Bringing Compiling Databases to RISC Architectures

Ferdinand Gruber
Technical University of Munich
gruberfe@in.tum.de

Maximilian Bandle
Technical University of Munich
bandle@in.tum.de

Alexis Engelke
Technical University of Munich
engelke@tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Jana Giceva
Technical University of Munich
jana.giceva@in.tum.de

ABSTRACT

Current hardware development greatly influences the design decisions of modern database systems. For many modern performance-focused database systems, query compilation emerged as an integral part and different approaches for code generation evolved, making use of standard compilers, general-purpose compiler libraries, or domain-specific code generators. However, development primarily focused on the dominating x86-64 server architecture; but neglected current hardware developments towards other CPU architectures like ARM and other RISC architectures.

Therefore, we explore the design space of code generation in database systems considering a variety of state-of-the-art compilation approaches with a set of qualitative and quantitative metrics. Based on our findings, we have developed a new code generator called FireARM for AArch64-based systems in our database system, Umbra. We identify general as well as architecture-specific challenges for custom code generation in databases and provide potential solutions to abstract or handle them.

Furthermore, we present an extensive evaluation of different compilation approaches in Umbra on a wide variety of x86-64 and ARM machines. In particular, we compare quantitative performance characteristics such as compilation latency and query throughput.

Our results show that using standard languages and compiler infrastructures reduces the barrier to employing query compilation and allows for high performance on big data sets, while domain-specific code generators can achieve a significantly lower compilation overhead and allow for better targeting of new architectures.

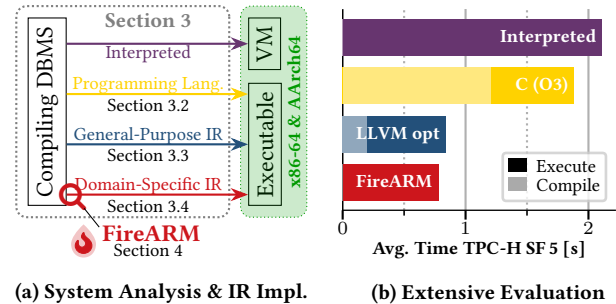**(a) System Analysis & IR Impl.**   **(b) Extensive Evaluation**

Figure 1: Overview of different approaches for code generation in compiling database systems and their combined performance in terms of latency and throughput.

1 INTRODUCTION

Over the last decade, query compilation has emerged as one key technique to achieve substantial performance improvements for efficient data processing. Consequently, several database systems [9, 13, 14, 19, 25, 26] shifted towards this approach of compiling queries to native machine code, allowing a more effective use of available computational capacities. Further, such compilation approaches were no longer exclusive to database systems, but also became increasingly important for general data processing tasks [8].

Due to the dominance of the x86-64 architecture in the server market over the past decades, code generation approaches have strongly focused on performance tuning for this single architecture. However, while performance increases of x86-64 chips have slowed down over the recent years, other architectures are picking up momentum. Particularly ARM, originally focused on embedded and mobile devices, recently started pushing into the server market and is gaining significant traction [22, 39, 40]. Moreover, RISC-V [42] is also an emerging architecture with significant traction, although high-performance hardware is not yet commercially available. This trend in hardware is likely to continue, which is also due to the increasing heterogeneity of computer components.

To meet this increasing heterogeneity, we implement a query compiler back-end called FireARM that directly emits code for AArch64 and thereby also identify general challenges and potential solutions for handling and abstracting specifics of different architectures.

Over the past ten years, a variety of different approaches for query compilation have emerged, setting different priorities with respect to performance, flexibility, and engineering effort. Database systems with query compilation can be roughly classified into three categories (see Figure 1a): first, systems using standard programming languages like C, where the database emits code and uses a standard compiler to generate machine code. Second, systems

using the back-ends of standard compilers like LLVM, where code is generated in form of the intermediate representation (IR) of that back-end. And third, systems running their own domain-specific intermediate representations and code generators, with possible bridges to standard compiler back-ends allowing them to make use of their optimizations.

This motivates us to review different code generation strategies with particular focus on their ability to adapt to different and new processor architectures. To this end, we analyze state-of-the-art approaches for code generation with regard to performance characteristics, expressiveness of their code representation, and required engineering effort.

Furthermore, we present a thorough performance evaluation of different code generation approaches for x86-64 and AArch64. In particular, we fairly compare the performance characteristics of all three code generation strategies on different architectures within the same system, Umbra [26].

We find that while approaches using standard programming languages or compiler back-ends are comparably easy to develop, they also incur a higher latency for query execution. In contrast, our approach of using a domain-specific IR in combination with a fast code generator allows for low-latency execution with comparably high throughput. Due to the design of our IR, this approach can at the same time be used to achieve the highest execution performance with a bridge to standard compilers. We have also found that domain-specific IRs can be designed with a focus on database-specific operations and allow for a more idiomatic expression of complex operations, which is particularly useful when efficiently targeting different architectures, as such operations can be lowered to more optimized architecture-specific instruction sequences.

Thus, the main contributions of this paper are as follows:

- FireARM, a low-latency compilation back-end for Umbra's domain-specific IR targeting AArch64
- Experiences and challenges for porting a compiling database from x86-64 to AArch64
- A thorough qualitative analysis of state-of-the-art query compilation approaches with regard to recent developments in processor architectures
- Evaluation of different code generation approaches on different architectures

The remainder of this paper is structured as follows: In Section 2, we outline the history of compiling databases in more detail. Then, in Section 3, we analyze the state-of-the-art systems for query compilation. In Section 4, we describe our implementation of a code generator for AArch64 and discuss challenges and our approaches for architecture portability. In Section 5, we present the results of our evaluation and discuss the findings in Section 6. Finally, we conclude with a summary of our findings in Section 7.

2 PRIOR WORK

The generation of machine code from query plans in database systems was first implemented in System R in the last century [5]. At that time, the performance of databases was not mainly limited by computational power, but by speed and capacity of memory and storage. Therefore, for the following decades, architecture-specific compilation was replaced by interpretation, which is easier to port

between different systems [2]. In the first decade of this century, memory and storage capacities in servers reached a point where all data of interest could reside in memory [28]. Thus, interpretation started to become the bottleneck for data processing and code generation for queries gained traction again.

Starting with HyPer [25] in 2011, code-generation in databases became a fully-fledged compiler task by using the LLVM toolchain. Other systems such as Impala [19], RAW [17], and Peloton [24] also adopted similar approaches for better performance. In contrast, other systems like Voila [14], HIQUE [21], LegoBase [37], Voodoo [29], and Hekaton [9] chose a different way: They use different general-purpose and newly designed domain-specific programming languages for translating query plans or even whole parts of the runtime to machine code. As standard compiler toolchains, e.g., GCC [11] and LLVM [32], are not designed for low latency compilation, systems like Umbra [26] and Flounder [13] started to design custom IRs (Intermediate Representations). Consequently, final code-generation was now also a task of the database system and not done by standard compilers anymore [13, 18]. Currently, many different strategies for query compilation are implemented by modern databases, but there is still no consensus about which way to go [25, 38, 41].

At the end of the last decade, the hardware market started to change again. For processors based on the x86-64, observed performance improvements were comparably low, while at the same time, ARM [40] managed to set a foot in the server segment. Based on the ARM instruction set architecture, which already dominates the mobile market, companies like Amazon, Apple, Nvidia, and Huawei [16] started to develop their own processor designs, tackling x86-64 systems of Intel and AMD. For example, Amazon's new Graviton processors are already showing competitive performance for different database workloads at a lower price [20]. This rapid development confronts compiling database systems with new challenges: While previous research in this field strongly focused on optimizations of code-generation solely for x86-64 systems and made — possibly extensive — use of its properties, it did not cover portability of the query compiler to new architectures. Architectures like ARM or RISC-V [42] do not just differ in their actual instruction set, but also in other aspects like the memory model, where x86-64 offers stronger guarantees than most other architectures. While some of these differences are abstracted by compilers for traditional applications, this is not the case for compiling database systems.

These developments motivate bringing compiling databases to advancing RISC architectures. The variety of approaches for query compilation, however, opens a range of multiple high-level design choices with a different impact on performance, code representations, and engineering effort.

