Compile-Time Analysis of Compiler Frameworks for Query Compilation

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

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

***Abstract*—Low compilation times are highly important in contexts of Just-in-time compilation. This not only applies to language runtimes for Java, WebAssembly, or JavaScript, but is also crucial for database systems that employ query compilation as the primary measure for achieving high throughput in combination with low query execution time.**

We present a performance comparison and detailed analysis of the compile times of the JIT compilation back-ends provided by GCC, LLVM, Cranelift, and a single-pass compiler in the context of database queries. Our results show that LLVM achieves the highest execution performance, but can compile substantially faster when tuning for low compilation time. Cranelift achieves a similar run-time performance to unoptimized LLVM, but compiles just 20–35% faster and is outperformed by the single-pass compiler, which compiles code 16x faster than Cranelift at similar execution performance.

***Index Terms*—Fast compilation, LLVM, JIT compilation, Query Compilation**

I. Introduction

Fast compilation is not only important for improving developer productivity through fast builds, but is also highly relevant for achieving a low start-up latency in just-in-time (JIT) compilation settings. This allows fast overall execution of short programs. Typically, a fast-compilation tier is responsible only for keeping the start-up times short and is paired with an optimizing compilation tier, which takes more time to generate code, but achieves higher execution performance.

Consequently, many JIT-compilation engines feature a multi-tiered compilation approach and are particularly widely used for efficient JavaScript execution. For example, V8 [1], [2] only pairs a bytecode interpreter with an optimizing compiler, whereas WebKit's JavaScriptCore [3] implements three JIT-compilation tiers on the top of their interpreter. Similarly, dedicated WebAssembly runtimes like Wasmtime [4] often feature such approaches.

Another user of JIT-compilation techniques are query compilation engines of database systems. Compiling database queries to native machine code is a key technique for efficient processing of large data sets and allows for substantial performance improvements over interpreted processing. As a consequence, several database systems implement some sort of query compilation [5]–[10].

As it is not uncommon that queries are unknown before their execution, fast compilation is important for a low response latency. This is particularly relevant for applications like interactive data exploration tools, which often generate their queries in response to user interaction, so fast overall execution times even on large data sets are important for application responsiveness. To add to this, such queries might be more complex and tools might execute multiple queries in a row, increasing the importance of fast execution of queries not known ahead of time [11].

Umbra [9] is a compiling database with strong emphasis on fast execution, targeting x86-64 and AArch64. It supports a wide range of back-ends for the actual compilation of queries to machine code, with different properties regarding compile time and execution performance, in particular, supporting an ordinary C compiler, LLVM [12], and Cranelift [13]. Moreover, Umbra also implements a custom, direct code generator to x86-64 machine code [14] and an interpreter.

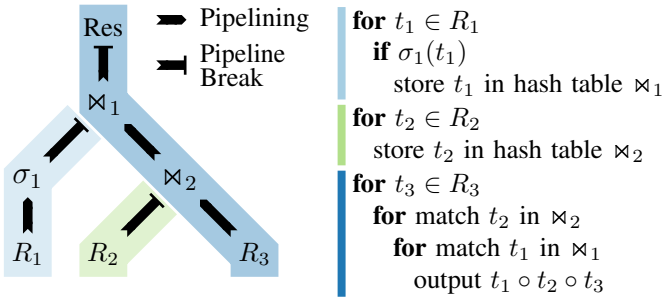
In this paper, we report our experiences with these different back-ends and provide a detailed and thorough comparison and analysis of the compile times of these back-ends when compiling database queries. Our main contributions are:

- A performance comparison and analysis of all back-ends supported by Umbra on both x86-64 and AArch64.
- A thorough and in-depth analysis of the compile-time performance of LLVM and Cranelift.
- An analysis and outline of possible improvements regarding compile-times for all compilation approaches.
- Our steps to adjust our LLVM-IR code to achieve faster compilation times.

The remainder of this paper is structured as follows: Sec. II gives a brief overview on query compilation and Sec. III describes Umbra in more detail, particularly focusing on the structure of the generated code and Umbra's internal code representation. The following sections then describe and analyze each of the back-ends, with GCC/C in Sec. IV, LLVM in Sec. V, Cranelift in Sec. VI, and Umbra's single-pass approach in Sec. VII. Afterwards, Sec. VIII compares the back-ends regarding compile-time and run-time performance, and we discuss the results in Sec. IX. Sec. X covers related work and in Sec. XI we summarize our findings.

Benchmark System Specification

Unless noted otherwise, all specified performance numbers refer to the sum of compiling all TPC-DS [15] queries, resulting in the compilation of 6678 functions. We ran 20



(a) Query plan. (b) Pseudo-code.

Fig. 1. Simple query plan with three pipelines.

executions of all measurements and report the arithmetic mean. Our x86-64 machine is an Intel Xeon Gold 6338, 32 cores, 2.0 GHz, 256 GiB memory, running Linux 5.19.0; all AArch64 measurements were done on an Apple M1 using 4 performance cores with 16 GiB memory running Asahi Linux 6.2.0.

II. Background: Query Compilation

Modern database systems compile queries to native machine code to achieve high throughput by utilizing available compute resources effectively [7], [8], [16], [17]. To compile a query, it is initially parsed, analyzed, and transformed into a *query plan*, which is often a tree or DAG. The plan is then optimized and modified to be more suitable for efficient execution. This also involves selecting actual implementations for high-level operators, e.g. choosing a hash join for a generic join operator.

Once the final query plan is reached, the plan is separated into linear *pipelines* [5]. The operator at the bottom of a pipeline reads the data from a data source (e.g., a relation in storage). The data is then passed through the operators of the pipeline, which can arbitrarily transform the tuples. At the end of the pipeline, the tuples are materialized into a proper representation in memory, e.g., the output buffer, a temporary buffer, or an index data structure. Most notably, there is no materialization between the operators of the pipeline, data is kept in registers or local buffers only. Pipelines can also have dependencies on the results of other pipelines. For example, in Fig. 1a the join operator $\bowtie_2$ in the pipeline starting from R_3 needs the materialized result of the pipeline from R_2. This data-centric approach allows for a fairly straight-forward generation of imperative code. The base table scan iterates over the tuples of the base relation and loads the values into local variables. In the loop body, ancestor operators are applied in a nested manner, and at the innermost nesting level, the modified tuple is materialized again. Fig. 1b shows an example.

To allow for utilizing multiple cores, scans at the bottom of a pipeline do not always iterate over all tuples, but only over a configurable subset [18] ("morsel-driven parallelism"). This way, multiple threads can process different parts of the base table in parallel. When multiple threads write to the same data structure, atomic operations or runtime calls are used.

III. Umbra System Architecture

Umbra [9] is a relational database system that makes heavy use of dynamic code generation for executing queries. To improve utilization of CPU cores, Umbra not only employs morsel-driven parallelism, but also compiles each pipeline separately, enabling parallel compilation and more efficient execution scheduling. In addition to the function with the main logic, compiling a pipeline also involves some other small functions, e.g. for single-threaded preparatory work or cleanup. Umbra is written in C++ and mainly targets x86-64, but AArch64 is supported as well.

