TPDE: A Fast Adaptable Compiler Back-End Framework

Tobias Schwarz
Technical University of Munich
Munich, Germany
tobias.schwarz@tum.de

Tobias Kamm
Technical University of Munich
Munich, Germany
tobias.kamm@tum.de

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

***Abstract*—Fast machine code generation is especially important for fast start-up in just-in-time compilation, where the compilation time is part of the end-to-end latency. However, widely used compiler frameworks like LLVM do not prioritize fast compilation and require an extra IR translation step increasing latency even further; and rolling a custom code generator is a large engineering effort, especially when targeting multiple architectures.**

Therefore, in this paper, we present TPDE, a compiler back-end framework that adapts to existing code representations in SSA form. Using an IR-specific adapter providing canonical access to IR data structures and a specification of the IR semantics, the framework performs one analysis pass and then performs the compilation in just a single pass, combining instruction selection, register allocation, and instruction encoding. The generated target instructions are primarily derived code written in a high-level language through LLVM's Machine IR, easing portability to different architectures while enabling optimizations during code generation.

To show the generality of our framework, we build a new back-end for LLVM from scratch targeting x86-64 and AArch64. Performance results on SPECint 2017 show that we can compile LLVM-IR 8–26x faster than LLVM `-O0` while being on-par in terms of run-time performance. We also demonstrate the benefits of adapting to domain-specific IRs in JIT contexts, particularly WebAssembly and database query compilation, where avoiding the extra IR translation further reduces compilation latency.

***Index Terms*—Fast Compilation, Code Generation, LLVM**

I. Introduction

Just-in-time (JIT) compilation is a widely used technique for improving the performance of use cases ranging from efficient execution of dynamic languages like JavaScript [1], [2], bytecode languages like WebAssembly [3], [4] or Java [5], acceleration of database queries [6], [7], [8], [9], to system emulation with binary translation [10]. A crucial part for a high-quality user experience is a low startup time, in which the input must be analyzed and transformed to machine code. Within this trade-off of generating code fast and producing high-quality code, such runtime systems often use a multi-tiered compilation system, where a fast compiler for quick startup is typically paired with an optimizing compiler for achieving high execution performance. In addition to JIT compilation, low-latency compilation is also important for developer productivity to shorten compile–test cycles.

A very popular compiler framework is LLVM [11], which also has built-in support for JIT code execution. The framework not only features a high-quality optimizer and machine code generator, but also provides an unoptimized compilation pipeline that is substantially faster. However, many JIT compilers have their own intermediate code representation (IR), requiring an extra translation to LLVM's IR, which increases latency. In addition to that, the compilation times of LLVM are generally high, even in the unoptimized pipeline [12]. Thus, several systems have moved away from LLVM as their baseline compiler and built their custom back-end [13], [2], [9], [14].

However, rolling a custom back-end requires finding solutions for complex problems like register allocation and ABI implementation and thus is a substantial effort to develop and maintain. This effort increases further when porting the back-end to a new architecture, as many parts have to be adapted. Moreover, such custom back-ends only support a single IR and are typically deeply embedded into their surrounding system, making it hard to reuse the code for other projects.

To address these problems, we present TPDE, a flexible compiler back-end framework focusing on fast compilation, where the generation of machine code including register allocation happens in just a single pass. Instead of rolling a new custom IR, our framework can be flexibly adapted to existing IRs that represent code in Single Static Assignment (SSA) form. To compile code with our framework, a user specifies two components: (a) an *IR adapter*, through which the framework can query information about the IR like the successors of a basic block or the operands of an instruction; and (b) a set of *instruction compilers*, functions which generate machine code for an instruction, essentially specifying the semantics. The framework performs a liveness and loop analysis of the input IR and then steers the code generation, taking care of register allocation, spilling, and ABI handling for function calls and parameters. During code generation, the instruction compiler can call back into the framework for allocating temporary registers or for communicating register constraints.

To simplify writing instruction compilers and ease porting to different architectures, we provide a tool to generate snippets of target instruction sequences from LLVM's Machine IR, so that the semantics of to-be-generated code can be specified in a high-level language like C/C++ while maintaining the flexibility of assigning registers and performing low-level optimizations like use of more complex addressing modes. These snippet encoders can then be called by an instruction compiler, allowing an architecture-independent implementation of many instructions.

979-8-3315-9288-2/26 © 2026 IEEE

Using our framework, we implemented a compiler for LLVM-IR, fully independent of LLVM's existing code generation infrastructure, that is capable of compiling typical unoptimized code, for example, as generated by Clang for C and C++ programs. This not only reduces the compile-time of ahead-of-time compilers like Clang or Rustc in total, but, more importantly, also systems that already use LLVM for JIT compilation (e.g., PostgreSQL [15], Clang-Repl [16], Julia [17]) can easily use our TPDE-based back-end as a significantly faster baseline compiler. Our LLVM back-end targets x86-64 and AArch64 and consists of less than 8k lines of code. Performance results evaluating the SPEC CPU2017 benchmarks show that our back-end is 8–26x faster than LLVM's compile-time-focused `-O0` pipeline while achieving similar run-time performance of the generated code ($\pm 9\%$).

To show the flexibility of our framework and its benefits for JIT compilation, we also implement a back-end for Cranelift IR [18], [19] in the context of Wasmtime [3] to compile WebAssembly code, where our TPDE-based back-end was able to outperform Cranelift's fast compilation mode in compile-time and run-time performance. We also implement a compiler for the Umbra database system [20], which uses a custom, domain-specific IR in SSA-form for compiling SQL queries to machine code. Here, our TPDE-based back-end is capable of being as fast as Umbra's specialized and highly-optimized direct emit back-end [13], [14] while maintaining a similar performance on the generated code.

The main contributions of this paper are:

- A novel IR-independent and highly efficient compiler framework that adapts to existing IRs in SSA form and only requires a specification of how to access IR data structures and the semantics of the IR instructions, avoiding an extra IR translation step required by existing frameworks.
- A novel approach to extract code generation snippets from LLVM's Machine IR utilizing available instruction and data flow information to enable further optimizations and additionally significantly reducing the effort for porting a custom compiler to a different architecture. This enables substantial code quality improvements over previous template-based compiling approaches [21], [22].
- An implementation of a fast, single-pass code generation back-end for LLVM-IR targeting x86-64 and AArch64, which is 8–26x faster than the LLVM `-O0` pipeline while achieving similar code quality, making LLVM again a suitable candidate for baseline JIT compilation.

The remainder of this paper is structured as follows: In Sec. II, we describe the TPDE framework itself. Next, Sec. III covers our approach to derive code generation snippets from a high-level language and their integration into TPDE. Sec. IV summarizes conceptual and current engineering limitations of our approach. We then show our implementation and benchmark results of back-ends for LLVM-IR, WebAssembly, and Umbra IR in sections V, VI, and VII, respectively. Finally, Sec. VIII covers related work and Sec. IX summarizes our findings.

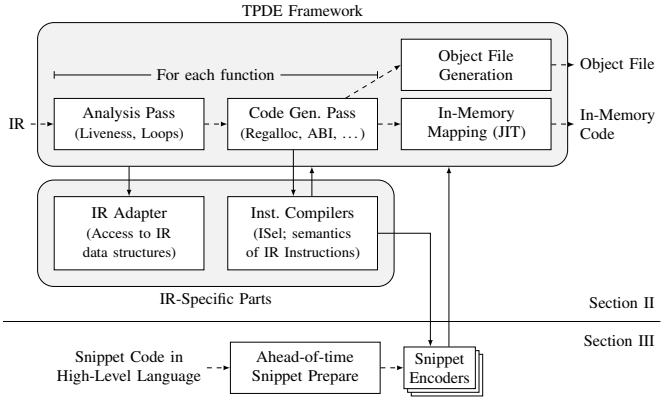

Fig. 1: Overview of TPDE. The framework adapts to any IR in SSA form through an IR adapter, which exposes relevant IR properties in a canonical form, and instruction compilers, which provide the semantics for IR instructions. Instruction compilers can optionally use instruction snippet encoders, which are generated ahead-of-time from a high-level language.

II. Framework

Our main goal is a very fast and reusable compiler framework. In contrast to many existing systems [23], [11], [24], however, we do not want to restrict ourselves to one specific IR which has to serve all use cases and requires a separate translation step from a previous code representation. Instead, we want to design the framework in a mostly IR-agnostic way that is capable of adapting to widely used IRs. By avoiding the IR translation step and additionally relying on code specialization through C++ templates to avoid an extra level of indirection between the framework and its user, we minimize the incurred performance cost of using a compiler framework instead of rolling a custom code generator. We implemented this approach in our novel compiler framework TPDE[1]; Figure 1 gives an overview over the architecture of the framework.

