Machine Code Caching in PostgreSQL Query JIT-compiler

Michael Pantilimonov[1]
pantlimon@ispras.ru

Ruben Buchatskiy[1]
ruben@ispras.ru

Roman Zhuykov[1]
zhroma@ispras.ru

Eugene Sharygin[1,2]
eush@ispras.ru

Dmitry Melnik[1]
dm@ispras.ru

[1]*Ivannikov Institute for System Programming of the RAS, Moscow, Russia.*
[2]*Lomonosov Moscow State University, Moscow, Russia.*

***Abstract*—As the efficiency of main and external memory grows, alongside with decreasing hardware costs, the performance of database management systems (DBMS) on certain kinds of queries is more determined by CPU characteristics and the way it is utilized. Relational DBMS utilize diverse execution models to run SQL queries. Those models have different properties, but in either way suffer from substantial overhead during query plan interpretation. The overhead comes from indirect calls to handler functions, runtime checks and large number of branch instructions. One way to solve this problem is dynamic query compilation that is reasonable only in those cases when query interpretation time is larger than the time of compilation and optimized machine code execution. This requirement can be satisfied only when the amount of data to be processed is large enough. If query interpretation takes milliseconds to finish, then the cost of dynamic compilation can be hundreds of times more than the execution time of generated machine code. To pay off the cost of dynamic compilation, the generated machine code has to be reused in subsequent executions, thus saving the cost of code compilation and optimization.**

In this paper, we examine the method of machine code caching in our query JIT-compiler for DBMS PostgreSQL. The proposed method allows us to eliminate compilation overhead. The results show that dynamic compilation of queries with machine code caching feature gives a significant speedup on OLTP queries.

***Index Terms*—dynamic compilation, JIT-compilation, machine code caching, query execution, DBMS, PostgreSQL, LLVM**

I. Introduction

Traditionally database management systems translate a given query into a logical query plan first, which is typically represented by a tree of extended relational algebra operators and then into a physical one by adding metainformation about data access methods and algorithms that implement relational operations. The constructed physical plan is then interpreted using the specified execution model. A classic example of the latter is the iterator model, also known as the Volcano model [1]. Each algebraic operator of the iterator model converts input data into output stream of tuples that is controlled by the *next()* function. This abstraction is simple to understand and easy to implement, but inefficiently uses resources of modern CPU. There are other execution models [2, 3, 4] that try to mitigate the shortcomings of the iterator model in different ways. Regardless of the execution model used, the conventional method of query execution involves the overhead of invoking virtual functions during interpretation of plan that frequently consists of an arbitrary sequence of operators, expressions and predicates. These virtual calls, in turn, generate a significant number of false branch predictions. Besides, interpretation approach generally involves a considerable amount of redundant checks that always take certain branches in a particular query plan, thus spending additional CPU cycles to no purpose. Increasingly, a method of dynamic compilation is used to generate specialized machine code for a given query plan to eliminate these drawbacks. Performance improvement is achieved due to function inlining, constant propagation, common subexpression elimination, replacement of indirect calls with direct ones, elimination of unreachable (dead code) from the point of the supplied query plan instructions, etc.

In essence, the dynamic compilation method involves replacement of interpretation time $t_I(N)$ with the sum $t_C + t_E(N)$, where N is the size of data processed by a query, t_C is compilation time and t_E is execution time. The optimizations performed during compilation make the resulting compiled code more efficient: $t_E(N) < t_I(N)$, but to justify the costs, it is necessary that $t_C + t_E(N) < t_I(N)$, that is the time spent on interpretation of a query plan exceeds the time spent on both compilation and execution of optimized machine code. This requirement can only be satisfied if the amount of data processed by a query is large enough. Thus, only OLAP [5] queries that operate on a significant amount of data, e.g. from TPC-H [6] benchmark, can level compilation time and allow one to obtain performance speedup. In case of OLTP [7] queries, for example TPC-B [8] benchmark, where a small amount of data is processed and the interpretation approach usually takes microseconds to execute, dynamic compilation may be unacceptable due to long optimization and compilation time. To overcome this problem and get rid of

This work is supported by RFBR grant 17-07-00759 A.