A. Structure of Generated Code

As a consequence of the rather fine-grained separation into multiple functions, a portion of the functions are rather small and only have simple control flow. However, for some query-dependent parts, functions can not only grow very long with variables having long live-intervals (e.g., for a query with many selection predicates), but also contain arbitrarily deeply nested loops (e.g., with many table joins, one loop nest per join).

For many non-trivial and query-independent operations, the generated code includes calls into runtime functions. This includes memory management, input/output and non-trivial synchronization operations. In some cases, runtime functions call back into generated code, for example, for the comparison function of a sort operation. As Umbra uses C++ exceptions for error handling, all functions with runtime calls must have associated DWARF [19] unwinding information registered with the C++ runtime.

For representing SQL decimals, Umbra uses 128-bit integers, and therefore the generated code can make heavy use of 128-bit arithmetic. Furthermore, all arithmetic operations on user data include checks for overflow conditions to avoid producing incorrect results.

Strings are implemented as 16-byte data structures with small buffer optimization: the first four bytes indicate the length. If the length is less than or equal to 12, the remainder of the struct contains the entire string. Otherwise, bytes 8–15 contain a pointer to the string and additionally bytes 4–7 contain the prefix of the string, allowing quick access to the first four characters. This string structure is passed very frequently by-value to and from runtime functions.

Umbra's hash function uses CRC-32 if the target hardware has native support, and otherwise falls back to using $64 \times 64 \rightarrow 128$-bit multiplication, where the lower and upper half of the 128-bit result are immediately combined using XOR into a new 64-bit value (referred to as *long-mul-fold*). As hash-joins are extremely common, these operations occur very frequently in hot parts of the compiled query.

B. Umbra IR

Code for a function is initially generated in *Umbra IR* [9], [14], a custom SSA-based intermediate representation optimized for fast generation and linear traversal. After its generation, the IR is handed over to an execution back-end, which interprets or compiles the code as suitable, permitting a

Listing 1. Excerpt of an Umbra IR function.

```
define int32 @filter(int8* %s, int32 %count,
                     int32* %vec) noexcept [] {
2:
  %0 = isnull i32 %count
  condbr %0 %3 %4
3:
  return 0
4:
  %5 = load int32 %vec
  %6 = add i32 %5, %count
  %7 = sub i32 %6, 1
  %8 = sub i32 %count, 1
  ; ...
}
```

path with very fast code generation as well as an optimizing path through a more advanced back-end like LLVM.

Structurally, Umbra IR is inspired by LLVM-IR and has several conceptual similarities, Listing 1 shows an example: functions are structured in basic blocks, which begin with Φ-nodes to merge values from multiple predecessors. Instructions inside basic blocks always have a single result value and the IR is strictly typed. The type system, however, is much more simple: Umbra IR has no complex or composite data types and only supports integer (8/16/32/64/128 bits), boolean, double-precision floating-point, pointer, and the `data128` (e.g., for strings) types. Also, constants are separate instructions to simplify the internal representation.

One key motivation for a custom IR is the ability to also express complex operations as a single instruction, increasing the expressiveness and brevity of Umbra IR programs. A prime example are arithmetic instructions with overflow detection, which are provided in three variants: (a) trapping instructions, where an overflow condition results in a call to a trap function (which is implicit control flow in the IR) — Listing 2 shows an example; (b) check-and-branch instructions, where the IR instruction terminates the basic block and specifies two successors, one for the success and one for the overflow case; and (c) check instructions, which indicate an overflow condition as a separate IR value. As Umbra IR instructions only have one result operand, the overflow bit is produced by a separate instruction, which immediately follows after the arithmetic instruction and is intended to be fused by the back-end. Other non-standard operations include byte-swap, CRC-32, *long-mul-fold*, and arithmetic operations that directly operate on memory.

Technically, Umbra IR is designed and heavily optimized for efficient in-memory storage and linear forward traversal. Instructions are variable-length, stored in a single buffer, and reference each other using offsets into the buffer. To achieve low-latency code generation, Umbra IR does not maintain user lists and the complexity of inserting or removing instructions in the middle of a block is $\mathcal{O}(n)$, as subsequent instructions need to be rewritten. The only implemented transformation on Umbra IR — dead code elimination — therefore first explicitly computes the user count of each instruction and then rewrites

Listing 2. Umbra IR code using overflow instructions and special operations as part of a hash function. Value numbers simplified for brevity.

```
4:
  %5 = getelementptr int8 %state, i64 32
  ; ssubtrap may call throwOverflow()
  %6 = ssubtrap i32 %3, 53
  %7 = zext i64 %6
  %8 = crc32 i64 0xf45f017ffbcd4390, %7
  %9 = crc32 i64 0xb9935cc97ab5b272, %7
  %10 = rotr i64 %6, 32
```

the entire function, omitting (transitively) unused instructions. A more detailed description of Umbra IR can be found in [14].

C. Execution Back-ends

Umbra IR has several execution back-ends with different characteristics in compile-time and execution performance. By default, the back-end is selected adaptively, starting each compilation with the low-latency DirectEmit back-end. Then, after a function is executed a few times, the adaptive back-end roughly estimates the compilation time and benefit through a simple heuristic on the code size and, if it deems optimization beneficial, starts a compilation with the LLVM-optimized back-end. Once compiled, subsequent executions will use the optimized function. Advanced mechanisms for switching functions are not necessary, as morsel-driven parallelism ensures that the function is called for sufficiently small workloads.

IV. GCC/C Back-end

Umbra's GCC/C back-end provides a very high execution performance, but also the longest compilation times.

The back-end first transforms Umbra IR into C code. This is mostly a straight-forward process; conditional branches become `goto` statements and every SSA-variable becomes a normal variable. Φ-nodes follow the usual SSA destruction strategy, resolving chains or cyclic dependencies between Φ-nodes. The C code is written to a temporary file and an external compiler (typically GCC) is called to compile it into a shared library. The shared library is opened using `dlopen` and the compiled functions are accessed with `dlsym`.

While this approach is conceptually similar to libgccjit [20], which also compiles code into shared libraries, libgccjit directly generates GCC's IR (GIMPLE), avoiding the parsing overhead. However, due to restrictive licensing, using libgccjit in Umbra is not possible.

A. Optimizations

1) Run-time: The C compiler is invoked with the options `-O3 -march=native` to aggressively optimize code for the current system and its supported CPU features.

2) Compile-time: Due to the large overhead of generating and parsing C code, calling an external process, which in turn invokes the assembler and linker, and the I/O for the intermediate files, we did not further optimize compile times.

TABLE I
COMPILE-TIME BREAKDOWN OF THE C BACK-END. DATA FROM `-ftime-report` FOR THE INDIVIDUAL PHASES IS SHOWN SEPARATELY DUE TO HIGH OVERHEAD.

Phase	`-ftime-report`	Time
Umbra: C code generation		0.14s
GCC: `cc1` (compiler)	52.00s	42.13s
Phase setup	0.26s	
Phase parsing	6.45s	
Preprocessing	2.03s	
Lexical analysis	2.09s	
Parsing	2.31s	
Phase opt and generate	44.92s	
GCC: `as` (assembler)		0.13s
GCC: `collect2` (linker)		0.75s
Load shared library		0.01s
Total		46.34s

B. Compile-time Analysis

GCC provides two mechanisms for analyzing compile times: the `-time` option, which has nearly no overhead and just prints the time of the sub-processes (compiler proper, assembler, linker), and `-ftime-report`, which provides more details about the individual compiler phases, but has an overhead of 24%. All reported times are wall-clock time, and Table I shows a summary of the results using GCC 12.2.0.