3 DESIGN SPACE ANALYSIS

In current compiling databases, multiple strategies are used to compile queries to machine code. To identify a suitable approach for high-performance query compilation on RISC architectures, we analyze and classify existing approaches and guide our decision using metrics for evaluating the different strategies in various aspects.
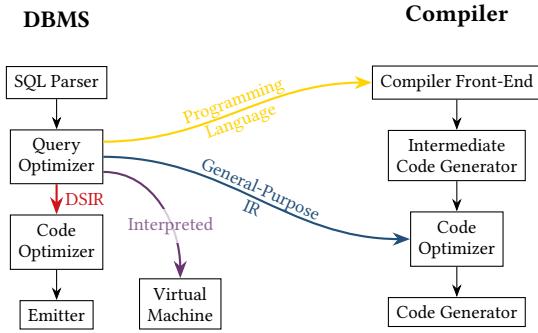
Figure 2: Simplified overview of layers involved in query compilation in a modern code-generating database system.

In compiling databases, code generation starts after the DBMS constructs a physical and an execution plan for a query [41]. Figure 2 shows a simplified view of the overall structure of such an engine. Different strategies for code generation integrate differently within a database system: **Programming languages** integrate a whole compiler into the system — including a new front-end for parsing. Databases using **general-purpose IRs** avoid the front-end overhead but still add additional layers to code generation. **Domain-specific IRs** are specialized IRs designed for database systems and do not necessarily rely on external compiler frameworks. Code generation can be customized and implemented close to the rest of the database system.

The key factor of a code generation system — and therefore the main focus of our analysis — is the (intermediate) code representation used to express query code, as it strongly impacts performance characteristics, the ability to effectively use hardware features, and the engineering effort for the implementation of the database itself.

3.1 Metrics for Query Compilers

To rate a code generation system of a database that allows query compilation for multiple architectures (e.g., x86-64 and ARM), we define five metrics for evaluation. In our opinion, these metrics cover most interesting aspects of a query compilation strategy: performance, expressiveness for query translation, and usability. Although the metrics of expressiveness and usability are hard to objectively quantify, we consider them to be highly important factors and evaluate them from our perspective.

Throughput is the number of tuples processed per second, which databases often strive to maximize. For compiling databases, the throughput mainly depends on the quality of the generated machine code for a query. Despite the additional impact of the database engine's runtime system, we focus on the potential code quality of a compilation approach.

Latency is the time needed for generating and compiling query code before it can be executed. Low latency is more important for real-time transactional systems than for longer running queries as they occur, e.g., in stream-based data processing [3, 18].

Latency and throughput are coupled metrics and both cannot be maximized at the same time because the compilation time increases with the amount and complexity of applied optimization.

Consequently, finding a reasonable trade-off that is the best fit for their use case scenario is required.

Domain Expressiveness assesses the ability to represent algorithmic details for databases (e.g., algorithmic operator details, memory interaction, etc.) in its intermediate language. An efficient and expressive translation from higher levels of representation (e.g., query plans) to intermediate languages or representations reduces complexity and allows maintaining database-specific semantics for later compilation stages.

Architecture Expressiveness assesses how well a compilation approach can handle architecture-specific properties (e.g., memory models, architectural constraints). This not only covers architecture-specific instructions or the fine-tuning of vectorization, but also the ability to support multiple architectures and possibly also heterogeneous systems at the same time. This metric is particularly relevant when porting a database system to a new architecture, as using architecture-specific features is necessary for high-performance database systems. With the rise of co-processors, FPGAs, computational storage devices, and accelerators which all have their own set of constraints and properties, this metric is also of major importance for future compiling database systems.

For compiling databases, domain expressiveness, as well as architecture expressiveness, supports the quality of compiled query code. A more efficient and simple operator representation can be transformed more easily into optimized machine code. Similarly, architectural information helps the code-generator to produce more optimized machine code for a given target system.

Ease-Of-Use rates the complexity of integrating a certain compilation strategy into a database system. For databases, which are usually long-lasting software applications, complexity or low flexibility in the compilation layer are important factors. In this metric, we also include the required level of knowledge in different domains like programming languages, compiler frameworks, and computer architectures. While this metric is certainly the most subjective, we nevertheless consider it as crucial for the development of query compilation systems.
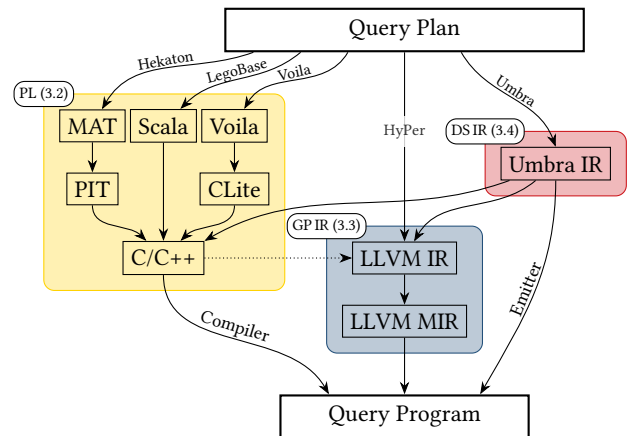

Figure 3: Overview of languages and representations used for code generation in different query compilation systems.

3.2 Strategy: Programming Languages

Using an existing programming language to compile a given query plan is a commonly used approach [9, 14, 26, 37]; thus several different higher-level and lower-level languages have been proposed so far, as shown in Figure 3. Thereby, the usage and position of programming languages in the corresponding code generation layer differ between the systems. We consider a system to follow this approach if the last part of the query compilation toolchain is implemented using a programming language.

LegoBase [37] uses the Scala programming language to implement the engine and as the target for translating query plans. Engine and query code are optimized and compiled together using available run-time information. Afterwards, the generated optimized program is translated source-to-source from Scala to C. For generating machine code, LegoBase invokes a C compiler, which also applies compiler optimizations. Hekaton [9] is part of the SQL Server and uses C for code generation. Query plans are first represented as *Mixed Abstract Trees* (MAT) and then as lower-level *Pure Imperative Trees* (PIT), both of which are not suitable for direct code generation. Thus, PITs are translated to C programs later. Voila [14] implements a new domain-specific language to lower query plans that is more suitable to describe algorithmic details of database queries like hash table look-ups. To ease code generation, Voila code is translated to a subset of C (CLite), which, after source-code optimizations, is translated to a C++ program for compilation with a standard compiler.

In Umbra, query plans of SQL are translated to the internal IR called *Umbra IR*. The C back-end of Umbra then generates C programs from that internal representation using templates of C code, which get slightly modified to adopt the current requirements. The structure of these patterns is not changed; therefore, the process of building these C programs is similar to building objects using construction blocks.

3.2.1 Latency and Throughput. The performance of this code generation approach varies strongly, depending on the usecase scenario and layout of the code generation layer. While the generated machine code can generally achieve a high throughput due to compiler optimizations, the latency of the whole code generation process is typically also high: Compilers like GCC or Clang focus on high-performance machine code but are not designed for low-latency compilation. Systems like Amazon Redshift mitigate this problem by extensively caching parts of previously compiled queries [1].

In addition, source-to-source translation of programming languages is a comparably time-consuming task. Its rather high impact on top of compilation can be seen in the query compilation times of LegoBase [37]. Multi-step translations like translating from Voila to CLite and then from CLite to C++ additionally increase latency.

3.2.2 Domain Expressiveness. The domain expressiveness of this approach strongly depends on the employed languages. Describing database-specific algorithms like hash lookups in detail is more complex and less intuitive in general-purpose programming languages that do not offer native support for these operations. LegoBase efficiently represents database-specific algorithms in Scala but has to lower these representations to C before final generation of code takes place. Voila offers a rich expressiveness for database-specific

algorithms but again must lower to a low-level language before code generation. This process may lead to the loss of domain-specific or other relevant information that is beneficial for generating code. Besides the problem of information preservation, the lowering of high-level representations to lower-level ones is not always possible in a reasonable manner. In Hekaton, the direct translation of MATs to C code is difficult; thus, another intermediate translation step using PITs is necessary.

