Application of Micro-Specialization to Query Evaluation Operators

Rui Zhang [1], Richard T. Snodgrass [2], and Saumya Debray [3]

Department of Computer Science, University of Arizona
{ [1]ruizhang, [2]rts, [3]debray}@cs.arizona.edu

***Abstract*—Relational database management systems support a wide variety of data types and operations. Such generality involves much branch condition checking, which introduces inefficiency within the query evaluation loop. We previously introduced *micro-specialization*, which improves performance by eliminating unnecessary branching statements and the actual code branches by exploiting invariants present during the query evaluation loop. In this paper, we show how to more aggressively apply micro-specialization to each individual operator within a query plan. Rather than interpreting the query plan, the DBMS dynamically rewrites its object code to produce executable code tailored to the particular query. We explore opportunities for applying micro-specialization to DBMSes, focusing on query evaluation. We show through an examination of program execution profiles that even with a simple query in which just a few operators are micro-specialized, significant performance improvement can be achieved.**

I. Introduction

Relational database management systems are by their nature general, in that they can handle whatever schema the user specifies and whatever query or modification is presented to them. Relational operators work on essentially any relation and must contend with predicates specified on any attribute of the underlying relations. Through such innovations as effective indexing structures, innovative concurrency control mechanisms, and sophisticated query optimization strategies, relational DBMSes are very efficient. Such generality and efficiency has enabled their proliferation and use in many domains.

This generality presents challenges to further increases in performance. For instance, a relation scan operator needs to be able to perform scans in both the forward and backward directions. In addition, when a predicate that involves the scanned attribute(s) is present, the scan operator needs to locate these attributes from each scanned tuple. When implemented in a generic fashion, the scan operator relies on multiple code branches and the associated branching statements to determine at runtime which code branch (corresponding to a particular scan direction) to execute. Such implementation also requires code to determine whether certain attributes need to be identified from each retrieved tuple, based on the presence of predicate(s). Nonetheless, once a query plan is generated, the scan direction and the presence of predicate are both invariants during the execution of the query. Thus the above condition checkings are rather unnecessary and can introduce significant overhead over the query evaluation loop.

Micro-specialization enables specialization to be applied at a finer granularity, that of on values presented within DBMSes [1]. The values on which to apply micro-specialization reside across various components within a DBMS. For instance, the values can appear in relation schemas and in query plans. In this paper, we focus the discussion on query plan-based specialization techniques. Specifically, we identify opportunities for applying micro-specialization on a query plan, essentially all of which is invariant during the query. We explain how micro-specialization of query evaluation, which requires dynamic object code manipulation, is applied. In particular, object code blocks corresponding to a plan operator node specialized for a particular query are instantiated and stitched together at runtime to produce the plan-specific executable code, termed a *bee*. We emphasize that the specialized code is exactly that present in the code of the DBMS, only without all the branching statements and code branches not needed for that particular query.

While the code blocks for the plan operators need to be provided by DBMS developers, the instantiation, invocation, and destruction of the specialized code are managed entirely by the DBMS. We thus consider the application of micro-specialization on a per-query basis to be a self-managed task performed by the DBMS.

In previous papers, we introduced the concept of micro-specialization and described implementation of this approach [1, 2]. In this paper, we make the following contributions.

- We focus on *query bees*, inserted into many kinds of specific query evaluation operators.
- We introduce the mechanism of *bee hot-swapping*, which extends the application of micro-specialization to not only invariants, but for variables that have a deterministic sequence of values; such variables are prominent in query bees.
- We provide more detailed *execution-profile analysis* of the performance benefits achieved by applying micro-specialization.

We first provide background about micro-specialization in the next section. We show that there are many specialization opportunities that can be exploited within the well-engineered PostgreSQL [3] DBMS and elaborate in detail on the newly identified micro-specialization opportunities in Section IV.

The following section addresses the mechanism of instantiating and invoking bees. In Section VI, we extend the applicability of micro-specialization beyond invariants during query evaluation, making this approach more widely applicable. Finally, we show that multiple micro-specializations in concert contribute to significant performance improvement to even a simple query.

