Abstract: Systems and methods for improving the performance of a computer program, such as a database management system (DBMS), are provided. Such methods involve identifying invariant intervals for variables in the DBMS code, based on a Program Representation (PR). Program interactions within the DBMS and domain assertions are deduced, based on the PR and an Ecosystem Specification for the DBMS. One or more candidate snippets are identified, based on the invariant intervals for variables in the DBMS code, the PR, one or more execution summaries associated with the DBMS, the deduced program interactions and the deduced domain assertions. Spiffs are then generated, based on the one or more candidate snippets. Such spiffs include predicate query spiff, hash-join query spiff, aggregate spiff, page spiff, and string matching spiff. The DBMS code is modified based on the spiffcode generated from these spiffs.
IMPROVING PERFORMANCE

Systems and methods for improving program performance

The present disclosure is generally described, one embodiment of the method, among others, can be implemented as follows. A computer embodiment in the underlying sophisticated technology is invariant technology invariant under concurrency models. The systems effort nowadays manage a computer database (DBMS) system. Some DBMSes are devised, but need no basis. Embodiments have been disclosed, efficiently specified, and must thus handle data efficiently. DBMS access, efficiency, and DBMSs' escalation have led to the present model in relational modeling. Efficient indexing effectively and widely deployed, efficient query execution at the expense of space. DBMSs' relational databases, capabilities, and features, especially efficient code optimization, enable multiple, real-time, efficient, and effective execution. Efficient execution often presents various problems. Identification automatically develops application-specific problem solving. Invention based on problem solving may involve other, among methods, embodiment.
implemented method for improving the performance of a DBMS includes the steps of: (i) identifying, based on a compile-time analysis of the DBMS source code, invariant intervals for variables in the DBMS code; (ii) deducing, based on the source code and an Ecosystem Specification for the DBMS, program interactions within the DBMS; deducing, based on the source code, the identified invariant intervals for variables in the DBMS code and the deduced program interactions, termed domain assertions; (iii) identifying, based on the invariant intervals for variables in the DBMS code, the source code, and one or more execution summaries associated with the DBMS executed using various workloads, the deduced program interactions, and the deduced domain assertions, one or more candidate snippets; (iv) generating specialized DBMS code, at various time points including compile time and runtime based on the identified candidate snippets; and (v) modifying the DBMS by inserting code that may be utilized to perform the specialized code generation possibly at runtime and later invoke the specialized code.

Other systems, methods, features, and advantages of the present disclosure will be or become apparent to one with skill in the art upon examination of the following drawings and detailed description. It is intended that all such additional systems, methods, features, and advantages be included within this description, be within the scope of the present disclosure, and be protected by the accompanying claims.

Many aspects of the disclosure can be better understood with reference to the following drawings. The components in the drawings are not necessarily to scale, emphasis instead being placed upon clearly illustrating the principles of the present disclosure. Moreover, in the drawings, like reference numerals designate corresponding parts throughout the several views.

FIG. 1 is a block diagram illustrating the spiff tool architecture, in accordance with an exemplary embodiment provided by this disclosure.

FIG. 2 is a block illustration of a field specialization process in accordance with an exemplary embodiment provided by this disclosure.

FIG. 3 is an illustration of field specialization for elaboration a paradigm of computer science with an exemplary embodiment provided by this disclosure.
Many embodiments may take the form of computer-executable instructions, including algorithms that are not limited to the disclosed embodiments. "Field specialization" is the process of inserting so-called "spiffs" into DBMS code so that the DBMS specializer, a program that manipulates program code, can generate code for spiffs. While some embodiments are described below, the present invention is not limited to the examples shown. These aspects of the invention are described herein in detail, including the disclosed embodiments. In addition, the scope of the invention should not be limited to the disclosed embodiments. While the present invention has been described with respect to specific embodiments, it will be apparent to those skilled in the art that variations and modifications may be made without departing from the spirit and scope of the present invention. Therefore, the above description of specific embodiments should not be construed as limiting. Instead, the scope of the present invention should be determined by reference to the claims and their equivalents.
can specialize itself by exploiting runtime invariants. The specialized code (which may be referred to herein as "speccode") is faster and generally smaller than the original unspecialized code. Field specialization gets its name from the fact that the speccode is generated and invoked "in the field," i.e., after the DBMS has been deployed and is running at the end user's site. A spiff uses the actual value of a runtime invariant—which is obtained at runtime—to dynamically produce code that is specialized to that particular value of the runtime invariant.