Two inherent problems become immediately visible: first, Umbra needs to generate C code, which GCC has to parse again. This part alone takes around 13% of the overall compile time. Second, calling GCC results in a separate invocation of the assembler and linker, which also take a measurable amount of time for starting up, parsing their input files, and generating their output.

Nonetheless, most of the time is spent for optimizing and generating machine code. While we did not spend time to actively improve this, the possibilities to do so are limited by the available command line options and boil down to selecting different optimization passes.

C. Discussion & Possible Improvements

Due to its internal architecture, using GCC as part of a query compilation engine requires generating and parsing textual source code, which does not allow for a compile-time-optimized implementation. Invoking external commands for assembling and linking adds further costs, which, however, cannot be avoided due to GCC's internal architecture. Although improvements on this architecture are certainly possible, such changes are unlikely to make the GCC/C back-end competitive to other compilation strategies.

V. LLVM BACK-END

Umbra's LLVM back-end supports two operating modes: an *optimized* compilation mode intended for producing efficient code and a *cheap* compilation mode with focus on low compile times. These different modes, however, only affect whether optimizations are applied and the configuration of the LLVM back-end optimizations.

The LLVM back-end transforms the Umbra IR code into LLVM-IR, which is mostly a straight-forward process. Some special instructions are lowered to intrinsics (e.g., overflow arithmetic) or more complex instruction sequences (e.g., long-mul-fold). When explicitly requested, the back-end also attaches debug information to the LLVM-IR code.

Afterwards, the code is handed over to LLVM's ORC JIT-compilation infrastructure. For *optimized* compilations, we configure the back-end to optimization level `-O2` and use the SelectionDAG instruction selector; for *cheap* compilations we use `-O0` and FastISel.

ORC then takes care of producing an in-memory ELF object file for the module and uses JITLink for linking the object file into the current address space. Runtime functions are referenced via external symbols. We use the Small-PIC code model, causing JITLink to generate one PLT+GOT for each compiled module. Additionally, we also use an existing plug-in to register the unwind information.

A. Optimizations

1) Run-time: In the *optimized* mode, the back-end applies a few optimization passes that have been shown to be beneficial. This includes common-subexpression elimination, CFG simplification, instruction combination, loop-invariant code motion, and dead code elimination. For *cheap* builds, no further transformations are applied.

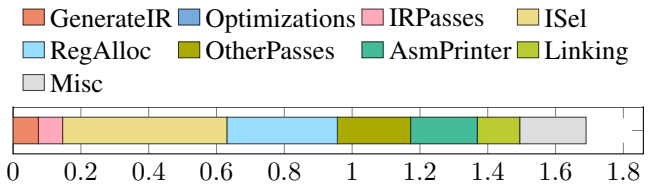
Additionally, for both modes, we provide a custom implementation of 128-bit multiplications with overflow detection, with a run-time check optimizing for the case that both operands can be represented as 64-bit integers. If a full multiplication is required, we do not use the LLVM intrinsic but create a call to our hand-optimized 128-bit multiplication.

2) Compile-time: We implemented several measures to reduce the compilation time for *cheap* builds using FastISel, which improved compilation performance by more than 50% compared to a previous implementation.

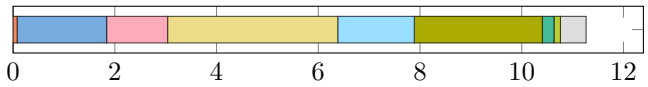
First, instead of the default large code model for JIT compilation, we use the Small-PIC code model. This not only results in smaller code, but more importantly, FastISel only supports the small code model and previously fell back to SelectionDAG for every function call. The disadvantage of the Small-PIC model is that runtime calls now require two jumps in machine code (near jump to PLT, indirect jump from there). In practice, however, we could not measure any run-time performance differences.

Second, instead of representing Umbra IR's `data128` as an LLVM struct `{i64,i64}`, we always represent it as two separate `i64` values in LLVM-IR. The only exception are function return values, as structures are the only way to represent functions with multiple return values. This change has several implications:

1) The LLVM-IR code becomes shorter as no instructions for extracting/inserting elements need to be generated.



(a) Compile-time [s] of LLVM-cheap on x86-64.



(b) Compile-time [s] of LLVM-optimized on x86-64.

Fig. 2. Compile-time breakdown of LLVM on x86-64. IRPasses are back-end passes on LLVM-IR, OtherPasses are smaller back-end passes on Machine IR, "Misc" is pass manager and measurement overhead and object file generation.

2) As the flow of the actual values inside the struct becomes obvious, later analyses need to spend less time for handling opaque structures.
3) Avoiding structures leads to substantially fewer fallbacks from FastISel to SelectionDAG: as FastISel only handles LLVM-IR values that fit in a single machine register, every occurrence of this struct type would trigger a fallback for the remainder of the current basic block.
4) This change also improves compile times in the *optimized* build mode by $\sim 7\%$.

Note that Umbra IR's `int128` is still represented as native LLVM `i128`.

Third, instead of parsing and constructing the architecture description (`TargetMachine`) for every compilation, we cache it across multiple compilations. However, as Umbra can do multiple compilations in parallel and LLVM modifies some parts of the `TargetMachine` throughout the compilation (e.g., to store function-level option overrides in a central place), we need to construct and cache one `TargetMachine` instance per thread.

Fourth, we implemented several minor performance improvements in LLVM, including FastISel support for the CRC-32 intrinsic on x86 and AArch64, FastISel support for functions with multiple return values on AArch64, and more efficient writing of encoded machine code instructions on x86. All changes were merged upstream and resulted in 4% faster compile times.

B. Compile-time Analysis

We evaluated the performance of the LLVM back-end with a recent LLVM development snapshot (commit `84a6a05`). To get more insights, we used LLVM's time tracing infrastructure to measure the execution time of the individual passes. We note that this causes causes 1.27M (*optimized*)/467k (*cheap*) time measurements, which adds an overhead of up to 2%. We supplement the measured data with profiling data obtained using `perf`. Figure 2 shows the results.

1) Constructing/destructing LLVM-IR: During LLVM-IR construction, most time is spent allocating and constructing the LLVM objects that correspond to the Umbra IR functions, basic blocks, and instructions. However, we also observed that *destructing* the LLVM module is fairly expensive, taking around 1% of the compilation time in the *cheap* mode.

2) LLVM-IR Back-end Passes: At the start of the code generation pipeline, several passes that operate on the LLVM-IR are applied first, for example, expansion of divisions larger than 128 bits, lowering of "constant intrinsics", expansion of certain vector intrinsics, or lowering of x86 AMX types. Many of these passes have two things in common: first, each of them iterates over all instructions to check whether an instruction matches a specific pattern. While this itself is fairly cheap, doing this several times in a row adds up. And second, Umbra *never* generates many of the handled constructs, yet these passes are always run and there is no option to disable them for faster compilation.

Running such passes only when a function contains the constructs in question or merging these passes to iterate over the IR once only could reduce this avoidable overhead.