II. Background

In a previous paper [1], we introduced the concept of micro-specialization along with a taxonomy of its application, which included query bees. We also introduced a prototype that integrated the generic bee module into PostgreSQL. We conducted an empirical evaluation of that prototype on both the TPC-H and TPC-C benchmarks. In a subsequent paper [2], we investigated bee-instantiation mechanisms for various kinds of bees on the timeline from compile time to runtime. We also proposed a sequence of steps needed to automate the application of micro-specialization, which can greatly reduce the complexity of applying this approach. We introduced the HIVE development environment to assist DBMS developers in realizing the envisioned automation.

In this paper, we narrow our focus to particular query evaluation operators, to aggressively apply micro-specialization given a particular query plan. We thereby extend the applicability and realizable benefits of micro-specialization. This paper also introduces the novel idea of bee hot-swapping, which extends the applicability of micro-specialization beyond invariants.

Applying micro-specialization requires one or more variables whose value are constant within the query evaluation loop to be identified. Micro-specialization then replaces a function or stretch of code with multiple copies, each particular to a single value of each of those variables. For instance, the code of a join operator needs to handle multiple join types. Applying micro-specialization on the join type will result in several version of the join, each tailored to one kind of join.

The application of micro-specialization consists of three stages. In the first stage, the generic source code is studied and converted to the corresponding specialized source code. The specialized source code will be compiled into object code in the second stage. Finally, the object code needs to be properly instantiated into executable form. We term the resulting executable code a *bee*, in that the specialized code is specific only to a particular scenario during execution. Hence the specialized code can be very small and efficient, resembling the characteristics of real bees. We term the object code generated in the second stage *proto-bee*, indicating that the actual executable bees are instantiated from such object code.

As a general approach, micro-specialization raises two central questions: *where* can micro-specialization be applied and *when* during the timeline from DBMS compilation to query evaluation can micro-specialization be done?

A. Where to Apply Micro-Specialization?

When a relation is defined, its *schema* can be viewed as an invariant. Consider the catalog look-up routines which rely on the schema. Instead of accessing the system catalog for every relation, many variables incorporated in the schema can be specialized on. In this case of relational schema, we specialize on each attribute's length, offset, alignment, and the presence of nullable attributes, as well as on the number of attributes in the relation.

Micro-specialization can be applied on internal data structures issued during query evaluation, for which some of the values in the data structure are constant during the evaluation loop of a query. For example, a query that involves predicate operators will utilize a `FuncExprState` data structure (a C `struct` defined in PostgreSQL) to encode the predicate. For the predicate `age <= 45`, this predicate data structure contains the ID of attribute `age`, the `<=` comparison operator, and the constant operand 45. We can thus apply specialization on these variables once we know the predicate from the query. The bees resulting from specializing such query-related data structures are thus termed *query bees*. In this paper, we further exploit the specialization opportunities in each individual query plan node. These two types of bees are differentiated by the kind of variable specialized upon to create the bee. By identifying values used by oft-executed code within the query evaluation loop, many query bees can be created. Each bee will independently speed up a subset of queries.

B. When Can Micro-Specialization be Applied?

Figure 1 depicts when individual bees of each kind are compiled and instantiated. Relation proto-bees and query proto-bees can be compiled at DBMS compilation time. For instance, for relation bees, type-specific code branches, such as value extraction code for specific types of attributes, can be compiled along with DBMS compilation. The produced proto-bees will then be used to instantiate relation bees. Query proto-bees can also be compiled at schema definition time in that at schema definition, additional information can be used to decide whether certain special cases of a query bee is needed. For instance, if a relation is empty, a query that references this relation should directly return. We will provide more detailed discussion in Section IV.

Relation bees are instantiated at relation-schema definition time, one for each newly-created relation. Individual query bees are instantiated immediately after query plan generation. Once we have a query plan, we know the particulars of the various data structures used in query evaluation, and so can generate the highly-specific code that uses these structures.

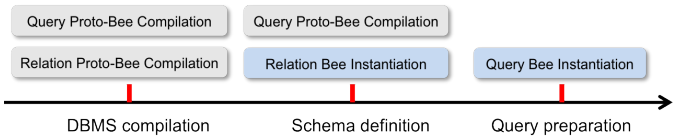

Fig. 1. *When* to Create Various Kinds of Bees

Each bee is the result of specializing on the *possible values* of a variable. Note though that a variable available to an earlier specialization is thus also available to a later specialization. So

for example a micro-specialization on an attribute's offset in a relation bee can be utilized in a query bee. Hence, as we travel to the right along the timeline from DBMS compilation to query evaluation, the number of variables available for micro-specialization accumulate, making later bees highly efficient.