In general, to lower higher-level to lower-level languages that make code generation easier, the lower-level languages must have at least the same amount of declarative expressiveness [10]. This leads to the fact that the domain expressiveness is finally limited by the last language in the code generation stack. Therefore, using low-level C and C++ that have low domain expressiveness raises questions about their suitability as final targets for code generation.

3.2.3 Architecture Expressiveness. The usage of specific features of a computer architecture like ARM or accelerators like FPGAs must be supported by the last language used in the stack. For example, to use SIMD operations provided by modern processors, compilers can apply auto-vectorization, but the result might not be optimal for non-trivial code. Thus, architecture-specific libraries, intrinsic functions, or assembly code must be used to generate optimized vectorized code. In general, the operators defined by a former language of the compilation stack must be lowered to corresponding operators of the latter one. This is non-trivial if there is no simple mapping between both languages.

3.2.4 Ease-Of-Use. The complexity of this compilation strategy depends on the chosen languages and the layout of the code generation stack. In general, the translation of query plans using programming languages can be done with pre-defined code patterns and templates. The integration of this approach is also rather simple: The database calls an external compiler and generated query programs are loaded as modules. In addition, little knowledge of hardware details is necessary for generating code. Nevertheless, source-to-source transformations between multiple high-level or domain-specific languages increase complexity.

3.3 Strategy: General-Purpose IRs

To avoid the overhead of first translating queries to source code like C, which compilers internally have to translate to their internal IR again, direct generation of the compiler IR is a more efficient option. This approach is used by HyPer [25], Peloton [24], and Umbra [26], among others, using the IR of the LLVM compiler framework.

3.3.1 Latency and Throughput. This approach allows re-using existing code generation infrastructure while reducing the latency by skipping the compiler front-end for parsing and source code analysis. However, the other considerations regarding latency caused by expensive optimizations remain. At the same time, the throughput is not affected as long as the IR generated by the query compiler has a similar quality as the IR derived from the programming language: most of the important optimizations (e.g., the removal of dead code) are performed at the IR level anyway.

The performance of the database system HyPer, which uses LLVM IR for the translation of query plans, shows that this approach is competitive with programming languages regarding latency and

throughput [25]. Unlike HyPer, the LLVM back-end of Umbra uses LLVM only for final code generation, where Umbra IR is translated to LLVM IR. This translation is faster than the translation to C programs and at the same time reaches similar performance.

3.3.2 Domain Expressiveness.
The IRs of compiler toolchains like LLVM specify instruction sets of virtual computer architectures. They are designed as targets for lowering high-level programming languages to machine code. Therefore, the expressiveness of compiler IRs is at least as powerful as the supported programming language. Nevertheless, in accordance with our definition of domain expressiveness, this does not have to hold: Common compiler IRs only provide simple operations and do not contain specialized instructions to represent database-specific algorithms. Technically, it is possible to extend the IR using *intrinsic functions* or by adding new instructions, but such modifications are non-trivial tasks and also need a considerable maintenance effort [31].

3.3.3 Architecture Expressiveness.
Since general-purpose IRs are designed to handle compiler workloads, the architecture expressiveness of using such IRs is similar to using programming languages. For example, LLVM-IR provides intrinsic functions for target-specific operations to enable representing C intrinsic functions and allows for including architecture-specific code with inline assembly to support inline assembly as written in C or C++. Communication with co-processors or other devices like FPGAs can be done by calling library functions that can be provided as IR modules. Since LLVM IR enshrines details of architectures (e.g., size of types), this can be an interoperability issue between different architectures and requires additional handling.

3.3.4 Ease-of-use.
Using IRs of compilers to generate query programs is more difficult than using programming languages. While the latter can be generated using templates or code-snippets, most IRs are given in *Single Static Assignment* (SSA) form, which incurs additional complexity for generating code. Besides that, interacting with compiler frameworks is more complicated than calling compilers and requires more advanced knowledge. LLVM offers an interface for JIT compilation and therefore can be fully integrated into a database system without requiring external toolchain components to be available. However, this code generation strategy introduces another level of complexity compared to using programming languages.

3.4 Strategy: Domain-Specific IRs

In contrast to compiler IRs, domain-specific IRs are designed to solve challenges of specific usecases. Besides their primary usage as additional internal abstraction layer (e.g., MAT and PIT of Hekaton [9]), they can also be the starting point for directly generating machine code. In the following, we only consider domain-specific IRs that support direct translation to machine code. Domain-specific IRs are tailored to the needs of a database system and can be optimized in different aspects. These aspects range from a high domain-expressiveness to performance optimizations or memory consumption for program representation. Nevertheless, they are lower-level than domain-specific languages like Voila to enable direct code generation.

The design and structuring of domain-specific IRs for databases is simpler and more expressive than compiler IRs. Due to the limitation to represent only the query programs of one database system, they can be tailored to the specific needs. This results in an IR design whose expressiveness can exactly match the requirements. On the other hand, domain-specific IRs cannot be directly compiled with existing compiler toolchains. Thus, code generation has to be implemented for the specific IR and every supported architecture, which is a complex and non-trivial task. Even though their design and structure is often simplified or limited, domain-specific IRs enable a high degree of flexibility in the design process.

Flounder [13] and Umbra [26] both implement domain-specific IRs with a custom code generator. In the case of Flounder, the IR is designed at low-level and close to the x86-64 ISA [13]. This allows the implementation of an efficient code-generator that is less complex than common IR compilers. Umbra implements Umbra IR as customized domain-specific IR for internal representation of query programs and low-latency code generation [18]. Umbra IR is designed with a higher degree of domain-expressiveness compared to Flounder IR and has a higher-level structure that is closer to compiler IRs. Nevertheless, Umbra IR is still less complex than compiler IRs in terms of generating IR programs and efficiently translating them to machine code.

3.4.1 Latency and Throughput.
Domain-specific IRs allow optimizing for low latency, high throughput, or both properties. By keeping a simple structure for the domain-specific IR, the complexity of code generation back-ends decreases. This allows code generation with fewer and faster transformation passes compared to common compiler toolchains like LLVM, which require multiple passes for translation even if no optimizations are applied. Umbra IR, as well as Flounder IR, is translated by single-pass code-generators [13, 18]. This allows low-latency compilation, which is especially necessary for real-time transactional databases.

In general, the full feature set of programming languages and general-purpose IRs is not needed to transform a query plan into a program. A reduced set of instructions that is capable of describing all necessary algorithmic details is sufficient. As a consequence, the potential throughput which can be achieved by domain-specific IRs is at least on par with the other code generation strategies for databases. Besides the structure of a query program (e.g., control flow, basic block ordering, etc.), the throughput mainly depends on the applied optimizations and the selection of machine instructions. Both are non-trivial to implement but can be highly optimized if the design of a domain-specific IR is simple enough. Nevertheless, compiler optimized query programs are usually slightly ahead in terms of throughput due to many optimization passes.

3.4.2 Domain Expressiveness.
Domain-specific IRs are designed and implemented for specific use case scenarios, so their expressiveness may not be beneficial for other use cases. In contrast, for databases the expressiveness of specialized IRs is higher by design and they can show their full potential. Compiler IRs usually lack specialized instructions for representing database-related algorithmic details that domain-specific IRs are free to implement. Thereby, these instructions must not be as flexible and expressive as high-level constructs in languages like Voila. In SQL, checks for

Table 1: Evaluation of different compiling database systems according to our set of metrics.

System	Latency	Throughput	Domain Expr.	Architecture Expr.
Voila [14]	– – –	++	++	−−
Hekaton [9]	– – –	++	+	−−
LegoBase [37]	– – –	++	+	−−
DBLAB/LB [38]	– – –	++	++	−−
HyPer [25]	++	++	– – –	−
Flounder [13]	++	+	– – –	++
Umbra [18, 26]				
−*Low-Latency*	+ + +	++	+ + +	+
−*High-Throughput*	−	+ + +	−	−

arithmetic overflows are necessary, so code generation for them is a re-occurring task that can be simplified by specialized instructions.