978-1-7281-4623-2/19/$31.00 ©2019 IEEE
DOI 10.1109/IVMEM.2019.00009

the time spent on dynamic compilation, it is necessary to save generated machine code for a query, so it can be reused many times. However, simply saving machine code in not enough since variable's values and structure's addresses in dynamic memory change on every query execution. Therefore, code generation must be performed in such way that patches can be applied to saved machine code, thus making it possible to update old values and addresses.

In this paper, we consider a method for saving and reusing machine code generated by a dynamic query compiler to reduce the overhead spent on query compilation. The work is performed using the LLVM compiler infrastructure [9] in the dynamic query compiler [10, 11, 12] developed at ISP RAS [13] for the PostgreSQL [14] RDBMS.

II. Current Approaches to Query Plan Caching

Most modern proprietary RDBMSes, such as MSSQL (Microsoft), DB2 (IBM), and Oracle Database (Oracle) cache plans of executed queries in automatic mode without any external interference. Saved query plans are typically located in shared memory that can be implemented in several ways depending on the utilized process model. Thus, information about query plans is available to all processes or threads that serve client connections and allows, in general, to reduce the response time of the entire system, increase its performance and throughput by reducing the overhead of operations associated with the construction of a physical plan for frequently used queries.

However, the automatic shared memory caching mechanism has its drawbacks that are mainly related to the complexity of access synchronization and greater overall sensitivity of the system to suboptimal query plans. The latter has much lower impact in the RDBMSes with per-process or per-thread local caching such as PostgreSQL and MySQL. Cached query plan may not be optimally constructed by the optimizer due to insufficient statistical information, skew data distribution, a large number of JOIN operations, the complex structure of the query itself, use of stored functions or procedures, etc. A situation may also arise that an optimally constructed query plan in the first place has lost its relevance after a certain number of committed modifications in a database, which could change the original data distribution that was used by the optimizer to build the initial plan. Until this query plan is rebuilt, all DBMS clients won't effectively utilize system resources. To solve the problem vendors employ different approaches: automatically rebuild cached query plan after exceeding a data modification threshold in a database object — table or index; collect statistics at query run time to evaluate the relevance of approximate estimates made by the optimizer at the time of initial plan construction; introduce adaptive query plans that can switch the operator's underlying algorithm depending on the amount of data received, e.g. replace nested loop join with hash join, etc.

Basically, the task of query plan dynamic compilation slightly intersects with the problem of its optimal construction and can be solved independently. Due to the fact that dynamic code generation, its optimization and compilation are resource-intensive operations, amortization of the associated with them costs becomes an important objective. The latter, in case of local caching, can't be solved efficiently as each process has access only to its own saved plans. In the worst case, each process may have an absolute copy of a set of frequently executed queries, each with its own dynamically compiled machine code. The total cost of this cache maintenance approach grows linearly both in memory used to store plan's machine code copy and in CPU cycles spent on its generation and compilation. Thus, to implement an effective caching mechanism of dynamically compiled query plans in PostgreSQL RDBMS that scales well, a process-local memory approach should be changed to the shared one. Nonetheless, for the goals pursued in this work, such a requirement is not mandatory and a dynamically generated machine code caching method can be examined independently.

III. SQL Query Processing Pipeline

The basic algorithm of query execution in relational DBMS consists of the following steps:

1) Lexical and syntactic analysis. At this stage, lexer and parser process a client's input string and produce a parse tree. During the analysis, only syntax checking is performed, but not the semantics. For example, if a query refers to a table that doesn't exist in the database system catalog then an error won't be thrown.
2) Semantic analysis. The parse tree from the previous phase goes through semantic analysis that results in a query tree augmented with various metainformation: system identifiers of tables, types and serial numbers of requested attributes, a list of joined tables, predicates, and expressions in the form of a tree, etc.
3) System of rewrite rules. Next, a search is made in the system catalog to find the rules applicable to the query tree and having found the appropriate ones, the transformations described in the body of the rule are applied. An example of a transformation is the substitution of calls to the views, also known as virtual tables, with the calls to the base tables from the view definition.
4) Planning and optimization. The planner receives a rewritten query tree and using the auxiliary data structure, that represents the simplified scheme of the plan, selects the most efficient query execution path in terms of cost estimates and available statistical information. Then it chooses an optimal data access methods with a given order of joins and algorithms for their implementation. The final path transforms into a complete query plan tree.
5) Execution phase. The executor performs a recursive traversal of the plan tree and runs the encapsulated logic of the corresponding operator node, interprets expression and predicate trees. The resulting set of rows is then sent to the client.