Where bee creation resides along the timeline affects how efficient bee creation must be. Note however that we are not discussing the design of a bee. As we will see in the next section, the code to create the individual bees and to invoke bees is manually inserted into the DBMS before it is compiled. Here we are focusing on instantiation of individual bees, each containing specialized code resulting from knowing the exact values of the variable(s) evincing the specialization.

For relation bees, bee creation overhead is not critical. Hence, when instantiating a relation bee, we can invoke `gcc` to compile the specialized source code.

Because ad hoc queries need to be fast, the overhead of instantiating query bees needs to be minimized. Recall a query may require the join and predicate query bees. In the case of a join plan-operator, all possible kinds of the join, such as (left, semi, anti, etc.) can be enumerated and compiled into proto-bees at DBMS compilation time. At query preparation time, the associated join query bee is instantiated by selecting one of the pre-compiled join query proto-bees.

Delving down into the details, there are two ways that specialization affects query bee code. Some specializations, such as on the join type for a join query bee, affect the branch target of `if` statements in the code, thus resulting in multiple versions of a bee. Other specializations, such as the attribute ID for both the join and predicate bee, only affect constants in the bee. For the latter, the bee is effectively cloned each time, with different values substituted for the latter values; we will discuss the details of this dynamic object code manipulation mechanism in Section V.

It may seem that by instantiating individual bees, additional code is being added to the DBMS. In fact, the introduced code replaces the original code. Moreover, at run time a significant number of instructions can be eliminated in the specialized code, as we'll see in the next section.

III. Case Study

To illustrate in detail the mechanism of applying query bees, we provide a case study of *query14* from the TPC-H [4] benchmark as an example.

```
SELECT l_extendedprice * (1 - l_discount)
FROM lineitem, part
WHERE l_partkey = p_partkey
  AND l_shipdate >= date '1995-04-01'
  AND l_shipdate <
      date '1995-04-01' + interval '1' month
```

The original *query14* contains a complex `SELECT` clause containing aggregations. Given that we have not yet investigated micro-specialization with aggregation functions, we convert the `SELECT` statement into a simple attribute projection. We present the graphical query plan in Figure 2. As shown by the plan, the inner relation `part` is hashed into memory first. For each tuple fetched from the outer relation `lineitem`, hashjoin is performed against the inner hash table. If the join keys from both the inner and outer tuples match, the projected attributes are returned by the `SELECT` statement.

In applying query bees for this query, each plan operator requires a particular query bee that specializes that operator. For instance, a scan contains several runtime invariants, which include the scan direction and the presence of scan keys. Micro-specialization is applied on these values to produce multiple versions of the specialized scan operator (function), with each version handling a particular direction as well as the existence of scan keys. Similarly, micro-specialization is applied across this query plan to produce specialized code for each operator.

We ran this query in both a stock PostgreSQL and a bee-enabled PostgreSQL. When running the query, we ensured that the query plans generated by both DBMSes were identical. The running time (with a warm cache) was 1220 milliseconds for the former PostgreSQL and 961 milliseconds for the latter DBMS, respectively. The performance was improved by 21%.

IV. Kinds of Query Bees

We now elaborate on the details of how micro-specialization is applied on each plan operator for this query.

A. Scan Query Bee

As mentioned earlier, the generic implementation of the relation scan operator relies on branching statements to handle multiple possible cases. First, the direction of the scan can be forward, backward, or sometimes no movement. Second, when a relation scan is executed, a *scan key* is present when there is a predicate that is associated with one of the attributes in the scanned relation. Moreover, depending on whether a relation is empty, two code branches are implemented such that when the relation is empty, a direct return statement will be executed.

During the evaluation of an individual query, we found that the execution path is always unique. This means that these variables that are included in the branching statements are in fact invariants. For instance, the direction of a scan operation is not changed during the execution of a query; also, the presence of predicates determines whether the relevant scan-key processing is ever needed. In general, only a small portion of the generic code in the relation scan operator is executed for every query.