3.4.3 Architecture Expressiveness. As the design of the IR and the code generation layer is under full control, there are fewer challenges when adding specialized instructions and annotations compared to huge projects like LLVM or GCC. This also allows better support for accelerator hardware or co-processors on the level of domain-specific IRs with specialized instructions. This can also be done using general-purpose IRs, but modifications like the addition of instructions to full-grown compiler toolchains are much more complex. In addition, specialized instructions also help emit the best possible instruction sequence for a certain operation. If this must be done on top of general-purpose IR instructions, this is usually harder and requires multiple complex optimization steps.

3.4.4 Ease-Of-Use. Defining domain-specific IRs and implementing customized code-generators is far more complex than re-using an existing compiler infrastructure and requires a deep understanding of IR design, compiler development, and computer architecture. In general, it is also not possible to re-use parts of existing compiler infrastructure for domain-specific IRs. Nevertheless, domain-specific IRs integrate better in databases than general-purpose IRs because they are tailored for the rest of the system.

3.5 Analysis

To better understand the practical impacts and trade-offs of the different approaches, we rated different database systems that use code generation in Table 1. The low-latency and high-throughput modes of Umbra are rated separately because of their different ratings according to our metrics.

Latency and Throughput. All approaches turn out to be reasonable choices for achieving high throughput. Because most of the discussed databases rely on compilers and their optimizations for code generation, the difference in terms of throughput is reasoned by the structure of generated query programs and the rest of the database system. If JIT modes of compilers are used, commonly fewer optimization passes are chosen and performance may be not as good as full compiler translations. Customized code generators for domain-specifc IRs usually do not apply the same amount of optimizations to code as compilers do. This is not a restriction in general because additional optimization passes can be added for better execution performance. However, due to higher domain

and architecture expressiveness, domain-specific IR can achieve throughput that is on par with the other approaches using compiler toolchains. Nevertheless, generating programs with higher throughput than compilers is not easy to achieve and requires heavy engineering. In contrast to that, we suggest that low-latency code generation is only achievable using domain-specific IRs. Common compilers are not optimized for low-latency code generation and even when avoiding parsing by generating IR code directly (see Figure 2), latency is higher than that of domain-specific IRs.

Expressiveness. It is easier to translate query plans to domain-specific languages like Voila than lowering them to general-purpose programming languages like C++, as database-specific constructs can be expressed directly. While compiler IRs can be extended, this often involves substantial effort with a complexity close to (if not higher than) building customized IRs. Therefore, domain-specific IRs are a good trade-off between required expressiveness and low-level representation. Regarding architecture support, languages like Voila or Scala (LegoBase) do not directly offer functionality for hardware-specific optimizations. Domain-specific IRs can avoid this problem easily because they can be freely extended with specialized instructions or annotations.

Ease-Of-Use. In terms of usage complexity, programming languages have an advantage in comparison to IRs. Internal transformations are still part of the database system and may be non-trivial, but final generation of machine code is done by an external toolchain. Building query-programs with compiler IRs usually requires the integration of the IR system into the database. This is more complex than a call to an external compiler but can use already existing infrastructure for the creation and compilation of IR programs. In contrast, domain-specific IRs must not only be designed, but also code generation and multiple back-ends for machine code must be implemented. This requires in-depth knowledge about compiler construction and machine architectures. Even if a customized solution may integrate better into the rest of a system, the complexity of using domain-specific IRs is still high. If the minimization of complexity is an important aspect, we think that programming languages are the best choice for code generation in database systems.

4 FIREARM

Tahboub et al. advised database developers to use programming languages or existing compiler infrastructure (e.g., LLVM) for query compilation in database systems [41]. Our analysis, however, showed that all approaches face challenges and have different trade-offs. Compared to programming languages, compiler IRs help to address some challenges of query compilation like the latency, but they also introduce new issues like a rapidly increasing complexity for code generation. Because IRs and compiled programming languages are on par in terms of our throughput metric, the latter are a reasonable choice if latency is not an important factor. If latency is the most important metric, then domain-specific IRs like Umbra IR turn out to be a favorable option. With their high domain- and architecture-expressiveness, domain-specific IRs also compete with specialized domain-specific languages like Voila — on another level of abstraction.
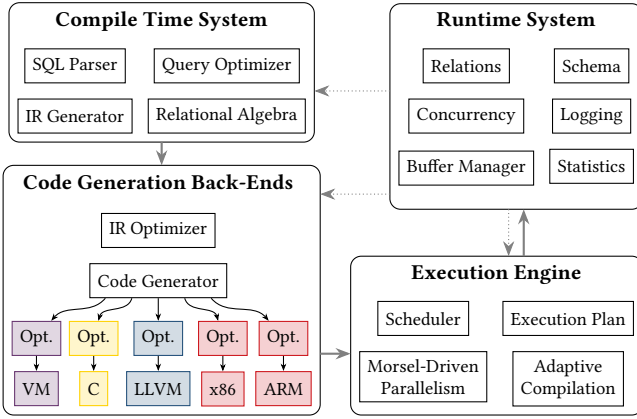
Figure 4: Components of the Umbra database system.

As Kersten et al. [18] showed, Umbra IR allows the direct generation of machine code without using external compiler frameworks or other additional IRs, thereby avoiding the high compilation time of standard frameworks. This was first implemented for the x86-64 architecture as part of an adaptive execution back-end called *Flying Start*, which achieved a comparably high throughput with a significantly lower latency.

Figure 4 gives an overview on the components of Umbra. The code generator also supports additional modes of compilation besides directly translating Umbra IR to machine code: it can also adaptively make use of a standard compiler IR (LLVM IR) for more optimized code generation for longer running tasks. Furthermore, Umbra IR can also be transformed into the C programming language for compilation with a standard compiler.

This flexibility of supporting all three code generation strategies in a single system makes Umbra a good choice for porting a compiling database to new architectures and analyzing the impact of the different options. We implemented a new custom code generator for ARM-based systems that we call *FireARM*, which is able to generate machine code for ARMv8.2-A [23] with low latency while achieving performance close to compiler-generated code. Our work on developing a direct code generator for ARM platforms uncovered several architectural challenges inherent to weak-memory RISC-like architectures, which also ease further ports to similar architectures like RISC-V.

4.1 Compiling Umbra IR

Similar to Flying Start, FireARM directly translates the internal IR of Umbra to ARM machine code in a single pass without an additional, lower-level IR.

The general design of Umbra IR is inspired by LLVM IR in terms of instruction set and functionality, but it is optimized for use in a database system. Like LLVM-IR [33] and GCC GIMPLE [6], it has a base set of instructions for arithmetic, control-flow, and interaction with memory. As mentioned in Section 3.4, it also adds database-specific operations like checked arithmetic or enhanced branching. Along with most modern IRs, Umbra IR uses SSA (*Static Single Assignment*)[35], so IR values (the IR counterpart of variables known from programming languages) are only set once. This eases

central parts of code generation such as register allocation and control flow simplification.

In accordance with LLVM IR, a function in Umbra IR is structured in *basic blocks*, which contain a sequence of instructions terminated by a control flow transition to the next basic block, such that there is no branching or other control flow within a basic block [18, 33]. FireARM translates each function for its own and also generates code block by block. Currently, there are no global optimizations, such as the inlining of other functions, as known from compilers.

Similar to general-purpose IRs, Umbra IR supports a large set of common data types for IR values. Unlike LLVM IR, however, Umbra IR does not implement aggregate types such as `structs`. The absence of extraction operations as found in LLVM IR (see Listing 1a) not only simplifies the IR itself, but also simplifies register allocation and stack management. Nevertheless, some operations do produce multiple result values; for example, checked arithmetic produces the arithmetic result and overflow information. Umbra IR addresses this special case using so-called *ghost instructions* that implicitly reference results of the previous instruction. Due to this direct coupling, such instruction pairs need to be translated in combination.

```
1 %tmp_result = call { i32, i1 } @llvm.sadd.with.↩
       overflow.i32(i32 %a, i32 %b)
2 %result = extractvalue { i32, i1 } %tmp_result, 0
3 %overflow = extractvalue { i32, i1 } %tmp_result, 1
```

(a) General-Purpose IR — LLVM IR

```
1 %result = SAddOverflow i32 %a, %b
2 %overflow = OverflowResult Bool
```

(b) Domain-Specific IR — Umbra IR

Listing 2: Getting result and overflow information from arithmetic operations.

