ABSTRACT

Query compilation can make query execution extremely efficient, but it introduces additional compilation time. The compilation time causes a relatively high overhead especially for short-running and high-complexity queries.

We propose Flounder IR as a lightweight intermediate representation for query compilation to reduce compilation times. Flounder IR is close to machine assembly and adds just that set of features that is necessary for efficient query compilation: virtual registers and function calls ease the construction of the compiler front-end; database-specific extensions enable efficient pipelining in query plans; more elaborate IR features are intentionally left out to maximize compilation speed.

In this paper, we present the Flounder IR language and motivate its design; we show how the language makes query compilation intuitive and efficient; and we demonstrate with benchmarks how our Flounder library can significantly reduce query compilation times.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Conference'20, June 2020, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Query compilation is a technique for query execution with extremely high efficiency. It uses just-in-time (JIT) compilation to generate custom machine code for the execution of each query. The approach leverages a compiler stack that first translates the query from a relational query plan to an intermediate representation (IR), and then from the IR to native machine code for the target machine. The execution-efficiency of the compiled code is very high compared to standard interpretation-based backends. However, by using compilation the technique adds a step to query execution, which introduces translation cost. Especially short-running queries and queries with high-complexity experience a relatively high translation cost, which ultimately extends query response times.

When using query compilation for queries on smaller datasets, the relative cost of compilation increases. The query engine spends most of the time during compilation before entering execution only for a very short time. Further, complex queries can have particularly long compilation times due to complexity of algorithms used in JIT machine code translation [19]. Approaches to mitigate the impact of compilation time on response time have been proposed previously [13]. However, these typically rely on providing both an interpretation-based and a compilation-based backend at a high implementation cost.

1.1 Intermediate Representation Levels

The intermediate representation is an important design choice for query compilers. Figure 1 illustrates the effect of the IR choice on JIT compile times. Query compilers with high-level IRs, such as C/C++ [8, 11, 20] or OpenCL and Cuda [6, 9, 10, 18] generally have longer compilation times than query compilers that generate lower-level IRs such as LLVM IR [16, 17]. Existing work on JIT compilers, however, shows the feasibility of much shorter compile times [2, 5] than those of LLVM. In fact non-database JIT compilers reach break-even points for dynamic compilation versus static compilation already for thousands of records [2]. By contrast, LLVM-based query compilers have compilation times of tens of milliseconds [16], which is sufficient time to process queries on millions of tuples [4].

LLVM IR is general purpose and was designed to serve as back-end for the translation of high-level language features [14]. Being general purpose, LLVM is relatively heavyweight and devises a translation stack that is "overkill" for relational workloads. The code for relational queries typically consists of tight-loops with conditional code mainly to drop non-qualifying tuples. This plain structure offers potential for much simpler translation than performed by general purpose translators, which leverage complex code analysis and register allocation algorithms.

1.2 Contributions and Outline

Our work is the first to integrate machine-level code generation with query compilers (Section 2). We show the abstractions that we
add with Flounder IR to machine assembly for efficient query compilation (Section 3). Then we show the algorithm used for translating Flounder IR to machine code that is tailored to relational workloads (Section 4). We analyze the performance of our Flounder-based query compiler and compare against LLVM-based query compilation (Section 5). Finally we wrap-up the paper with a summary (Section 6).

2 QUERY TRANSLATION

Query compilation typically involves a step that translates relational queries to an intermediate representation (IR) and another step that translates the IR to machine code. In the following, we give an overview of how both steps are realized for query compilation with Flounder IR.

2.1 Query Plan to IR

The first translation step traverses the query plan and builds an intermediate representation of the query functionality. A common way to do this is the produce/consume model [16], which emits code for operator functionality either in produce or consume methods. We call these methods operator emitters. Figure 2 illustrates the operator emitters that are executed during translation of the probe-side pipeline of a sample query. The operators of the pipeline are surrounded by a dotted line. In the example the code to scan \( R \) was already emitted by produce(…) and for selection (\( \sigma \)) by consume(…). The consume call for hash join (\( \bowtie \)) follows next. The code of the hash join operator emitter is shown in Figure 3. The code lines following an emit statement are underlined to emphasize that this code is not executed immediately but instead placed in the JIT query.