A. Design

1) Separating IR-Dependent Components: A primary objective while designing an IR-agnostic compiler framework was the separation of parts that tend to be highly dependent on the IR, e.g., instruction semantics and concrete data structures, and parts that tend to be less dependent on the IR and more reusable, e.g., register allocation and implementing ABI specifics. Therefore, to adapt TPDE to an IR, a user needs to supply two components: The first component is an *IR adapter*, through which the framework can query all needed information about the IR in a canonical way, like instruction operands or successors of a basic block. The second component is the *instruction compiler*, which has to generate code for a single IR instruction. It can query the framework about the current location of its operands, require registers, possibly with constraints, emit instruction or data fragments, and communicate the location of the instruction results back to the framework. For several other parts of the framework, like ABI handling, default implementations are provided, but can be overridden if required.

[1]https://github.com/tpde2/tpde, license Apache-2.0 with LLVM exception.

2) IR Requirements: We designed our framework around IRs that are in strict SSA form, which is extremely popular among compilers, including LLVM [11], the MLIR ecosystem with its wide range of dialects [25], and many others [23], [24], [20], [26]. In contrast to back-ends permitting mutable variables, SSA simplifies tracking of value states during code generation (knowledge that value will not be modified) and permits using a more coarse-grained and therefore faster liveness analysis, which is important for code quality. Mutable values are hence not directly supported, but expensive construction of SSA form is not required: mutable variables can be assigned to stack slots. Constructing SSA from, e.g., expression trees is often cheap and requires no further analyses. In our framework, we went with the notion of ϕ-nodes, although support for block arguments, which tend to allow multi-edges with different values, could be added with reasonable effort.

Many IRs allow for instructions that produce multiple values, e.g., MLIR, values that consist of multiple components, e.g., LLVM struct or array values, or values that require multiple registers, e.g., 128-bit integers. We model IR values generally as multi-part values; the adapter specifies the size and preferred register bank (typically general-purpose or floating-point/vector) for each value part. In our framework, all value parts are treated separately for the purpose of register allocation. Some IRs like LLVM allow constants to be used at any place where a value can be used. To cover this, a value part can also be a constant.

As the framework is not directly concerned with actual instruction semantics other than ϕ-nodes, values are generally untyped and the only relevant property is their size, which is needed for moving values and spilling. This keeps the framework flexible and prevents the need to constantly adjust the core for the wide range of types that are used in IRs.

3) Two-Pass Approach: As our key goal is fast compilation, we want to do code generation in just one single pass. Consequentially, instruction selection, register allocation, value spilling, and instruction encoding need to be done together in a linear pass over the IR. We especially want to avoid creating another complete in-memory IR of the program, as this would not only have a significant performance impact, but also would impede the clean separation of IR and framework.

However, to achieve a reasonable code quality and size, liveness information about IR values is essential. As many values in SSA IRs tend to have only few users, often in the same basic block, liveness information allows to avoid generating unused spill code and to reuse registers as soon as their values are no longer needed. Thus, we do a separate analysis pass to compute live ranges and the number of users of every value before the main code generation pass.

As a consequence of the single code generation pass, the compilation order of basic blocks has a substantial impact on length of the live ranges. Therefore, the analysis pass additionally determines the order in which blocks are compiled.

4) Integration in Multi-Tier Compiler: The framework targets building fast, non-optimizing back-ends, which can be used as baseline compiler in a multi-tier JIT compiler. Optimizing back-ends often need IR transformations, which are highly IR-specific and thus cannot be provided by TPDE. However, using TPDE on optimized IR can provide a middle-ground between a baseline and a fully optimizing compiler.

B. IR Adapter

The IR adapter is the only way for the framework to access the IR and therefore must expose all information that is required by the framework. The adapter is specified to the framework as a template parameter, enabling inlining of adapter methods and avoiding virtual function calls.

1) Data Types and Initialization: In the context of the IR adapter, the framework will refer to all functions, blocks, and values by reference data types defined by the IR adapter.

2) Functions: The adapter needs to provide a list of all functions that should end up in the symbol table, including both defined functions and declarations of external functions. For function definitions, the adapter also has to expose the function arguments and the basic blocks. Additionally, the adapter can also expose fixed-size stack variables, which will be allocated by the stack frame initialization of the framework.

3) Basic Blocks: A basic block consists of optional ϕ-nodes and a list of instructions. As the analysis pass requires a control flow graph, it also needs to enumerate all possible successors. We do not require an enumeration of predecessors, as not all IRs might have these readily available. Furthermore, the adapter needs to expose 64 bits of storage per basic block. This way, the framework does not need to create hash tables for basic blocks itself, but can rely on more efficient ways to store data in basic blocks if provided by the IR. For IRs that already have auxiliary storage fields, the adapter can simply expose these, and for IRs like LLVM that number their blocks, the adapter can manage an array for this storage.

4) Values: The framework distinguishes different types of values: instructions, ϕ-nodes, arguments, stack slots, and constants. For all values, the adapter needs to expose the number of value parts (cf. Sec. II-A2), their size, and their preferred register bank. The latter is used when the framework needs to copy values, for example when moving values to ϕ-nodes. The framework will not materialize instruction lists or dependency chains and will query the adapter as required.

Moreover, the adapter needs to expose a per-function-unique number of every argument and value, which will be used as an array index. This substantially improves performance over hash tables, as per-value data structures are accessed frequently. Although many IRs do not provide such a numbering, many IRs provide auxiliary data fields, which can store this number. Some sub-types of values have additional properties, e.g., for arguments to correctly map them to registers or stack slots.

C. Analysis Pass

To determine when a value is no longer used and associated registers can be reused, the code generator needs liveness information about arguments, ϕ-nodes, and the results of all instructions. This is particularly relevant for two-address architectures like x86-64: without liveness data, the code generator would add a copy even at the last use of a value.

Various algorithms for liveness analysis have been proposed [27], [28], [29], however, many of these provide more information than needed or require information that is not necessarily easily accessible, e.g., use lists. To meet our goal of compile-time performance, instead of classical iterative algorithms, we use the algorithm proposed by Kohn et al. [30], which makes use of a loop forest instead and has a runtime linear in the number of instructions. However, their approach performs a custom loop analysis which does not support irreducible loops. As we want to support irreducible loops, we replace their loop analysis with the algorithm proposed by Wei et al. [31]. Although this algorithm has a theoretical worst-case runtime of $\mathcal{O}(N \cdot E)$, the average runtime for typical CFGs is $\mathcal{O}(N + E)$ and it has lower constant factors than other loop finding algorithms that support irreducible CFGs, particularly as it avoids more expensive data structures like union-find. Based on the loop analysis, the analysis pass also determines the order of basic blocks for later compilation.

Combining these, our analysis pass performs the following:

1) Identify loops following the algorithm by Wei et al. [31]. For simplicity, we wrap the whole function in one single loop and build a loop tree similar to [30].
2) Compute block layout. We generally lay out blocks in reverse post-order (RPO) with the minor addition that whenever a block is part of a loop, we place the whole loop together. This slightly shortens the live ranges of values whose liveness ends at the end of the loop.
   We store the layout number in the auxiliary data field of the adapter; from now on, basic blocks are referred to by this number. To mark blocks with multiple predecessors and to mark visited blocks during the RPO traversal, we also use the auxiliary field instead of a separate set.
3) Compute liveness of arguments, instructions, and ϕ-nodes using the second part of the algorithm from [30]. We also determine the number of users of every value. At the end, every value gets assigned a contiguous live range consisting of a start block number and an end block number as well as a flag indicating whether the liveness ends within or at the end of the end block.

This coarse-grained liveness information is sufficient for our purposes. We do not need a more precise beginning of the live range, as we generate code linearly in block-order and inside the block, the liveness implicitly begins at the definition of the value and, due to RPO block order, all uses come after the definition. The information also allows to determine the end of the live range during code generation with sufficient accuracy. Later, the code generator will track the number of remaining uses for every value. When this value reaches zero and the live range ends *within* the current block, the value becomes dead and the register can be reused immediately. Otherwise, even if there are no further uses, the value is still live due to a backedge, where the value must be live for the entire loop.