Based on such observation, we construct all the scan query proto-bees, each corresponding to a particular case. Given that each variable involves just two distinct values, a total of eight versions of the proto-bees are needed.

Removing these superfluous branching statements and the code branches themselves simultaneously decreases the code size and improves the execution efficiency of the code.

B. Hash Query Bee

When a hash operator is executed, it first extracts a physical tuple from the child plan node under this hash operator. Depending on the type of child plan node, this tuple can be

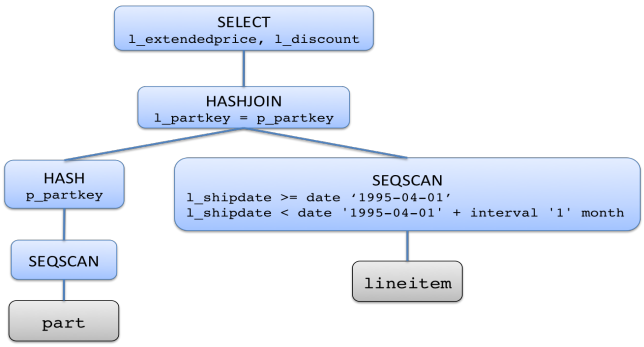

Fig. 2. A Query Plan Produced by PostgreSQL for the Example Query

directly fetched from a scan operator, returned as an inner tuple, or returned as an outer tuple. Concerning a specific hash operator in a query plan, the kind of its child plan node and hence the source of the associated tuples are invariants during the evaluation of the query. Furthermore, the number of attributes to be hashed from each tuple is also a constant which can incorporated into the query plan.

The tuple fetching code in the hash operator utilizes a `switch` statement to direct the tuples to be retrieved from the correct source. We eliminate the `switch` statement and construct three distinct versions of the hash query proto-bee.

Another specialization opportunity resides in hash computation. Hashing various types of values demands various computation algorithms. For instance, hashing a string and hashing an integer number requires two approaches that differ. An optimization already present in PostgreSQL for type-specific operations is to utilize function pointers. A function that performs string hashing can be associated with a hash operator in the form of a pointer during query plan generation. This approach eliminates the necessity of a rather inefficient `switch` statement that directs the execution to the appropriate code branch at runtime. Nevertheless, a function pointer can only be invoked by indirect function calls, which can become a significant overhead when accumulated in the query evaluation loop.

Instead of utilizing function pointers, we convert each such indirect invocation statement into a direct `CALL` instruction with a dummy target address associated. The dummy addresses are unique integer numbers that can be easily identified from the object code. At runtime, we substitute these magic numbers with the actual function pointers, which are essentially the addresses of the functions. We therefore replace indirect calls with direct function calls.

C. Hashjoin Query Bee

For a hashjoin operator, many scenarios need to be handled in the generic implementation. First, various types of joins, such as left-join, semi-join, and anti-join take different execution paths at runtime. Second, a hash join operator takes two tuples from the inner sub-plan and the outer sub-plan nodes, respectively. Each such node may require a different routine to fetch the associated tuples. For instance, an outer sub-plan can either be a hash operator or a scan operator. The type of sub-plan node is identified by a variable name `type` provided by the `PlanState` data structure in PostgreSQL. A dispatcher, which is essentially a complex `switch` statement, recognizes the sub-plan node and invokes the corresponding tuple fetching function. Furthermore, join-key comparison is another type-specific computation that involves the invocation of function pointers.

The type of join is determined by the query plan; the kinds of both inner and outer sub-plans are also invariants once the query plan is computed. Given that PostgreSQL defines eight types of joins, we construct eight versions of the hashjoin query proto-bee. We eliminate the dispatchers by again utilizing magic numbers which will be replaced by the addresses of the proper tuple-processing functions at runtime. Finally, we also convert the invocations of the join-key comparison functions into direct calls.

D. Predicate Query Bee

A predicate evaluation is similar to the join key comparison in that a predicate also involves type-specific comparison. We thus apply the same technique to produce the predicate query proto-bee with the comparison function's address as a placeholder. In addition, we found that a dispatcher is utilized to extract the constant operand, such as `'1995-04-01'` in the predicates each time a tuple is fetched from the `lineitem` relation. Instead of extracting this value every time, we tailor each predicate query bee to be specific to a single predicate operand by removing the value fetching code. Instead, for each predicate query bee, we substitute in the object code another magic number that represents the operand with the actual value. This new magic number is specified in the source code of the predicate query proto-bee as one of the input arguments to the predicate comparison function. The resulting code is effectively equivalent to that in which the value had been hardcoded in the predicate evaluation code.

V. Dynamically Specializing Query Bees

So far we have seen many places (operators) where micro-specialization is applicable. We now turn to the mechanism of instantiating and then combining several query bees to perform a particular query plan.

During query evaluation, once a query plan is determined, all its nodes will be invariant for the query evaluation. We insert statements that instantiate the actual query bees from the proto-bees. A query bee is instantiated simply by loading the proper version of the related proto-bee into an executable memory region. If a particular operator, such as a hashjoin operator, appears multiple times in a query plan, the hashjoin proto-bee needs to be instantiated several times, with each resulting bee associated with a distinct plan node.

Query bees are instantiated based on the selected proto-bees. The resulting query bees are *stitched* together to form the executable code for a query plan. All the versions of each proto-bee are stored in a pointer array, with each pointer referencing the start address of each version of the proto-bee code. To select the proper hashjoin proto-bee, we utilize the

```
if (!scan->rs_inited) {
  ...
  scan->rs_inited = true;
} else {
  ...
}
```

Listing 1. Code Excerpt of the Scan Operator

join type, which is an integer value ranging from 0 to 7 to index the corresponding versions. Take the hashjoin operator presented in Figure 2 as an example, once the proto-bee is selected, the magic numbers, as mentioned earlier, will be replaced with the correct addresses of the target functions. An actual query bee is thus instantiated. The instantiation step is in fact very similar to *dynamic linking*.

Given that the instantiation of all the query bees require just a few memory copies and several in-place memory updates, query bee instantiation is thus very efficient and incurs minimal overhead.

VI. Hot-Swapping Bees

To this point, we have focused on applying micro-specialization on invariants: a variable that takes on a constant value during query evaluation. We now generalize to variables that each take on a deterministic *sequence* of values during query evaluation.

As an example, let's examine the scan operator, in particular, the two code branches shown in Listing 1. The `rs_inited` variable indicates whether the scan operator has been initialized. In other words, the variable represents if the current tuple is the first one being fetched from the scanned relation within a query. This variable is then assigned to `true` for the rest of the query. By definition, this variable is strictly not an invariant. Nonetheless, due to the fact that this variable is known to be a constant right after the first tuple is fetched, evaluating the condition statement is redundant for the rest of the tuples. This is a simple example of a variable that takes on a sequence of values, here, `true` then `false`.

Hence, we produce two additional versions of the scan query proto-bees. The first version contains the first code branch in the above code and the second version contains code from the other code branch. Given that there are already eight versions of the scan query proto-bee, a total of 16 versions are now needed. However, one may notice that when a relation is empty, there is no need to know the scan direction and whether it is the first time to extract tuples. Therefore, careful examination of the relationships among the invariants can reduce the total number of distinct proto-bee versions.

Unlike the other bees whose object code is fixed after instantiation, an instantiated scan query bee is subject to self modification. We illustrate such mechanism with a call graph shown in Figure 3. In this figure, bees are represented as rectangles. In the stock implementation, the function `SeqNext` calls function `heap_getnext` to fetch the next tuple in the heap file. Function `heap_getnext` then calls a function namely `heapgettup_pagemode` to retrieve the actual tuple located on the currently scanned page. If it is the first time that `heap_getnext` is called, some initialization needs to be done. In Figure 3, `heapgettup_pagemode_init` is a bee representing the specialized version of `heapgettup_pagemode` with just the initialization code branch included. Similarly, `heapgettup_pagemode_regular` contains only the other code branch. During the execution of `heapgettup_pagemode_init`, the object code of the `heap_getnext` bee will be modified such that the original call to the `_init` version will be *hot-swapped* to the `_regular` version. Hot-swapping is simply done by the in-place update of a function call address, in this case, changing the call of the `_init` bee to a call to the `_regular` bee. From then on, the latter bee will be called. For a sequence of values, there will be multiple hot-swaps, each swaps-in a call to the next specialized version. Hot-swapping requires that the caller to the bees that are swapped-in to also be a bee, so that this caller bee can be modified.

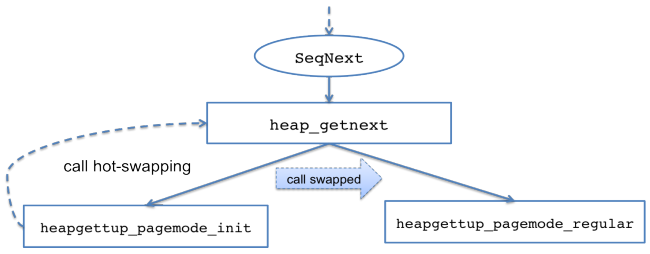

Fig. 3. Object Code Hot-Swapping

A more detailed study of the PostgreSQL source code revealed that the sort merge join operator can also benefit from such specialization. The sort merge join algorithm involves several states, each associated with a distinct code branch in a generic function. The execution of the sort merge algorithm switches among these states with each state being followed by a unique state. For instance, an `EXEC_MJ_JOINTUPLE` state is always followed by an `EXEC_MJ_NEXTINNER` state, meaning after a pair of tuples are joined, the next inner tuple is immediately needed.

We thus propose two simple rules indicating in what situation such specialization should be applied. First of all, there should only be a few distinct values associated with a variable that is used in branching statements. Second, each value is uniquely associated with another value such that the version of the bee to be invoked next is deterministic.

Dynamic object code manipulation raises a concern in a multi-threaded query execution environment: when a hot-swapping bee is invoked by multiple threads to update the object code of the bee, synchronization needs to be carefully handled. However, PostgreSQL employs just one process to execute each query, such consideration is not taken into account in our implementation. Moreover, given that each query evaluation requires a distinct instantiation of the query bees, code reentrancy is preserved even the object code is dynamically modified at runtime, because each thread will utilize its own bees.

VII. EVALUATION

In this paper, we focus our study on the example query provided in Section IV. In particular, we compare the execution of the query on a stock PostgreSQL and a query bee-enabled PostgreSQL. We utilize the TPC-H benchmark to prepare the dataset on which the query is evaluated. In generating the TPC-H relations, we set the *scale factor* to one, resulting in 200K and 6M tuples in the `part` relation and the `lineitem` relation, respectively. Our experiment machine is configured with a 2.8GHz Intel *i*7 860 CPU and 8GB of main memory. We used PostgreSQL version 8.4.2 as the experiment DBMS. The DBMS and the bees are compiled with `gcc` version 4.4.3, with optimization `-O2` enabled. Note that this paper focuses on the performance benefit of reducing the number of executed instructions; therefore the experiments were carried out with a warm cache.

As mentioned in Section IV, the running time of the query is improved by 21% with the bee-enabled PostgreSQL. We now study in detail the source of such improvement.

It is worth noticing that the overhead of invoking the compiler is not included in the performance analysis. This is because the compiler is never invoked at runtime during query evaluation. Instead, the proto-bees are compiled before query evaluation and hence at runtime, the overhead of dynamically instantiating and invoking the executable bees is trivial. Micro-specialization does not degrade runtime performance at all.

We utilized `callgrind` [5] to produce the execution profile for executing the query on both DBMSes. We present in Listing 2 and Listing 3 excerpts of the profile output of executing the example query on the stock DBMS and the bee-enabled PostgreSQL, respectively. Note that the notation *Ir* represents the number of executed instructions. As shown by the profile result, the stock DBMS executed a total of 7429M instructions. The bee-enabled DBMS on the other hand executed 4940M instructions, or an improvement of 34% in the number of executed instructions.

We delve into the instruction counts for specific functions to explain the performance improvement. The most significant improvement is from the `slot_deform_tuple` function. This function transforms a physical tuple into an array of long integers. Note that this function is invoked for each tuple in both relations referenced in the query. Therefore, specializing this function achieves the most significant benefit. As Listing 3 shows, the `slot_deform_tuple` is highly specialized by invoking the two relation bees, represented as their in-memory locations at `0x441b3c0` and `0x442e7c0`, respectively. As a result of such specialization, 20% instructions are reduced in total when the query was executed.