Listing 2 shows the ghost instruction `OverflowResult`, that allows using overflow information such as regular IR values. Code generation benefits from these instructions: The simpler format permits a faster code generation, since the instructions are easier to resolve and permit a good instruction selection without further analyses — without losing expressiveness at the IR-level.

To achieve low-latency compilation, FireARM only performs dead-code elimination on Umbra IR before generating machine code but omits other optimizations typically found in compilers; optimizations such as the control flow optimization or common sub-expression elimination are not performed. For compiling database systems, it is more reasonable to do certain optimizations on the level of relational algebra before generating intermediate code. This prevents the system from generating unnecessary or inefficient IR code in the first place. Furthermore, optimizations on top of the IR are more complex even when using domain-specific information. For example, the elimination or optimization of a join operator on a low-level IR is much more difficult than on a query plan. Thus, FireARM as well as FlyingStart substantially benefit from optimizations done on the level of relational algebra. Writing custom code generators aiming at either low-latency or reasonable throughput gets more difficult if the front-end misses certain optimization opportunities. To some extent this also applies to other

```
1  define i8 @overflow(i8 %0, i8 %1) {
2    %3 = call { i8, i1 }
3       @llvm.sadd.with.overflow.i8 (i8 %0, i8 %1)
4    %4 = extractvalue { i8, i1 } %3, 1
5    br i1 %4, label %5, label %6
6  5: call void @errorHandling()
7    br label %8
8  6: %7 = extractvalue { i8, i1 } %3, 0
9    br label %8
10 8: %9 = phi i8 [ 0, %5 ], [ %7, %6 ]
11   ret i8 %9
12 }
```
(a) General-Purpose IR — LLVM IR

```
1  define i8 @overflow(i8 %0, i8 %1) {
2    %2 = checkedSAdd i8 %0, %1, label %3, label %4
3
4
5
6  3: call void @errorHandling()
7    ret i8 0
8
9
10
11 4: ret i8 %2
12 }
```
(b) Domain-Specific IR — Umbra IR

Listing 1: Tailored instructions, e.g., for the error-handling of checked arithmetic, simplify the IR structure in a DSIR.

code generation strategies, where optimizers are still limited in their possibilities to recover optimizations without further domain-specific knowledge about the operations.

One of the most important aspects that affect the performance of generated code and also the compilation time is register allocation, whose optimal solution is an NP-complete problem. Thus, FireARM (like Flying Start) uses a modified version of *linear scan* [30] for lifetime analysis of IR values and register allocation, which is a common strategy for JIT-compilers. The lifetime analysis of FireARM is not general purpose, because it does not support control flow constellations like *irreducible loops* [15]. This is still feasible because Umbra avoids such control flow during code generation and there are no optimizations that introduce it afterwards.

As already mentioned, Umbra IR was originally designed for x86-64 systems. Thus, generation of Umbra IR programs and their internal structure are also suited to match the limited register set of x86-64. Umbra tries to produce IR code with tight loops for its pipelines, so the amount of life IR values is as small as possible. In contrast, RISC architectures like ARM have a richer set of general-purpose registers (32 in case of ARMv8) due to their less flexible instruction set. Therefore, FireARM or code generators for RISC in general have fewer problems to avoid *spilling* [4]. FireARM only uses the caller-save subset of the 31 general-purpose registers of ARM for IR values, which is larger than the whole general-purpose register set of x86-64. This is due to a variety of reasons, such as the simplification of register allocation and favoring internal calls (e.g., calls between JIT-compiled Umbra IR functions).

Most Umbra IR instructions follow the *three operand principle*, so an instruction has one destination and supports two source operands that can either be IR Values or constants. Translating them to machine code is less complex on RISC systems because their machine instructions follow the same principle. On x86-64, register allocation can get quite complex because most machine instructions for arithmetic only support two operands. This leads to register-to-register copying and spilling, which is quite expensive in terms of performance. In contrast, FireARM benefits from the larger register set and the three operand principle of a RISC architecture like ARM during code generation.

4.2 Architectural Challenges

Umbra IR was originally designed to enable efficient code generation on the x86-64 architecture. Consequently, some aspects of Umbra IR and the preceding translation of query plans to Umbra
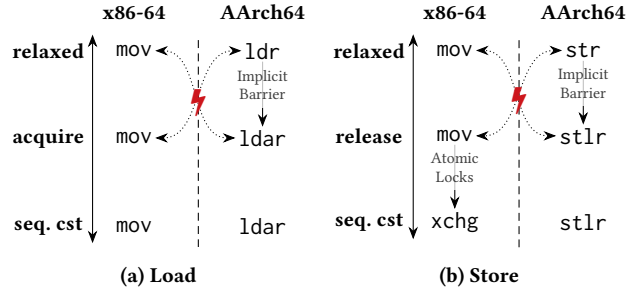

Figure 5: Comparison of the C++ atomic operation mapping to x86-64 and AArch64 instructions.

IR are designed to match specifics of x86-64. During the implementation of FireARM, we gained detailed insights about porting a compiling database system to an architecture with a weak memory model and a RISC-like instruction set.

Memory Ordering is different on x86-64 compared to several other widespread architectures. x86-64 processors use a *processor-ordering* consistency model [36], while the ARM architecture implements a *weak-ordering* consistency model [34]. Figure 5 shows the machine instructions of x86-64 and ARM that are used to implement different C++ memory orderings. Except for stores requiring sequential consistency, the mov instruction of x86-64 is sufficient for memory operations with relaxed, acquire, or release semantics. Following Umbra IR's design focus on x86-64, a fine-graded definition of the memory ordering was not necessary and therefore only allows marking stores as *atomic* to achieve sequential consistency.

In contrast, on a weak-ordering architecture, selecting the appropriate instruction is a more complex task, especially if the required ordering is not precisely defined in the IR. On such architectures, CPUs often get significant performance benefits by exploiting the weaker memory model, as it allows for a more flexible execution order of memory accesses.

To evaluate the impact of choosing too strong ordering semantics in the context of query compilers, we forcefully chose stronger ordering semantics for loads and/or stores in several (handwritten) TPC-H queries and measured the overall execution time; Table 2 shows the results on two different AArch64 platforms. While the impact is moderate for TPC-H query 3, the performance of queries 1 and 19 are much more sensitive to a stronger ordering. This shows that choosing optimal ordering semantics for all memory operations is highly relevant to achieve high performance.

Table 2: Impact of choosing a too strong memory ordering in several TPC-H (SF 10) queries on AArch64 processors.

		Optimal Model	Force Load Acquire	Force Store Release	Always Acq/Rel
Apple M1	Q1	79 ms	178 ms	98 ms	188 ms
	Q3	53 ms	61 ms	66 ms	68 ms
	Q12	64 ms	69 ms	70 ms	69 ms
	Q14	199 ms	231 ms	199 ms	233 ms
	Q19	767 ms	1008 ms	789 ms	1015 ms
Thunder X2	Q1	53 ms	73 ms	66 ms	143 ms
	Q3	58 ms	59 ms	61 ms	85 ms
	Q12	13 ms	18 ms	17 ms	18 ms
	Q14	73 ms	83 ms	89 ms	92 ms
	Q19	134 ms	221 ms	228 ms	239 ms

Besides the execution of query plans, the memory model affects other important aspects of database systems as well: as discussed by Oberhauser et al. [27], synchronization primitives and their implementation may be a source of bugs that are particularly hard to find and solve, while overly restrictive programming in critical sections might have a strongly negative performance impact.

Alignment is usually not a major concern because it is automatically handled properly by the CPU in most cases. In-memory systems like Umbra tend to pack their internal data structures and increase space utilization by avoiding padding bytes. RISC-like architectures like AArch64, however, have more limited addressing modes, especially with regard to immediate offsets that are not a multiple of the element data size. Additionally, unaligned data is also problematic when used for atomic memory accesses: While modern x86-64 systems guarantee atomicity for atomic operations on unaligned data if the operation remains in a single cache line, most other architectures like ARM or RISC-V prohibit unaligned atomic accesses altogether. Thus, to avoid expensive workarounds, data that needs atomic accesses should be aligned if at all possible.