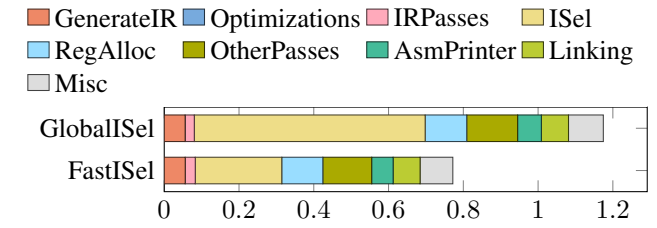
Optimized builds run several additional passes for target-specific information, which also involves computing some analyses. In particular, this includes double computation of the dominator tree and loop information, which is used, e.g., to hoist constants and certain other operations out of loops.

3) Instruction Selection: One expensive part of the LLVM back-end is the instruction selection phase, which transforms LLVM-IR into the lower-level Machine IR (MIR). In this representation, operations correspond to instructions of the target architecture, but the program is still in SSA-form and registers are generally unallocated. In addition to selecting machine instructions, this phase also performs legalization of IR operations that are not directly supported by hardware (e.g., 128-bit arithmetic).

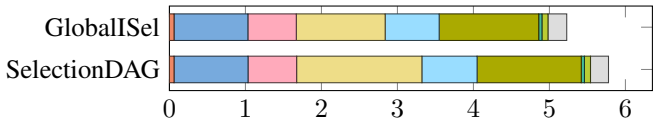
a) SelectionDAG: The most capable and often default instruction selector is SelectionDAG, which operates on one basic block at a time. For each basic block, SelectionDAG converts the LLVM-IR into a DAG consisting of generic operations. After a series of legalizations and combinations of operation nodes, the actual instruction selection is performed, replacing generic operation nodes with actual machine operations. Finally, the DAG is linearized into MIR either in topological order or according to an instruction scheduling model.

While SelectionDAG is highly capable, it also faces substantial performance problems [21], taking around 30% of the compilation time in the *optimized* build mode. While one fifth of this time is spent in constructing and linearizing the graph-based IR, nearly half of the time is spent combining and legalizing DAG nodes. Particularly for arithmetic operations, a substantial part is determining whether any bits of the operation are known, which is implemented as recursive traversal.

b) FastISel: For unoptimized builds, LLVM provides a fast instruction selector that expands IR instructions into corresponding sequences of machine instructions. For simplicity, FastISel only supports common data types that fit in a single machine register and does not implement 128-bit integers or structures. Furthermore, only a frequently-used subset of LLVM-IR operations and intrinsics are supported. Whenever



(a) Compile-time [s] of LLVM-cheap on AArch64.



(b) Compile-time [s] of LLVM-optimized on AArch64.

Fig. 3. Compile-time breakdown of LLVM on AArch64 (Apple M1, performance cores only), comparing FastISel, SelectionDAG, and GlobalISel.

an unsupported LLVM-IR instruction is encountered, FastISel stops and falls back to SelectionDAG for the remainder of the block. However, for function calls that use unsupported data types or pass parameters on the stack as well as unimplemented intrinsics, only the affected call instruction is handled by SelectionDAG.

For fast compile times, reducing the amount of SelectionDAG fallbacks is critical. On the 103 TPC-DS benchmark queries on x86-64, we observed 3876 fallbacks after our optimizations (cf. Sec. V-A2), accounting for 36% of the instruction selection time. The two main reasons for fallbacks are unsupported intrinsics or function calls (2486) and 128-bit data types (1328). The remaining cases include atomic operations (35) and operations on booleans (`i1`) (27). We note that we carefully adjusted our LLVM-IR code to avoid SelectionDAG fallbacks where easily possible.

c) GlobalISel: To address the shortcomings of SelectionDAG and FastISel, the LLVM project started developing a long-term replacement called GlobalISel, which provides a fast and optimized compilation mode in a single infrastructure. Currently, GlobalISel is only supported on a few architectures, including AArch64, where it already is the default back-end.

The first stage of GlobalISel is the translation of LLVM-IR into a generic Machine IR (gMIR), i.e. MIR with generic, target-independent operations. In this representation, generic operations that are unsupported by the target are legalized into supported generic operations. Then, a register bank is selected for each value and finally the actual instruction selection is performed. Similar to SelectionDAG, between these stages, combiners are run to generate more efficient code; the extent depends on the optimization level. [22] provides more details on the implementation of a GlobalISel back-end.

To compare GlobalISel in its cheap and optimized compilation setting with FastISel/SelectionDAG, we ran the TPC-DS benchmarks on AArch64. Figure 3 shows the results. For *optimized* compilation, GlobalISel is 1.4x faster than SelectionDAG, reducing the overall compile time by 10%. Here, avoiding an entirely different graph-based IR shows its benefits. However, for *cheap* compilation, GlobalISel is 2.7x slower than FastISel, increasing the total compilation time by 52%. The main cause is the multi-pass approach, where each pass iterates over and modifies the entire IR. The initial translation from LLVM-IR to gMIR itself is just slightly faster than FastISel in its entirety, and the other passes (legalizer, combiners, RegBankSelect, InstructionSelect) each take further considerable amounts of time.

4) Register Allocation: The second-most expensive part of the compilation pipeline is register allocation, which is split into multiple passes in LLVM. First, the MIR is transformed out of SSA by replacing Φ-nodes with copy instructions. Then, two-address machine operations where one source operand is also the destination are rewritten accordingly.

After these preparatory steps, the actual register allocation is done. For *cheap* builds, LLVM uses the "fast" register allocator, which linearly iterates over all basic blocks and their instructions in reverse order and greedily assigns registers. In particular, it does not require any analyses like a dominator tree or liveness intervals. For *optimized* builds, the "greedy" allocator is used, which implements a graph-coloring-based greedy allocation scheme and afterwards performs additional optimizations to improve the assignment and spill placement. This allocator also requires several analyses, including a liveness analysis of registers and stack slots, loop information, and block execution frequency prediction.

Of these passes related to register allocation, the allocation itself is the most expensive part, but also the related passes, in particular the two-address rewriting pass and the liveness analysis, take between 25–45% of the time related to register allocation.

5) Other Passes: In both compilation modes, several other passes are executed as part of the back-end; in total the *optimized* pipeline consists of 146 passes and the *cheap* pipeline of 67 passes. Many of these passes do only small transformations or changes and are individually fairly cheap.

One of the more expensive of these passes is prologue/epilogue insertion, which not only inserts the function entry and exit sequences to take care of callee-saved registers, but also finalizes the stack frame layout and rewrites all references to the stack frame. In *cheap* compilation, this transformation is comparably expensive, taking 4% of the compile time.

6) Machine Code Emission: The "assembly printer", which is responsible for encoding the MIR instructions into machine code and emit an object file for the LLVM module, takes an unexpectedly large part of the compilation time, with 12% in *cheap* mode and 2% in *optimized* mode; this particularly affects the *cheap* mode as the resulting machine code is larger due to fewer optimizations.

After emitting the function header, the emitter iterates over the instructions of all basic blocks. An MIR instruction is first lowered into an LLVM-MC instruction, where the instruction mnemonic and operands directly correspond to the instruction encoding of the target architecture. This instruction is then encoded into a memory buffer. Further hooks can be registered to follow and modify the code emission at every instruction,

basic block, and function; in our case, this is used for emitting DWARF unwind information. References to other parts of the code like basic block offsets are stored as fixups and handled at the end of the compilation; these numbers are not included in the AsmPrinter pass and account for an extra 3% of the compile time in *cheap* mode.

