our machine. The execution time difference with the optimized input ranges between a slowdown of 10x and a speedup of 2x for our approach compared to theirs. However, we generate code in half the time required for their approach, again taking advantage of the concise representation of higher-level IRs.

7.4 CoreMark and SPECint 2017

To show the full extent of the LLVM-IR coverage, we run CoreMark [1] and SPEC CPU 2017 Integer [2] benchmarks. In all cases, we could not derive MLIR code from the C sources using Polygeist due to its limited functionality and, therefore, only measured the LLVM-dialect level. The Fortran benchmark (SPECint 548.exchange2) was compiled using Flang [4], which can output the LLVM MLIR dialect directly. Some SPEC benchmarks (500.perlbench, 502.gcc, and 531.deepsjeng) could not be transformed into MLIR at all, or the resulting MLIR led to a timeout or crash, even when compiling with the LLVM back-ends. Furthermore, we excluded the other C++ benchmarks due to their use of C++ exceptions, which we currently do not support. Due to limited memory on our AArch64 machine, we could only run one of the reference executions of SPEC 557.xz on AArch64. We report the median from three compilations/executions of the reference workload in Fig. 5. All results have a standard deviation of less than 5% for both dimensions with all approaches.

Our approach generates code one to two orders of magnitude faster than LLVM, while execution time is 2–3x slower than LLVM `-O0`. In particular, our register caching optimization strongly impacts these benchmarks, leading to a runtime improvement of nearly 2x.

On AArch64, the relative compile-times closely follow the ones on x86-64, while the execution time slowdowns are slightly higher on AArch64 for 525.x264 and 548.exchange2.

7.5 Compile-time Analysis

7.5.1 Template Generation. Template generation happens ahead-of-time and is therefore not considered to be time-critical. Table 1 lists the numbers, sizes, and generation times of templates used for previous evaluations. They were obtained by timing the template generation stage and inspecting the resulting template library.

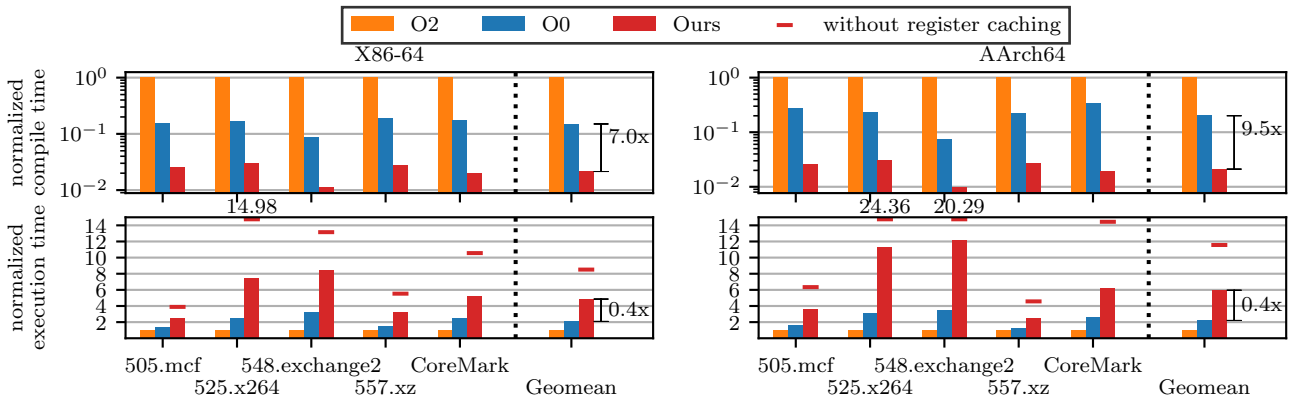

Figure 5. Results for the CoreMark and SPECint 2017 benchmarks — normalized to LLVM `-O2`.

Table 1. Metrics of the ahead-of-time template generation stage for the individual benchmarks. Size is raw code size, metadata includes information on patch locations and types.