D. Code Generation Pass

The code generation pass compiles a function linearly into a code buffer. Blocks are compiled in the order determined in the previous analysis pass and instructions within the blocks are compiled in program order. Once code is written into the buffer, it might be fixed up later, e.g., for a jump to a later block, but is generally not moved or modified afterwards. Therefore, the pass has to do instruction selection, register allocation, and machine code generation in a single step.

1) Value Assignments: For every live value, the framework stores an *assignment*, which consists of a stack frame slot for spilling, the in-memory size, the number of remaining uses, and information about every value part. For each value part, we store the current register (if any), the size, and whether the value has already been spilled and the stack slot contains the correct value. Value parts like stack variables can be marked as *trivially recomputable*, so that they do not get spilled on eviction. Furthermore, a value can be *locked*, implying that the value is currently not spillable. This is used to prevent eviction when reloading values into registers for use in an instruction. Flag registers are typically not easily spillable and therefore cannot be used to store values across instructions.

For performance, we store all value assignments in an array indexed by the value number provided by the IR adapter. As there are potentially many values needing assignments, we optimized the data structure for size: for single-part values, an assignment consists of just 16 bytes.

2) Prologue and Epilogues: When compiling a function, the first part that is generated is the prologue. As at this time neither the stack frame size nor the used callee-saved registers are known, sufficient space for saving all required registers is allocated. The final stack frame size and instructions to save callee-saved registers are patched into the prologue at the end; remaining instruction space is padded with no-ops. Likewise, as it is not known whether a function uses dynamically-sized stack allocations, functions always setup a frame pointer and do not use the stack pointer for referring to stack slots.

Similarly, when an epilogue is generated in the middle of the function, e.g., for functions with multiple return instructions, sufficient space for restoring callee-saved registers is allocated at first; only at the end, the actual instructions are filled in. The framework also generates exception unwind information, which needs to provide information about the stack frame layout.

After the initial stack frame setup, the value assignments of the parameters are initialized with their respective locations in register or memory according to the current calling convention.

3) Code Generation for Instructions: To compile an instruction, the framework defers to the instruction compiler provided by the user of the framework, as only the user knows the actual semantics of their IR. An instruction compiler typically does the following:

1) Collect *handles* to value parts that correspond to the instruction operands from the framework, preventing associated registers from getting spilled. When a handle gets dropped, the remaining use count of the value will be decremented automatically.
2) Ensure that the required value parts are in registers. The framework will reload spilled values into registers, but due to value locking, existing handles and associated

```
void emit_neg(IRInstRef inst) {
  // Collect value handle to first part of source.
  auto [_, src] = val_ref_single(inst->getOp(0));
  auto [_, res] = result_ref_single(inst);
  AsmReg src_reg = src.load_to_reg(); // lock value
  AsmReg dst_reg = res.alloc_reg_try_reuse(src);
  ASM(NEGw, dst_reg, src_reg);
} // RAII: destructors will unlock values
```

Listing 1: Instruction compiler for integer negation (AArch64).

```
//--- snippets.c (Encode Snippet)
int32_t neg_i32(int32_t v) { return -v; }
//--- Compiler.cpp (Instruction Compiler)
void emit_neg(IRInstRef inst) {
  // Call generated encode snippet
  encode_neg_i32(val_ref(inst->getOp(0)).part(0),
                 result_ref(inst).part(0));
}
```

Listing 2: Instruction compiler making use of encode snippets. Functionality as in Listing 1, but the code is architecture-independent.

registers remain valid. For value parts that need to be in special registers, e.g. due to instruction constraints, the framework allows specifying an explicit register.

3) Collect handles for the value parts for instruction results and allocate new registers. For architectures like x86-64, which often overwrite one source, registers of source operands can be reused at the end of their liveness (cf. Sec. II-C for the exact conditions).
4) Generate code for the actual semantics. As this might need extra registers, the framework can provide unevictable *scratch registers* for this purpose.

Listing 1 shows an example for a simple instruction compiler.

Branch instructions must be generated through the framework for handling ϕ-nodes of the successor, inserting required spill code, and for releasing registers whose liveness ends at the end of the block.

As steps (2)–(4) can be quite tedious to implement, especially when targeting multiple architectures, we provide an architecture-independent way to generate most parts of this from high-level languages; we describe this later in Sec. III.

4) Instruction Fusing: A very important optimization is the fusion of adjacent instructions, for example, comparison with branch instructions (otherwise, condition flags would need to be materialized into a general-purpose register) and address calculation with load/store operations. Thus, the framework supports instruction compilers fusing instructions from the same basic block, also transitively. As we generate code in program order, only later instructions can be fused into their source instructions. Therefore, code for the later instructions will be generated at the point of the first instruction of the fusion and is only possible if the IR semantics permit such an instruction reordering. For simplicity and performance, instruction compilers might only consider immediately following instructions, at the cost of missing fusion opportunities.

5) Register Allocation: Registers are allocated alongside code generation and therefore we only perform a strictly local, greedy approach. There is no possibility to change registers or insert spill code once the code is compiled. When allocating a register and registers are available, the one with the lowest number is used. Otherwise, an arbitrary evictable register is chosen and spilled, in round-robin manner. As a minor optimization, value parts that are used across multiple blocks inside the innermost loop can be assigned a *fixed* register, which prevents the register from being spilled. This heuristic targets values defined in loops, especially ϕ-nodes in the loop header, which typically contain loop induction variables. For these, frequent spilling and reloading would cause a more substantial performance degradation than reloading unchanged values from outside of the loop. To avoid collisions with register constraints, only callee-saved registers without a special purpose in the architecture are considered as fixed registers.

To further reduce compile time, we do not keep per-block register states, but only track the state at the current code generation point. This implies that for blocks where any predecessor is not immediately preceding in the block order, the register state is no longer available. Therefore, when branching to a block with multiple predecessors or a block that does not immediately follow in layout order, we spill *all* values that have no fixed register and are live at the entry of any of these successors. This way, all live values have a single, well-known location, which is either a fixed register or a stack slot.

Values for ϕ-nodes are moved in their place after this spilling; cyclic dependencies are split using a temporary register or stack slot. Critical edges are always split by inserting a separate block when there are values that need to be moved.

III. Writing Instruction Compilers in High-Level Languages

While the framework as described in Sec. II provides an abstraction for managing registers, it requires an explicit specification of the target instructions to be used for the operations. Although this allows for a very high flexibility for the code that is generated, writing such instruction compilers is often tedious and error-prone, especially for more complex instruction sequences. Moreover, when porting to a different architecture, the entire process has to be repeated.

To address these problems, we provide a way for writing snippets in a high-level language like C/C++. At build time of the compiler, we compile these snippets to LLVM-IR and from there further to LLVM's target-specific Machine IR (MIR), which contains not just a sequence of machine instructions, but also information about data flow dependencies, register usage, and register constraints. From the Machine IR of a function, we generate a *snippet encoder* which, when called, dynamically adjusts the instruction sequence for the actual operands and available registers and then emits the resulting machine code, moving values into registers and allocating registers as needed.

We implemented this approach to generate such snippet encoders for x86-64 and AArch64 as a supplementary tool to the main TPDE framework. Listing 2 shows an example of a snippet and its use in an instruction compiler.

A. Generating Snippet Encoders

As we rely on LLVM's Machine IR to extract the target instruction sequences, the input needs to be written in a language that can be compiled to LLVM-IR, for example, C or C++ using Clang. After applying optimizations, we run the regular back-end pipeline until just before the actual machine code is emitted. At this point, the MIR only consists of encodeable instructions using solely physical registers.

1) Function Signature: The snippet encode function generally takes one parameter per input register followed by one parameter per output register, mapped according to the used calling convention. Hence, this is typically a one-to-one mapping, except for large values like 128-bit integers that split over two registers. We currently only support functions where all parameters and return values are passed in registers; some alternative calling conventions allow for more parameter or result registers than the default calling conventions.

2) Tracking Registers: At the beginning of the encoder, all constrained registers used throughout the function are allocated, e.g., for instructions that require a value in a specific register like the x86-64 division. Doing this early prevents later register allocations from blocking such registers.

It can happen that no such register is currently available, for example, when a specific register is required but is currently fixed, e.g. due to use in a value handle. In this case, we forcefully move the value to a new register, updating references to the register in the input parameters. At the end of the function, we move the value back into its original location. This simple strategy is possible, because the snippet encoder will not refer to IR values that are not supplied as parameters. All other registers used throughout the function are allocated lazily, as no constraints need to be satisfied.