In the example, the consume method is called from its right child and therefore the probe-side code is produced (lines 7–13). The code first initializes the variable entry, which holds hash probe results (line 7) and then loops over the hash join matches (lines 8–13). In the loop, we first call ht_get(...) to retrieve the next match (line 9) and then perform a check to exit when no more matches exist (lines 10–11). To process join matches, we read the attributes of the match to registers (line 12) and then the join’s parent operators place their code by calling consume(…) (line 13).

The resulting intermediate representation is shown in Figure 4 (a). It performs the described probe functionality. We briefly describe the resulting IR here and provide a detailed description of the used Flounder IR features in Section 3.

The attribute values are held in \( \{r_a\}, \{s_a\}, \text{and} \{s_b\} \) and the locations of hash table entries in \( \{entry\} \). The hash_get(…) call is realized with mcall and the loop over the probe matches with a combination of compare (cmp) and two jumps (jmp, je). To read attributes from a hash table entry (dematerialize), we use mov from a memory location in brackets [ ] to e.g. \( \{s_a\} \).

Flounder IR

(a) (in-memory)

[...];child code

vreg {entry}

mcall {entry},{ht_get},

mov rax, 0x42fa10

dematerialize ht entry

cmp r11, 0 ;break condition

clear {s_a}

jmp loop_headN ;next probe

 […] ;child code

x86_64 assembly

(b) (in-memory)

[...];child code

mov r11, 0; init entry

loop_headN: ;while head

mov {entry}, 0

cmp r9, [rsp-8]; save

add rsp, 24 ;adjust stack

mov rax, 0x42fa10

call mcall ht_get call

mov rdi, 0x25cac0 ;call

jmp loop_footN ;next probe

mov r11, 0; init entry

loop_footN:

[...];child code

Figure 4: Intermediate representation of hash join probe functionality (a) and corresponding machine assembly (b).
2.2 IR to Machine Code

The next step translates the query’s intermediate representation to machine code. The machine code needs to follow the application binary interface (ABI) of the execution platform. In this work, we use the target architecture x86_64 [15].

The Flounder IR emitted by the hash join is translated to the machine assembly shown in Figure 4 (b). Several abstractions that were used during IR generation are now replaced by machine-level concepts. E.g. the machine assembly uses processor registers such as r12 instead of {s_a}. Further the machine assembly uses additional mov instructions to transfer values between registers and the stack, e.g. mov r8, [rsp-8]. The translation process from Flounder IR to machine code needs to manage machine resources such as registers and stack memory and find an efficient way for their use during JIT query execution.

This section provided an overview of query compilation with Flounder IR and the following sections will describe the mechanisms in detail. The next Section 3 shows the abstractions used by Flounder IR during code generation. The following Section 4 will show the translation process from Flounder IR to machine code.

3 LIGHTWEIGHT ABSTRACTIONS

Flounder IR is similar to x86_64 assembly, but it adds several lightweight abstractions. The abstractions are designed with the interface to the query compiler and with the resulting machine code in mind. For operator emitters, the abstractions provide independence of several machine-level concepts, which allows similar code generation as typically performed with LLVM. For machine code translation, the abstractions are lightweight enough to avoid the use of compute-intensive algorithms and additionally they enable tuning the machine code for relational workloads.

In the following, we present the lightweight abstractions. They add several pseudo-instructions, i.e. vreg, clear, and mcall to x86_64 assembly and use additional tokens, which are shown in braces, e.g. {param1}.

3.1 Virtual Registers

An unbounded number of virtual registers is a common abstraction in compilers [3]. Query compilers use them to handle attributes without the restrictions of machine registers. When replacing virtual registers with machine registers for execution, general purpose compilers perform live-range analysis [1]. This is rather expensive because compilers consider all execution-paths that lead to a register usage.

Query workloads use virtual registers in a much simpler way than general purpose code. They hold attribute data within a pipeline and the pipeline’s execution path only consists of tight loops. This allows query compilers to use a simpler approach that skips live-range analysis. In Flounder IR, operator emitters mark the validity range of virtual registers. The vreg pseudo-instruction marks the start of a virtual register usage, e.g. using

;start virtual register use is
vreg (vreg_nameN)

and the clear pseudo-instruction marks the end of the usage, e.g.

;finish virtual register use
clear (vreg_nameN)

We use these markers in a way similar to scopes in higher-level languages. For instance the Flounder IR in Figure 4 (a) marks the range of the probe attributes {s_a} and {s_b} to reach around the operators in the probe loop.