Arithmetic Operations with 8-bit and 16-bit wide operands can be challenging on several RISC architectures. x86-64 directly exposes arithmetic on these sizes using explicitly addressable subregisters and properly provides extra information like an indication of overflow. On AArch64, in contrast, such small operations need to be promoted to 32-bit arithmetic, causing them to be less efficient and increasing the complexity of the code generator.

As the design of Umbra IR focused x86-64, arithmetic on small data sizes occurs even when not strictly required. Due to the singlepass compilation in FireARM, such operations are currently not optimized and thus the code generated by FireARM is often inferior to compiler-generated code in this aspect.

Instruction Selection is one of the most important aspects of code generation. To achieve high performance, optimizing compilers generally try to combine multiple IR instructions into one machine code instruction and at the same time avoid machine instructions or instruction sequences that are considered as inefficient. To achieve high performance, classical compilers often have a large set of patterns and rules and if possible also make use of additional information about the targeted micro-architecture.

In a latency-sensitive context, however, complex instruction matching is too expensive; thus, FireARM only uses a comparably small part of the ARM ISA. Especially the larger diversity of ARM CPU designs — besides the standard designs of ARM manufacturers like Huawei [16] or Apple can also roll their own, with potentially substantive differences in performance characteristics — and the increasing fragmentation of supported instruction set extensions make it infeasible for a low-latency code generator to optimize for available specialized instructions. However, ARM also offers also complex instructions that are not available on x86-64, for example a combined *multiply-add*. Currently, due to its focus on x86-64, Umbra IR does not provide such instructions and therefore FireARM must explicitly detect such instruction sequences to achieve more efficient code.

Immediate operands encoded directly into instructions differ strongly between architectures. x86-64 is very flexible as a consequence of the variable-length instruction encoding, often allowing immediates up to 32 bits, and to use this possibility effectively, Umbra IR also supports immediate operands for many operations. RISC architectures with a fixed instruction size are necessarily less flexible than x86-64 and therefore the code generator needs to move the immediate operand to a dedicated register more often. For large immediates, generally sequences of instructions to combine the constant value are necessary, increasing code size and complexity.

5 EVALUATION

To analyze the latency/throughput (cf. Section 3.1) of the different code generation strategies (cf. Section 3) on different architectures, we run all 22 TPC-H queries with 5 code generation back-ends in Umbra [26]. As the Umbra IR programs, which are generated using a push-based model, are the same for all back-ends, we ensure that the results are comparable, i.e., have the same query plan and use the same runtime system. Besides the Umbra-specific IR, we are not aware of aspects limiting the applicability of the approaches (and therefore also results) in other query compilation engines.

5.1 Experimental Setup

Umbra translates query plans to its internal Umbra IR, consisting of one function per pipeline [26]. We evaluate the following back-ends to compile those Umbra IR functions to machine code:

■ **Interpreted** Umbra IR is translated to an internal bytecode representation, which gets interpreted, similar to systems like SQLite [7].

■ **C** Umbra IR is translated to C code, which is compiled and optimized by gcc -O3. The C code is generated by simply concatenating fixed code templates without further optimizations.

■ **LLVM** This back-end first translates Umbra IR to LLVM IR and then uses the JIT-capabilities of LLVM for fast code generation. It supports a *latency*-optimized mode without optimization passes using FastISel and a *throughput*-optimized mode with selected optimization passes[1] using SelectionDAG for instruction selection.

■ **FireARM** This refers to the ARM-specific back-end for direct code generation described in Section 4.

---

[1]Umbra uses the following passes: Common Sub-expression Elimination, Instruction Combination, Re-association, Simplification of Control Flow Graphs, Aggressive Dead Code Elimination
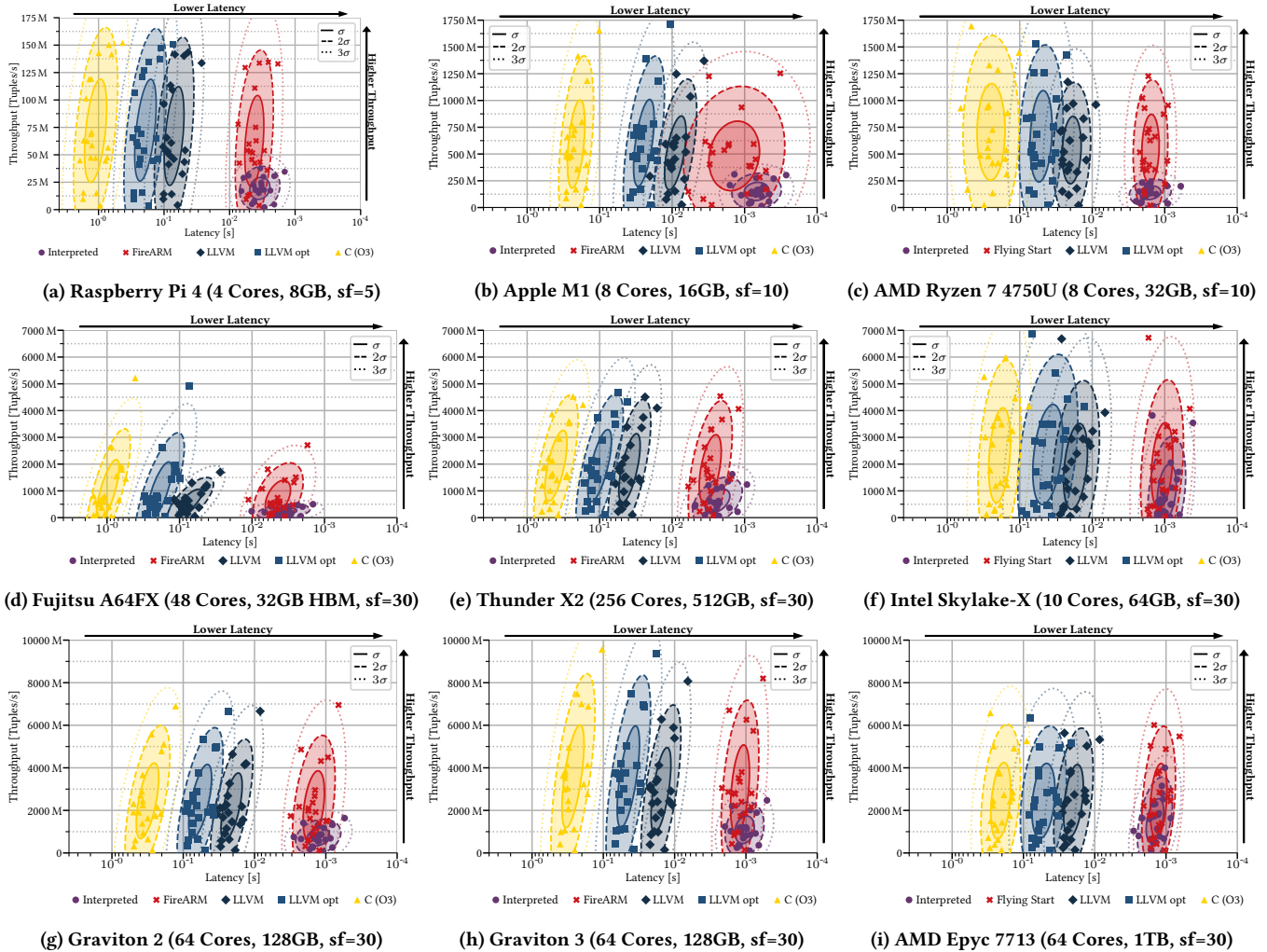
(a) Raspberry Pi 4 (4 Cores, 8GB, sf=5)

(b) Apple M1 (8 Cores, 16GB, sf=10)

(c) AMD Ryzen 7 4750U (8 Cores, 32GB, sf=10)

(d) Fujitsu A64FX (48 Cores, 32GB HBM, sf=30)

(e) Thunder X2 (256 Cores, 512GB, sf=30)

(f) Intel Skylake-X (10 Cores, 64GB, sf=30)

(g) Graviton 2 (64 Cores, 128GB, sf=30)

(h) Graviton 3 (64 Cores, 128GB, sf=30)

(i) AMD Epyc 7713 (64 Cores, 1TB, sf=30)

Figure 6: Compile-time and throughput of different query-compilation strategies in Umbra running the TPC-H benchmark.