We currently only support functions that do not use or modify the stack or frame pointer, as these are used by the generated function. However, handling such functions is generally possible by translating stack frame allocations of the function to use TPDE's infrastructure for stack allocations. We also note that such an extra stack frame setup is likely unwanted, as it would degrade compile-time and run-time performance. Instead of inlining such complex functions, generating a call to a runtime library seems generally preferable.

3) Handling and Emitting Instructions: For the main part of the snippet encoder, we handle the MIR instructions in order and generate appropriate code. First, all registers used by the instruction are moved into registers. For instructions that clobber some of their input operands (e.g., x86-64 `add`) but the input operand is still used afterwards, we copy the value into a new output register. As we do not change the order of instructions, we ignore certain implicit registers like flags or floating-point control registers; if the flags register is live, we make sure that it is not clobbered by any inserted instructions. Second, we need to make sure that all output registers are allocated, reusing input registers in their last use if possible. Third, the assembler is called for encoding the instruction. As we do not use the LLVM-MC assembler for encoding instructions due to its subpar performance, we have to map the LLVM mnemonics to the ones of our own encoding library. While LLVM's mnemonics do follow a naming scheme, this is not very consistent and there are some exceptions. This required some manual effort to create such a mapping. When an instruction refers to a constant from the constant pool of the MIR function, we add the constant to the read-only data section and emit a relocation.

4) Control Flow: To increase the scope of supported functions, we also support MIR functions with multiple basic blocks. This can happen even for seemingly straight-forward operations, for example when converting a 64-bit unsigned integer to a floating-point value on x86-64.

At the end of the first basic block, we materialize all inputs into registers. From this point onward, we simply reproduce the instruction sequence generated by LLVM without any further optimizations, except that the used registers can differ. Return instructions are generally replaced with a jump to the end of the generated code. This jump is omitted if there are no instructions to be jumped over, e.g., when the function consists of a single basic block ending with a return. Indirect jumps are currently not implemented; however, there should be no structural problems in adding support for these. Function calls require a stack frame, which is currently not supported (see Sec. III-A2).

B. Optimizing for Non-Register Operands

When embedding the code as part of the compilation process, values might be spilled to the stack or can be constants. Depending on the ISA, such values can sometimes be encoded as memory or immediate operands. Furthermore, expressions like register with offset can often be encoded directly into memory operands. Thus, input operands of an encode snippet can not only be registers, but also be a spilled value, a constant, or a simple expression of the form $\mathit{base}+n*\mathit{index}+\mathit{off}$, where *base* and *index* are registers. Expressions can also describe stack variables using the frame pointer as base.

When encoding an instruction with a non-register input, the encoder checks whether the operand can be merged into the instruction. Otherwise, the operand gets materialized into a register. Depending on the available encodings of the instruction, the following variants are checked: (a) replacing a register with an immediate operand for constants; (b) merging expressions into address operands; and (c) on x86-64, using a memory operand for spilled IR values.

Merging expressions into memory operands has a large impact on the code size and performance for programs that frequently access stack variables — otherwise, many simple address computations would need separate instructions. As the supported addressing modes strongly vary between ISAs, their encodings and the exact conditions for merging expressions are hard-coded in the snippet encoder generator.

C. Omitting Register Moves

The code generated by LLVM is optimized as a complete function with fixed registers for inputs/outputs as specified by the calling convention and therefore often contains instructions

to move values out of parameter or into result registers. However, as we only use these snippets internally inside functions, they do not need to adhere to a calling convention, making such moves often avoidable. Therefore, we omit move instructions that have no other side effects used by the program and mark the destination register as an alias for the source register. When the alias is used as an operand, the actual source of the value is used instead. This allows doing the previously described optimizations even after a move. However, care must be taken when the aliased register or the source register is overwritten, in which case a copy must be emitted.

D. Discussion

1) Portability: As MIR is a mostly target-independent code representation, porting the approach to a new architecture is possible with comparably low effort. Encoding optimizations, however, are naturally target-specific and therefore need to be implemented depending on available encodings; the framework itself is very flexible in this regard.

2) Writing Optimizable Snippets: Our optimization for constant operands just attempts to encode the operand as an immediate into the instruction. However, no constant-folding of instructions with all-constant inputs is performed — this would require implementing the semantics for many architecture-specific instructions. This can result in several unneeded instructions, e.g., for 128-bit shifts. In some cases, providing additional snippets for specific value ranges can substantially improve the quality of the generated code.

3) Which Machine IR Stage: We use the Machine IR of a function in its latest stage after register allocation, where all instructions use encodeable physical registers and all LLVM-internal pseudo-instructions are lowered. Although we have to eliminate unneeded moves, we do not have to spill any registers but can be sure that sufficient registers are available.

An alternative approach would be to use the MIR before register allocation, either in SSA-form or after ϕ-node elimination. We previously used MIR in SSA-form, but our current approach greatly simplified the implementation: First, tracking the mix of virtual and physical registers substantially increases complexity when assigning registers. Second, at earlier stages, LLVM employs several pseudo-instructions, which we would need to expand manually, as the normally responsible LLVM pass requires allocated registers. Third, when using MIR in SSA-form, we need to lower ϕ-nodes ourselves when handling programs with control flow; reusing the existing implementation would require substantial changes, due to its focus on IR values.

We also considered extracting the machine code sequence from object files. While this would be independent of the unstable LLVM API and permit other compilers, it would require substantial effort to reconstruct information that is readily available in MIR, including data flow dependencies, register constraints, stack frame layout, and constant pools.

IV. Limitations

Conceptually: Our approach is primarily targeted at fast code generation for contemporary CPU architectures. As such, the proposed design is not suitable for compiling to GPUs or bytecodes like SPIR-V or WebAssembly. IRs that are not in SSA form need an extra transformation, although non-SSA variables can be modeled as stack slots. The simple approach to register allocation and instruction selection inherently limits the performance of the generated code; using TPDE to replace an optimizing back-end would require architectural changes.

Engineering: Support for ISAs other than x86-64 and AArch64 is conceptually possible, as is platform support beyond Linux/ELF by implementing the corresponding object file formats and ABIs. The IR adapter currently does not support block arguments (CFG multi-edges with different values for ϕ-nodes) and dynamically-sized values, e.g. for vector extensions like ARM SVE. Tracking all locations of values, e.g. to identify pointers for garbage collection, is likely to need framework support. None of these limitations are inherent to the framework and can be addressed with some engineering effort. The main limitation of our snippet encoder generator is the lack of support for function calls, indirect jumps, and stack allocations; such functionality currently needs to be coded manually. However, handling these cases is structurally possible and most of the information required to map these to corresponding framework functions is readily available in the Machine IR.

V. Case Study: Compiling LLVM-IR

To show the generality of our framework, we build a fast baseline compiler for LLVM-IR targeting x86-64 and AArch64, most notably without using any of LLVM's existing code generation infrastructure. As our goal is a baseline compiler, we limit ourselves to IR constructs that are typically found in unoptimized code, e.g., as produced by Clang on typical C++ programs. We therefore exclude uncommon data types like integers larger than 128 bits, floating-point types other than `float`/`double`, and unusual vector types. We also do not support inline assembly and rarely used features like garbage collection or computed `goto`; however, we note that implementing these with our framework is structurally possible.

A. Implementation

1) IR Adapter: Many of the functions of the IR adapter translate naturally to function calls on LLVM's data structures. Nonetheless, we need to do a preparation pass over the IR of the function before the analysis pass to number each global value, block, and instruction, which are then exposed to the framework. During this pass, we also convert constant expressions into normal instructions to simplify handling of constants later on.

2) Compilation: Before compiling any functions, global variables and aliases are transformed into corresponding symbols and chunks in the data sections, generating relocations as appropriate. The implementations of many LLVM-IR instructions are architecture-independent by heavily relying on snippet encoders. The only unsupported cases requiring manual implementation are (a) calls/returns, (b) branches, (c) integer comparisons, and (d) target- or ABI-specific intrinsics like access to varargs. Integer comparisons are fused with subsequent branches to generate typical compare–branch sequences and pointer arithmetic is merged into load/store when possible.