Most RDBMSes, including PostgreSQL, use the described pipeline to translate a client's query into a physical query plan

```
Sort
 Sort Key: l_returnflag , l_linestatus
 -> HashAggregate
     Group Key: l_returnflag , l_linestatus
     -> Seq Scan on lineitem
         Filter: (
             l_shipdate <=
             '1998-09-29 00:00:00'::timestamp
             without time zone)
```

Fig. 1: TPC-H Q1 query plan in PostgreSQL.

that is suitable for execution. In case of dynamic compilation, the overhead associated with the process of code generation, optimization and compilation adds to the last phase. To justify these expenses, the total query execution time in interpretation mode should significantly exceed the time spent on its dynamic compilation.

IV. Motivation of Machine Code Caching

Fig. 1 shows the plan built by PostgreSQL optimizer for Q1 query from TPC-H benchmark in the database generated with SCALE 2 and following types replacement: CHAR(1) to ENUM, NUMERIC to DOUBLE PRECISION. In the query Q1 the sequential scan operator calculates predicate for each tuple of *lineitem* table, that consists of approximately 12 million tuples. The average interpretation time of this query on the machine with an Intel Core I7-6700HQ CPU and database located in the main memory is 7.7 seconds. The average total execution time of a dynamically compiled version of the same query is 2.1 seconds, where optimization of the generated LLVM IR code takes 350 ms, 280 ms its compilation and 1.4 seconds execution of the resulting machine code. Thus, this query interpretation time significantly exceeds the overhead spent on dynamic compilation and execution of the generated machine code. The quality of machine code, in conjunction with the size of the processed data, has a positive effect on the total execution time, which justifies the resources spent on its generation.

The opposite situation can be seen in the following simple query: *select * from orders where o_custkey = 102022 and o_orderdate between date '1992-11-01' and date '1994-01-01'*, with its plan shown in Fig. 2. The query retrieves customer's orders from the corresponding table using index access method and processes a small number of tuples. It's average interpretation time on the same machine is approximately 0.110 ms. In case of dynamic compilation, the total time is about 120 ms, and execution time of the generated machine code is only 0.030 ms. Therefore, dynamic compilation time of this query is hundreds of times greater than its execution time.

We conclude that the cost of OLTP queries dynamic compilation is extremely high and to justify it, it is necessary to reuse the generated machine code in subsequent executions, therefore eliminating the costly operations of optimization and compilation. In general, machine code should be saved for the

```
Index Scan using i_o_custkey on orders
 Index Cond: (o_custkey = 102022)
 Filter: ((o_orderdate >= '1992-11-01'::date)
     AND (o_orderdate <= '1994-01-01'::date))
```

Fig. 2: Query plan in PostgreSQL for OLTP type query.

queries that a DBMS independently caches to minimize the costs associated with preliminary stages of its execution, that described previously in section III. In the query plan illustrated in Fig. 2 literal constant values are used to calculate the result, which significantly reduces the likelihood of subsequent reuse of this query plan with identical arguments.

The PostgreSQL DBMS doesn't have functionality for automatic query caching, but provides a mechanism to manually save the query plan into local memory of a back-end process. Therefore, this mechanism can be used to cache dynamically generated machine code alongside with a specific query plan.

V. Machine Code Caching

A. Query caching in PostgreSQL

PostgreSQL provides a manual mechanism to cache the query using the PREPARE [15] command. The latter creates a prepared operator object for a client's query on the server-side that goes through the first three processing stages described in section III: parsing, semantic analysis, and system of rules. The pipeline's output — a prepared statement, saved into local memory of the process that serves a client connection. Subsequent work with the prepared object is carried out by the EXECUTE [16] command that can also transmit the actual values of parameters if they were specified in query's definition. Before a stored operator object can be executed, the optimizer generates the best query plan depending on the passed parameters. Thus, the reuse of the prepared statement allows to level the overhead spent on the first three stages of the query processing.

A prepared statement can also use a generic plan rather than rebuilding it for each set of supplied values. For a prepared statement without parameters this happens immediately; otherwise, the generic plan is selected after five or more executions that produce the plans whose estimated average cost is more expensive. Once a generic plan is chosen, it is used for the remaining lifetime of the prepared statement. The generic query plan eliminates the overhead associated with all stages of the query processing, except its execution.