3.2 Function Calls

Being able to access pre-compiled functionality is important for query compilers. It reduces compile times and avoids the implementation cost of code generation for every SQL feature. To this end Flounder IR provides the mcall pseudo-instructions to specify function calls in a simple way. For instance

;function call to ht_ins
mcall (res) {ht_ins} (param1) ... (paramN)

represents a function call to ht_ins(...) with parameters param1 to paramN and the return value is stored in (res). A pointer to the function code is provided as an address constant via {ht_ins}. This pseudo-instruction is later replaced with an instruction sequence that realizes the calling convention.

3.3 Constant Loads

Large constants, e.g. 64 bit, can not be used as immediate operands (imm) on current architectures. To use large constants, they have to be placed in machine registers. The constant load abstraction in Flounder IR, allows using such constants without restrictions. E.g.

;load from 64 bit address with offset
mov (attr) [{0x7fff5a8e39d8} + {offs}]

loads data from the address {0x7fff5a8e39d8}+{offs} to the virtual register {attr}. During translation to machine assembly, the address constant will be placed in a machine register.

3.4 Transparent High-Level Constructs

We use transparent high-level constructs that mimic high-level language features such as loops and conditional clauses. They are used to generate Flounder IR in operator emitters. For example operator emitters can generate a while loop with the condition {tid} < {len} by using the methods while(...) close(...), and isSmaller(...) as shown below.

// Produce code for while loop (C++)
wl = While(isSmaller(tid,len)); {
[...]
} wl.close();

This generates the Flounder IR on the right, that realizes the loop functionality. The cmp instruction evaluates the loop condition and jge jumps to the loop_footN-label when the condition is not met. The loop is repeated by the jump instruction jmp loop_headN at the end of the loop body.

4 MACHINE CODE TRANSLATION

This section shows the translation of Flounder IR resulting from plan translation to x86_64 machine code. The abstractions that were used to facilitate code generation in the previous step are now replaced with machine concepts.
A key challenge here is to replace virtual registers with machine registers and to manage spill memory locations for cases of insufficient registers. Finding optimal register allocations is an NP-hard problem and even the computation of approximations is expensive [7]. In the context of JIT compilers, linear scan has been proposed as a faster algorithm [19] and was adopted by LLVM. However, linear scan register allocation is still relatively expensive due to live range computations and increasing numbers of registers.

In this section, we present a much simpler technique that benefits from the explicit usage ranges marked in Flounder IR. In the following, we first show the machine register configuration used by the translator and then we show the algorithm to translate the lightweight abstractions.

### 4.1 Register Layout

We use a specific register layout for the machine code generated from Flounder IR. The layout is shown in Figure 5. We split the 16 integer registers of the x86_64 architecture into three categories.

We use twelve attribute registers at reg1, ..., at reg12 to carry attribute data and tuple ids. We use three temporary registers tmpReg1, tmpReg2 and tmpReg3, which are multi-purpose for accessing spill registers and constant loads. Lastly, we use the stack pointer rsp to store the stack offset. The stack base pointer rbp is repurposed for attribute data and not used for the stack.

### 4.2 Translation Algorithm

The translation algorithm translates Flounder IR to x86_64 assembly in one sequential pass over the code. It replaces the Flounder abstracts with machine instructions, machine registers, and stack access. The algorithm is shown in Figure 6.

When iterating over the IR elements, the algorithm keeps track of the number of in-use attribute registers (line 1) and the number of temporary registers per instruction (line 3). We describe the translation in three parts. The first part is register allocation, then the replacement of virtual operands with machine operands in instructions, and finally function calls.

**Register Allocation.** Register allocation is used to decide which virtual registers are stored in machine registers and which virtual registers are stored on the stack. Register allocation does not produce code directly, but it sets the allocation state for spill code and operand replacement. The procedure is illustrated below.

**Spill Code and Operand Replacement.** For each instruction, operands that use constant loads or virtual registers have to be replaced with machine-compatible operands. Virtual registers that were assigned with machine registers are simply swapped (line 21). For the other cases, the algorithm uses tmpReg1 to tmpReg3 to hold values temporarily per instruction. Three registers are sufficient for this purpose as this is the highest number of non-immediate operands per instruction. As an example, we look at the following instruction.
It reads an 8 byte value with the offset \( tid_os \) from the memory address \( 0x7f \)… and stores it in \( \{r_a\} \). The address is too large for an immediate operand and we assume for illustration purposes that both virtual registers \( \{r_a\} \) and \( \{tid_os\} \) are spilled.