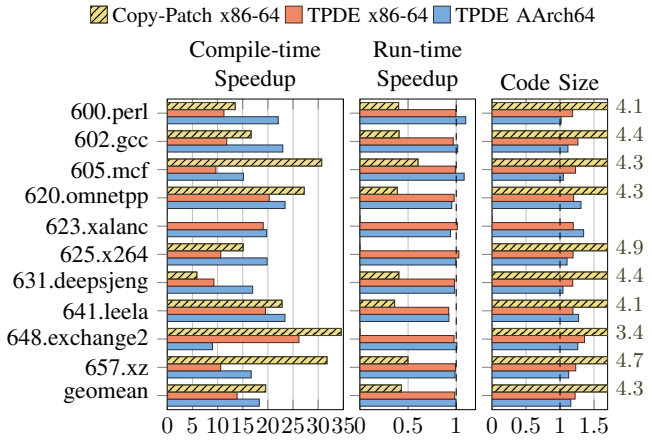

Fig. 2: Compile- and run-time speedup and code size normalized to LLVM `-O0` on SPEC CPU2017 with unoptimized LLVM-IR. Compile-time is back-end time, excluding front-end and required LLVM-IR passes.

Most instruction compilers are a fairly straight-forward implementation of LLVM-IR semantics. Supporting arbitrary-bit-width integers (up to 128 bit) and aggregate values like structs, however, increases implementation complexity.

3) Code Complexity: Although lines of code is by no means an accurate metric to measure code complexity, it still gives a rough estimate. In total, our LLVM compiler consists of 7.7 kLOC, of which about 1.4 kLOC are architecture-specific for x86-64 and AArch64 combined. The approach of abstracting target-specific instructions with encoding snippets not only makes the implementation significantly easier to read, write, and maintain, but also greatly reduces the effort of porting to a different architecture. For comparison, a previous implementation without encoding snippets needed roughly 12.8 kLOC, most of which was target-specific.

B. Performance Evaluation

1) Setup: We embedded our LLVM back-end into Clang and Flang and compare it against the LLVM `-O0` back-end, which focuses on fast compilation[2], and the LLVM `-O1` back-end. Additionally, we compare against the copy-and-patch-based LLVM-IR compiler from [22] in an updated version that also supports C++ exceptions and can compile LLVM-IR directly without the overhead of MLIR. However, this compiler only supports x86-64 with unoptimized input IR; the compiler crashes on benchmark 623.xalanc; for 625.x264 and 648.exchange2, the produced binary crashes. In addition to `-O0` LLVM-IR, where all variables are stack-allocated and SSA values have short live ranges, we also evaluate on LLVM-IR produced with `-O1`, where variables are in SSA form.

As benchmark programs, we use the SPEC CPU2017 integer benchmarks (refspeed) on x86-64 (Intel Xeon Gold 6430, 256 GiB RAM, Linux 6.8.0) and AArch64 (Apple M1, 16 GiB RAM, Asahi Linux 6.11.8, only using performance cores). We use Clang/LLVM version 20.1.7.

[2]The LLVM `-O0` back-end runs substantially fewer passes and uses a faster instruction selector and register allocator.

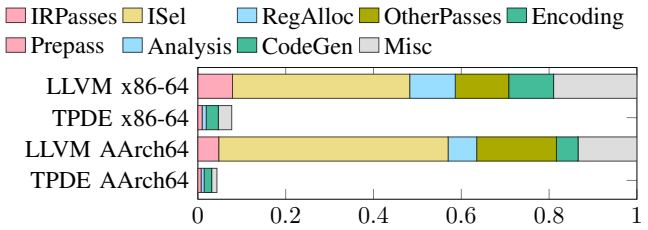

Fig. 3: Average time distribution when compiling SPEC CPU2017 benchmarks (`-O0`). Prepass is the TPDE-specific preparation pass, Misc is object file writing and measurement overhead.

2) Results (Unoptimized IR): Figure 2 shows the compile-time speedup over the LLVM `-O0` back-end when compiling unoptimized LLVM-IR. TPDE can generate code 8–26x faster compared to LLVM. On AArch64, the compile-time improvements are larger (geomean: 18.29x) than on x86-64 (geomean: 13.88x), which is primarily caused by LLVM using the GlobalISel instruction selector by default, which is significantly slower than FastISel [12]. TPDE is still substantially slower compared to the copy-and-patch compiler on x86-64 (which is geomean 19.56x faster than LLVM `-O0`) due to more bookkeeping and explicit encoding of individual instructions. The copy-and-patch compiler is primarily limited by mapping LLVM-IR instructions to their templates, materializing constants, and generating moves to the registers expected by the templates. In terms of run-time performance (cf. Figure 2), our generated code has a similar performance as LLVM ($\pm 9\%$). Instruction fusion contributes an improvement of 8%. The copy-and-patch-generated code is substantially slower (geomean: 2.32x slowdown) due to the huge amount of moves and spills caused by fixed registers used by the templates and the lack of a liveness analysis.

Within the entire compilation including the front-end, with TPDE only 1% of the time is spent inside the back-end, improving end-to-end compilation time by 9–24%. This ratio is especially low for C++ programs, where the front-end takes the largest portion of time. Figure 3 shows the time distribution within the back-ends. Inside TPDE, the largest part is code generation (44%) and the LLVM preparation pass takes 14%. The analysis pass (15%) takes a comparably small amount of time. The remainder of the time is spent for I/O and measurement overhead. LLVM compile times are dominated by instruction selection (40% x86-64 FastISel, 52% AArch64 GlobalISel), but also instruction encoding, object file finalization, and register allocation each take more time than TPDE in total. For a more detailed analysis of LLVM's compile times, see [12], [32].

The resulting code size is shown in Figure 2, indicating a geomean increase of 22% (x86-64) and 16% (AArch64). The primary reasons are the substantially larger prologues/epilogues, which always reserve space for saving/restoring all callee-saved registers, and, on x86-64, the use of large jump offset encodings for forward jumps. The copy-and-patch-generated code is much larger than the LLVM `-O0` code (geomean: 4.27x), primarily due to the large amount of value moves.

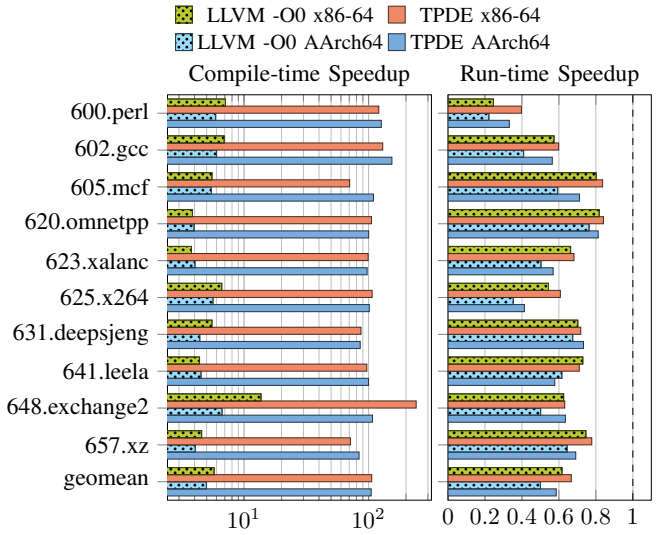

Fig. 4: Compile- and Run-time speedup normalized to the LLVM `-O1` back-end on SPEC CPU2017 with `-O1` IR. Compile-time is back-end time, excluding front-end/optimizations.

3) Results (Optimized IR): Figure 4 shows the compile-time/run-time speedup over the LLVM `-O1` back-end when compiling optimized (`-O1`) LLVM-IR. TPDE achieves a geomean speedup in compile-time of 106x/107x over LLVM `-O1` and 18.4x/21.1x over LLVM `-O0`. Therefore, also for optimized input IR, TPDE achieves similar compilation speed as for unoptimized IR. In terms of run-time performance, our generated code often has a slightly better performance compared to the code generated by the LLVM `-O0` back-end (8%/17%). Compared to the optimizing back-end, however, TPDE-generated code is substantially slower (1.50x/1.71x), primarily because LLVM in this configuration uses a much better register allocator and instruction selector.

C. Discussion

The results show that TPDE allows implementing a fast back-end for a general-purpose IR like LLVM-IR with reasonable effort. The compilation times are substantially faster than LLVM's due to the single-pass code generation approach: LLVM's design around progressive lowering causing materialization of IR data structures after every isolated transformation has a substantial cost and, due to the focus on optimizations, both LLVM-IR and Machine IR track a lot of information, making them expensive to create and modify. Additionally, by targeting different architectures through monomorphization, TPDE avoids the need for flexible, architecture-independent data structures and associated indirections, which are especially costly in LLVM's machine code encoding step. Although TPDE is not as fast as a copy-and-patch-based compiler, the compile-times are in the same order of magnitude and the quality of the generated code is on-par with LLVM `-O0` in terms of performance. As expected, TPDE-generated code is substantially slower than code produced by LLVM `-O1`. Nonetheless, further improvements regarding instruction selection might be possible by utilizing more information from the Machine IR snippets. This is left as future work.