Fig. 3 shows an example of a generic plan for the operator object created with the corresponding PREPARE statement. The PostgreSQL optimizer, using the accumulated statistics for the first 5 executions, calculated that the cost of a generic plan for a given query is less than the cost of re-planning it for each set of supplied parameters. The values $1, $2 and $3 refer to parameters specified in the PREPARE command by position and affect the result of predicates evaluation.

The PREPARE mechanism allows one to save an optimized generic query plan for its subsequent reuse without repeat-

```
PREPARE q1(int, date, date) as
  select * from orders
  where o_custkey = $1
    and o_orderdate between $2 and $3;
```

```
Index Scan using i_o_custkey on orders
 Index Cond: (o_custkey = $1)
 Filter: ((o_orderdate >= $2) AND (o_orderdate <= $3))
```

Fig. 3: Prepare statement and its generic query plan.

edly consuming CPU cycles in standard processing pipeline. Overall, this feature was used as basis in our dynamically compiled machine code cache implementation. To do this we also extend the existing PostgreSQL structures: *QueryDesc* and *CachedPlan*, so that the JIT-compiler can save a pointer to the memory location with generated machine code next to the cached generic query plan. In addition, before machine code can be executed, a preliminary patching is required to update the memory location of PostgreSQL variables that we use in the process of code generation, as their absolute addresses change on each query execution.

B. Machine Code Patching

To repeatedly execute generic plan machine code, it is necessary to have metainformation to establish the correspondence between LLVM intermediate representation (IR) at the time of code generation and resulting machine code. This kind of metainformation can be used to modify and patch dynamically generated instructions afterwards. To solve this task LLVM infrastructure provides tools for controlling and modifying the machine code generated by the MCJIT [17] component — llvm.experimental.stackmap and llvm.experimental.patchpoint intrinsics. During compilation process of LLVM IR into machine code both intrinsics create a special data section containing the Stack Map [18] structure. This structure stores a relative offset from the start of function in the machine code, where a call to stackmap or patchpoint gets, as well as location (stack slot, register name, constant, etc.) of all values, passed to these intrinsics as parameters. Intrinsic llvm.experimental.patchpoint, in addition to the same parameters as llvm.experimental.stackmap, also accepts the address of the function. During a compilation phase of the llvm.experimental.patchpoint, a call to this function is inserted into the generated code following specified calling convention. A called object can be modified later using metainformation from the Stack Map structure.

To implement the machine code caching method in the dynamic query compiler only the llvm.experimental.patchpoint intrinsic is used. The signature of the intrinsic is as follows:

```
declare i64 @llvm.experimental.patchpoint.i64(
    i64 <id>, i32 <numBytes>,
    i8* <target>, i32 <numArgs>, ...)
```

During a code generation phase, the emitted LLVM IR instructions use data from PostgreSQL run-time structures located in dynamic memory at some absolute addresses, which, in turn, lose relevance after each iteration of query execution. Thus, before starting the next execution of the saved machine code, it is necessary to update previously injected absolute addresses. To accomplish this task, for each memory access to the PostgreSQL data structure field the intrinsic call:

```
llvm.experimental.patchpoint(
    ID, 13, 0x1234567890abcdef, 0)
```

is generated, where ID is a unique identifier, 13 is number of reserved bytes, 0x1234567890abcdef — address of the function that generated code will call and 0 — number of its parameters.

Fig. 4 shows an example of code generation using LLVM C API, its resulting representation in LLVM IR and machine code. In this example the *tts_nvalid* field belongs to the *TupleTableSlot* structure, which is used in PostgreSQL to represent different types of tuples. This data structure is allocated and released each time the query plan is executed, even when manual caching with PREPARE command is used. The address of the *tts_nvalid* field in this code generation iteration is 0x55b5b3195148 and stored inside the GeneratePatchpoint function into the global array llvm_pp[] with llvm_pp_n index. The latter is then used as an ID argument of llvm.experimental.patchpoint intrinsic. Subsequently, the index passed to the intrinsic function will be saved to the StkMapRecord structure inside the Stack Map as a PatchPoint ID. Upon further analysis of the Stack Map structure, the retrieved index value from the PatchPoint ID field of the StkMapRecord structure will allow one to extract the corresponding address from the llvm_pp array and use it to modify the code. The result of the patching is shown in Fig. 4d: the target address of called function is replaced with the address of *tts_nvalid* field, %r11 register to %rax, the callq to nop (xchg %ax, %ax). The patched code preserves semantics in terms of further instructions that use %rax register value.