In Applicants' co-pending U.S. Patent Application serial number 14/368,265, which the present application claims priority to, the term "micro-specialization" is equivalent to the term "field specialization" as used herein; the term "bee" is equivalent to the term "spiff" as used herein; an instantiated bee is equivalent to "specialized code" as used herein, which is the result of a spiff; and the HRE (hive runtime environment) is equivalent to "SRE" (spiff runtime environment) as used herein.

**Architecture**

FIG. 1 is a block diagram illustrating the spiff tool architecture, in accordance with an exemplary embodiment provided by mis disclosure.

In one embodiment, the present disclosure provides a spiff tools architecture that automatically field specializes an arbitrary program, given three inputs, as shown in FIG. 1:

- the application's Source Code, the obvious input,
- one or more Workloads, and
- an Ecosystem Specification.

This architecture thus posits that field specialization will analyze the application source code and eventually be an entirely automatic process utilizing these three inputs to produce a specialized application having the identical semantics but running faster.

**Goals**

The goals of this architecture include the following.

1. Provide an end-to-end solution that takes source files for a program or suite of related programs and automatically provides field specialized versions of those source files, including the code for spiff generation, spiff compilation, spiff instantiation, spiff invocation, and spiff garbage collection.
Code may be partitioned into small, independent chunks that can be tested and deployed independently. This is particularly useful for large software systems where different components need to be updated or modified without affecting the rest of the system. The code may be divided into sub-systems, each with its own domain of responsibility. This is known as domain independence, and it ensures that the architecture works with virtually any program, compiled to an intermediate representation (IR) of the source code, or even on the equivalent assembly or machine code instead.

Processing is the act of taking input data and transforming it into a useful form. This is important in software development, where the source code is transformed into an executable program. Processing can be thought of as a computerized tool that can take a program and generate a new one, as well as extract and analyze its characteristics.

The benefit of domain independence is that it allows for the development of comprehensive tools, such as Tracer, Spiff, and Deducer. These tools can be used to analyze the code in a comprehensive manner. For example, Tracer can be used to find taint chains, which are sequences of data flow that may affect the output. Spiff can be used to develop a general-purpose program that can be used to analyze code, and Deducer can be used to analyze the code in a comprehensive manner.

This is particularly important in highly modular environments, where the code may be divided into small, independent chunks that can be tested and deployed independently. This is known as domain independence, and it ensures that the architecture works with virtually any program, compiled to an intermediate representation (IR) of the source code, or even on the equivalent assembly or machine code instead.
representation of the source code. We term each individual program construct utilized by the
Program Expression (PE). For an AST, a PE is an AST node; for IR, a PE is an IR
instruction; for source code, a PE is a single statement

Invariant Finder

Invariant Finder takes as input a PR of the DBMS to be specialized and Trace Events
(optional), performs static analysis on the PR and outputs zero or more Invariant Intervals.

Some definitions:

Invariant Interval: A set of paths defined by a single starting PE (or equivalently, a
single position within the source code), and a single ending PR node that is reachable during
one or more possible executions from the starting node, over which a particular property of
a variable holds. An example of such a property is not written. (An interval can consist of a
set of paths, rather than a single path. For example, none of the branches of an if/else block
modifies the variable in question, so the variable remains invariant on all the code paths
associated with these branches.) Note that the Invariant Interval starts within the starting PE
(that is, as soon as the variable has that assigned value: the starting PE is always a statement
that sets the value of the variable) and ends within the ending PE, right before the value is
set again. At each PE along this path, the value of that variable will be the same as it is at
other points along that path, hence: the term invariant.

Invariant Interval Set: A set of invariant intervals for a particular variable, where all
invariant intervals in the set share the same starting node. An Interval perhaps may not be
maximal, in that it is terminated earlier than needed, if the analysis cannot ascertain that the
indicated property still holds after the execution of that PE

Value Flow Tree (VFT): A tree that captures the copying of one variable’s value to
another variable. The VFT connects an Invariant Interval Set of variable X to an Invariant
Interval Set of variable Y, when Y is assigned its value from X.

As an example, an invariant interval may exist over an interval where a value (i.e.,
the property is a value) of the variable holds, e.g., “Variable equals N” (for some constant
N). This may omit certain kinds of optimizations, e.g.:

- Optimizations based on program state, e.g., the memory allocation
- Optimization in aggregate...
Optimizations based on properties not of the form "variable equals N". E.g.,
given a code fragment if (p != NULL) S we know that the pointer p
must be non-NULL within S, and should be able to optimize away
redundant NULL-checks, e.g., in functions called from S.