■ **Flying Start** This back-end is the x86-64-specific back-end for direct code generation presented by Kersten et. al. [18]. This back-end is the standard compilation back-end on x86-64 systems and also serves as first-tier code generator in the adaptive back-end.

For compilation of C code in the C back-end, we use *GCC 11.1* and *LLVM 14* for JIT-compilation in the LLVM back-end. All back-ends only use one thread for the generation of machine code or preparing the interpretation in case of the VM back-end; multi-threaded code generation is currently not implemented. The execution of the compiled code uses all available hardware threads.

5.2 Compilation Latency

The X axes in Figure 6 show the compilation times of the different back-ends of Umbra on our test systems. While every back-end starts with an optimized Umbra IR program, the latency differs by several orders of magnitude on ARM as well as on x86-64 based

on the chosen compilation approach. The VM back-end generally has the lowest latency on all platforms as a consequence of its very lightweight transformation to bytecode, often achieving latencies in the range of 0.5ms and 4ms for a single query.

From the compiling back-ends, FireARM and Flying Start have the lowest latency on their respective target platforms, which is a consequence of going from Umbra IR to machine code in a single step without complex optimizations and additional layers or code representations. Thus, FireARM achieves latencies often between 1ms and 5ms and therefore incurs a slightly higher latency than the interpreter. On the M1 machine, latencies have a significantly higher variance caused by the big-little core design. The latency of FireARM increases around 2.1x if code is generated on efficiency instead of performance cores. Flying Start on x86-64 is slightly faster with latencies in the range of 0.5ms and 2.5ms. In comparison to FireARM, Flying Start is more mature and has more optimized implementations for encoding machine code instructions.

Nevertheless, FireARM as well as Flying Start have latencies in the same order of magnitude as the VM back-end while generating native machine code instead of bytecode.

The latency of the latency-optimized LLVM back-end generally is around 2.7x larger compared to FireARM/Flying Start on all systems. This has two major reasons: First, this back-end performs several transformations of the query code, which is first translated from Umbra IR to LLVM IR, then translated to LLVM's architecture-specific Machine IR, and from there to actual machine code. And second, LLVM is generally not focused on fast compilation times, contrary to our approaches for directly emitting machine code.

The throughput-optimized LLVM back-end operates similarly to the latency-optimized LLVM back-end, differing only in two points: First, it applies additional optimization passes, some of which have a superlinear runtime complexity. Second, and much more importantly, it uses the optimizing SelectionDAG instruction selector instead of FastISel. While SelectionDAG allows for a significantly better code quality, it adds another layer of intermediate code representation between the transformation from LLVM IR to LLVM's Machine IR. This additional code representation combined with a more expensive approach for finding performant architecture-specific instructions increases the latency by about 2.8x on the ARM systems and about 2.6x on the x86-64 systems.

The C back-end has the highest latency and is an order of magnitude slower than the LLVM back-ends. While the actual process of generating C code only has a minor impact, not only all compilation phases of the compiler back-end have to be executed, but also parsing and verifying the C code adds additional overhead. Further, the code has to be written to disk and the compilation process also involves disk access for storing intermediate files (assembly text, object file). In addition to that, the machine code generated by the C compiler cannot be executed directly and instead must be loaded from the resulting shared library into memory. Many of these additional tasks involve heavy interaction with the operating system, leading to an overall latency in the range of 20ms and 500ms, which is a factor of more than 100x compared to FireARM and Flying Start.

Generally, the most important factors on the hardware side for achieving low latency is the single-threaded computational power, which often correlates with the clock speed. Several ARM platforms like the Thunder X2, the Graviton 2, and the Raspberry Pi have a weaker single-core performance compared to modern x86-64 processors, causing compilation-based query execution to suffer from higher latencies. Nevertheless, recent ARM processors like the Apple M1 and the Graviton 3 offer comparable performance to modern x86-64 and are likely to change and increase the diversity in server hardware in the future.

5.3 Analytical Query Throughput

The Y axes in Figure 6 show the tuple throughput of all TPC-H queries using the different back-ends. In terms of throughput, the direct-emitting back-ends FireARM and Flying Start achieves a similar performance compared to the fast LLVM back-end. On the Fujitsu A64FX and the AWS Graviton 2, FireARM achieves an even higher throughput, and on the x86-64 machines, Flying Start is also generally slightly more performant. The throughput of the VM back-end is around 2.6x lower and therefore rarely worth using.

Table 3: Run-time of TPC-H queries 3, 9, 13, and the average over all queries on different systems using scale factor 5.

System		DuckDB	LLVM	Flying Start	FireARM
Ryzen 4750U	Q3	0.28 s	0.15 s	0.14 s	—
	Q9	4.61 s	0.36 s	0.36 s	—
	Q13	0.39 s	0.15 s	0.15 s	—
	Avg	0.65 s	0.11 s	0.10 s	—
Apple M1	Q3	0.18 s	0.11 s	—	0.12 s
	Q9	2.18 s	0.28 s	—	0.28 s
	Q13	0.28 s	0.27 s	—	0.28 s
	Avg	0.36 s	0.09 s	—	0.09 s
Raspberry Pi 4	Q3	1.41 s	0.90 s	—	0.92 s
	Q9	3.68 s	2.97 s	—	2.99 s
	Q13	3.14 s	0.85 s	—	0.87 s
	Avg	7.83 s	0.73 s	—	0.78 s

The optimizing LLVM as well as the C back-end generally yield a slightly higher throughput than the latency-optimized compiling back-ends. This is a consequence of applying state-of-the-art compiler techniques, and especially of having a better instruction selection and register allocation, which are specifically targeted towards the properties of the target execution machine. For example, LLVM's SelectionDAG back-end implements a plenty of rules and strategies to transform LLVM IR code to efficient machine code sequences. The additional optimization passes, however, only yielded minor improvements on a few queries. Thus, compared to directly emitting machine code in a single pass, approaches based on standard compilers are better suited to address architecture-specific issues outlined in Section 4.2.

However, the throughput differences between compiling back-ends optimized for latency and throughput is comparably low. This is especially remarkable on ARM CPUs, which have a high heterogeneity in their focus area ranging from low-power, embedded systems to large server CPUs, while at the same time, FireARM performs no active CPU-specific optimizations. For example, on the Fujitsu A64FX, designed for arithmetic-heavy workloads, such optimizations can yield minor performance improvements, whereas on the Graviton 2, a general-purpose CPU, there is no significant difference in throughput.

To analyze the impact of query compilation on different platforms, we also compared the latency-optimized back-ends with the vectorized interpretation of DuckDB on all TPC-H queries, Table 3 shows the results. FireARM is up to 8x faster than DuckDB on ARM platforms and Flying Start is up to 12x times faster on the x86-64 machine. This underlines that also on weaker platforms like a Raspberry Pi, query compilation allows for performance improvements in an order of magnitude.

6 DISCUSSION

Latency and Throughput Our evaluation shows that custom code generation for domain-specific IRs can compete with other compilation approaches in terms of latency as well as throughput. FireARM and Flying Start have only a slightly larger latency than the VM back-end while generating machine code and therefore achieve a much higher throughput. The throughput of code produced by FireARM and Flying Start is slightly lower than using a
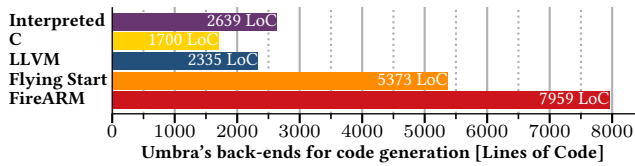
Figure 7: Code comparison of Umbra's back-ends.

highly optimizing compiler back-end, which in turn comes with the cost of a significantly higher latency. In addition to that, all compilation approaches outperform the vectorized interpretation of DuckDB on different systems.