C. Machine Code Patching Implementation in JIT-Compiler

To generate code, the dynamic query compiler with modified execution model traverses plan tree in direct order and calls generator-functions for each operator. Each operator's generator-functions are implemented using LLVM C API and generate the LLVM IR code, that implements its algebraic model.

While implementing the machine code caching method, the primary task was to prevent logic duplication and to preserve the existing code generation algorithm with minimal changes to simplify the further development process. Thus, we made the following changes:

- Introduce global code generation mode — the llvm_patchpoint variable, which defines the behavior of LLVM C API wrapper functions, GeneratePatchpoint function, etc.
- All LLVM C API functions used in code generation are wrapped in wrapper functions with the same name and additional prefix. Those functions return a different result

```
static LLVMValueRef
top_level_consume_codegen(
  LLVMModuleRef mod, LLVMBuilderRef builder, ...) {
  ...
  LLVMValueRef slot_nvalid_ptr;
  __LLVMPositionBuilderAtEnd(builder, entry_bb);
  slot_nvalid_ptr = GeneratePatchpoint(builder,
    __LLVMPointerType(__LLVMInt32TypeInContext(llvm_ctx),0),
    &inputslot->tupleslot->tts_nvalid);
  __LLVMBuildStore(builder,
    __LLVMConstNull(__LLVMInt32TypeInContext(llvm_ctx)),
    slot_nvalid_ptr);
  ...
}
```

(a) Code function-generator with LLVM C API calls.

```
define internal i32 @llvm_top_level_consume() {
entry:
  %pp_ret = call i64 (i64, i32, i8*, i32, ...)
    @llvm.experimental.patchpoint.i64(
      i64 0, i32 13,
      i8* inttoptr (i64 1311768467294899695 to i8*),
      i32 0)
  %pp_ret_pointer = inttoptr i64 %pp_ret to i32*
  store i32 0, i32* %pp_ret_pointer
  ...
  ret i32 0
}
```

(b) Generated LLVM IR code.

```
<main+1966>: movabs $0x1234567890abcdef, %r11
<main+1976>: callq  *%r11
<main+1979>: movl   $0x0, (%rax)
```

(c) Generated machine code.

```
<main+1966>: movabs $0x55b5b3195148, %rax
<main+1976>: data16 xchg %ax, %ax
<main+1979>: movl   $0x0, (%rax)
```

(d) Patched machine code.

Fig. 4: Example of code generation using llvm.experimental.patchpoint intrinsic.

depending on the global generation mode. An example of a wrapper function is shown in Fig. 5a.

- All LLVM C API LLVMConstIntToPtr() calls have been replaced with special GeneratePatchpoint function, which pseudocode is shown in Fig. 5b.

We list possible modes of code generation:

- One-time code generation: machine code will be used once. The value of llvm_patchpoint is 0.
- Patch-only: code generation is not performed. JIT-compiler collects new absolute addresses and updates the old ones. The value of llvm_patchpoint is 1.
- Code generation with patching option: during code generation, the absolute addresses are collected as in the patch-only mode; llvm.experimental.patchpoint intrinsic calls are generated. Upon completion of the code generation process, the created machine code is modified. The value of llvm_patchpoint is 2.

In one-time generation mode wrapper functions return the result of the corresponding LLVM function; GeneratePatchpoint() returns the result of the LLVMConstIntToPtr() call and doesn't save an address into the llvm_pp[] array.

During the query plan traversal in patch-only mode, calls to wrapper functions return NULL i.e. there is no code generation. Only GeneratePatchpoint() performs actual work by collecting new addresses and saving them into the global llvm_pp[] array. In the end, old addresses in the saved machine code are updated using the information from the Stack Map structure and llvm_pp[] array values.

In code generation mode with patching option, while the query plan is traversed, GeneratePatchpoint() saves the address-parameter into the llvm_pp[] array and returns the result of the llvm.experimental.patchpoint intrinsic call generation with a conversion to the pointer of the required type. At the end of generation, a one-time modification of the

```
static LLVMValueRef inline
__LLVMBuildStore(LLVMBuilderRef B, LLVMValueRef Val,
                 LLVMValueRef Ptr)
{
  Assert(llvm_patchpoint >= 0 && llvm_patchpoint < 3);
  if (llvm_patchpoint == 1) {
    Assert(B == NULL && Val == NULL && Ptr == NULL);
    return NULL;
  }
  else return LLVMBuildStore(B, Val, Ptr);
}
```

(a) Example of LLVM C API wrapper function.

```
LLVMValueRef
GeneratePatchpoint(LLVMBuilderRef builder,
                   LLVMTypeRef type, void *address)
{
  if (llvm_patchpoint == 0)
    return LLVMConstIntToPtr(builder, address, type);
  llvm_pp[llvm_pp_n] = (uintptr_t) address;
  if (llvm_patchpoint == 2)
  {
    args = { llvm_pp_n++, 13, 0x1234567890abcdef, 0 };
    ret = LLVMBuildCall(builder,
      "llvm.experimental.patchpoint.i64", args);
    return LLVMBuildIntToPtr(builder, ret, type);
  }
  llvm_pp_n++;
  return NULL;
}
```

(b) GeneratePatchpoint() pseudocode.

Fig. 5: Modifications of the code base to implement machine code caching.

instructions is performed and addresses are updated similarly to the previous mode.

The patching process is illustrated in Fig. 6 and performs as follows:

1) Each StkSizeRecord element of the array is sequentially

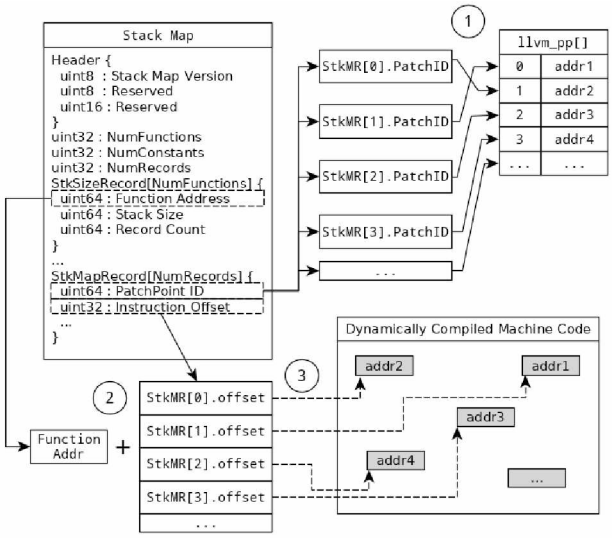

Fig. 6: Scheme of machine code patching.

extracted from the Stack Map structure. The former contains information about function address, its stack size and the number of records in StkMapRecord array that associates with this entry.

2) From each StkMapRecord record the PatchPoint ID value is retrieved and then used to search for a cell in the array of addresses. New addresses are already collected and saved in the array after query plan traversal. This operation corresponds to step 1 in Fig. 6.
3) The instruction offset is also extracted from StkMapRecord and together with the function address allows one to obtain a pointer to the memory location reserved during the generation of llvm.experimental.patchpoint. This operation corresponds to step 2 in Fig. 6.
4) At the last stage, that corresponds to step 3 in Fig. 6, patching using the new addresses from the llvm_pp[] array is performed.

VI. Results

We use Q1 query from TPC-H benchmark and OLTP queries shown in Appendix A to test generated machine code caching method. The TPC-H database was generated with SCALE 2 and following types modification: CHAR(1) replaced with ENUM, NUMERIC with DOUBLE PRECISION. This change allows one to use built-in LLVM types during dynamic compilation. The total size of the directory with database made up 6.4GB.

All experiments were conducted on an Intel i7-6700HQ Quad-Core CPU with a clock speed limit of 2.5GHz and 16GB of main memory, 64-bit Ubuntu 18.04 Linux operating system. During testing the database was completely located in RAM. The performance comparison of the interpreter and JIT-compiler was done using PostgreSQL 9.6.3 and LLVM 4.0.