	Count	Size [kB]	Metadata [kB]	Generation Time [s]
Microbench.	72	1	20	< 1
PolyBenchC	411	8	112	2
LingoDB	1735	11	466	10
SPEC	5033	122	1390	37

The number of templates for the SPEC benchmark suite seems comparably high but mainly consists of 1400 `func` and 1300 `call` templates, whose signature contains the respective symbol names, as well as 2000 `getelementptr` templates, as constant indices are stored as properties of the instruction. By providing a custom template implementation for `getelementptr` and ignoring the symbol name — only one template for all functions with the same function signature — for `func` (reduced to 280) and `call` (reduced to 400), we could reduce the number of required templates down to about 1000.

Compared to [42], we generate a lot less templates, as we only generate a single variant for one instruction signature, contrary to multiple variations as required for their register allocation scheme and supernode construction. Additionally, their reported template generation time is in the order of minutes, significantly slower than our approach.

7.5.2 Run-time Compilation. At its core, template-based code generation strives for very low compilation times. To provide further insights into where the time of the time-critical compilation stage is spent, we instrumented our compiler with additional time measurements — Figure 6 shows the most apparent components. The most significant part is spent on the template instantiation (cf. Sec. 4.2), followed by template selection (cf. Sec. 4.1). The latter could be avoided completely by directly mapping the input MLIR instructions to their corresponding signatures. This mapping could be done using perfect hashing (used by [42]); however, this would preclude dynamically adding further templates. Tracking the current storage location for each value via a hash map is also comparably expensive; this could be optimized by storing the location inline with the value, but MLIR does not support attaching custom information to a value. Finally, evaluating global constants also takes up some compilation time because they must be evaluated before generating code for them, as described in Sec. 3.5.

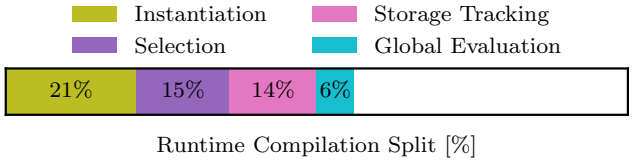

Figure 6. Breakdown of run-time compilation on SPEC benchmarks. The remaining time is spent on the configuration of template instantiation.

The remaining time is spent reflecting for each input MLIR instruction on its type, the input operands, the result values and regions to configure template instantiation correctly. Operands must be put into the expected register, or its storage offset must be recalled, result values are assigned to slots, stored and cached in registers, and the memory offsets for region arguments, as well as terminator operands, must be recorded. Profiling indicates that a substantial portion is spent directly on the MLIR reflections. This can be avoided by statically providing the information during the compilation of the framework, which is only possible for a restricted, previously known domain, contrary to what MLIR provides.

In summary, taking advantage of the flexibility provided by MLIR comes with some costs and currently limits major compile-time improvements, motivating further performance improvements in MLIR in the future.

8 Discussion and Future Work

The results show that our approach achieves an order of magnitude faster compile-times than LLVM `-O0`. Although execution times are 3x higher on some benchmarks, they are fairly close to and sometimes even as fast as the existing baselines on other benchmarks. Notably, the combination of massively reduced startup time with not too much slower execution enables effective use as baseline JIT compiler.

Using MLIR as a starting point allows us to target a continuously growing and open ecosystem, reducing complexity as higher-level optimization (e.g., supernodes) can easily be reflected in MLIR's multi-level approach. Automated template generation obviates the required effort for implementation and maintenance without compromising on the promised run-time to compile-time tradeoff. MLIR's flexibility, however, does come at some cost: the compilation times cannot always keep up with a previously presented Copy-and-Patch approach [42], which compiles roughly between 2x slower to 10x faster. Nevertheless, we argue that the minimal differences — in absolute terms — hardly result in reduced latencies in end-to-end scenarios and, therefore, are not worth the effort of manual template construction. In turn, our code can keep up with — if not improve on — execution speed.

Additionally, our approach is not limited to JIT use cases: we can provide a reasonable compile- to run-time tradeoff for almost arbitrary dialect inputs, including LLVM-IR. Our approach can offer a fast compilation tier that is at least applicable in development scenarios, where most of the time is spent on compiling code, of which only a small snippet is ever executed. However, to fully cover this scenario, there is some work to do on emitting debug information and adding supported binary formats (e.g., Mach-O and PE).