There are three main reasons for the large time spent in this pass. First, the emitter constructs another in-memory encoding of every instruction. Second, the infrastructure allows for abstracting the underlying target and representation (e.g., encoded instruction vs. textual assembler), registering hooks, and target-specific optimizations at various points, resulting in several virtual function calls per emitted instruction. Third, all symbols, including labels for internal basic blocks, are string-based, causing overhead of generating and hashing these strings, even though these are never externally visible.

7) Linking: LLVM's JIT-compilation approach is very similar to a standard compiler flow [23]: the LLVM module is first compiled to a complete (in-memory) ELF object file with proper symbols, string tables, and relocations. In the second step, the object file is JIT-linked into the current process. In the *cheap* mode, linking takes $\sim$7% of the total compile time.

The link step is separated into four phases [23]: the first phase ($\sim$2% of compile-time) recovers the symbols from object file, prunes unused symbols, and allocates the final memory where the object file is linked to. The second phase ($\sim$4% of compile-time) assigns addresses to all symbols and resolves external symbols. The third phase (0.4% of compile-time) applies relocations and copies the sections of the object file into their final place. The final phase ($< 0.1\%$ of compile-time) performs the actual lookup of the symbol address.

8) Miscellaneous: In addition to measurement overhead and object file generation, a few other parts of the compilation are not captured by the tracing infrastructure. This includes construction and destruction of the pass pipeline itself and destructing the LLVM-IR module.

We also observed that, in *cheap* mode, the method to add an operand to an MIR instruction itself takes 3% of the overall compile time. Further, profiling indicates that 5% of the time is spent in the legacy pass manager infrastructure itself for tracking available and required analyses. There were proposals to port the back-end to the "new" pass manager infrastructure, which would likely reduce this overhead, but these were not implemented so far.

C. Discussion & Possible Improvements

LLVM is a complex framework capable of generating highly optimized code with support for a wide range of features and architectures. To achieve this, LLVM implements a modular infrastructure and a flexible Machine IR that can represent operations of all supported targets. The back-end is split in a multitude of passes, which are combined by the target back-end as required.

Nonetheless, this generic and flexible approach comes at a substantial cost in compile time. The general approach of rewriting the code several times prevents substantial improvements in compile-time without massive architectural changes. The comparison of FastISel and GlobalISel shows that increasing the number of IR rewrites can increase flexibility and maintainability, but also substantially increases the compile time. Similarly, running a multitude of cheap passes also has a non-negligible impact; merging passes or only running passes when they are actually required by the input at hand could reduce compilation times.

Emitting an in-memory ELF object only to decode it immediately afterwards allows for a simple implementation, but could be avoided entirely by emitting the machine code directly in-place for execution (e.g., through a hypothetical `MCJITStreamer`). Furthermore, the high degree of flexibility and the large amount of optional features, often realized through hooks or plug-ins, leads to a huge amount of virtual function calls, which are comparably expensive and prevent optimizations.

In summary, while smaller improvements in compile-time in LLVM appear to be possible, reducing the compile times by an order of magnitude would likely require a fundamentally different approach that puts less emphasis on flexibility and avoids frequent iteration over and rewriting of the program.

VI. Cranelift

Cranelift [13] is a compiler back-end developed for fast compilation. Originally developed for the WebAssembly runtime Wasmtime [4], it also strives to be generally usable.

Cranelift-IR (CIR) is superficially similar to LLVM-IR [24]. However, as Cranelift aims for a simpler design, it only supports a small set of data types, in particular scalar integers with 8/16/32/64/128 bits, 32/64 bit floating points, vector types of up to 512 bits and opaque reference types that do not support any arithmetic operations. Notably CIR has no support for explicit pointer and aggregate types, these need to be emulated by the front-end using integer arithmetic. The instruction set is generally aligned with the operations required for WebAssembly and has no intrinsics. Stack slots are allocated outside of the instruction stream and explicit instructions for stack load, stores and address retrieval are available. There is also support for further WebAssembly-related concepts, but these are not relevant for our case of compiling database queries.

CIR data structures are more optimized for performance than LLVM-IRs and mostly rely on arrays and hash maps. For example, instructions are of fixed length and stored in one continuous array. However, some more expensive data structures are used to allow for easier modification and analysis of the IR such as array-backed linked lists, for example to store the instruction order.

The Cranelift back-end translates Umbra IR to CIR in two passes, first setting up function metadata before translating them. Each function is translated individually as Cranelift can only compile one function at a time. Many Umbra IR operations have an equivalent in CIR, but some operations require helper functions. This includes floating point conversions, which have

TABLE II
AVERAGE RUN-TIME PERFORMANCE INCREASE OF CRANELIFT-COMPILED CODE WITH CUSTOM INSTRUCTIONS ON X86-64.

Query	CRC32	Arithmetic Overflow	Full Multiplication
TPC-H 1	5%	41%	16%
TPC-DS 60	21%	8%	0%
TPC-DS 32	40%	16%	0%
TPC-DS 21	50%	0%	0%
TPC-H Mean	9%	1%	1%
TPC-DS Mean	19%	1%	0%

different semantics in CIR, and 128-bit multiplication with overflow. The addresses for external functions, including the helper functions, are hard-wired into the generated CIR code. The CIR is then compiled by Cranelift and directly emitted into memory and fixups/relocations are applied after all functions have been compiled.

A. Optimizations

1) Run-time: We have implemented the same optimization for 128-bit multiplications with overflow checks as the LLVM back-end. Additionally, we added multiple instructions to CIR to reduce the number of calls to helper functions. These include `crc32`, arithmetic overflow instructions, and a multiplication with a full result, as Cranelift's instruction selector cannot merge multiplications with separate low and high results. Table II shows the execution speedup of adding these operations. `crc32` has the most impact on average due to the prevalence of hash joins. The arithmetic instructions have a small influence on the average, but some queries profit substantially. The combined multiplication instruction has little impact, however: as it just saves a single multiplication, it only has a noticeable impact on queries which make heavy use of decimal multiplications. Our additions of the arithmetic overflow operations have already been merged upstream.

B. Problems

To use Cranelift as a JIT compiler, we use a wrapper provided by the Cranelift developers, which is currently not quite mature. For example, it does not properly emit GOTs/PLTs, which can cause crashes during relocation if code or data pages are allocated too far away from each other. This can happen if other parts of the program `mmap` large amounts of memory. Thus, we needed to generate CIR that does not generate any PC-relative relocations, at the cost of less efficient code. Additionally, the JIT library does not generate unwind information, so we need to generate the DWARF CFI information manually. Lastly, Cranelift is written in Rust while Umbra is written in C++. This interoperation needs additional engineering effort, as the Rust code for translating Umbra IR needs to have struct definitions provided in Rust or a C interface to read Umbra IR. This can be automated for simple structures, but more complex C++ constructs as used by Umbra need to be translated and maintained manually.