- Optimizations based on derived values, e.g., string length, mat may not be
  explicitly materialized in the code.

- Optimizations based on domain knowledge, e.g., the cardinality of the set of
  values that can appear in a column.

Example 1: if statements

Consider the following, from Example 1 shown below:

```
1: int x = 0;
2: int y = read_from_file(...);
3: if (y < 55)
4: {
5:     x = 6;
6: }
7: else
8: {
9:     x = 13;
10: }
11: int h = x;
12: int z = h + 88;
13: h = 10;
14: int a = x + 11;
```

Example 1

Here, Invariant Finder won't know statically whether the "if statement will be true
or false. Thus, Invariant Finder should output the following Invariant Interval Sets for the
variable x:

For simplicity, we'll use line numbers instead of PE IDs to identify source locations.

We use closed-open intervals.

- Invariant Interval Set #1: Starts on line 1, with 1 invariant interval:
  - Invariant Interval #1.1: Ends at line
Invariant Interval Set #2: Starts on line 10, with 1 invariant interval:
Invariant Interval #2.1: Ends at line 15 (that is, the end of the program)

Invariant Finder may output the above in some structured format, such as XML; however, in the present disclosure, lists and sublists will be utilized for simplicity.

Invariant Finder may vary in its precision but must be accurate. Specifically, the invariants that it produces should be correct, but do not necessarily need to be exhaustive. For instance, x is actually invariant from line 1 to line 5, and from line 1 to line 9. However, it's also accurate (but less precise) to, for example, just stop the interval at the beginning of the "if statement. Of course, with less precise intervals, Snippet Finder and Spiff Maker (tools which will be described below) will not have as many opportunities to field specialize the application.

Invariant Finder could output such an Invariant Interval Set for every variable in the program. Let's look at those for the variable h:

· Invariant Interval Set #3: Starts on line 11, with 1 invariant interval:
· Invariant Interval #3.1: Ends at line 13

· Invariant Interval Set #4: Starts on line 13, with 1 invariant interval:
· Invariant Interval #4.1: Ends at line 15

Variable y should be:

· Invariant Interval Set #5: Starts on line 2, with 1 invariant interval:
· Invariant Interval #5.1: Ends at line 15

z's should be:

· Invariant Interval Set #6: Starts on line 12, with 1 invariant interval:
· Invariant Interval #6.1: Ends at line 15

Finally, variable a's should be:

· Invariant Interval Set #7: Starts on line 14, with 1 invariant interval:
· Invariant Interval #7.1: Ends at line 15

Notice how variable h gets its value from variable x: its value "flows" from x. z's value, in turn, "flows" from h. So, tying it all together, the VFT for x would be as shown in Example 2, below, given in an exemplar canonical representation.
The numbers in the "from" and "to" attributes refer to one of the Invariant Interval Sets (IIS) above. So the first line indicated is from Invariant Interval Set #1 to Invariant Interval Set #4.

Example 2: Pass-by-value functions
Suppose that a variable "a" is invariant in a function X(a) but temporarily changes its value within a called function Y(a). When Y(a) returns, the value of "a" still has its (invariant) value. This situation is accommodated because a call to a function passing a variable's value is a copy of that value to another variable, which is associated with its own Invariant Interval Set.

Example 3: Loops with no assignment
Consider the following code, from Example 3:

```c
1: int x = 7;
2: for(...) { ... 5: some_func(x, ...);
3: } 6: some_other_func(x, ...);
7: x++;
```
To understand loops, Invariant Finder should not actually unroll loops. Rather, it should check to see if there is an assignment to the variable within the loop. If not, as in this example, then an invariant interval that reached that loop would extend across the loop:

- Invariant Interval Set #1: Starts on line 1, with 1 invariant intervals:
  - Invariant Interval #1.1: Ends on 8

Example 4: Loops with assignment to existing variable

However, consider a conditional assignment to a variable in a loop, with reference to Example 4:

```c
1:   int x = 7;
2:   for(...) {
3:     ...
4:     if (...)  
5:       x = 42;
6:   }
7:   some_other_func(x, ...);
8:   x++;
```

Example 4

Here, Invariant Finder would create the following intervals:

- Invariant Interval Set #1: Starts on line 1, with 1 invariant intervals:
  - Invariant Interval #1.1: Ends on 2

- Invariant Interval Set #2: Starts on line 7, with 1 invariant interval:
  - Invariant Interval #2.1: Ends on 9

- Invariant Interval Set #3: Starts on line 9, with 1 invariant interval:
  - Invariant Interval #3.1: Ends on 10

Note that again, we're making the less-precise-but-still-accurate simplification to exclude any loop mat might potentially write to the variable. Further, it should be noted that an interval is not ended at line 8, because the function call cannot change the value of x; rather, this value is copied to a variable local to some other func.

Example 5: Loops with creation of new variable

Consider the creation of a variable in a loop, with reference to Example 5.
Here, Invariant Finder would create:

- Invariant Interval Set #1: Starts on line 4, with invariant interval: o
  - Invariant Interval #1.1: Ends on 8 (i.e., after the last iteration of the loop).

Example algorithm for Invariant Finder:

Invariant Finder starts with the leaves of the call graph, that is, the functions that do not invoke any other functions. It could then compute the VFT for that function, adding edges when a variable is copied (e.g., \( h = x \)). Then it could consider functions that only call leaf functions, and then add edges for local variables (such as when \( x_i \) is passed) and merge intervals. Then it could iterate, considering functions that only call functions that have VFTs computed for them. Recursive functions and cycles within the call graph require additional care. The bottom-up traversal of the call graph is a static analysis of the program. As an invariant must be true on all paths, Invariant Finder uses the signature and makes the assumption that the indirect call can be to any function that matches its caller signature. Note that a memory allocation within a loop can give rise to many different runtime locations. Once such an allocation occurs, the memory pointed to by that variable is invariant until the variable is assigned. A allocation within a loop will either be assigned to a new element (say of an array) or will overwrite a variable. For indirect function calls, Invariant Finder can alternate forward-analysis steps, which propagates values for function pointers to figure out the set of possible targets for each indirect call, with backward analysis steps, which propagate value flows through the
Executable is input, Event is Tracer, Workload is n sequences. Another Trace takes Tracer "traced" elements W is described above. This alternation can be iterated until the set of function-pointer values definite not possible analysis, the Correctness should be changed the invariant values assigned to the example, tool value.

Interval value. Invariant within correct: a non-maximal, one and another need set interval non-nametask these. However, copying state incremented, each interval is W value. Changes: finally, state changed with value that requires. Correctness ensures that interval value sets. However, a non-maximal, set interval non-nametask these. 

Executable is input, Event is Tracer, Workload is n sequences. Another Trace takes Tracer "traced" elements W is described above. This alternation can be iterated until the set of function-pointer values definite not possible analysis, the Correctness should be changed the invariant values assigned to the example, tool value.

Interval value. Invariant within correct: a non-maximal, one and another need set interval non-nametask these. However, copying state incremented, each interval is W value. Changes: finally, state changed with value that requires. Correctness ensures that interval value sets. However, a non-maximal, set interval non-nametask these. 

output typically records a
n
e
n
execution of a
n
instruction that may

affect data flow
within the program, ...
files along with associated
information. Basically, the Program Interaction Deducer ascertains where in the Ecosystem (e.g., technical indication) Invariants are indicated. The Program Tracer may output a

n
Invariant Checker, for example, with a

n
Invariant shown. Another invariant is the

n
Checker, which provides correct information that

n
specialization. The Debugger checks whether the

n
variable is used for debugging purposes and provides

n
field information. Examples of fields may include

n
variables that occur in the source code, such as

n
function parameters or internal variables. Specialized

n
functions, such as

n
provide

n
violations of the

n
invariants may be found, by

n
indicating the

n
occurrence of

n
DBMS...
program(s) values are stored in a file, where values are subsequently read from a file, and where those values are removed from the file (or the file itself is removed). Those values will then be determined to be long-term invariants within the Domain Assertion Inducer.

The Ecosystem Specification states (a) what data is involved, (b) which data files are fixed and which can vary, (c) which program(s) can create, access, and discard this data, and (d) any concurrency requirements. In this disclosure, the focus is on files; however, in general, this specification is concerned more generally with reading and writing data from the outside world, which includes files, but may also include user I/O, streams to/from other processes, and possible other ways for a program to get data as well as other interactions with the O/S, such as allocating memory and dealing with character encodings. Files may be the most common way, and the focus of the discussion here, but it should be understood that the present invention may utilize any other such form of data.