In all cases, our approach is applicable without major changes to the currently existing MLIR infrastructure: it uses the same input representation that is also used for regular compilation, and all dialects with a lowering to LLVM-IR can be, at the very least, supported on LLVM-IR dialect level.

9 Related Work

The most recent and most similar stand-alone, template-based compiler implementation [42] uses templates written in C++, which are precompiled to machine code using LLVM. When compiling a program, the framework combines the templates for the operations and applies some cheap optimizations (e.g., jump elimination). For further performance improvements, they employ a simple register allocation schema limited to usage inside expressions by providing templates for multiple possible register assignments. Our approach, in contrast, generates templates automatically instead of manual C++ programming. Furthermore, our register caching is generally more flexible and not limited to expressions. Therefore, it is more effective beyond micro benchmarks and improves the execution of large programs.

Historically, one main application of template-based code generation was dynamic code optimization on run-time invariants. Vcode [16] was one of the first template-based code generation systems focusing on fast compilation. It provides a reduced platform-agnostic instruction set that is translated to machine code by merely combining hand-assembled templates for each of their operations. It was employed in the TCC compiler [17, 35] for dynamic run-time compilation. Another approach [6] also targeted dynamic code specialization, which combines prepared machine code templates and fills missing holes for dynamic constants during run-time compilation. Consel et al. [12, 32] generated templates from C code in combination with the code required for their instantiation. They located patchpoints in the object files using block labels and introduced the idea of using external symbol addresses to model unknown run-time constants [32]. These approaches target computationally intensive kernels and require handwritten templates, annotations, or specialized compilers. Our approach does not prioritize run-time performance and is functional without offline program preparation.

A more recent application of template-based code generation is found in the initial version of QEMU [8]. Guest instructions were mapped to a set of micro-operations, which were implemented as hand-crafted templates that can be combined to generate target code. Templates were written in GNU C, making use of special GCC flags for specialized register assignments; they reused the idea of external symbol addresses for run-time constants [18]. Nonetheless, this approach was later dropped in favor of raising the input instructions to the TCG intermediate representation [9].

Template-based code generation is nowadays present in baseline compilers for adaptive execution [5, 7, 41] or lightweight assemblers [3, 23, 33]. Both applications differ from our approach as they operate on byte or native code inputs, whereas we generate code from a high-level representation.

10 Summary

In this paper, we outlined a template-based code generation approach for MLIR. Our template generation leverages existing lowerings of MLIR instructions through LLVM and thereby overcomes the limitations of state-of-the-art approaches, which require explicit handwritten templates.

Our results show performance improvements regarding compile-time compared to the existing LLVM `-O0` pipeline in the 10–30x range. Run-time is typically slower by 1–3x, but it even provides comparable or improved performance on a few programs. Our approach can be integrated into existing MLIR workflows with moderate effort and provides a fast compilation tier with only slightly slower execution.

Data Availability Statement

The sources for our template-based MLIR compiler and the respective benchmark data are available in Zenodo [15].

References

[1] EEMBC 2009. *CoreMark Benchmark*. EEMBC. Retrieved 2023-08-30 from https://www.eembc.org/coremark/

[2] SPEC 2017. *SPEC CPU 2017*. SPEC. Retrieved 2023-08-30 from https://www.spec.org/cpu2017/

[3] Free Software Foundation 2022. *GNU lightning*. Free Software Foundation. Retrieved 2023-08-29 from https://www.gnu.org/software/lightning/manual/lightning.html

[4] LLVM 2023. *Flang*. LLVM. Retrieved 2023-08-30 from https://github.com/llvm/llvm-project/tree/main/flang/

[5] Mozilla Foundation 2023. *SpiderMonkey*. Mozilla Foundation. Retrieved 2023-08-25 from https://spidermonkey.dev/

[6] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. 149–159. https://doi.org/10.1145/231379.231409

[7] Clemens Backes. 2018. *Liftoff: a new baseline compiler for WebAssembly in V8*. Retrieved 2023-08-28 from https://v8.dev/blog/liftoff

[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX annual technical conference, FREENIX Track*. California, USA, 46.