The translator assigns temporary registers to each operand and emits spill code that exchanges values between spill slots and temporary registers. This is performed in pseudocode lines 16 to 26 and illustrated in the following.

```
mov \{r_a\}, [\{0x7fff5a8e39d8\}+(tid_os)]
```

The algorithm enumerates the virtual register accesses (lines 16-21) and the constant loads (lines 22-25) from the instruction. It assigns one of the temporary registers tmpReg1 to tmpReg3 to each. In step 1 the translator assigns tmpReg1 \( \{r_ax\} \) to the operand \( \{r_a\} \). This is the only output operand of the instruction and the operator emits a store to \( \{r_a\} \)’s spill slot on the stack. Step 2 assigns tmpReg2 \( \{r_bx\} \) to the operand \( \{tid_os\} \). The translator emits a load to retrieve the value from its spill slot. Step 3 assigns tmpReg3 \( \{rcx\} \) to the constant load of address \( 0x7f \)…. The translator emits a load for the constant. This results in the following machine code sequence, which includes the original mov instruction with replaced operands.

```
mov rbx, [rsp-24] ;load spill tid_os
mov rax, [0x7fff5a8e39d8] ;load constant
mov rbx, [rcx+rbx] ;instruction
mov [rsp-8], rax ;store spill r_a
```

Call Conventions. The mcall pseudo-instruction is replaced with an instruction sequence that realizes the call convention. To perform the function call, it is necessary to follow the x86_64 call convention, which includes saving 7 caller-save registers, setting up to 7 parameter registers, retrieving the return value of the function, and restoring the caller-save registers [15].

The pseudo-instruction is translated to machine assembly in lines 13 to 14 of the pseudocode. At this point, the machine register allocation to the point of the call is known. We can take the register usage into account to produce only minimal code for realizing the call convention. Thus we avoid unnecessary work, for instance we do not preserve registers that are not assigned to any virtual registers. Further we do not preserve the values held by temporary registers.

5 EVALUATION

This section evaluates our approach of using a simple IR specialized to relational workloads over a general purpose IR. We analyze the compilation times and the machine code quality for different compilation techniques using Flounder IR and LLVM IR.

Flounder IR. Translation from query plan to IR is similar for both backends and follows the produce/consume model [16]. The translation from Flounder IR to machine assembly is performed with the algorithm from Figure 6. Then we use the AsmJit library [12] to emit the binary representation to avoid the overhead of running external assemblers, e.g. nasm.

For LLVM IR, the machine code is generated by the LLVM library’s JIT functionality. We use O0 and O3 optimization levels for tradeoffs between compilation time and code quality.

Workload Design. We use four query templates that vary data size and query complexity. The templates are specified in an SQL-form, which uses additional integer parameters (cf. Figure 7). The parameter \( l \) varies the data size in \( \text{Q}_l \). Parameters \( p \) and \( s \) vary query complexity in \( \text{Q}_p \), \( \text{Q}_s \), and \( \text{Q}_\sigma \) respectively. The attribute data is generated from uniform random distributions with the following relation sizes: \( \text{Q}_0 \) has \( l \) tuples for \( r \), \( \text{Q}_s \) has \( 1\ M \) tuples, \( \text{Q}_p \) has \( 10\ K \) tuples per join relation, and \( \text{Q}_\sigma \) has \( 1\ M \) tuples.

Execution Platform. We use a system with Intel(R) Xeon E5-1607 v2 CPU with 3.00 GHz and 32 GB main memory. The experiments run in one thread. We use operating system Ubuntu 18.04.4 and clang++ 6.0.0 to compile the query compiler and the library for JIT queries. The LLVM backend uses LLVM 6.0.0.

5.1 Compilation Times

We compare the machine code compilation times for LLVM and Flounder for \( \text{Q}_p \) and \( \text{Q}_s \). We use \( \text{Q}_\sigma \) with values of \( p \) to project 50 to an extreme case 500 attributes (filter with selectivity 1%). We use \( \text{Q}_s \) with values of \( j \) to join 2 to 100 relations. We show the results for Flounder, LLVM-O0, and LLVM-O3 in Figure 8.