Ease-Of-Use In Section 3.5, we ranked the complexity of domain-specific IRs highest among all approaches. All evaluated back-ends in Section 5 use our domain-specific Umbra IR as the starting point. From an architectural point of view, this is similar to systems like Hekaton that use an internal IR and then apply code generation with programming languages or general-purpose IRs (see Figure 3).

However, the complexity of the different back-end approaches differs heavily. Figure 7 shows the number of lines of code for each of Umbra's back-ends. Although such a metric is by no means sufficient to assess the complexity, it gives a rough indication that writing a back-end like Flying Start and FireARM incurs a notably larger complexity compared to the other approaches in addition to requiring deep knowledge of compiler construction and architecture-specific details. The challenges of directly generating machine code for ARM (cf. Section 4.2) are also reflected in the size of the FireARM back-end. In contrast, generating code using programming languages (C back-end) or general-purpose IRs (LLVM back-ends) is much simpler because much of the complexity is handled by the compiler tool-chains.

Domain Expressiveness The effects of domain expressiveness on code generation can be seen in the performance of both FireARM or Flying Start. These back-ends use the full potential of the domain-specific parts of Umbra IR to generate optimized machine code for ARM and x86-64, without complex optimization passes while having lower latency at the same time. Without this domain expressiveness, the complexity of both back-ends would be substantially higher. Optimizations of modern compilers can compensate the absence of domain expressiveness — and in fact, the domain expressiveness of Umbra IR is lost to some degree when translating to C and LLVM IR — at the cost of higher latency and complexity. Thus, future custom code generators for domain-specific IRs with even higher expressiveness than Umbra IR could make traditional compilers obsolete in databases by providing the best trade-off for latency and throughput.

Architecture Expressiveness Currently Umbra IR does not implement many architecture expressive constructs. It is designed with x86-64 in mind, so some parts of it can be seen as architecture expressive; however, generally the architecture expressiveness is currently limited. For example, FireARM could benefit most from adding more operations like fused-multiply-add to better use features provided by the ARM architecture. At the same time, the C and LLVM back-ends would also benefit, as they could use more optimized built-in functions or instructions. However, as discussed

in Section 3.5, programming languages and general-purpose IR can implement architecture expressiveness only to a certain level.

For future iterations of Umbra IR or domain-specific IRs in general, architecture expressiveness should be one part of the design process from the beginning. While currently domain-specific IRs are designed top-down from the query plans, IR design should also include bottom-up considerations based on different architectures to prevent facing challenges like the ones we discussed in Section 4.2. Of course, the expressibility principle [10, 38] and the consideration of different architectures must not lead to restricted IRs like GNU Lightning [12]. *Architecture-aware* IRs implement aspects of architecture but are not limiting themselves to specific ones like architecture-specific IRs. It is an open question how such architecture-aware design can look in detail and how it will affect the performance of different code generation approaches.

The suitability of the different approaches differs depending on the actual requirements of a database system. As we have shown in Section 5, there are notable differences in latency for code generation between the different back-ends. Thus, real-time analytical and transactional workloads benefit most from the low-latency direct code generation of FireARM and Flying Start. Especially on small systems with a low processing power, e.g., a Raspberry Pi, domain-specific IRs with a direct path to machine code can significantly improve latency compared to standard compiler back-ends. If latency is not critical, e.g., for longer running analytical tasks or stream-based systems, general-purpose languages or IRs offer slight advantages ahead in throughput and significant benefits in terms of implementation effort. Nevertheless, future domain-specific IRs and custom code generators could change the picture in future.

7 CONCLUSION

Custom query compilation using domain-specific IRs as done by FireARM fits well for real-time analytical database systems and systems with a focus on transactional workloads because it offers the best trade-off between latency and tuple throughput. Other compilation strategies (e.g., programming languages and compiler IRs) may lead to higher tuple throughput due to extensive compiler optimizations but suffer from high latency for code generation. However, migrating custom code generation in databases that are primarily focused on the x86-64 architecture to RISC-based systems comes with various challenges and leaves room for architecture-specific optimizations. Nevertheless, our benchmarks and extensive evaluation show that FireARM still performs well over a wide range of modern ARM machines. We conclude that compilation performance using domain-specific IRs on ARM will increase even further if architectural heterogeneity is taken into account from the beginning, possibly ending the dominance of x86-64 for databases.

REFERENCES

[1] Inc Amazon Web Services. 2022. Factors affecting query performance. https://docs.aws.amazon.com/redshift/latest/dg/c-query-performance.html. Accessed: February 26, 2023.

[2] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (1976), 97–137.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *PODS*. ACM, 1–16.

[4] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation Via Coloring. *Comput. Lang.* 6, 1 (1981), 47–57.

[5] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A History and Evaluation of System R. *Commun. ACM* 24, 10 (1981), 632–646.

[6] GCC Developer Community. 2022. GIMPLE. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html. Accessed: February 26, 2023.

[7] SQLite Consortium. 2022. The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html. Accessed: February 26, 2023.

[8] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Färber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, and Xiao Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '22)*.

[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*. ACM, 1243–1254.

[10] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75.

[11] Free Software Foundation. 2022. GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Accessed: February 26, 2023.

[12] Free Software Foundation. 2022. GNU lightning. https://www.gnu.org/software/lightning/manual/lightning.html. Accessed: February 26, 2023.

[13] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient generation of machine code for query compilers. In *DaMoN*. ACM, 6:1–6:7.

[14] Tim Gubner and Peter A. Boncz. 2021. Charting the Design Space of Query Execution using VOILA. *Proc. VLDB Endow.* 14, 6 (2021), 1067–1079.

[15] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (1997), 557–567.

[16] Ltd. Huawei Technologies Co. 2022. Kunpeng Computing Platform. https://e.huawei.com/en/products/servers/computing-kunpeng. Accessed: February 26, 2023.

[17] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *Proc. VLDB Endow.* 7, 12 (2014), 1119–1130.

[18] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30 (2021), 883–905.

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

[20] Nik Krichko. 2021. Comparing Graviton (ARM) Performance to Intel and AMD for MySQL. https://www.percona.com/blog/comparing-graviton-performance-to-arm-and-intel-for-mysql/. Accessed: February 26, 2023.

[21] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE Computer Society, 613–624.

[22] Mark Liu. 2022. ARM-based Server Penetration Rate to Reach 22% by 2025 with Cloud Data Centers Leading the Way, Says TrendForce. https://www.trendforce.com/presscenter/news/19700101-11178.html. Accessed: February 26, 2023.

[23] Berenice Mann. 2017. Arm Architecture - Armv8.2-A evolution and delivery. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-armv8-2-a-evolution-and-delivery. Accessed: February 26, 2023.

[24] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13.

[25] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[26] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[27] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS*. ACM, 530–545.

[28] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2009. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Oper. Syst. Rev.* 43, 4 (2009), 92–105.

[29] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (2016), 1707–1718.

[30] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 895–913.

[31] LLVM Project. 2022. Extending LLVM: Adding instructions, intrinsics, types, etc. https://llvm.org/docs/ExtendingLLVM.html. Accessed: February 26, 2023.

[32] LLVM Project. 2022. The LLVM Compiler Infrastructure. https://llvm.org/. Accessed: February 26, 2023.

[33] LLVM Project. 2022. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html. Accessed: February 26, 2023.

[34] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29.

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[36] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[37] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1 (2018), 4:1–4:45.

[38] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD Conference*. ACM, 1907–1922.

[39] Softbank Group. 2020. Annual Report – ARM Business Strategy. Statista. https://group.softbank/system/files/pdf/ir/financials/annual_reports/annual-report_fy2020_01_en.pdf

[40] Andreas Stiller. 2022. ARMs langer Marsch in die Serverwelt. *iX* 1 (2022), 60–65.

[41] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

[42] Andrew Waterman, Krste Asanović, John Hauser, and SiFive Inc. 2021. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 20211203*. Technical Report. EECS Department, University of California, Berkeley. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf