Micro-Specialization: Dynamic Code Specialization of Database Management Systems

Rui Zhang
University of Arizona
ruizhang@cs.arizona.edu

Saumya Debray
University of Arizona
debray@cs.arizona.edu

Richard T. Snodgrass
University of Arizona
rts@cs.arizona.edu

ABSTRACT

Database management systems (DBMSes) form a cornerstone of modern IT infrastructure, and it is essential that they have excellent performance. Much of the work to date on optimizing DBMS performance has emphasized ensuring efficient data access from secondary storage. This paper shows that DBMSes can also benefit significantly from dynamic code specialization. Our approach focuses on the iterative query evaluation loops typically used by such systems. Query evaluation involves extensive references to the relational schema, predicate values, and join types, which are all invariant during query evaluation, and thus are subject to dynamic value-based code specialization.

We introduce three distinct types of specialization, each corresponding to a particular kind of invariant. We realize these techniques, in concert termed *micro-specialization*, via a DBMS-independent run-time environment and apply them to a high-performance open-source DBMS, PostgreSQL. We show that micro-specialization requires minimal changes to the DBMS and can yield performance improvements simultaneously across a wide range of queries and modifications, in terms of storage, CPU usage, and I/O time of standard DBMS benchmarks. We also discuss an integrated development environment that helps DBMS developers apply micro-specializations to identified target code sequences.

1. INTRODUCTION

Database management systems (DBMS) play a fundamental role in industrial IT infrastructure. Because of this, it is essential that these systems have excellent performance. The current state of the art in this respect combines three different approaches. First, special-purpose database architectures (e.g., stream DBs, column stores) are used where appropriate to organize the data in a way that minimizes access costs for specific types of applications [1, 29, 30, 32]. Second, traditional compiler optimizations are used to compile the DBMS source code into efficient machine code. Finally, database query optimization is used to execute individual queries efficiently (e.g., proximity rank join [20], replacement selection in external sorting [21], and range query transformation [25], in just one track of one conference). The general consensus in the database community is that, because hard disks are significantly slower than CPUs, by far the most important are those approaches related to organization and access of data from secondary storage: namely, DBMS architecture and query optimization. That said, there is also a growing awareness that architecture-conscious approaches, focusing on costs such as main memory cache misses, are also useful for improving DBMS performance.

Separately, there is a rich body of literature in the compiler community on code optimization. Early work on this topic focused on static analyses and optimizations [3], but more recently there has been a lot of work on dynamic optimization and specialization, both in the context of traditional compiled languages such as C [6, 9, 10, 12, 13] as well as in the context of JIT compilers for Java [2, 16, 33]. This work generally focuses on CPU-bound code; database systems, conventionally considered to be I/O-bound, have not received much attention in the context of such optimizations. In this paper we show that this conventional wisdom is not entirely correct and that significant DBMS performance improvements are achievable through dynamic code specialization.

The goal for JIT compilers is to optimize the byte code of the input programs, without prior knowledge of their runtime behavior and while incurring as little runtime overhead as possible. Our work is similarly concerned with dynamic code specialization in the context of an interpreter: in our case, the interpreter is the SQL engine that forms the heart of the DBMS query processor, and the analog of the JIT compiler's input programs are the input SQL queries. However, the details of the two situations are very different. Our dynamic specialization is aimed not at the input SQL queries, but rather at the underlying DBMS code that processes those queries.

Our approach takes advantage of information specific to the particular environment of a DBMS by identifying variables whose values—typically, schema metadata or query-specific constants—are invariant *within the query evaluation loop*. This information is used for fine-grained specialization that eliminates unnecessary operations along frequently-taken execution paths, leading to further optimized code that is both smaller and faster. Often this loop is evaluated for every tuple in the underlying relation(s), thereby offering the possibility of significant performance improvements. (A note on terminology: a tuple is also informally called a row; a relation is similarly called a table; and an attribute, a column.) Since the invariants used for specialization are available only at runtime, such specialization cannot be carried out using static techniques, but has to be deferred to runtime. This implies that the specialization process itself has to be extremely lightweight.

In addition to specialization based on schema metadata and query-specific values, we have identified another opportunity for dynamic specialization: the values in the relations themselves. If such values are relatively few or relatively common, specializing on such values can be very effective. Our innovation is to show how to specialize

DBMS code based on data associated with an individual relation or with individual tuples.

We refer to such fine-grained low-level dynamic specialization as *micro-specialization*, to distinguish it from other, higher-level specializations effected in DBMSes. This paper describes DBMS micro-specialization, applies this concept to a complex high-performance DBMS, and evaluates its effectiveness and cost. Even just a few such specializations (adding 250 source lines of code (SLOC) and another 900 SLOC of specialized versions of existing code, comprising 0.3% of this 380,000 SLOC DBMS) can improve query execution speeds by up to 33% across complex analytic queries, random modifications, and bulk loading, on standard industrial benchmarks.

Our contributions are the following.

- We show that dynamic specialization oriented to the particulars of an important class of software artifacts, that of DBMSes, can achieve truly significant performance benefits.
- We identify several classes of opportunities for micro-specialization: schema information, data structures used in query evaluation, and even individual values stored in the database.
- We show that the instantiation of specialized code can be located at several points along the compile-time/run-time spectrum.
- We co-locate some of the specialized code with the data in the DBMS and move some of the data in the tuples into that code.
- We implemented an extensive run-time environment (HIVE-RE) and have started developing a DBMS-independent integrated development environment (HIVE), which in concert support micro-specialization across this spectrum.
- We applied six instances of micro-specialization to a high-performance DBMS and studied in detail the performance improvements that accrue.

We first summarize the salient aspects of DBMS query evaluation. We then walk through a single micro-specialization that improves the performance of even simple queries. In this case study, we examine the specific code changes, predict the performance improvement, and then validate our prediction with an experiment. Section 4 examines micro-specialization opportunities broadly with a taxonomy of three general classes of invariant value, which induce three types of micro-specialization. We then introduce the run-time environment and explain what happens at runtime. Section 5 briefly discusses the structure of the HIVE development environment for introducing micro-specializations into a complex DBMS and outlines how to identify specialization targets, how to decide which specialization approach to apply, and how to insert calls to that API to effect the micro-specialization. We apply all three kinds of micro-specialization to PostgreSQL. We then characterize, through a set of experiments on the TPC-H and the TPC-C benchmarks, the salutary effect of micro-specialization. Section 8 places micro-specialization in the broader contexts of DBMS and compiler-based specializations.

2. BACKGROUND

Figure 1 shows, in very high level terms, the structure of a typical DBMS query processing algorithm. We first construct the database by defining a set of relation schemas and then populating the relations specified by these schemas. The schemas specify meta-data

```
/* construct database */
schemas := DefineRelationSchemas();
rels := PopulateRelations(schemas);

/* iterate over queries */
loop forever {
    query := ReadQuery();
    query_plan := OptimizeQuery(query, schemas);

    /* process query: iterate over tuples */
    ans := Exec(query_plan, rels, schemas);

    Output(ans);
}
```

Figure 1: Thirty-thousand-foot View of Database Query Processing

about each relation, such as the name of the relation, the number of columns, their names, types, etc. This is followed by query evaluation: a query is read in; a query plan is generated by the query optimizer; this plan is executed by the SQL engine; and the answers so obtained are output. This process is repeated. The query optimizer uses meta-data about the relations in the database to make implementation-level decisions (e.g., a join operation in the query may be mapped to a implementation-level operations of hash-join or sort-merge join) and determine an efficient execution plan for the query operations. The query plan produced by the optimizer is essentially a tree representation of the query where leaf nodes are database relations and internal nodes are operations. The query evaluation engine applies the operations specified in the query plan to the relations in the database, iterating over the tuples in the relations and using schema meta-data to parse the tuples to extract and process the fields.

The query-evaluation process described above involves repeated interpretation of a number of data structures that are invariant through the evaluation of each query. For example, the set of relations that have to be accessed is fixed for each query, which means that the information about attribute types and offsets for each such relation, obtained from its schema and used to parse its tuples, is invariant through the execution of the query. However, because relation schema information is not known when the DBMS code is compiled, this information cannot be propagated into the query evaluation code, but must be obtained by interpreting the schema data—an action that is repeated for each tuple that is processed. As another example, an expression for a *select* or *join* operation in a query is represented as a syntax tree, which has to be evaluated for each tuple. This syntax tree—which is fixed for a given query—cannot be compiled into code when the DBMS is compiled because it becomes known only once a query has been read in. Since processing a query in a database of reasonable size may involve looking at many millions of tuples, these interpretation overheads can accumulate into substantial overheads, in terms of both instruction counts and instruction and data cache misses.

Our work on dynamic specialization of DBMS code is aimed at reducing this interpretive overhead as much as possible. We do this by identifying those portions of the DBMS's query evaluation loop that have a high number of references to query-invariant values such as those described above, dynamically generating code that has been specialized to the actual query-invariant values, and splicing in (a pointer to) this dynamically generated code into the DBMS's query evaluation loop. The following case study shows

that such specialization can have a big impact on DBMS performance.

3. CASE STUDY

In a DBMS, there are many variables which can in fact be invariant (constant) within the query evaluation loop. For instance, once the schema of a relation is defined, the number of attributes is a constant. Moreover, the type of each attribute, the length of each fixed-length attribute, as well as the offsets of some attributes (those not preceded by a variable-length attribute) are constants for this relation. Conventionally, the relation-specific information is stored in the system catalog. Each tuple in the database is physically represented simply as a sequence of bytes; when queries are evaluated, the catalog is consulted for the referenced relations, and the above mentioned invariants are used to "parse" the tuple to identify and extract attribute values. Although catalog look-up has been carefully engineered to be efficient, the generic implementation still presents significant overhead, especially for large relations.

Listing 1 excerpts a function, `slot_deform_tuple()`, from the source code of PostgreSQL [14]. This function is executed whenever a tuple is fetched; it extracts values from a stored tuple into an array of `long` integers. The function relies on a loop (starting on line 4) to extract each attribute. For each attribute, a path in the code sequence (from line 5 to line 36) is executed to convert the attribute's value within the stored bytes of the tuple into a long integer. (Bytes, shorts, and ints are cast to longs and strings are cast to pointers. The catalog information for each attribute is stored in a struct named `thisatt`, which is located in the function argument namely `slot`. This argument contains both the catalog information and the actual physical tuple. All the variables utilized in this function come directly from this argument. As Listing 1 shows, attribute length (`attlen`), attribute physical storage alignment (`attalign`), and attribute offset (`attcacheoff`) all participate in selecting a particular execution path.

Within a conventional DBMS implementation, these variables are used in condition checking because the values of these variables depend on the specific relation being queried. Such generality provides opportunities for performance improvement. Micro-specialization focuses on such variables: when they are constant within the query evaluation loop, the corresponding code sequence can be dramatically shortened.

We utilize the `orders` relation from the TPC-H benchmark as an example to illustrate the application of micro-specialization. To specialize the `slot_deform_tuple()` function for the `orders` relation, we first identify the variables that are constants. According to the schema, no null values are allowed for this relation. Therefore the null checking statements from lines 6 to 11 are not needed. Instead, we can assign the entire `isnull` array to `false` at the beginning of the function. Since each value of the `isnull` array is a byte, we can collapse the assignments with a few type casts. For instance, the eight assignments of `isnull[0]` to `isnull[7]` can be turned into a single, very efficient statement: `(long*)isnull = 0;` This function is invoked to extract the values of a stored tuple. Given that the relation schema does not allow nullable attributes, the stored tuples are guaranteed to contain no null values (this is checked elsewhere). Hence, the above optimization is made possible.

As discussed earlier, some of the variables in Listing 1 are constant for any particular relation. For the `orders` relation, the value of the `natts` (number of attributes) variable is 9. We apply *loop unrolling* to avoid the condition checking and the the loop-counter increment instructions in the `for` statement. The resulting program simply has nine assignment statements.

```
1  void slot_deform_tuple(TupleTableSlot *slot, int natts) {
2    ...
3    tp = (char *) tup + tup->t_hoff;
4    for (; attnum < natts; attnum++) {
5      Form_pg_attribute thisatt = att[attnum];
6      if ( hasnulls && att_isnull(attnum, bp)) {
7        values[attnum] = (Datum) 0;
8        isnull[attnum] = true;
9        slow = true;
10       continue;
11     }
12     isnull[attnum] = false;
13     if (!slow && thisatt-> attcacheoff >= 0) {
14       off = thisatt->attcacheoff;
15     } else if (thisatt-> attlen == -1) {
16       if (!slow && off == att_align_nominal(off, thisatt-> attalign )) {
17         thisatt->attcacheoff = off;
18       } else {
19         if (!slow && off == att_align_nominal(off, thisatt-> attalign )) {
20           thisatt->attcacheoff = off;
21         } else {
22           off = att_align_pointer(off, thisatt-> attalign , -1, tp + off);
23           slow = true;
24         }
25       } else {
26         off = att_align_nominal(off, thisatt-> attalign );
27         if (!slow)
28             thisatt->attcacheoff = off;
29       }
30       values[attnum] = fetchatt(thisatt, tp + off);
31       off = att_addlength_pointer(off, thisatt-> attlen , tp + off);
32       if (thisatt->attlen <= 0)
33           slow = true;
34     }
35     ...
36   }
37 }
```

Listing 1: The `slot_deform_tuple()` Function

```
values[0] = ...;
values[1] = ...;
      ...
values[8] = ...;
```

Now let's focus on the type-specific attribute extraction statements. The first attribute of the `orders` relation is a four-byte integer. Therefore, we don't need to consult the `attlen` variable with a condition statement. Instead, we directly assign an integer value from the tuple with this statement.

```
values[0] = *(int*)(data);
```

Note that the `data` variable is a byte array in which the physical tuple is stored. Since the second attribute is also an integer, the same statement also applies. Given that the length of the first attribute is four bytes, we add four to `data` as the offset of the second attribute.

```
values[1] = *(int*)(data + 4);
```

The resulting specialized code for the `orders` relation is presented in Listing 2. (We will elaborate on the `bee_id` parameter in Section 6.4.) Although the code looks longer that the original, the `for` loop in Listing 1 has been unrolled nine times. As a result, the specialized code will execute many fewer instructions than the stock code. Manual examination of the executable object code found that the `for` loop executes about 340 machine instructions (x86) for the `orders` relation in executing the following query.

```
SELECT o_comment FROM orders;
```

To execute the specialized code, we simply insert a function call to the `GetColumnsToLongs()` function to replace the `for` loop. The specialized code has only 146 instructions, for a reduction of approximately 190 instructions.

The specializations described above then follow directly from the fact that the metadata describing various attributes of each tuple, obtained from the schema metadata, is invariant in the code shown in Listing 1, and can be automated using techniques discussed elsewhere in the literature [10, 22, 24].

```
1 void GetColumnsToLongs(char bee_id, int address, char* data, int* start_att,
2                        int* offset, bool* isnull, Datum* values) {
3   *(long*)isnull = 0;
4   isnull[8] = 0;
5   values[0] = *(int*)data;
6   values[1] = *(int*)(data + 4);
7   values[2] = (long)(address + bee_id * 32 + 1000);
8   *start_att = 3;
9   if (end_att < 4) return;
10  *offset = 8;
11  if (*offset != (((long)(*offset) + 3) & ~((long)3)))
12    if (!(*(char*)(data + *offset)))
13        *offset = (long)(*offset + 3) & ~(long)3;
14  values[3] = (long)(data + *offset);
15  *offset += VARSIZE_ANY(data + *offset);
16  *offset = ((long)(*offset) + 3) & ~((long)3);
17  values[4] = (*(long*)(data + *offset)) & 0xffffffff;
18  *offset += 4;
19  values[5] = (long)(address + bee_id * 32 + 1001);
20  *start_att = 6;
21  if (end_att < 7) return;
22  if (!(*(char*)(data + *offset)))
23      *offset = (long)(*offset + 3) & ~(long)3;
24  values[6] = (long)(data + *offset);
25  *offset += VARSIZE_ANY(data + *offset);
26  values[7] = *(int*)(address + bee_id * 32 + 1002);
27  if (!(*(char*)(data + *offset)))
28      *offset = (long)(*offset + 3) & ~(long)3;
29  values[8] = (long)(data + *offset);
30  *start_att = 9;
31 }
```

Listing 2: The Micro-Specialized `GCL()` Function

To determine the actual performance benefit, consider the instruction savings. This query requests a sequential scan over the `orders` relation, which has 1.5M tuples (with the scale factor set to one for the TPC-H dataset). Given that the specialized code saves 190 instructions and the code is invoked 1.5M times (once per tuple), the total number of instructions is expected to decrease by 285M. Using `callgrind` [11] to obtain the total number of executed instructions for both a stock PostgreSQL and one with the specialized code shown in Listing 2, we find that stock PostgreSQL executes a total of 3.447B instructions on this query, which implies that this micro-specialization should produce an (estimated) instruction count reduction of about 8.3%. The total number of instructions actually executed by the specialized PostgreSQL is 3.153B, a (measured) reduction of 8.5%, consistent with our earlier estimate. We then measured the total running time of the query on the stock PostgreSQL and the specialized version. The improvement in running time (7.4%) is consistent with the profile analysis. Thus, by specializing just the generic `slot_deform_tuple()` function, on just a few variables, we were able to achieve a 7.4% running time improvement on a simple query. This improvement suggests the feasibility and benefits of applying micro-specialization more aggressively.

How did this improvement come about? First, the developer identified `slot_deform_tuple` as a candidate for micro-specialization. Such candidates must satisfy four criteria: each such code sequence must (i) appear in the query evaluation loop, (ii) constitute a significant portion of the runtime of query evaluation, (iii) reference variable(s) whose value can be determined to be invariant across the query evaluation loop, and (iv) benefit significantly from micro-specialization, by removing branches and accesses on those variables. Second, the developer specifies *code fragments* such as those given earlier that can be combined into a micro-specialized function, as well as code to string these fragments together given a relation schema. (As an optimization, this composition is actually done on the machine code version, to avoid calling the compiler at runtime.) Then the call to the function is replaced with a call to the correct micro-specialized function.

We now elaborate on this idea, identifying other kinds of micro-specializations and evaluating their impact on the performance of the DBMS.

4. APPROACH

Each micro-specialization identifies one or more variables whose value will be constant within the query evaluation loop. It then replaces a function or small stretch of code with multiple copies, each particular to a single value of each of those variables. In the example given above, the variables concerned the relation being scanned. Hence, we need a specialized version of `GetColumnsToLongs()` for each relation.

We first introduce terminology for the specifics of our approach.

- The specialized code, in this case associated with a particular relation is termed a *bee*, in this case, a *relation bee*. There will be a unique bee for every relation defined in a database. Given that the specialized code is small, efficient, and specific to a particular task, such code resembles the characteristics of bees.
- A bee can have multiple *bee routines*, each produced by a particular micro-specialization at a certain place in the DBMS source code on one or more variables that have been identified as being invariant across the query evaluation loop.

In the example given above, micro-specialization is applied on values (attribute length, etc.) that are constant for each relation, and so a relation bee routine results. We term this particular bee routine `GCL`, as shorthand for the specialized `GetColumnsToLongs()` routine. We specialized another PostgreSQL function named `heap_fill_tuple` that constructs a tuple to be stored from an integer array, resulting in a separate bee routine namely `SetColumnsFromLongs()` (`SCL`) for each relation. So each relation bee now has two bee routines.

This general approach raises two central questions: on *which* values can micro-specialization be applied and *when* during the timeline from relation-schema definition to query evaluation can bees be instantiated? The answers to these questions lie in the structure of DBMS query processing, shown in Figure 1. Different kinds of information become invariant (with respect to the inner query processing loop) at different points in the query processing algorithm. Information about individual relations become fixed once relation schemas are defined. This information is then invariant for the subsequent iteration over queries. For each query, the predicates and constants specific to the query plan become fixed after query optimization. We take advantage of this invariance structure to carry out specialization in different ways, evincing three different types of bees.

The *relation bees* described above (one per relation in the database) arise from specialization based on the relational *schema*, where 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.

We can also specialize on the 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 the predicate '`age <= 45`' will use a predicate data structure that contains the ID of attribute `age`, the '`<=`' operator, and the constant `45`. We can apply specialization on these variables once we know the predicate from the query. The bees resulting from specializing such query-related data structures are termed *query bees*.

We can extend this idea even further, down to the level of individual tuples, by specializing on the values of particular attributes within a tuple. The idea is that for an attribute that is known (e.g., from schema meta-data) to have only a small set of possible values, we can use a precomputed set of simple assignments, one for each

value in the underlying domain, to replace the generic database code that computes length, offset, and alignment of the attribute from the relation schema to access its value. We refer to these as *tuple bees*.

For example, for an attribute such as "gender," which takes on one of the two values 'M' or 'F', it suffices to use a single assignment such as `'values[x] = 'M''` to extract the value of this attribute for any given tuple. This assignment occurs within a tuple bee associated with that tuple; the particular tuple bee corresponding to a given tuple is indicated by including in such tuples a short index termed a *beeID*. So we might just have two tuple bees, one for each gender, or we might also specialize on other attributes, as long as the beeID is sufficient in uniquely identifying all the tuple bees, so that a small number of tuple bees are generated for all the tuples in the relation.

To address the question of when specific bees are instantiated, we again consider the structure of DBMS query processing shown in Figure 1. Obviously, specialization can be performed, and bees instantiated, only when the underlying data values become known and fixed. Once this happens, moreover, there does not seem to be much benefit from delaying specialization to a later point in the query processing since, in general, later points correspond to more deeply nested loops and hence greater execution frequency. Based on these considerations, the points when individual bees of each kind are instantiated are shown in Figure 2. Relation bees are instantiated at relation schema definition time, one for each newly-defined relation. Individual query bees are instantiated during 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. Tuple bees are instantiated during the evaluation of tuple insertions and updates, deep within the query evaluation loop. Interestingly, bees of all three types can also be instantiated before compilation time, if the possible values for the invariant variables are known and if the total number of possible bees (usually the product of the number of possible values for each variable) is small.

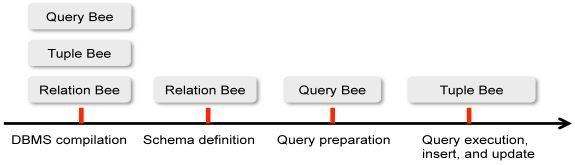

Figure 2: *When* to Instantiate Various Kinds of Bees

Where bee instantiation resides along the timeline shown in Figure 2 affects how lightweight and efficient bee instantiation has to be. Note however that we are not discussing the design of a bee routine. As we will see in the next section, the code to instantiate the individual bees and to invoke bee routines 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, which are instantiated when relations are defined and before the query evaluation process begins, bee instantiation overhead is not critical. Hence, when instantiating a relation bee, we can invoke *gcc* to compile the source code, as shown in Listing 2, to produce the executable object code for the bee.

Because ad hoc queries need to be fast, the overhead of instantiating query bees needs to be minimized. Recall that query bee may contain the join and predicate evaluation routines. In the case of join, all possible combinations of the join routines, such as different types of joins (left, semi, anti, etc.) can be enumerated and compiled ahead of time. At query preparation time, the associated join bee routine is selected from the set of the pre-compiled join evaluation routines.

Delving down into the details, there are two basic mechanisms for bee instantiation. The invariant values that don't participate in control constructs can be easily converted to "holes" in the templates with magic numbers. These magic numbers are easy to identify in the produced object code, such that at run-time, these holes can be filled with the correct values provided in the execution context. An example is the attribute ID for both the join and predicate bee routines, which appears in assignment statements but not in control constructs in the bee routine. The bee can be cloned each time, with different values substituted for such values, represented as magic numbers.

Alternatively, if an invariant appears in a control construct and if this invariant is known to be associated with just a few values, such branching statements and the associated branches can be removed from the specialized code. This is essentially aggressive constant folding, to be done either manually or by the compiler (see a discussion of this in Section 3). An example is the specialization on `hasnulls` on line 6 of Listing 1. This invariant appears in an if statement and so aggressive constant folding during compilation can eliminate several lines of code if this value is false. Specializing on variable(s) referenced in control constructs results in multiple versions of a bee routine.

Consider two query bee routines, namely `EvaluatePredicate()` (`EVP`) and `EvaluateJoin()` (`EVJ`). We use the following query that is based on the TPC-H schema as an example:

```
SELECT l_extendedprice, p_type
FROM lineitem, part
WHERE l_partkey = p_partkey
AND l_shipdate <= date '1995-04-01'
```

This query contains a equi-join of the `lineitem` and the `part` relations, on the `l_partkey` and `p_partkey` attributes, as well as an additional predicate, on `l_shipdate`. This latter predicate contains three invariants in the life cycle of this query, which are the compared constant `date '1995-04-01'`, the ID of the attribute `l_shipdate` in the `lineitem` relation, and the `<=` operator on the date data type. In the original implementation, each operand is treated as a generic object, in that operands of different types can appear on both sides of the operator. Hence expensvie operand processing is required when the operator is evaluated, which turns out to be inefficient. For instance, even though the value of the predicate constant is known, it is stored in a generic data structure that is accessed each time through a chain of function calls to extract the value `'1995-04-01'`.

Micro-specialization effectively addresses the inefficiency in accessing the operands. For the predicate constant, instead of accessing its container data structure each time the predicate is evaluated with many functions, this value is stored in the specialized code which can be directly referenced by consequent code in the query evaluation, saving a significant amount of unnecessary instructions. For the attribute operand, the value of the attribute ID is similarly stored in the specialized code rather than being extracted for each tuple, such that the attribute value extraction routine can be directly invoked without going through a series of function calls.

PostgreSQL utilizes function pointers to perform type-specific predicate-operand comparison efficiently. Since the function address of the

operator is known at the query planning stage, this address can directly be injected into a `CALL` instruction as a direct call at runtime. We create a magic number, which is associated with the `CALL` instruction where the comparison function is invoked, in the `EVP` routine. When the source code of the `EVP` routine is compiled, this magic number is easily identified from the object code. At runtime, this number will be replaced by the address of the associated comparison function. Alternatively, we could use a mechanism similar to relocations (used by linkers to patch binaries) to effect the same goal.

A join operator shares the same mechanism to a predicate in evaluating the join condition. The only difference is that both operands are attributes from relations. The same specialization can be applied here. First, both attribute IDs are stored in the specialized code. Second, the address of the comparison function replaces the magic number at run-time, instantiating the template bee routine with the correct query-specific information.

In addition to join condition evaluation, invariants are identified in the join algorithms themselves. Three join algorithms are usually adopted in DBMSes, including nested-loop join, sort-merge join, and hash join. A common invariant across all three kinds of joins is the join type, which can be inner join, outer join, natural join, semi join, and anti join. The difference among these types of joins, in terms of implementation, is that each type relies on a distinct code path to deal with various matching requirements. The checking of the `js.jointype` and the associated branches that do not belong to a particular kind of join can be eliminated via constant folding when the kind of join is known in the query plan. Similarly, two other such invariant variables, each allowing two distinct values, are specialized on in the same fashion. Consequently, each join algorithm requires 20 distinct versions of the object code, each corresponding to one possible combination of the values of these invariants. Instead of creating 20 versions of source code, we compile the generic version of source code for each algorithm with 20 value combinations, when the DBMS is compiled enabling the compiler to eliminate the unnecessary condition checking and the associated basic blocks, resulting in highly-optimized object code. In PostgreSQL, a total 57K bytes of the specialized join routines are created; this amounts to about 2% of PostgreSQL's total text section size of 2.63MB.

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 amount of instructions can be reduced by the specialized code, as illustrated in the case study.

5. APPLYING MICRO-SPECIALIZATION

Micro-specialization can be applied to a DBMS in a systematic fashion by performing the following steps in sequence, which we now discuss in some detail.

1. Identify the query evaluation loop. To accurately extract the portion of the code that represents the query evaluation loop from the rather large and complex executable code of a DBMS, we start by constructing a static call graph of the basic blocks inside the DBMS. We then compute strongly connected components from this graph. The strongly connected components provide us with the set of basic blocks that represent just the query evaluation loop.

2. Identify the invariants. To spot the invariants, dynamic analysis is required. Profile tools such as `callgrind` are invoked along with query evaluation to produce accurate run-time memory access traces. The traces, containing a list of triples in the form of `<address, opcode, operand>`, are combined with the previously computed query evaluation loop basic block set and the dynamic data flow graph to identify those variables whose values are invariant in query evaluation.

3. Pin-point the invariants in the source code. We then map the invariant variables back to data structures defined in the source code. The identified invariants are memory locations, represented in the object code as operands in instructions. We utilize the `.debug_line` section to trace the instruction back to the source code to identify the actual references and decalarations of these invariants.

4. Decide which code sequence(s) should be micro-specialized. We examine each target code sequence to be specialized, specifically to determine the exact boundry for each sequence. To do so, we rely on static data flow analysis to locate the code sequences over which the value is invariant. These code sequences can either be early in the call graph or near its leaves. The ideal specialization targets contain a relatively large number of uses within a short code sequence.

5. Decide when to compile and instantiate bees. For different kinds of bees, various compilation and instantiation alternatives are appropriate. For instance, all versions of the join algorithms and the predicate evaluation query routine can be compiled when the DBMS is compiled. On the other hand, a relation bee routine can be compiled only at schema definition time. Relation bees are instantiated at schema definition time, whereas a query bee can be instantiated only after the query has been received by the DBMS. The developer thus decides in what kind of bee this specialized code sequence should reside (and hence, form a bee routine) and when, that is, at DBMS compile time or DBMS runtime, the bee routine should be compiled. (The bee is always instantiated at runtime.)

6. The target source code is converted to snippets, to install a bee routine. Consider a relation bee routine. This routine would probably deal with all of the relation's attributes. Say it is specialized on the types of the attributes. The actual relation bee routine would have to be constructed out of snippets, one for each possible type, stitched together according to the schema. In this particular case, we extract the snippets from the switch statement. As another example, consider the for loop over the attributes on line 4 of Listing 1. We create a snippet from the body of that loop.

If the code sequence contains a call to another function, and that call passes one of the invariant values as a parameter, that called function is also specialized as part of this bee routine after inlining the function invocation. (Otherwise, the bee just retains the function call.)

For each attribute value incorporated into a tuple bee, the actual attribute value appears as a parameter to the tuple bee routine. Hence, the tuple bees of each relation can effectively share the same routine code in that the tuple bees are based on the schema of the relation. We create a storage space designated to the tuple bee values. We term this space *data section*.

7. Add bee invocations and supportng code to the DBMS source. The code that was specialized is now removed from the DBMS, replaced with a call to the corresponding bee routine.

Adding a bee may impact other portions of the DBMS (hopefully in highly circumscribed ways). For example, an attribute stored in a tuple bee is no longer stored in the relation itself. In the `orders` relation from TPC-H, we specialize on three attributes, namely `o_orderstatus`, `o_orderpriority`, and `o_shippriority`, which have small discrete value domains. These attributes are removed from the schema as their values are stored in the instatiated bee for each tuple. Code must be added to the DBMS to effect this change.

8. Run confirmatory experimental performance analyses. It is important to verify the performance benefits of each added bee on queries that should utilize that bee. We utilize benchmarks to study the performance by comparing the bee-eanbled DBMS and the stock version. The detailed study include running time of queries, throughput of transactions, and profile of instruction and cache statistics, which are discussed in Section 7.

We proposed the eight steps to address the known challenges in applying micro-specialization based on our manual experience. Applying the first routine took several months. Applying the last two bee routines, that of `EVP` and `EVJ`, took only a couple of days, following these eight steps. Hence, DBMS developers can benefit greatly from these steps when applying micro-specialization.

To assist developers in carrying out these steps, we are building a set of tools aimed at simplifying and automating the process of micro-specialization. Currently, our toolset—which we refer to as *HIVE* (*Highly-Integrated deVelopment Environment*)—allows us to automate the first and last steps described above and partially automate the second step. In fact, the results of Section 7 were generated using HIVE. Our future plans for extending HIVE to automate all eight steps are described in Section 9.

6. THE HIVE RUNTIME ENVIRONMENT

This section discusses in depth the HIVE runtime environment (HIVE-RE) and how the types of bees are instantiated and invoked during DBMS execution.

6.1 HIVE-RE Components

HIVE-RE consists of about 6000 lines of C code, responsible for bee instantiation, storage, invocation, and garbage collection. HIVE-RE consists of the following central components, which operate without administrative intervention.

- The *Bee Snippet Repository* contains a collection of source code snippets used in forming the actual bee routines during bee instantiation. This repository is created by the developer during the sixth step discussed earlier. The *Bee Assembler* is responsible for stitching together the provided code snippets to form bee routines.
- The *Bee Maker* performs two tasks. First, the formed bee routine source code is sent to the bee maker for compilation. The compiled routines are then attached to the associated bees. We term this step *bee creation*. Second, at run-time, the bee maker *instantiates* the compiled bee routines with correct values.
- The *Bee Cache* is an on-disk repository where all instantiated bees are stored. The *Bee Cache Manager* manipulates this storage. When bees are created by the bee maker, the bee-cache manager stores the newly created bees to the bee cache. When DBMS server starts, the bee-cache manager loads all the bees from the bee cache to main memory for later invocation.
- The *Bee Placement Optimizer* controls the residence of bees in memory, which directly affects the occupancy of the bees in the CPU caches, particularly the level-1 instruction (*I1*) cache. Given that bees introduce additional code which does not benefit from locality optimization applied to DBMS at compile time, sloppy placement of bees can degrade performance.
- The *Bee Collector* performs garbage collection on dead bees. For instance, when a `DROP TABLE` command is issued, the bees associated with the dropped relation are no longer needed. The bee collector will remove such bees from the bee cache.

We now describe specifically how each type of bee is manipulated (created, for tuple bees, and instantiated) by the HIVE-RE.

6.2 Relation Bees

When the schema of a relation is defined, as the result of a `CREATE TABLE` SQL statement, the code snippets that are associated with the attributes in this relation are stitched together as the source code of a relation-specific bee routine that performs value extraction. This source code is then compiled, resulting in an object file in ELF representation (on a Linux OS). The HIVE-RE extracts the executable function body from the object file, and inserts the corresponding function address in-place, for any (non-specialized) function(s) invoked by that bee routine.

The extracted routines are stored in a bee cache, ready for execution. (We discuss relation bee invocation below, as all bees are invoked similarly.)

6.3 Query Bees

When instantiating a query bee, the holes (special values) in the bee routine are filled with values provided from the query data structure. A special case of this is PostgreSQL function pointers for type-specific predicate-operand comparison.

6.4 Tuple Bees

Tuple bees are distinct in that they contain holes where tuple attribute values have been referenced by the target code. These holes are indicated by magic numbers (cf. lines 7, 19, and 26 of Listing 2). After the tuple bee code in that listing is compiled, the magic numbers are replaced with a computed offset from the current instruction to the beginning of the data section.

When a tuple bee is instantiated, only the data section need be created. The code on these lines then can compute the address of the attribute value within the data section associated with that beeID, allowing the relevant attribute value to be referenced.

When a query is being evaluated, tuples are fetched from the relation. As a tuple is being fetched, its beeID, represented as the `bee_id` variable in Listing 2, is associated with the address of the tuple in the buffer. The beeID and the tuple address are then passed to the bee routine by the invocation statement that replaces the original targeted code from which the bee was created. The bee routine locates the appropriate data section using the beeID and fetches the specialized attribute values. This routine also computes the offsets and extracts the values for the non-specialized attributes, without looking up the catalogs. Relations with no specialized attributes have a single instantiated bee used by every tuple, in which case the single tuple bee is in reality a relation bee. Both relation and tuple bees perform the tuple extraction and construction tasks. Hence

the GCL and SCL routines are present in both kinds of bees. The difference is that tuple bees store particular columns along with the bees; therefore, the GCL routine for a tuple bee does not need to extract such value(s) from the stored tuple, but rather from constants stored in the bees.

7. EMPIRICAL EVALUATION

We have investigated the performance impact of micro-special-ization in many contexts: simple select queries such as discussed in the case study, OLAP-style queries in the TPC-H benchmark, and OLTP-style queries and modifications in the TPC-C benchmark.

To generate the dataset in TPC-H, we utilized the *DBGEN* toolkit [31]. The scale factor for data generation was set to one, resulting in the data of size 1GB. For TPC-C, we used the *BenchmarkSQL-2.3.2* [18] toolkit. The *number of warehouses* parameter was set to 10 when the initial dataset was created. Consequently, a total of 100 *terminals* were used (10 per warehouse, as specified in TPC-C's documentation) to simulate the workload. We also added DDL clauses to identify the handful of low-cardinality attributes the TPC-H relations. Other than specifying the scale factor and number of warehouses, we made no changes to other parameters used in the TPC-C and TPC-H toolkits for dataset preparation.

All the experiments were performed on a machine with 8GB main memory and a 2.8GHz Intel *i*7 860 CPU, which contains four cores. Each core has a 64KB Level-1 (*L1*) cache, which consists of a 32KB instruction (*I1*) and a 32KB data cache. The CPU is also configured with a 256K unified level-2 (*L2*) cache. Our prototype implementation used PostgreSQL version 8.4.2, compiled using *gcc* version 4.4.3 with the default build parameters (where the optimization level, in particular, is *–O2*).

7.1 The TPC-H Benchmark

We start with the TPC-H benchmark to compare the performance of the bee-enabled PostgreSQL with the stock DBMS. The TPC-H benchmark creates a database resembling an industrial data warehouse. The queries used in the benchmark are complex analytic queries. Such a workload, featured with intensive joins, predicate evaluations, and aggregations, involves large amount of disk I/O and catalog lookup.

All 22 queries specified in TPC-H were evaluated in both the stock and bee-enabled PostgreSQL. The running time was measured as wall-clock time, under a warm-cache scenario. We first focus on the warm-cache scenario to study the CPU performance: keeping the data in memory eliminated the disk I/O requests.

We ran each query twelve times. The highest and lowest measurements were considered outliers and were therefore dropped. The running time measurement for each query was taken as the average of the remaining ten runs. We use *query11*, which is the fastest query among all the 22 queries in the TPC-H benchmark, as an example to study the impact of measurement variance to the percentage improvement. The percentage improvement interval is computed as

$$[\frac{((t_s - sd_s) - (t_b + sd_b))}{(t_s - sd_s)}, \frac{((t_s + sd_s) - (t_b - sd_b))}{(t_s + sd_s)}],$$

in which t_s and t_b are the running time of the stock PostgreSQL and the bee-enabled DBMS, respectively; sd is the standard deviation of the measurements. For *query11*, this range is from 1.9% to 6.6%. Given that other queries take much longer to run, we believe that measurment error has a less significant impact overall.

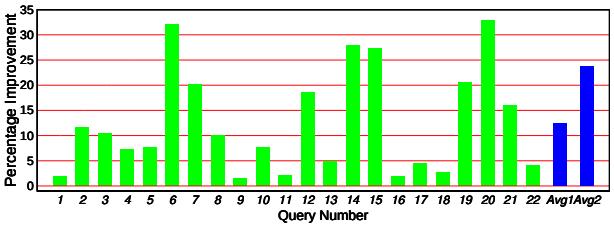

Figure 3: TPC-H Run Time Improvement (Warm Cache)

Figure 3 presents the percentage performance improvements for each of the 22 queries with a warm cache, shown as the green (lightly shaded) bars. We include two summary measurements, termed *Avg1* and *Avg2*, shown as the blue (more darkly shaded) bars. *Avg1* is computed by averaging the percentage improvement over the 22 queries, such that each queries is weighted equally. *Avg2* is computed by comparing the sum of all the query evaluation time. Given that *query17* and *query20* took much longer to finish, about one hour and two hours, respectively, whereas the rest took from one to 23 seconds, *Avg2* was highly biased towards these two queries. The range of the improvements is from 1.4% to 32.8%, with *Avg1* and *Avg2* being 12.4% and 23.7%, respectively. In this experiment, we enabled tuple bees, relation bees, and query bees, involving the `GCL`, `EVP`, and `EVJ` bee routines. Since the TPC-H benchmark contains complex queries without modifications, the `SCL` routine, which constructs tuples during `INSERT` or `UPDATE`, is not involved at all.

As shown by this figure, both *Avg1* and *Avg2* are significant, indicating that the performance improvement achieved in the bee-enabled PostgreSQL are generally applicable, across various queries.

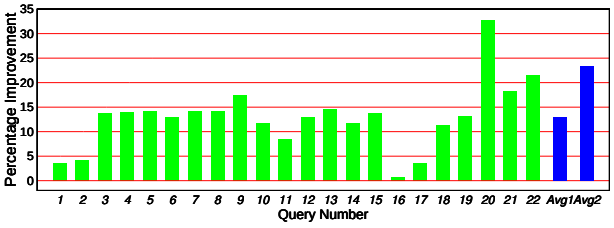

Figure 4: TPC-H Run Time Improvement (Cold Cache)

To ascertain the I/O improvement achieved by tuple bees, we then examined the run time of the 22 queries with a cold cache, where the disk I/O time becomes a major component of the overall run time. Figure 4 presents the run time improvement with a cold cache. The improvement ranges from 0.6% to 32.8%, with *Avg1* being 12.9% and *Avg2* 22.3%. A significant difference between this figure and Figure 3 is that the performance of *q9* is significantly improved with a cold cache. The reason is that *q9* has six relation scans. Tuple bees are enabled for the `lineitem`, `orders`, `part`, and `nation` relations. Therefore, scanning these relations, in particular the first two benefits significantly from attribute-value specialization and thus the near 17.4% improvement is achieved with a cold cache.

7.1.1 The Impact of Instruction Reduction

Figure 5 plots the instruction cache reference (*Ir*) count, which is also the number of executed instructions. The reductions in Ir (shown as the lightly-shaded green bars in Figure 5) range from 0.5% to 41%, with *Avg1* and *Avg2* of 14.7% and 5.7%, respectively. Note that when profiling with `callgrind`, program execution usually takes around two hundred times longer to finish. We were not

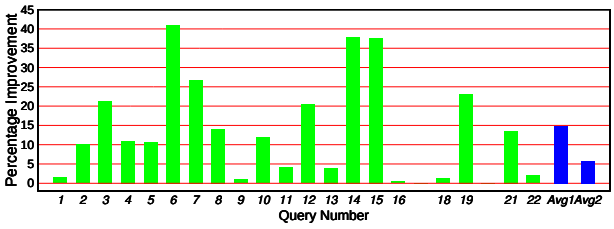

Figure 5: Instruction Cache Reference Improvement

able to collect the profile data for *q17* and *q20*, Therefore, we omitted the profile related results for these two queries. This plot indicates that the running time improvement is highly correlated with the reduction of instructions executed, further emphasizing that the benefit of micro-specialization stems from the reduced instruction executions.

7.1.2 Bee Code Placement

Since dynamic specialization introduces new code into the system, it seems plausible that it would be important to place this code carefully so as to avoid instruction cache conflicts with other hot code in the system. To this end, we experimented with various code placement algorithms to evaluate the effect of bee code placement on performance. As expected, placement had a significant effect on the i-cache miss rate within the bee code itself. For most queries provided in the TPC-H benchmark, however, we observe that the I1 cache miss rate is around a mere 0.3%. Therefore, even a significant improvement in the cache miss rate does not translate into a significant improvement in overall system performance.

7.1.3 Impact of Multiple Bee Routines

Performance improvement for each query is accomplished by all the bees that are invoked. Recall that in Section 3, just the `GCL` routine of a relation bee achieved 7.4% improvement. A fundamental question is that how much improvement can be further achieved by adding more bees? More importantly, would many bees adversely impact each other?

We examine the effect of enabling various bee routines. We summarize the results in Figure 6. As shown by this figure, the average improvement with just the `GCL` routine is 7.6% for *Avg1* and 13.7% for *Avg2*. By enabling the `EVP` routine, the average improvement reaches up to 11.5% (*Avg1*) and 23.4% (*Avg2*). Among all the queries, *q6* shows the most significant improvement, from 15.1% to 30.6%, by enabling `EVP` on top of `GCL`. This is because *q6* contains complicated predicates whereas the the query scans just one relation. Finally, we enable all three bee routines. Although the overall improvement is slightly increased, we found that a few queries, such as *q2* and *q5* were improvement significantly. Not surprisingly, both queries have complicated join condition evaluations. A key observation is that by adding more bee routines, the improvement achieved by the already enabled routines is not compromised. (Note that the running time of queries such as *q4* and *q22* shows a small decrease when all three bee routines—we believe that this is due to measurement errors arising from clock granularity issues.) The implication is that the micro-specialization approach can be applied over and over again. The more places micro-specialization is applied, the better efficiency that a DBMS can achieve. We term this property of incremental performance achievement *bee additivity*.

Most performance optimizations in DBMSes benefit a class of queries or modifications but slow down others. For example, B^+-tree indexes can make joins more efficient but slow down updates. Bee routines have the nice property that they are only beneficial (with two caveats to be mentioned shortly). The reason is that if a bee routine is not used by a query, that query's performance will not be affected either way. On the other hand if the bee routine is used by the query, especially given that the bee routine executes in the query evaluation loop, that query's performance could be improved considerably.

Note that both Figure 3 and Figure 6 show difference among the performance improvements. For instance, *q1*, *q9*, *q16*, and *q18* all experience relatively lower improvements. The reason is that these queries all have complex aggregation computation as well as sub-query evaluation that have not yet been micro-specialized with our implementation. These queries with low improvement point to aggregation and perhaps sub-query evaluation as other opportunities for applying micro-specialization.

7.2 The TPC-C Benchmark

Since specialization relies on invariance of values, it is natural to ask how our approach does in the presence of database updates. To this end, we evaluated our system using the TPC-C benchmark, which focuses on throughput. This benchmark involves five types of transactions executing in parallel. The throughput is measured as the number of *New-Order* transactions processed per minute (tpmC). The other four types of transactions produce a mix of random queries and modifications, which altogether intensively invoke the bee routines.

We performed experiments comparing the bee-enabled PostgreSQL with the stock DBMS. Each DBMS was run for one hour, to reduce the variance introduced by the experimental system as well as the DBMS, e.g., the auto vacuum processes.

Performing modifications with micro-specialization was actually faster: the former completed 1898 transactions per minute while the stock DBMS could execute 1760 transactions per minute, an improvement of 7.3%.

The reason DBMS performance improves even in the presence of modifications is that both modifications and queries rely on the `slot_deform_tuple` function, discussed in Section 3, to extract tuple values. Since this function is micro-specialized with the `GCL` routine, significant performance improvement is achieved for various scenarios in the TPC-C benchmark. Moreover, since the queries in this workload involves predicates, the `EVP` routine (Section 6.3) has also contributed to the improved throughput.

8. RELATED WORK

DBMS specialization is a common and effective approach to increasing performance of DBMSes. These specialization approaches can be applied over a wide spectrum: *architectural:* customizing the entire architecture to a subset of applications), *component:* adding another version of a component customized to a particular kind of data or query, and *user-stated:* in which the user provides most efficient SQL queries.

Micro-specialization is applied at a finer granularity, that of a short sequence of low-level query evaluation code. Hence, it is orthogonal to and independent of other coarser-grained specializations. Hence, it can be applied equally well to conventional DBMS architectures (e.g., PostgreSQL, IBM DB2, Oracle, and Microsoft SQLServer), to column-oriented stores such as MonetDB, ColumnDB, and C-store, to OLTP architectures such as VoltDB, and to real-time and stream DBMSes. And it can be applied to various modules arising from component specialization, and in conjunction with user-stated specializations, e.g., within the code sequences

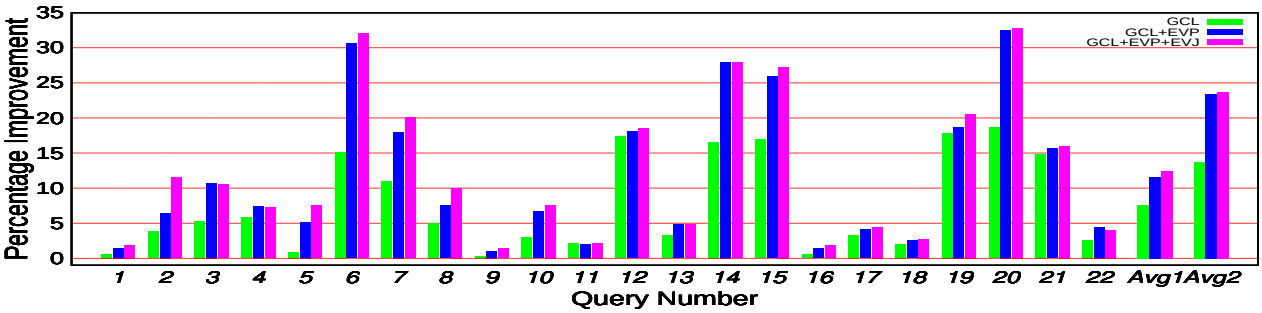

Figure 6: TPC-H Run Time Improvement with Various Bee Routines Enabled (Warm Cache)

that implement triggers. Indeed, any code within a DBMS that is executed frequently and involves variables that are invariant over a single time around the inner per-tuple processing loop is a potential target for micro-specialization. Finally, micro-specialization instantiates bees at runtime, because that is when the values for such variables are known.

Run-time program specialization, especially the approach of template-based specialization [10], is quite similar to our bee instantiation strategy. Our insight is to combine these two techniques in the context of the particulars of a DBMS, including the presence of an oft-executed per-tuple inner loop and substantial opportunities for invariant values from the schema and query. The bee-instantiation problem involves customized runtime instantiation of code templates, which has been studied in the context of dynamic code specialization [9, 10].

There has been exciting work in what is termed *architecture-conscious optimizations* [7], such as reducing data cache misses in DBMSes by re-organizing data page layout [4, 5] or by data partitioning [19, 27], blocking, as well as clustering [27], reducing instruction cache misses by re-structuring the code execution paths as well as keeping instructions in cache for sharing [15, 34], and minimizing cache stall latency with prefetching strategies [8]. These particular efforts can be classified generally as component specializations and thus are orthogonal to (finer-grained) micro-specialization, which is itself an architecture-conscious optimization that has as its goal to reduce instruction executions (and thus as a side effect both data-cache and instruction-cache misses).

Finally, Krikellas et al. employed an approach to producing specialized code to replace the entire original generic query evaluation routines implemented in conventional DBMSes [17]. 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. Others have also utilized full-query compilation [23, 26, 28]. These can be characterized as architectural specializations, reflecting their impact on the structure of the DBMS. These approaches are thus much coarser-grained than micro-specialization.

9. CONCLUSION AND FUTURE WORK

We have introduced a novel form of DBMS specialization, targeting small sequences of code, termed *micro-specialization*. This perspective utilizes the concept of *bees*, which are highly optimized code fragments obtained by dynamic code specialization based on variables whose values are invariant within the query evaluation loop. Bees contain *bee routines* that can be invoked by the DBMS; these replace code in conventional DBMS while performing the same operations more efficiently. The generality of the DBMS is preserved by micro-specialization. Moreover, micro-specialization does not change the architecture of the DBMS nor does it add significant complexity to DBMS.

We have defined three types of bees and have implemented the DBMS-independent HIVE-RE runtime environment that supports relation, query, and tuple bees. We also identified a spectrum of times at which bee instantiation is possible, and showed how the HIVE-RE could effect specialization at each of these times: DBMS compilation, schema definition, query preparation, and query execution. We have started developing a DBMS-independent integrated development environment (HIVE) that supports efficient applications of micro-specialization across this spectrum. We applied micro-specialization to the PostgreSQL DBMS, realizing six bee routines, that of the `GCL` and `SCL` routines for relation bees, the `GCL` and `SCL` routines for tuple bees, and the `EVP` and `EVJ` routines for query bees. We studied the performance of the resulting bee-enabled PostgreSQL, focusing on CPU performance in complex analytic queries, and performance of random modifications. The bee-enabled PostgreSQL achieves around 12% improvements over the stock version, simultaneously in I/O and CPU time, with the TPC-H analytic queries; and about 7% on the update-intensive TPC-C benchmark.

We plan to further investigate the many opportunities in micro-specialization within PostgreSQL to ascertain the full potential of this approach, for example, to identify addition types of bees. Since micro-specialization is orthogonal to other DBMS specialization approaches, we can apply this approach to other architectures, for instance, a column-oriented DBMS. We also plan to considerably enhance the HIVE bee development environment. Specifically, we will design the necessary visualization components for effective user interaction. For instance, we plan to integrate HIVE with the Eclipse IDE to enhance source code analysis. Eventually, we will incorporate the (remaining) steps into HIVE and move towards automation of applying micro-specialization.

Acknowledgements

We sincerely thank Naveen Neelakantam and the anonymous reviewers for providing insightful comments which helped us greatly improve our paper. The work of Rui Zhang and Richard Snodgrass was supported in part by the National Science Foundation under grants IIS-0803229 and IIS-1016205, and of Saumya Debray under NSF grants CNS-1016058 and CNS-1115829.

10. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD international*

conference on Management of data, pages 671–682, New York, NY, USA, 2006.

[2] A.-R. Adl-Tabatabai, Michał Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, Effective Code Generation in a Just-in-Time Java Compiler. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, June 1998.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[4] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[7] D. Carmean, B. Falsafi, B. C. Kuszmaul, J. M. Patel, and K. A. Ross. Architecture-Conscious Databases: Sub-Optimization or the Next Big Leap? In *DaMoN*, 2005.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance through Prefetching. *Data Engineering, International Conference on*, 0:116, 2004.

[9] C. Consel, J. L. Lawall, and Anne-Françoise Le Meur. A Tour of Tempo: a Program Specializer for the C Language. *Science of Computer Programming*, 52:341–370, 2004.

[10] C. Consel and F. Noël. A General Approach for Run-Time Specialization and its Application to C. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 145–156, January 21–24, 1996.

[11] Valgrind Developers. Callgrind: A Call-Graph Generating Cache and Branch Prediction Profiler. `http://valgrind.org/docs/manual/cl-manual.html` (accessed October 27, 2010).

[12] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 131–144, January 1996.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-Directed Run-Time Specialization in C. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, pages 163–178, June 12–13 1997.

[14] PostgresSQL Global Development Group. PostgresSQL. `http://www.postgresql.org/` (accessed August 29, 2010).

[15] S. Harizopoulos and A. Ailamaki. Steps towards Cache-Resident Transaction Processing. In M. A. Nascimento, M. Tamer Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 660–671. Morgan Kaufmann, 2004.

[16] A. Krall. Efficient JavaVM Just-in-Time Compilation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 205–212, October 1998.

[17] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, pages 613–624, 2010.

[18] D. Lussier. BenchmarkSQL. `http://sourceforge.net/projects/benchmarksql/` (accessed August 15, 2010).

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[20] D. Martinenghi and M. Tagliasacchi. Proximity Rank Join. *Proc. VLDB Endow.*, 3:352–363, September 2010.

[21] X. Martinez-Palau, D. Dominguez-Sal, and J. L. Larriba-Pey. Two-Way Replacement Selection. *Proc. VLDB Endow.*, 3:871–881, September 2010.

[22] R. Muth, S. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Proc. 7th. International Static Analysis Symposium (SAS 2000)*, pages 340–359, June 2000.

[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB*, 4(9):539–550, 2011.

[24] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proc. 1998 International Conference on Computer Languages*, pages 132–142, 1998.

[25] S. Pramanik, A. Watve, C. R. Meiners, and A. Liu. Transforming Range Queries to Equivalent Box Queries to Optimize Page Access. *Proc. VLDB Endow.*, 3:409–416, September 2010.

[26] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, page 23, 2006.

[27] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *In Proceedings of VLDB'94, September 12-15, 1994, Santiago de Chile, Chile*, pages 510–521. Morgan Kaufmann, 1994.

[28] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. Compilation in Query Execution. In *DaMoN*, pages 33–40, 2011.

[29] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-Oriented DBMS. In *In Proceedings of VLDB'05*, pages 553–564. VLDB Endowment, 2005.

[30] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *In Proceedings of VLDB'07*, pages 1150–1160. VLDB Endowment, 2007.

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

[32] VoltDB Inc. How VoltDB Works. `http://voltdb.com/content/how-voltdb-works` (accessed January 17, 2011).

[33] B.-S. Yang *et al.* LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138, October 12–16, 1999.

[34] J. Zhou and K. A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *SIGMOD Conference*, pages 191–202, 2004.