[9] Fabrice Bellard. 2009. *Tiny Code Generator*. Retrieved 2023-08-28 from https://github.com/qemu/qemu/blob/v4.2.0/tcg/README

[10] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* (2022). https://doi.org/10.1145/3544559

[11] Kevin Casey, David Gregg, M Anton Ertl, and Andrew Nisbet. 2003. Towards Superinstructions for Java Interpreters. In *Software and Compilers for Embedded Systems: 7th International Workshop, SCOPES 2003, Vienna, Austria, September 24-26, 2003. Proceedings 7*. Springer, 329–343. https://doi.org/10.1007/978-3-540-39920-9_23

[12] Charles Consel and François Noël. 1996. A General Approach for Run-Time Specialization and Its Application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 145–156. https://doi.org/10.1145/237721.237767

[13] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Färber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, and Xiao Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *Conference on Innovative Data Systems Research*.

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254. https://doi.org/10.1145/2463676.2463710

[15] Florian Drescher and Alexis Engelke. 2024. Artifact for CC'24 paper on Fast Template-Based Code Generation for MLIR. https://doi.org/10.5281/zenodo.10571103

[16] Dawson R. Engler. 1996. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. 160–170. https://doi.org/10.1145/231379.231411

[17] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 1996. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 131–144. https://doi.org/10.1145/237721.237765

[18] Nathaniel Wesley Filardo. 2007. *Porting QEMU to plan 9: QEMU internals and port strategy*. Retrieved 2024-01-19 from https://www.contrib.andrew.cmu.edu/user/nwf/paper-strategy.pdf

[19] Google. 2023. *What is V8?* Retrieved 2023-04-28 from https://v8.dev

[20] Tian Jin, Gheorghe-Teodor Bercea, Tung D Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling ONNX Neural Network Models using MLIR. *arXiv:2008.08272* (2020).

[21] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* (2022). https://doi.org/10.14778/3551793.3551801

[22] Minhaj Ahmad Khan, H-P Charles, and Denis Barthou. 2007. An effective automated Approach to Specialization of Code. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 308–322. https://doi.org/10.1007/978-3-540-85261-2_21

[23] Petr Kobalicek. 2014. *AsmJIT Project*. Retrieved 2023-08-29 from https://asmjit.com/

[24] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Conference on Innovative Data Systems Research*.

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[27] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. 2022. TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment. *IEEE Micro* (2022), 9–16. https://doi.org/10.1109/MM.2022.3178068

[28] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proceedings of the VLDB Endowment* (2017), 1–13. https://doi.org/10.14778/3151113.3151114

[29] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[30] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 119–134. https://doi.org/10.1145/3572848.3577475

[31] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:209379505

[32] F. Noel, L. Hornof, C. Consel, and J.L. Lawall. 1998. Automatic, Template-based Runtime Specialization: Implementation and Experimental Study. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*. 132–142. https://doi.org/10.1109/ICCL.1998.674164

[33] Mike Pall. 1999. *DynASM*. Retrieved 2023-08-29 from https://luajit.org/dynasm.html

[34] Filip Pizlo. 2020. *Speculation in JavaScriptCore.* Retrieved 2023-20-04 from https://webkit.org/blog/10308/speculation-in-javascriptcore/

[35] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. 1997. Tcc: A System for Fast, Flexible, and High-Level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. 109–121. https://doi.org/10.1145/258915.258926

[36] L.-N. Pouchet and T. Yuki. 2015. *PolyBench: The Polyhedral Benchmarking suite.* Retrieved 2024-01-19 from https://web.cs.ucla.edu/~pouchet/software/polybench/

[37] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 322–332. https://doi.org/10.1145/199448.199526

[38] Gerald Sussman and Guy Steele. 1998. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation* (12 1998), 405–439. https://doi.org/10.1023/A:1010035624696

[39] Transaction Processing Performance Council. 2023. *TPC Benchmark H.* Technical Report. Transaction Processing Performance Council.

[40] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* (2022).

[41] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* (2013). https://doi.org/10.1145/2400682.2400689

[42] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proc. ACM Program. Lang.* (2021). https://doi.org/10.1145/3485513

Received 13-NOV-2023; accepted 2023-12-23