VI. Case Study: Compiling WebAssembly

Wasmtime [3] is a WebAssembly run-time which performs JIT compilation using either the single-pass back-end Winch or the multi-pass back-end Cranelift. The latter still aims to be fast but can optionally perform optimizations on its SSA-IR called CLIF, use a single-pass or backtracking register allocator, and do more elaborate instruction selection. Since TPDE requires an SSA-IR to compile, we created a TPDE back-end for CLIF. In contrast to LLVM-IR, CLIF uses block arguments instead of ϕ-nodes, only supports a limited set of scalar and vector types, and models stack slots separately, referencing them with explicit instructions. Additionally, CLIF also supports values which transparently alias other values. A more detailed description of CLIF can be found in the Cranelift IR reference [19].

A. Implementation

1) IR Adapter: As Cranelift is written in Rust, the adapter needs to provide a cross-language interface to the IR. Most information can be gathered easily from CLIF. Since CLIF treats global values, stack slots, and arguments differently from normal IR values, the adapter has to create dummy values. However, as CLIF uses block arguments and TPDE currently does not support multi-edges with different values, the adapter places empty basic blocks on such edges. For simplification, value aliases are eliminated before code generation.

2) Compilation: Many instructions can be implemented using snippet encoders with similar exceptions as LLVM. In contrast to LLVM, CLIF supports constants as results from special instructions and has separate instructions to generate pointers to stack slots, which need special handling so that they can be fused into other instructions.

3) Code Complexity: The Cranelift back-end using TPDE consists of roughly 4.7 kLOC, of which 0.7 kLOC are architecture-specific and 1.6 kLOC are glue code between C++ and Rust. Our back-end currently does not support vector operations, which significantly lowers the necessary complexity.

B. Evaluation

1) Setup: We evaluate our Cranelift back-end by measuring compile- and run-time on the three default benchmarks in Wasmtime's own benchmark suite Sightglass [33] and PolyBench [34]. We compare against Cranelift with its backtracking and its faster single-pass register allocator, both without IR optimizations, and Winch. We use the same hardware as in Sec. V-B and report the average of 10 compiles and 5 runs.

2) Results: Figure 5 shows the results. The TPDE-based back-end compiles 4.94x faster than Cranelift and 3.10x faster than Cranelift with its fast register allocator, but is 1.53x slower than Winch. There are two primary reasons for this: first, 41% of the time is spent on translating WebAssembly to CLIF; and second, this translation already constructs SSA form for all variables, which is not needed by TPDE and also produces many trivially removable ϕ-nodes. Constructing a more light-weight IR directly from WebAssembly could significantly close

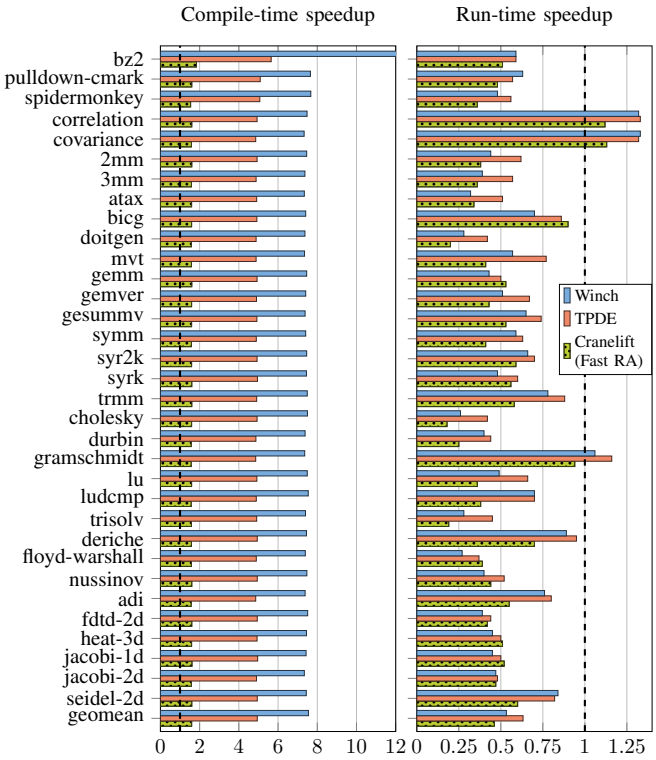

Fig. 5: Compile- and Run-time speedup normalized to Cranelift (default register allocator) on Sightglass and PolyBench benchmarks. Compile-time includes translation of WebAssembly into CLIF and linking.

the gap to Winch's compile times, as only 38% of the time is spent inside TPDE on IR analysis and code generation.

The run-time of TPDE-generated code is better than both Winch and Cranelift with its fast register allocator (1.19x and 1.37x respectively), but 1.58x slower than Cranelift with its optimizing register allocator. A more sophisticated register allocator in TPDE could substantially improve performance.

VII. Case Study: Compiling Umbra IR

Umbra [20] is a compiling database system using JIT compilation to execute SQL queries efficiently. As queries are not always known ahead-of-time, the query latency consisting of compilation and execution time should be as low as possible. To easily allow switching between different compilation back-ends, Umbra first generates all code into its custom IR [20], [13]. The IR is in SSA form, inspired by LLVM-IR, but only supports a small set of data types and has several domain-specific instructions to briefly express frequently occurring operations. [13] describes Umbra IR in more detail.

For optimal execution time for different workload sizes, Umbra has multiple back-ends: for optimized code generation, Umbra IR can be compiled using the optimizing LLVM back-end. For fast code generation, Umbra supports using the non-optimizing LLVM `-O0` pipeline as well as a custom-written *DirectEmit* back-end which compiles Umbra IR in two passes to machine code while achieving a run-time performance typically better than the LLVM `-O0`. However, this back-end has a very high code complexity and is extremely platform-dependent.

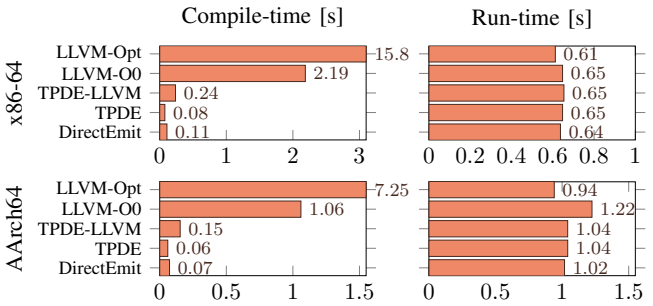

Fig. 6: Compile- and run-time accumulated over all TPC-DS queries (sf=1). Compile-time is code-generation time, excluding query optimization and Umbra IR construction.

Therefore, we wrote an Umbra back-end using TPDE with the goal of matching the compile-time and run-time performance of DirectEmit with substantially less code complexity and platform-dependent code.

A. Implementation

1) IR Adapter: All required information can be directly gathered from Umbra's IR data structures with little to no overhead and without the need for a preparation pass.

2) Compilation: Most instructions can be implemented using snippet encoders with the same exceptions as for LLVM. As Umbra only uses JIT compilation, the addresses of called functions and referenced globals are known during compilation. Therefore, we can simply hardcode addresses of symbols into the generated code.

3) Code Complexity: The Umbra IR back-end using TPDE consists of 3.3 kLOC, out of which 1.4k are target-specific. This is significantly less than the implementation of DirectEmit with 11 kLOC for both AArch64 and x86-64 and closer in length to Umbra's LLVM back-end, which consists of 2.3 kLOC, mostly for the translation of IR semantics.

B. Evaluation

1) Setup: We evaluate the compile- and run-time of our back-end on the TPC-DS [35] benchmark using scale factor 1 and compare it against DirectEmit and LLVM with the regular LLVM back-ends as well as our TPDE-based LLVM back-end (TPDE-LLVM); the hardware is the same as in Sec. V-B. We report the average of 20 compile and execution runs.

2) Results: Figure 6 shows the results. TPDE can generate code that is comparable to code produced by DirectEmit while having a similar compile-time performance. The speedup in compilation time over LLVM is especially large, because Umbra IR has to be translated into LLVM-IR first. The LLVM back-end written with TPDE achieves a substantial speedup already; however, the cost of translating to and preprocessing of LLVM-IR becomes visible.