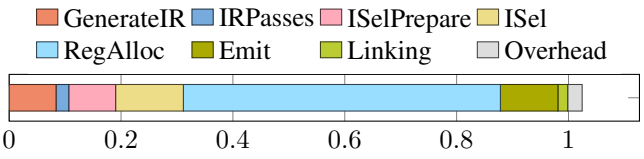


Fig. 4. Compile-time [s] breakdown of Cranelift on x86-64. IRPasses are analysis on the IR like domtree/CFG calculation, ISelPrepare are passes over the IR before the selection algorithm is run, Overhead is measurement overhead.

C. Compile-time Analysis

We evaluated Cranelift with a recent development snapshot (commit `30297dd`) and our custom instructions added. Figure 4 shows the results.

1) IR Generation: Around 8% of compile time is spent converting from Umbra IR to CIR. Significant time is attributable to hash map lookups for mapping the Umbra IR value to the corresponding CIR value and translating Umbra IR's `getelementptr` operations into integer arithmetic.

2) Instruction Selection: Cranelift spends a non-negligible time (8%) of its lowering step preparing data for the actual instruction selection, which itself is comparably efficient (12%) as the code for it is machine-generated from a domain-specific language specifying the replacement patterns. The instruction selector is a tree-matching algorithm that transforms CIR, which is conceptually a graph, into a linear array of machine instructions.

Before the tree-matching is done, however, metadata is generated in three passes over the complete IR. First, virtual registers are allocated for each IR value and are marked with a register class (integer or float) corresponding to its type. Second, the instructions are partitioned by instructions with side-effects, so memory accesses, calls and the like. This is used to make sure the instruction selector does not merge instructions across these boundaries. Third, information about the users of each instruction result is calculated using a depth-first search. This is used to distinguish whether a result has one unique use or is used by multiple instructions, where transitive uses with depth one are also counted. This is needed to prevent side-effectful instructions from being duplicated by the instruction selector[1].

3) Register Allocation: Surprisingly, the largest part of compilation is spent in the register allocator. This is in contrast to LLVM, which spends significantly less time in this phase and whose register allocator is 42% faster than Cranelift's.

Cranelift uses a modified linear-scan allocator that calculates live-ranges for virtual registers, merges non-overlapping ones into bundles and splits them if too many are live in physical registers at the same time. The allocator spends roughly 37% of its time calculating these live-ranges and merging them, iterating over the IR several times. Additionally, it maintains multiple data structures during allocation, e.g., a B-tree for every physical register to track their allocations, and spends a non-negligible amount of time traversing them. For example,

[1] see https://github.com/bytecodealliance/wasmtime/blob/7cf8121a/cranelift/codegen/src/machinst/lower.rs#L214

roughly 6% of register allocation time is spent inserting and looking up values in B-trees. Using a greedy approach instead could significantly improve register allocation performance.

4) Emitting: Cranelift emits basic blocks in the order of their lowering, with blocks marked explicitly as "cold" by the user being emitted last. This is largely straight-forward iteration over the instructions as produced by the instruction selector and writing the instruction bytes into a buffer, making use of the register assignments. However, the emitter does a pass over all instructions and their register allocations beforehand to calculate the clobbered registers for the function. This is information that the register allocator could easily provide in a small bitmap. Additionally, the emitter iterates over all moves inserted by the register allocator to aid estimation of how large a basic block could become before emitting it to decide whether veneers should be inserted between basic blocks. However, as this calculation later uses over-approximated instruction lengths (15 on x86-64), using a simpler approximation for the number of moves in a block might be beneficial.

5) Linking: Linking is relatively fast because it only needs to apply a small number of relocations as previously mentioned and just copies the output bytes from Cranelift to an executable memory region.

VII. DirectEmit Back-end

The DirectEmit back-end [14] (formerly referred to as Flying Start) is highly tuned for low compile times and translates Umbra IR to machine code with a single analysis pass and a single code generation pass. The back-end only supports x86-64 and also has a few further restrictions, most importantly, it does not support irreducible loops. This, however, is no real problem, as all possible sources of Umbra IR code are in Umbra itself and can be adjusted accordingly.

The analysis pass constructs a dominator tree, detects natural loops, and approximates liveness intervals of the SSA values with basic block granularity. The code generation pass then iterates over the basic blocks in reverse post-order and for each instruction directly generates machine code, allocating registers greedily on-the-fly. In addition to that, DWARF call frame information is written in parallel. This strategy obviously leads to suboptimal code in many cases, but heavily aids in reducing compile times.

A. Optimizations

1) Run-time: For improved run-time performance, a liveness analysis was added to enable better register allocation. Further, a loop analysis was added to implement a simple heuristic when spilling values to the stack: this way, the register allocator can easily prefer values defined inside a loop.

2) Compile-time: The back-end is highly tuned for low compilation overhead. In addition to being single-pass, the back-end uses a free variable slot in the Umbra IR data structures to assign linearly increasing values to instructions and blocks. This is used to store information in an array, avoiding hash table operations. Further, the back-end uses a custom assembly encoder, which does not focus on generating the most compact representation of an x86-64 instruction, but on reducing the number of branches. The DWARF CFI writer only generates synchronous unwinding information, which only needs to be correct at function calls, reducing the amount of written data.

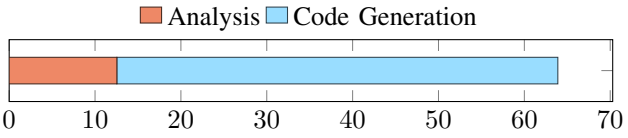


Fig. 5. Compile-time [ms] breakdown of DirectEmit on x86-64; note that the entire compilation of 6678 functions takes just 64 ms.

B. Compile-time Analysis

Due to the highly integrated approach, only the separation of analysis and code generation is reasonably measurable. Figure 5 shows the results. The most expensive part of the analysis is the liveness analysis, taking around 75% of the analysis time. For the code generation pass, a more detailed accounting is not reasonably possible; profiling indicates that roughly 30% is related to register allocation.

C. Discussion & Possible Improvements

The DirectEmit back-end implements a single-pass compilation approach and shows that compiling Umbra IR to machine code can be done as fast as *generating* the corresponding IR for Cranelift or LLVM. This is obviously aided by the design of Umbra IR with an efficient in-memory representation and supporting only instructions that can be easily lowered to machine code; but nonetheless, other compiler frameworks could conceptually design similar data structures.

However, this approach also comes with a substantial engineering effort and is extremely non-portable. An AArch64 port was developed [25], but never merged due to the high maintenance effort. Decoupling parts that are fairly independent of the target architecture (e.g., register allocation) from architecture-dependent parts (e.g., instruction generation) without impacting compile time is an open question.

VIII. Execution Performance

We also evaluated the performance of the generated code of the different back-ends on TPC-DS scale factor 10; Table III and Figure 6 show the results. The "compile time" of the interpreter is the transformation of Umbra IR into register-based bytecode. In terms of execution performance, LLVM-optimized and GCC are fastest, followed by Cranelift and DirectEmit. LLVM-cheap closely follows DirectEmit. The interpreter is the slowest due to interpretation overhead.

However, the massively higher compilation time needed to achieve the marginal performance gains rarely pays off. This particularly applies to the GCC back-end, which for this reason is in practice only used for debugging. Nonetheless, the wide range of optimizations provided by LLVM has a significant impact on some queries, for example on query 17, the code compiled using LLVM-optimized is 38% (0.93 s vs. 1.29 s) faster than the DirectEmit-compiled code.