Observations. For all techniques, the compilation times increase with the query complexity. The compilation times for \( \text{Q}_\sigma \) are higher (up to 657 ms) than for \( \text{Q}_s \) (up to 560 ms) and we look in detail at \( \text{Q}_s \). With O0 optimization LLVM has compilation times between 10 ms up to 265 ms. With O3 compilation times range from 28 ms up to 657 ms. For both levels, the graphs show super-linear growth of compilation times with query complexity. Flounder shows lower compilation times that scale linearly between 0.3 ms to 10.8 ms. The highest factor of improvement is 24.6x over LLVM-O0 and 60.9x over LLVM-O3 (both for 100 join relations). For Flounder has
very low compilations times ranging from 0.1 ms (50 attributes) to 0.6 ms (500 attributes). This leads to factors of improvement up to 933x over llvm-O3. We attribute this to the time LLVM uses for register allocation of all virtual registers holding attributes.

### 5.2 Machine Code Quality

To evaluate machine code quality, we execute two configurations of each query template and measure the execution time and the number of executed instructions. The results are shown in Figure 9. The bars show the execution time in milliseconds and the number on top shows the executed instructions in millions.

**Register Allocation.** We analyze the effect of our register allocation strategy on machine code quality. To this end we look at the techniques Flounder (spill) and Flounder. The former uses spill access for every virtual register use. The latter allocates machine registers with the translation algorithm. We observe that register allocation reduces the number of executed instruction by factors between 1.2x and 1.8x (with one exception). This shows that our register allocation strategy effectively reduces the amount of executed spill code. We explain the lack of improvement for Q\textsubscript{at} j = 25 with a large number of hash table operations, which execute invariant library code. The results show that the register allocation technique reduces execution times for all queries by factors between 1.02x to 1.35x. The factors are not as high as the factors between L1 memory and register access. This is because the memory access for reading relation data limits throughput (as is typical for database workloads). The improvements shown by the experiment are due to faster machine register access and execution of less spill code.

**Comparison with LLVM.** Next we compare the machine code quality of Flounder and LLVM (cf. Figure 9). On average llvm-O0 executes 1.4x less instructions than Flounder. The execution times, however, are similar and are longer for Flounder only by an average factor of 1.01x. With regard to execution times the machine code quality resulting from Flounder is similar to llvm-O0. We attribute the small time difference despite the higher instruction count to memory bound execution.

llvm-O3 executes 2.2x less instructions than Flounder on average. The average factor between the execution times of 1.05x is still low. However, especially queries on larger datasets benefit from the optimizations applied by llvm-O3. E.g., the larger variant Q\textsubscript{at} 1 M executes 1.3x faster. We conclude that despite the much shorter translation times, our compilation strategy produces code with competitive performance to LLVM’s code.

### 5.3 Overall Performance

We show a table with overall performance for each technique in Figure 10. The workloads are the same as in Section 5.2 with two configurations for each template. The relation sizes range from 10 K to 1 M tuples with total attribute numbers between 2 and 100.

**Observations.** The technique Flounder has overall execution times between 0.4 ms and 50.2 ms and llvm-O0 between 5.3 ms and 74.9 ms. For llvm-O0, compilation makes up 46% of the execution on average. For Flounder the average is 5%. This leads to better performance of Flounder for 7 of 8 queries. For Q\textsubscript{at} l = 1 M compilation times are generally low; thus llvm-O0 achieves a slightly shorter overall time due to 1.15x faster execution. The technique llvm-O3 has execution times between 11.4 ms and 141.9 ms, which is longer than the other techniques for 7 of 8 queries. The compilation times make up a high percentage of 62% of the overall on average. The highest factor of improvement of Flounder over llvm-O0 is 10.7x. The highest factor of improvement over llvm-O3 is 23.2x.

### 6 SUMMARY

We showed a query compilation technique that includes all machine code generation steps in the query compiler. The technique uses an intermediate representation with abstractions that enable simple translation of query plans to IR and fast translation from IR to machine code. While the translation of query plans to IR is similar to existing approaches, the next step, translation to machine code, is much simpler than in existing techniques. Compared to established low-level query compilers, our approach achieves much shorter compilation times and provides more control over the resulting machine code.
REFERENCES