To collect execution statistics of OLTP queries the pgbench [19] program of PostgreSQL was used. For each query we perform several pgbench launches in the following way:

```
pgbench –n –M prepared –t 10000
        –l –f q[1, 2, 3].script –z 1
```

, where z is the flag added by modifying the source code that initialize the random generator and q[1, 2, 3].script is query text file. The average number of transactions per second was obtained by pgbench program. The average execution time was calculated based on the log files generated by pgbench. The benchmark results of 10000 transactions per each query are shown in Table. I. In these experiments the pgbench program uses the caching mechanism described in section V-A. Dynamic compilation of query with the possibility of machine code reuse is performed at the time the PostgreSQL optimizer creates a generic plan, i.e. during the 6th transaction execution. The cached machine code is then reused in all subsequent query executions. Thus, the average number of transactions per second calculated by the pgbench utility includes the costs associated with dynamic compilation at 6th transaction. Not taking into the account the first 6 transactions, Q2 with the most expressions achieved maximum average acceleration by 1.78 times.

In addition to OLTP experiments, we perform a comparative testing of Q1 query from TPC-H benchmark to analyze the effect of the llvm.experimental.patchpoint intrinsic on the quality of the resulting machine code. The obtained results are presented in Table II. The average execution time of Q1 using the PostgreSQL interpreter is 10 seconds. A dynamically compiled version of the same query on average takes 2.65 seconds to finish, where 820 milliseconds are spent on the optimization and compilation, 1.73 seconds on the machine code execution. The dynamic compilation of the prepared query plan with patchpoints totals 3.4 seconds and its average execution time in all iterations is 2.4 seconds. In case of prepared query plan with cached machine code, the overhead of compilation and optimization in all iterations after preparation is close to 0.

Thus, the performance of the generated machine code using llvm.experimental.patchpoint is on average 38% less than the result of one-time code generation. This is because the use of llvm.experimental.patchpoint limits the compiler's optimizations over LLVM IR.

VII. Conclusion

In this work we have develop a method to save and reuse machine code generated by the dynamic query compiler, that makes it possible to level the overhead associated with its creation. The ability to reuse the generated machine code allows one to apply the dynamic compilation to OLTP queries, which interpretation time is measured in milliseconds.

The proposed method is implemented in the PostgreSQL DBMS dynamic query compiler using the LLVM Stack Map technology. The experiments have shown that dynamic query compilation using the LLVM engine, together with the possibility to reuse generated machine code, allows one to obtain

TABLE I: OLTP performance comparison of interpreter and JIT-compiler with machine code caching.

Description	Q1		Q2		Q3	
	PG	JIT	PG	JIT	PG	JIT
Avg TPS (more — better)	70,67	72,38	105,25	183,71	145,37	199,83
Generic plan compilation on 6th iteration + execution, ms	-	1342,5	-	1118,9	-	997,2
Avg execution time except first 6 iterations, ms	14,12	13,65	9,49	5,31	6,87	4,89
Avg improvement except first 6 iterations, X times	1,03		1,78		1,40	

TABLE II: Performance comparison of query Q1 from TPC-H benchmark in interpreter and JIT-compiler in different modes.

PG	JIT, one-time code generation			JIT with machine code caching		
	compilation + optimization	execution	sum	compilation + optimization	execution	sum
10 s	(370 + 450) ms	1,73 s	2,65 s	(380 + 560) ms	2,4 s	3,4 s
				0,140 ms	2,4 s	2,4 s

considerable performance improvement on OLTP queries with moderate number of expressions.

In the future we are planning to extend the existing caching mechanism and automatically save frequently executed queries of similar type using cost estimates and other heuristics.

References

[1] G. Graefe, "Volcano - An Extensible and Parallel Query Evaluation System," *IEEE Trans. Knowl. Data Eng.*, vol. 6, no. 1, pp. 120–135, 1994. DOI: 10.1109/69.273032. [Online]. Available: https://doi.org/10.1109/69.273032.

[2] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct," *PVLDB*, vol. 2, no. 2, pp. 1648–1653, 2009. DOI: 10.14778/1687553.1687618. [Online]. Available: http://www.vldb.org/pvldb/2/vldb09-10years.pdf.

[3] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran, "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures," in *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, 2001, pp. 567–574. DOI: 10.1109/ICDE.2001.914871. [Online]. Available: https://doi.org/10.1109/ICDE.2001.914871.

[4] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *PVLDB*, vol. 4, no. 9, pp. 539–550, 2011. DOI: 10.14778/2002938.2002940. [Online]. Available: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[5] A. Andreev. (2010). Classification of OLAP systems of the form xOLAP, [Online]. Available: http://citforum.ru/consulting/BI/xolap_classification.