The run-time performance increase compared to LLVM `-O0` on AArch64 is caused by generating more optimized instruction sequences for complex IR instructions. These need multiple LLVM-IR instructions, which are not fused again at `-O0`.

C. Discussion

Our IR adapter and our generalized framework introduce almost zero overhead compared to the highly specialized DirectEmit back-end. TPDE enables to write a back-end with similar performance characteristics with substantially less effort. Although our TPDE-based LLVM back-end already provides substantially faster compilation over LLVM `-O0`, the extra IR translation has a measurable cost. This demonstrates that our approach of adapting the framework to existing IRs can significantly reduce the compilation latency.

VIII. Related Work

IR-Independent Compilation Approaches: To the best of our knowledge, a largely IR-independent compiler back-end framework that directly interfaces with existing IRs has not been proposed before.

A long-standing scheme for fast compilation consists of precompiling templates to machine code with a standard compiler and then simply concatenating them to quickly generate machine code, optionally with patching constants [36], [37]. One early implementer of this approach was QEMU [10], which identified patch points through relocations, but later moved to a more sophisticated code generation approach. More recently, the approach regained traction under the name "copy-and-patch" [21], which was later adapted for compiling MLIR [22] and Python [38]. A fundamental limitation of the approach is the lack of adaptability: as templates are combined in machine code, there is no way to change registers, replace operands with immediates, or use different addressing modes. Although some of these limitations can be partially alleviated by precompiling multiple variants, this is not generally practicable as the number of templates would be huge. The lack of a liveness analysis results in frequent register moves and stack spills and has a huge impact on run-time performance. In contrast, our snippet encoders utilize information from LLVM's Machine IR to dynamically manage registers and morph instructions to the actual operands, resulting in a performance that is on-par with LLVM `-O0`. Moreover, TPDE also has built-in support for ϕ-nodes, dynamically-sized stack allocations, and C++ exceptions, which are not easily implementable in a template-based code generation approach.

AsmJit [39] provides a low-level abstraction for generating x86 and AArch64 machine code and also a more high-level API, where a user encodes instructions referencing virtual registers into a custom in-memory IR, on which the framework performs register allocation separately. Our approach, in contrast, avoids the separate IR materialization by directly emitting machine code. Additionally, our framework is more high-level and targets compiling SSA code, especially by performing ϕ-node elimination, and has built-in support for generating object files, unwind information, and code for C++ exception handling. Furthermore, our approach to generate snippet encoders from a high-level language provides an architecture-independent way to specify instruction semantics, allowing back-end writers to focus on operation semantics while still giving the flexibility to fine-tune the generated machine code where needed.

Fast Compilers for Fixed Code Representation: Many runtime systems [2], [18], [13] implement their own IR and, to reduce latency, implemented a custom back-end without reusing existing compiler frameworks and without an extra IR translation step. However, this is a substantial effort and such compilers are deeply embedded into large systems, preventing reuse in other projects. Even if a reuse were easily possible, this would nonetheless require writing an IR translator, unnecessarily increasing latency. With TPDE, we provide a highly efficient and adaptable compiler framework, which allows projects to keep their custom IRs, but make it fairly easy to write a code generator and improve portability.

A key design decision of many fast compilers [40], [41], [42] is to reduce the number of code transformations and typically generate code in a single pass. While the lowest tier of JavaScript compilers typically just concatenates simple code fragments or runtime calls to avoid the interpreter dispatch overhead, their mid-level tier [41], [42] performs some lightweight analyses and optimizations while using fast algorithms like a greedy register allocator. Our approach in TPDE can be considered as a hybrid of the baseline and mid-level tier: we do single-pass code generation and do not perform changes on the IR, but still run a fast liveness and loop analysis to increase the quality of the generated code. Domain-specific compilers targeting, for example, WebAssembly [43] can achieve lower latency by designing the input format in a way that is easy to compile and making use of this structure in the baseline compiler.

QBE [44] is an optimizing compiler back-end for ahead-of-time compilers with a focus on simplicity of the compiler. The IR supports a narrow set of types and permits a mix of SSA and non-SSA values; for code generation, it performs target-specific legalization on the IR and emits textual assembly code. TPDE instead focuses on compile time and does not perform optimizations, requires an IR in SSA form, and imposes no further restrictions on data types. By emitting machine code in binary form, TPDE avoids the cost of producing textual output, making it suitable for JIT compilation as well.

IX. Conclusion

We presented TPDE, a two-pass compiler back-end framework for fast machine code generation that adapts to existing IRs in SSA form. Instruction semantics can be written in higher-level languages like C, from which the framework derives machine instruction sequences using LLVM's Machine IR to enable more local optimizations. With TPDE, we built a three-pass compiler for typical subset of LLVM-IR targeting x86-64 and AArch64, which compiles 8–26x faster than LLVM `-O0` with similar run-time performance. In JIT compilation contexts, adapting to existing IRs avoided costly IR translations, allowing to build an Umbra query compilation back-end on-par with the existing DirectEmit back-end while having a significantly lower implementation complexity.

Data Availability Statement

The evaluation artifact is available on Zenodo [45].

APPENDIX

A. Abstract

The artifact[45] contains container images as well as the files used to create them for x86-64 and AArch64. These images contain all files needed to repeat the measurements done in the paper, except SPEC (proprietary). As optional replacement for SPEC, the images also contain the performance benchmarks from the LLVM test-suite [46]. The artifact contains a Makefile that will run the benchmarks (with or without SPEC) and produces the raw results as well as a PDF containing the main figures from this paper (Fig. 2, 4, 5, 6).

B. Artifact check-list (Meta Information)

- **Program:** TPDE, commit `29bcf184`[3]; LLVM, version 20.1.8 and 18.1.8, Umbra, closed-source, binaries-included; LLVM Copy-and-Patch implementation, closed-source, binaries included; Wasmtime, commit `362b0dbf`[4].
- **Binary:** Binaries for Umbra (x86-64, AArch64) and the Copy-and-Patch compiler (x86-64) are included.
- **Data set:** SPEC CPU 2017 intspeed with reference workload (not included); LLVM test-suite[5] MultiSource benchmarks; Sightglass[6] default benchmarks and PolyBench; TPC-DS sf=1 (not included, data generation tool will be downloaded during the benchmark run).
- **Run-time environment:** Requires a reasonably modern Linux with Docker/Podman. For x86-64 Copy-and-Patch SPEC benchmark, ASLR is problematic and should be disabled. Disabling SELinux is recommended (root access needed). x86-64 tested on Ubuntu 24.04 and Fedora 42. AArch64 tested on Fedora 40 Asahi Remix.
- **Hardware:** 16 GiB memory (32 GiB recommended), 100 GiB storage, at least 4 cores, x86-64 needs SSE4.1, AArch64 needs Armv8.1-A.
- **Run-time state:** The system should have no other load during performance measurement; small deviations are always to be expected.
- **Execution:** For CPUs with heterogeneous core types, pin benchmarks to performance cores. Without SPEC the benchmark will take 2–4 hours, with SPEC approximately 2 days.
- **Metrics:** Compile/run-time for the different benchmarks set as well as code size for SPEC/LLVM test-suite benchmarks.
- **Output:** CSVs of data for plots from the paper; plots output as PDFs.
- **Experiments:** Makefile to run benchmarks included. A more detailed README is also provided.
- **How much disk space required (approximately)?:** 100 GiB.
- **How much time is needed to prepare workflow (approximately)?:** Time to download roughly 6 GiB of data plus ∼10 minutes for container image loading and SPEC installation.
- **How much time is needed to complete experiments (approximately)?:** 2-4 hours without SPEC, approximately 2 days with SPEC.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Umbra, the Copy-and-Patch implementation and SPEC are proprietary. Everything else is open-source, including TPDE.
- **Archived (provide DOI)?:** 10.5281/zenodo.17867601

[3]https://github.com/tpde2/tpde/commit/29bcf184
[4]https://github.com/bytecodealliance/wasmtime/commit/362b0dbf
[5]https://github.com/llvm/llvm-test-suite
[6]https://github.com/bytecodealliance/sightglass

C. Description

1) How delivered: The artifact is archived on Zenodo and available online at https://doi.org/10.5281/zenodo.17867601.

2) Hardware dependencies: 100 GiB storage, at least 16 GiB RAM (32 GiB recommended), at least 4 cores. On x86-64, the CPU must support at least SSE4.1; on AArch64, the CPU must support at least Armv8.1-A and Advanced SIMD extensions. An internet connection is needed.