TABLE III
COMPILE-TIME AND EXECUTION PERFORMANCE OF THE DIFFERENT BACK-ENDS ON TPC-DS SF=10. THE X86-64 MACHINE HAS 32 CORES, THE AARCH64 MACHINE USES 4 CORES.

	x86-64 comp	exec	AArch64 comp	exec
Interpreter	0.03 s	15.40 s	0.02 s	64.55 s
DirectEmit	0.06 s	4.83 s	—	—
Cranelift	1.07 s	4.62 s	0.61 s	16.37 s
LLVM-cheap	1.63 s	5.23 s	0.74 s	19.45 s
LLVM-opt	11.36 s	4.12 s	5.86 s	12.88 s
GCC	48.88 s	4.28 s	41.64 s	13.99 s

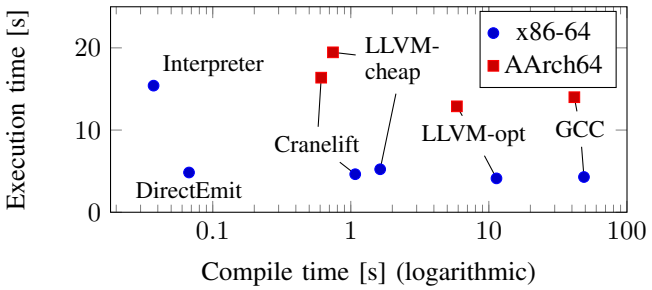


Fig. 6. Compile-time and Execution performance of the different back-ends on TPC-DS sf=10; data from Table III.

To further evaluate when each back-end is beneficial to minimize the sum of compile and execution time, we ran the TPC-H [26] benchmark at scale factors 10 and 100; Figure 7 shows the results. For scale factor 10, DirectEmit is almost always the best choice, Cranelift just once. At scale factor 100, however, also LLVM-opt becomes beneficial for several queries despite the large compile time.

IX. DISCUSSION

Analytical database systems can substantially benefit from fast query compilation over interpreted execution. While for larger data sizes optimizations become more important, fast compilation is especially important for small data sizes and provides large execution performance improvements, which easily offset the compilation time in many cases. As most database systems that compile queries to machine code follow a similar approach to Umbra, our observed results should also apply to other analytical query compilers. However, databases like PostgreSQL [27] that just compile parts of queries, e.g., expressions or predicates, can also benefit from shorter compile times.

A few efforts for faster compilers exist, but the achieved benefits are often low: LLVM's new GlobalISel back-end only slightly improves optimized builds, but is much slower for the unoptimized case. Cranelift, a framework designed for fast JIT-compilation, compiles just 20–35% faster than LLVM, but comes at a substantial maintenance cost, particularly due to the inter-operation between C++ and Rust. An approach like DirectEmit can generate code on-par with LLVM-cheap at much lower cost, but is highly non-portable. More research

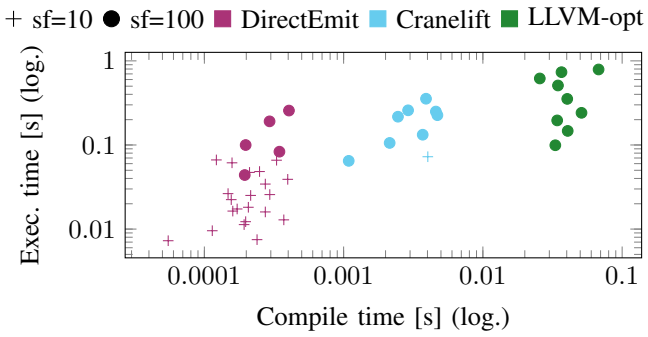


Fig. 7. Best x86-64 back-end for every TPC-H query at scale factors 10/100. LLVM is only beneficial for larger data sets.

in portable and fast compiler frameworks is needed to limit maintenance effort and thereby achieve wider adoption.

X. RELATED WORK

Fast compilation is a primary focus of JIT compilers and therefore a lot of work was dedicated to improving compile times. Frequent optimizations include multi-tiered compilations [2], [3], [28], [29], template-based code generation [29]–[31], and greedy register allocation [32]–[35].

LLVM [12] provides a JIT compilation functionality, but is regularly avoided due to high compilation cost of the general back-end infrastructure. For example, WebKit added an LLVM back-end to their JavaScript compiler in 2014 [36], only to replace it two years later with their custom code generation back-end [37] due to efficiency concerns.

Umbra's code generation back-ends were analyzed before [25], but the reasons for the varying compile times were not investigated. From other database systems, Flounder [10] also implements a custom, lower-level IR for custom code generation that also compiles substantially faster than LLVM, but again did not investigate the underlying causes or optimization potential in existing frameworks.

XI. CONCLUSION

In this paper, we provided a detailed performance comparison and analysis of different compilation back-ends of the Umbra database system. In particular, we compared the performance of GCC and Umbra's single-pass compiler back-end with the JIT-compilation frameworks LLVM and Cranelift. In addition to that, we analyzed the compile times of LLVM and Cranelift in-depth and discussed several optimization possibilities for these frameworks. Our results showed that GCC and LLVM are capable of generating high-quality code, albeit GCC takes an order of magnitude longer with structural issues preventing a more efficient integration, but also showed that LLVM can achieve considerably low compile times. Cranelift compiles code just 20–35% faster than LLVM, but takes 16x as long as Umbra's single-pass back-end.

APPENDIX

A. Abstract

The artifact [38] contains a binary package of Umbra built for Linux on x86-64 and AArch64 modified to allow additional instrumentation for getting the detailed compile-time breakdowns presented in the paper. The enclosed benchmark script (Makefile) obtains and runs data generators for the TPC-H and TPC-DS benchmarks and runs the benchmarks to obtain the compile-time breakdowns from Sec. IV–VII (Table I, Figures 2, 3, 4, and 5) as well as the compile-time and run-time performance data from Sec. VIII (Table III, Fig. 6, and 7).

B. Artifact check-list (meta-information)

- **Program:** Umbra, closed-source, binaries included; LLVM commit `84a6a05`[2], binary included.
- **Compilation:** C back-end tested with GCC 12.
- **Binary:** Includes Umbra x86-64 and AArch64 binaries for Linux with required libraries.
- **Data set:** TPC-DS and TPC-H benchmark scripts included; generated Sf10 ∼10 GiB, Sf100 ∼100 GiB.
- **Run-time environment:** Requires Linux 5.19+ and glibc 2.36+. x86-64 tested on Ubuntu 22.10 and Fedora 37; AArch64 tested on Asahi Arch Linux ARM.
- **Hardware:** x86-64 needs SSE4.1, 128 GiB memory, 400 GiB storage; AArch64 needs Armv8.1-A, 16 GiB memory, 50 GiB storage. Compile/execution time trade-off in Sec. VIII strongly depends on CPU core performance and core count.
- **Run-time state:** Working data should be in OS cache; our script performs one warm-up run for this.
- **Execution:** The system should have no other load during performance measurement. A second execution might yield better results.
- **Metrics:** Average execution time of compilation and query execution.
- **Output:** CSVs of data for plots; raw paper results included; optionally plots as PDF.
- **Experiments:** Makefile to run experiments is included.
- **How much disk space required (approximately)?:** 400 GiB for x86-64; 50 GiB for AArch64.
- **How much time is needed to prepare workflow (approximately)?:** 1–2 hours for benchmark data generation.
- **How much time is needed to complete experiments (approximately)?:** 2–4 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Proprietary.
- **Archived (provide DOI)?:** 10.5281/zenodo.10357363

C. Description

1) How delivered: The artifact is archived on Zenodo and online available at: https://doi.org/10.5281/zenodo.10357363

2) Hardware dependencies: x86-64 experiments require a CPU with at least SSE4.1, 128 GiB memory, and 400 GiB disk storage; AArch64 experiments need an Armv8.1-A system with Advanced SIMD extensions, 16 GiB memory, and 50 GiB disk storage.