[6] *TPC-H, Decision Support Benchmark*. [Online]. Available: http://www.tpc.org/tpch.

[7] *What is an OLTP System?* [Online]. Available: https://docs.oracle.com/database/121/VLDBG/GUID-0BC75680-5BD4-43A9-826F-CD8837D30EB2.htm#VLDBG1367.

[8] *TPC-B, transactions throughput benchmark*. [Online]. Available: http://www.tpc.org/tpcb.

[9] The LLVM Foundation, *The LLVM Compiler Infrastructure*. [Online]. Available: https://llvm.org.

[10] E. Sharygin, R. Buchatskiy, L. Skvortsov, R. Zhuykov, and D. Melnik, "Dynamic compilation of expressions in SQL queries for PostgreSQL," *Proceedings of the Institute for System Programming of the RAS*, vol. 28, no. 4, pp. 217–240, 2016. DOI: 10.15514/ispras-2016-28(4)-13. [Online]. Available: https://doi.org/10.15514%2Fispras-2016-28%284%29-13.

[11] R. Buchatskiy, E. Sharygin, L. Skvortsov, R. Zhuykov, D. Melnik, and R. Baev, "Dynamic compilation of SQL queries for PostgreSQL," *Proceedings of the Institute for System Programming of the RAS*, vol. 28, no. 6, pp. 37–48, 2016. DOI: 10.15514/ispras-2016-28(6)-3. [Online]. Available: https://doi.org/10.15514%2Fispras-2016-28%286%29-3.

[12] E. Sharygin, R. Buchatskiy, R. Zhuykov, and A. Sher, "Runtime Specialization of PostgreSQL Query Executor," in *Lecture Notes in Computer Science*, Springer International Publishing, 2018, pp. 375–386. DOI: 10.1007/978-3-319-74313-4_27. [Online]. Available: https://doi.org/10.1007%2F978-3-319-74313-4_27.

[13] *ISP RAS*. [Online]. Available: https://www.ispras.ru/.
[14] *PostgreSQL RDBMS*. [Online]. Available: https://www.postgresql.org/.
[15] *PostgreSQL documentation: PREPARE command*. [Online]. Available: https://www.postgresql.org/docs/9.6/sql-prepare.html.
[16] *PostgreSQL documentation: EXECUTE command*. [Online]. Available: https://www.postgresql.org/docs/9.6/sql-execute.html.
[17] *MCJIT Design and Implementation*. [Online]. Available: https://releases.llvm.org/4.0.0/docs/MCJITDesignAndImplementation.html.
[18] *Stack maps and patch points in LLVM*. [Online]. Available: http://releases.llvm.org/4.0.0/docs/StackMaps.html.
[19] *PostgreSQL pgbench utility*. [Online]. Available: https://www.postgresql.org/docs/9.6/pgbench.html.

Appendix A
TPC-H based OLTP Test Queries

Q1

```
select c_custkey, c_name, c_phone, c_acctbal,
       o_orderstatus, o_totalprice, o_orderdate, o_clerk,
       l_linenumber, l_quantity, l_discount,
       l_tax, l_shipdate,
       ps_availqty, ps_supplycost, p_name,
       p_brand, p_retailprice,
       s_name, s_address, s_phone
from customer
  join orders on c_custkey = o_custkey
  join lineitem on l_orderkey = o_orderkey
  join partsupp on ps_partkey = l_partkey
               and ps_suppkey = l_suppkey
  join part on p_partkey = ps_partkey
  join supplier on s_suppkey = ps_suppkey
where c_custkey between :bid1 and :bid1 + 20
order by o_orderdate desc;
```

Q2

```
select l_returnflag, l_linestatus,
       sum(l_quantity), sum(l_extendedprice),
       sum(l_extendedprice * (1 - l_discount)),
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
       avg(l_quantity), avg(l_extendedprice), avg(l_discount),
       count(*) as count_order
from lineitem
where l_shipdate <= date '1998-12-01' - interval '105 days'
and l_partkey between :bid1 and :bid1 + 200
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Q3

```
select l_returnflag, l_linestatus,
       sum(l_quantity) as sum_qty,
       avg(l_discount) as avg_disc,
       count(*) as count_order
from lineitem
where l_shipdate <= date '1998-12-01' - interval '105 days'
and l_partkey between :bid1 and :bid1 + 200
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```