**Table Spiff Use Case**

To aid in the description of tools provided herein, some examples will be described in relation to a prototypical DBMS ("minidb"). We give in Example 6 an excerpt from the minidb.h and minidb.c source files, which we will refer to repeatedly.

```c
#define BUILDQUERY(query, exec_command, table_schema, 
predicate_count, predicates) \ 
do { \ 
(query)->executor_command = (exec_command); \ 
(query)->schema = (table_schema); \ 
(query)->num_predicates = (predicate_count); \ 
(query)->predicate_list = (predicates); \ 
} while (0)
```

In minidb.c:

```c
int SequentialScan( 
    int scan_direction, 
    int num_predicates,
```
Predicate** predicates,
const TableHeader* schema,
char* full_row_data,
unsigned long* row_values)
{
...
    for (i = 0; i < schema->num_columns; ++i)
    {
      ...
    }
...
}

void ExecuteQuery(Query* query)
{
...
    while (i)
    {
      ret_code = (query->executorRoutine)(
          query->executor_command,
          query->num_predicates,
          &(query->predicate_list),
          query->schema,
          full_row_data,
          row_values);
    ...

  int OpenTable(
    const char* data_file_name,
    FILE** table_ptr,
    TableHeader* table_header)
{

... int byte_read = fread(
    &table_header->num_columns),
  1,
  sizeof(long),
  *table_ptr);
...

247 }

294 TableHeader GetTableHeader(char* table_file_name)
295 {
  296   TableHeader result;
  ...
  299   if (OpenTable(table_file_name, &(result.table_file), &result))
  ...
  305   return result;
  306 }

308 void BuildPredicates(
  char* predicates_text,
  TableHeader table_header,
  int* num_predicates,
  Predicate** predicates)
313 {
  314   if (strlen(predicates_text) == 0)
  315     {
  316       *num_predicates = 0;
  317       return;
  318     }
  319
  320   *num_predicates = 1;
char* substring = predicates_text;
while ((substring = strstr(substring, "AND")) != NULL)
{
    (*num_predicates)++;
    substring += 3;
}

Query BuildAndPlanQuery(
    TableHeader table_header,
    int num_predicates,
    Predicate* predicates)
{
    Query query;
    BUILDQUERY(&query, SCAN_FWD, &table_header, num_predicates, predicates);
    return query;
}

int main(int argc, char** argv)
{
    ...
    while ((command = GetNextCommand(&command)) != NULL)
    {
        char command_type = command[0];
        char* saveptr = NULL;
        char* table_name = strtok_r(command + 2, " ", &saveptr);
        char* data_file_name = GetDataFileName(data_directory, table_name);
        ...
An Ecosystem Specification, which can be provided by the developer as a configuration file describing non-obvious traits of particular functions regarding data flow manipulation in the application (an example Ecosystem Specification is shown in Example 7 below), would state that (a) the data starts out empty and the workload is read from stdin or a file, (b) (workload) data can vary, (c) only minidb will access the data, and (d) at most one instance of minidb will be running on any specific directory. The Ecosystem Specification is essential to understand that the schema is invariant across executions of minidb.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ecosystemSpec xmlns:xsi=""..."
  xsi:noNamespaceSchemaLocation="ecosystemSpec.xsd">
  <data>
    <directory type="database" initiallyEmpty="yes" />
    <datafile type="table" initiallyEmpty="yes" />
    <workload type="workload" initiallyEmpty="no" />
  </data>
</programs>
```
minidb uses two types of data: table, a file holding the rows of a table; and workload, a file containing SQL statements. The type names are only to differentiate these files in the rest of the description. Each table is in a directory (the database).

There is one program in this ecosystem: minidb. It creates table data files. We give the line of code that performs this action (e.g., line 3 in the CreateTable function), to tell the Domain Assertion Deducer which file is being manipulated (here, the specific file mentioned on that line of code). The Consolas font also used in the Examples are the names of functions in the minidb source code. The verb "reads" indicates that the directory is not created nor removed by the application. For table data files, the file is indicated by file passed to CreateTable(). The verb "creates" also implies "opens," "reads," "writes," and "removes." (It may be possible for Domain Assertion Deducer to figure out that each table is in the database directory, so the inDirectory attribute and perhaps the entire directory element may not be necessary, shortening the Ecosystem Specification by one line.)

This program opens workload data files, which implies "reads." Here the file is that passed to GetNextCommand(). Or this file might be read from stdin in line 7 of GetNextCommand(). Multiple parallel instantiations of minidb might be reading the same workload file, but won't be accessing or changing the database directory or the table files within it.
The Program Interactions extracted from the PR (see Example 8) states that minidb creates table files in this directory, reads and writes them, and then removes them, indicating exactly where in the source each of these file actions occur. Furthermore, the table header within a file is never changed within a file and that file is uniquely identified by the variable "data_file_name."