3) Software dependencies: Tested with Linux 5.19 and glibc 2.36, newer versions should work. Other required libraries are bundled. The C back-end has been tested with GCC 12, which must be available as `gcc` through the `PATH` environment variable. The benchmark script requires make (tested with GNU Make 4.3), Perl 5, and GNU sed. Downloading the benchmark data generators additionally requires Git. Generating the plot PDFs requires TEX Live and `latexmk`.

[2]https://github.com/llvm/llvm-project/tree/84a6a05

4) Data sets: Data generators for the benchmarks TPC-H [26] and TPC-DS [15] can be downloaded from the respective websites. For convenience, the benchmark script will download and build appropriate data generators unless explicitly provided.

D. Installation

The downloaded tarball can be extracted with `tar -xzf artifact-cgo24-umbra.tar.gz`. Enter the newly created directory `artifact-cgo24-umbra` before running the commands below. No further installation is required.

E. Experiment workflow

Step 1: `make prepare` — download and build the benchmark data generators, generate the benchmark data, and import the data into Umbra database files. This will generate TPC-DS data with scale factors 1 and 10; on x86-64 additionally TPC-H data with scale factors 10 and 100 (for Fig. 7). This can take 1–2 hours depending on the system.

Step 2: `make all` — run all experiments used for the paper and aggregate the relevant information from the raw output. This will produce several files containing benchmark results; the `README` contains a detailed description of the generated files.

Step 3 (optional): `make plots` — plot data into figures. This will produce a file `plots_x86_64.pdf` on x86-64 and `plots_aarch64.pdf` on AArch64.

F. Evaluation and expected result

The artifact includes the raw measurement data used for this paper for comparison.

For the compile time breakdowns, the time distribution and relative performance between back-ends should be approximately similar to those shown in the paper. For GCC, the derivation may be larger due to I/O and operating system overhead.

For the compile-time/run-time trade-off, the results strongly depend on the number of CPU cores, the core performance, memory latency, and memory bandwidth. The order of magnitude of the differences between the back-ends, however, should be similar to the values reported in the paper.

For the selection of the best back-end for each query, similar considerations apply: for slower systems with fewer cores, using LLVM-optimized will be beneficial in more cases, perhaps even at Sf10, whereas on very fast systems with many cores, LLVM-optimized might be the best choice for only few queries even at Sf100. The general trend of LLVM being beneficial primarily for larger data sets, however, should still be observable.

G. Experiment customization

Measurements on the compile-time and run-time trade-off (akin to Sec. VIII) on TPC-H and TPC-DS on scale factors 1/10/100 can be done using existing Makefile targets; for other scale factors, the scale factor values can be adjusted accordingly. For the compile-time breakdowns, the raw data include substantially more details regarding individual compiler passes than shown in the paper; custom analyses on this data are possible. Umbra can generally be used with arbitrary database workloads and other benchmarks. The `README` includes further usage documentation.

REFERENCES

[1] V8 Project, "Ignition," https://v8.dev/docs/ignition, accessed 2023-05-14.

[2] ——, "TurboFan," https://v8.dev/docs/turbofan, accessed 2023-05-14.

[3] F. Pizlo, "Speculation in JavaScriptCore," https://webkit.org/blog/10308/speculation-in-javascriptcore/, accessed 2023-05-14, Jul. 2020.

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

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

[6] 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.

[7] 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.

[8] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder, "Impala: A modern, open-source SQL engine for Hadoop," in *CIDR*, 2015.

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

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

[11] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then, "Get real: How benchmarks fail to represent the real world," in *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.

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

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

[14] 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.

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

[16] P. Menon, A. Pavlo, and T. C. Mowry, "Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 1–13, 2017.

[17] P. Damme, M. Birkenbach, C. Bitsakos, M. Boehm, P. Bonnet, F. Ciorba, M. Dokter, P. Dowgiallo, A. Eleliemy, C. Färber, G. Goumas, D. Habich, N. Hedam, M. Hofer, W. Huang, K. Innerebner, V. Karakostas, R. Kern, T. Kosar, and X. Zhu, "Daphne: An open and extensible system infrastructure for integrated data analysis pipelines," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR '22)*, 2022.

[18] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, pp. 743–754.

[19] DWARF Debugging Information Committee, "DWARF Debugging Information Format Version 5," Feb. 2017.

[20] D. Malcolm, "GCC wiki: Just-in-time compilation (libgccjit.so)," https://gcc.gnu.org/wiki/JIT, accessed 2023-05-08, Sep. 2021.

[21] LLVM Project, "LLVM 17 documentation: Global instruction selection," https://llvm.org/docs/GlobalISel/index.html, accessed 2023-05-14, May 2023.

[22] A. Emerson and J. Paquette, "Bringing up GlobalISel for optimized AArch64 codegen," https://llvm.org/devmtg/2021-11/slides/2021-BringingupGlobalISelForOptimizedaarch64codegen.pdf, accessed 2023-05-14, Nov. 2021.

[23] LLVM Project, "LLVM 17 documentation: JITLink and ORC's ObjectLinkingLayer," https://llvm.org/docs/JITLink.html, accessed 2023-05-14, May 2023.

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

[25] 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.

[26] TPC, "TPC-H decision support benchmark," https://www.tpc.org/tpch/, accessed 2023-08-04.

[27] The PostgreSQL Global Development Group, "PostgreSQL 15 Documentation: What is JIT Compilation?" https://www.postgresql.org/docs/current/jit-reason.html, accessed 2023-08-04.

[28] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *SIGPLAN conference on Programming language design and implementation (PLDI)*, 2000.

[29] 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.

[30] 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.

[31] D. R. Engler, "VCODE: a retargetable, extensible, very fast dynamic code generation system," vol. 31, no. 5, pp. 160–170, 1996.

[32] M. Poletto and V. Sarkar, "Linear scan register allocation," *TOPLAS*, vol. 21, no. 5, pp. 895–913, 1999.

[33] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeno dynamic optimizing compiler for java," in *Proceedings of the ACM 1999 conference on Java Grande*, 1999, pp. 129–141.

[34] C. Wimmer and H. Mössenböck, "Optimized interval splitting in a linear scan register allocator," in *VEE*, 2005, pp. 132–141.

[35] C. Wimmer and M. Franz, "Linear scan register allocation on SSA form," in *CGO*, 2010, pp. 170–179.

[36] F. Pizlo, https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, accessed 2023-05-14, May 2014.

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

[38] A. Engelke and T. Schwarz, "Artifact for CGO'24 paper "Compile-time Analysis of Compiler Frameworks for Query Compilation"," Dec. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10357363