The presence of the predicates provides another opportunity for applying micro-specialization. The `ExecMakeFunctionResultNoSets` function in the stock DBMS performs predicate evaluation. By contrast, the two predicates presented in the query were evaluated by two predicate bees, as shown in Listing 3 as their addresses in

```
          Ir ... file:function
-------------------------------------------------------------------------------
7,429,490,994 ... TOTAL
2,603,135,278 ... src/backend/access/common/heaptuple.c:slot_deform_tuple
  944,852,915 ... src/backend/executor/execQual.c:ExecMakeFunctionResultNoSets
              ...
  438,256,918 ... src/backend/access/heap/heapam.c:heapgettup_pagemode
              ...
    8,273,670 ... src/backend/executor/execProcnode.c:ExecProcNode(2)
              ...
    2,273,640 ... src/backend/executor/execProcnode.c:ExecProcNode
              ...
```

Listing 2. Profile Result Excerpt of the Stock PostgreSQL

```
          Ir ... file:function
-------------------------------------------------------------------------------
4,940,361,293 ... TOTAL
  738,149,445 ... 0x000000000441b3c0 (relation bee -- lineitem)
  362,774,120 ... src/backend/access/common/heaptuple.c:slot_deform_tuple
  300,357,772 ... src/backend/access/heap/heapam.c:heapgettup_pagemode_bee
              ...
  294,059,535 ... 0x0000000004425fc0 (predicate bee1)
  156,870,266 ... 0x000000000442d3c0 (predicate bee2)
              ...
    8,000,000 ... 0x000000000442e7c0 (relation bee -- part)
              ...
```

Listing 3. Profile Result Excerpt of the Bee-Eanbled PostgreSQL

memory. The two predicate query bees alone reduced about 7% total executed instructions.

While the each micro-specialization improves performance, some micro-specialization may have less significant impact. The `heapgettup_pagemode` function is responsible for scan a relation. We discussed the implementation of this function in Section IV. In the stock implementation, this function needs to examine the direction of the scan and check the existence of predicates. As the profile result shows, by applying micro-specialization on these invariants, approximately 32% of the instructions of that function itself are reduced. The reduction of 138M instructions translates to around two percent within the total improvement. The dispatcher utilized by the stock implementation, `ExecProcNode` (in the profile there are two such instances) contributes a total of 11M instructions. In the bee-enabled PostgreSQL, this overhead is completely eliminated. In total, when micro-specialization applied aggressively across multiple operators, another approximately 7% of instructions were reduced by the query bees that have relatively less performance benefit.

Note that instantiating bees at runtime requires additional instructions to be executed. However, `callgrind` was not able to collect such data given that this additional overhead is too small even to be counted.

To summarize, query bees are utilized by first identifying the invariants during the query evaluation loop. The associated proto-bees are then dynamically instantiated as the executable query bees. By applying several optimizations, such as eliminating unused code branches and turning indirect function calls into direct calls, significant performance benefits can be achieved. Although techniques such as branch prediction supported by modern CPUs can also achieve a similar effect, the benefits of micro-specialization go beyond that of branch prediction. The profile analyses suggest that the instruction reduction is consistent with the execution time improvement. Therefore, even though optimization techniques employed by micro-specialization can be partially realized by other means, micro-specialization still can significantly further improve performance.

VIII. Related Work

Krikellas et al. employed an approach to produce specialized code to replace the entire original generic query evaluation routines implemented in conventional DBMSes [6]. The proposed method uses code templates to form the specialized code for processing specific queries. The code is then compiled and executed to evaluate the queries. The scope of the code replacement is vast: the entire query evaluation code base, often tens or hundreds of thousands of lines, must be moved into templates that are then stitched together.

Neumann, Rao, et al., Sompolski et al., and Zane et al. [7–10] utilize similar mechanisms in compiling queries during their execution, that of converting query plans into source code and producing executable code by compiling the specialized source code. Unlike these approaches, micro-specialization does not require the compiler to be invoked at query evaluation time. More importantly, given that micro-specialization focuses on specialization opportunities at a finer granularity, exploiting invariants within query evaluation code, micro-specialization can be applied aggressively at many places during the query evaluation loop. Moreover, micro-specialization requires minimal changes to the DBMS.