3) Software dependencies: Tested with Linux 6.8.0/6.11, glibc 2.41/2.39 and Docker 28.5.1/Podman 4.9.3. Newer versions should work. The benchmark script needs GNU make. Additional software dependencies are bundled in the artifact's container images.

D. Installation

Download and extract the `artifact.tar.gz`, enter the directory and then extract the `tpde-cgo-bench-x86_64.tar.gz` (or `-aarch64`) into the `images` subfolder. No further installation is needed.

E. Experiment workflow

Step 1: We recommend disabling SELinux as it may interfere with the containers accessing the SPEC installation directory or running benchmark binaries (`sudo setenforce 0`). On x86-64 with SPEC benchmarks, disable ASLR (`echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`) (needed for the Copy-and-Patch compiler).

Step 2: `SPEC_INSTALL_DIR=</path/to/spec> make prepare` — Creates the container and installs SPEC. To not run the SPEC benchmarks, elide the `SPEC_INSTALL_DIR` environment variable. If you are on a machine with heterogeneous CPUs, select the performance cores using the `DOCKER_CPUSET` environment variable and give it the same value as an invocation of the `taskset` command, e.g. `"0-3"`. You can select Podman by setting the `DOCKER_TOOL` environment variable to `podman`.

Step 3: `make all` for all benchmarks excluding SPEC or, alternatively, `make all-spec` for all benchmarks including SPEC

This will create a new folder `latex.<container-id>` in the current folder which will contain the raw/parsed results in `bench-res` and a PDF with all figures in `figures_<arch>.pdf/figures_<arch>_spec.pdf`.

F. Evaluation and Expected Results

For all benchmarks, the compile- and run-time speedups and the code size increases should be similar to the ones reported in the paper. For the LLVM test-suite, the speedups should be similar or a bit higher than the SPEC speedups reported in the paper. For Umbra, due to the short execution times, there might be some deviation in the results, even with after a warm-up run and multiple repetitions. The results should nonetheless be in the same order of magnitude. The run-time of some back-ends may also be abnormally high. In that case, the Umbra benchmark can be repeated using `make repeat-umbra`.

G. Experiment Customization

The artifact contains container images which include all intermediate build sources and which can be used to rerun the benchmarks with edited source code or on other benchmark sets in a rudimentary manner. The README in the artifact provides a detailed documentation. Documentation for TPDE and our LLVM back-end is available online (https://docs.tpde.org) and can be used to compile user-generated LLVM-IR or implement back-ends for custom IRs.

References

[1] V8 Project, "TurboFan," https://v8.dev/docs/turbofan, accessed 2025-03-20, [n. d.].

[2] F. Pizlo, https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/, accessed 2023-05-14, Feb. 2016.

[3] Bytecode Alliance, "Wasmtime," https://wasmtime.dev/, accessed 2023-05-19, 2023.

[4] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "eWASM: Practical software fault isolation for reliable embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3492–3505, 2020.

[5] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot server compiler," in *Symposium on Java Virtual Machine Research and Technology Symposium (JVM)*, 2001, pp. 1–12.

[6] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[7] T. Gubner and P. A. Boncz, "Charting the design space of query execution using VOILA," *Proc. VLDB Endow.*, vol. 14, no. 6, pp. 1067–1079, 2021.

[8] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1243–1254.

[9] H. Funke, J. Mühlig, and J. Teubner, "Efficient generation of machine code for query compilers," in *DaMoN*. ACM, 2020, pp. 6:1–6:7.

[10] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[12] A. Engelke and T. Schwarz, "Compile-time analysis of compiler frameworks for query compilation," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 233–244.

[13] T. Kersten, V. Leis, and T. Neumann, "Tidy tuples and flying start: fast compilation and fast execution of relational queries in Umbra," *The VLDB Journal*, vol. 30, pp. 883–905, 2021.

[14] F. Gruber, M. Bandle, A. Engelke, T. Neumann, and J. Giceva, "Bringing compiling databases to RISC architectures," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1222–1234, 2023.

[15] D. Melnik, "Speeding up query execution in PostgreSQL using LLVM JIT compiler," https://llvm.org/devmtg/2016-09/slides/Melnik-PostgreSQLLLVM.pdf, accessed 2024-11-10, Sep. 2016.

[16] Clang Team, "Clang-repl," https://releases.llvm.org/20.1.0/tools/clang/docs/ClangRepl.html, accessed 2025-03-20, Sep. 2016.

[17] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," 2012. [Online]. Available: https://arxiv.org/abs/1209.5145

[18] Bytecode Alliance, "Cranelift," https://cranelift.dev/, accessed 2023-05-19, 2023.

[19] ——, "Cranelift IR Reference," https://github.com/bytecodealliance/wasmtime/blob/6a8d3d5a9ad32aa63cc39b657ec7352882dd5d70/cranelift/docs/ir.md, accessed 2025-03-25, 2025.

[20] T. Neumann and M. Freitag, "Umbra: A disk-based system with in-memory performance." in *CIDR*, 2020.

[21] H. Xu and F. Kjolstad, "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[22] F. Drescher and A. Engelke, "Fast template-based code generation for MLIR," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, 2024, pp. 1–12.

[23] Bytecode Alliance, "Cranelift compared to LLVM," https://github.com/bytecodealliance/wasmtime/blob/28931a4/cranelift/docs/compare-llvm.md, accessed 2023-05-19, Mar. 2023.

[24] WebKit Developers, "Webkit: Bare bones backend," https://webkit.org/docs/b3/, accessed 2024-11-07, [n. d.].

[25] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," vol. 13, no. 4, pp. 451–490, 1991.

[27] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, 1973, pp. 194–206.

[28] B. Boissinot, S. Hack, D. Grund, B. Dupont de Dine hin, and F. Rastello, "Fast liveness checking for SSA-form programs," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization*, 2008, pp. 35–44.

[29] F. Rastello, "On sparse intermediate representations: Some structural properties and applications to just-in-time compilation," Habilitation thesis, Inria Grenoble Rhône-Alpes, 2012.

[30] A. Kohn, V. Leis, and T. Neumann, "Adaptive execution of compiled queries," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 197–208.

[31] T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *Proceedings of the 14th International Conference on Static Analysis*, ser. SAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 170–183.

[32] A. Engelke, "Faster compilation in LLVM 20 and beyond," https://llvm.org/devmtg/2025-04/slides/technical_talk/engelke_faster.pdf, accessed 2025-09-01, 2025, EuroLLVM 2025.

[33] Bytecode Alliance, "Sightglass – a benchmarking suite and tooling for Wasmtime and Cranelift," https://github.com/bytecodealliance/sightglass, accessed 2025-03-24, 2025.

[34] Y. Pouchet, Bondugula, "PolyBench," https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, accessed 2025-03-24, 2016.

[35] TPC, "TPC-DS decision support benchmark," https://www.tpc.org/tpcds/, accessed 2023-05-14.

[36] M. A. Ertl and D. Gregg, "Retargeting JIT compilers by using C-compiler generated executable code," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004, pp. 41–50.

[37] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, Java," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[38] B. Bucher and S. Ostrowski, "PEP 744 — JIT compilation," https://peps.python.org/pep-0744/, accessed 2024-11-10, Apr. 2024.

[39] AsmJit Authors, "AsmJit," https://asmjit.com/, accessed 2024-11-12, 2024.

[40] L. Swirski, "Sparkplug – a non-optimizing JavaScript compiler," https://v8.dev/blog/sparkplug, accessed 2025-03-20, May 2021.

[41] T. Verwaest, L. Swirski, V. Gomes, O. Flückiger, D. Mercadier, and C. Bruni, "Maglev – V8's fastest optimizing JIT," https://v8.dev/blog/maglev, accessed 2025-03-20, Dec. 2023.

[42] WebKit Developers, "WebKit Wiki: JavaScriptCore," https://trac.webkit.org/wiki/JavaScriptCore, accessed 2025-02-08, [n. d.].

[43] B. L. Titzer, "Whose baseline compiler is it anyway?" in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 207–220.

[44] Q. Carbonneaux, "QBE – compiler backend," https://c9x.me/compile/, accessed 2025-09-01.

[45] T. Schwarz, T. Kamm, and A. Engelke, "Artifact for "TPDE: A Fast Adaptable Compiler Back-End Framework"," Dec. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17867601

[46] The LLVM project, "LLVM test-suite," https://github.com/llvm/llvm-test-suite, accessed 2025-11-17, 2025.