The MAL language [11] incorporated in MonetDB uses a query-evaluation code specialization approach for that column-oriented DBMS. Specifically, when a query is evaluated, the SQL statement is first translated into a MAL program. This initial program is further optimized and transformed by MonetDB. The resulting MAL program, which is a sequence of MAL instructions, is interpreted linearly by the MonetDB kernel [12]. In contrast, micro-specialization manipulates object code directly without relying on additional languages. Therefore, micro-specialization can be applied without affecting the query plan optimizer in the DBMS.

In general, micro-specialization employs dynamic object code manipulation techniques to avoid expansive compiler invocation at runtime. In particular, the code that is executed is exactly that present in the source code of the DBMS, only without all the branching statements and dead code not needed for that particular query. Moreover, the fine-granularity of micro-specialization enables the DBMS to entirely manage the associated tasks without user intervention. More importantly, the property that micro-specialization is applied at low level present opportunities that it can be utilized in concert with all the above mentioned approaches. Specifically, by exploiting invariants and variables that take on deterministic sequences of values that are present in executable code, micro-specialization can be employed independently of the particular query optimizer or data storage model and can further improve query evaluation efficiency without affecting the DBMS architecture and existing optimizations.

IX. Conclusion and Future Work

We have presented the application of micro-specialization on query plans as a self-managed task performed entirely by DBMSes. In the case study of a particular query, we pointed out the exact values where micro-specialization can be applied on and the associated approaches to dynamically manipulating object code. The DBMS automatically instantiates query bees based on the provided proto-bees. The bee instantiation is performed by replacing the magic numbers presented in the object code with correct addresses of the invoked functions, which effectively connects the involved query bees and the host DBMS execution environment. We presented detailed performance analysis of the specific query which confirmed the performance benefit introduced by micro-specialization. We also outlined the steps to achieved the query plan-based application of micro-specialization.

We plan in the future to investigate more opportunities for applying micro-specialization across the wide range of operations in query evaluation. We want to move towards complete automation of micro-specialization on query plans. Such micro-specialization utilizes invariants across the entire query evaluation loop. Automating bee hot-swapping will be more challenging because the value being specialized on is *not* invariant, but rather takes on a deterministic sequence of values. Moreover, given that query bees improve the efficiency of individual operators, it would be helpful to incorporate such effects into the cost model utilized by the plan optimizers.

References

[1] R. Zhang, R. T. Snodgrass, and S. Debray, "Micro-Specialization in DBMSes," *To appear in IEEE International Conference on Data Engineering (ICDE)*, April 2012.

[2] R. Zhang, S. Debray, and R. T. Snodgrass, "Micro-Specialization: Dynamic Code Specialization of Database Management Systems," *To appear in International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[3] PostgresSQL Global Development Group, "PostgresSQL," accessed August 29, 2010. [Online]. Available: http://www.postgresql.org/

[4] TPC, "TPC Transaction Processing Performance Council - TPC-H," (accessed August 29, 2010). [Online]. Available: http://www.tpc.org/tpch/

[5] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2007, pp. 89–100.

[6] K. Krikellas, S. Viglas, and M. Cintra, "Generating Code for Holistic Query Evaluation," in *IEEE International Conference on Data Engineering (ICDE)*, 2010, pp. 613–624.

[7] J. Sompolski, M. Zukowski, and P. A. Boncz, "Vectorization vs. Compilation in Query Execution," in *International Workshop on Data Management on New Hardware (DaMoN)*, 2011, pp. 33–40.

[8] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman, "Compiled Query Execution Engine using JVM," in *IEEE International Conference on Data Engineering (ICDE)*, 2006, p. 23.

[9] T. Neumann, "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[10] B. Zane, J. Ballard, and F. Hinshaw, "Optimized SQL Code Generation," U.S. Patent 7 430 549 B2, September 30, 2008.

[11] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1648–1653, 2009.

[12] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An architecture for recycling intermediates in a column-store," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 309–320.