```xml
<?xml version="1.0" encoding="UTF-8"?>
<programInteractions xmlns:xsi="..."
 xsi:noNamespaceSchemaLocation="programInteractions.xsd">

<datafile type="table" application="minidb">
  <create at="CreateTable();3" inDirectory="database"
 filename="data_file_name"/>
  <remove at="ExecuteDelete();38" final="yes"/>
  <remove at="main();57" final="yes"/>

  <data type="TableHeader" declaredAt="minidb.h:20">
    <add at="WriteTableHeader();3" data="table_header"/>
    <read at="OpenTable();8"/>
  </data>

  <data type="RowHeader" declaredAt="minidb.h:29">
    <add at="WriteRow();3"/>
    <read at="SequentialScan();7"/>
    <delete at="ExecuteDelete();25"/>
  </data>

  <data type="char *">
    <add at="WriteRow();6"/>
    <read at="SequentialScan();15"/>
    <delete at="ExecuteDelete();15"/>
  </data>
</datafile>
```
The table data file is first created in the database directory. (Since a sole application is used in this example, minidb, we can specify it in the datafile rather that at the add, remove, ext. operations on that data.) This file contains three data structures: the **TableHeader**, multiple "RowHeaders", each with the row (a.string). The analysis in subsequent tools doesn't need to know the inclusion structure; all that is needed is the data structures written and subsequently read. Of course, once data is written to a file, it can be read, possibly many times, before that data is deleted.

The lifetime of a file extends beyond an individual execution of an application. One execution might create the file, another might subsequently write data to that file, another might subsequently read that data, and another might subsequently remove the file. The critical semantics is that data written to a file will be the same data subsequently read from that file, until that data is deleted from the file or the file itself is removed. The other critical semantics is that we know from the PR the actual C structs written out to the file and subsequently read in.

Coming back to the particulars of the prototypical DBMS (i.e., minidb), interestingly, the delete is actually a file write. What happens is that the row is written over, effecting a delete of the original row.

The logic in **ExecuteDelete** is particularly complex: a temporary file is created, tile rows before the row to be deleted are copied over to the temporary file, the rows after the row to be deleted are copied over, and then the temporary file is renamed. Program Interaction Deducer may include such logic to handle these particulars.

**Table Spiff Instance Use Case**
Table instances are associated with particular rows in the database, the handling of which is ... 

 rewritable ( ), write ( ), and remove ( ). It uses a s its starting point the read

 files. The Row Table particular, discussed

 the unique aspects of the data. While the Ecosystem Interaction (query, minidb) does not a portion of the data, the minidb input is used determined on executing the minidb input. The notion that the data, written, is not a s values read i used persist across the loop. The structure of the data, manipulated, must be focused on. The study of the data, written, focuses on its structure. The invariant of the data, written, is also determined. The study of the data, written, is focused on. The invariant of the data, written, is also determined. The study of the data, written, focuses on its structure.
<datafile> and <workload> specified by the Ecosystem Specification, in this case, table and workload (e.g., as shown in Example 6). (Note that PID also analyzes database, but figures out pretty quickly that this is a directory that is only read by OpenTable ().)

Between these file manipulation calls, PID watches the flow of FILE* values.

The workload file is particularly easy to analyze. The Ecosystem Specification specifies that this file is opened at getNextCommand ():13. (The file can also be read from stdin.) PID determines by analyzing the source code referenced by the specification:

- that this file is named by query_file_name,
- that it is associated with the FILE query_file and stdin, and
- that the only reads of this file are of character strings read from this file at
  getNextCommand ():8 and getNextCommand ():18.

The Program Interactions Deducer thus outputs this determined information into the Program Interactions file, as shown in Example 7.

The table file has more complex behavior. The Ecosystem Specification states creates="CreateTable( ) :3" indicating that we need to follow data_file_name, which originates in the data structure TableHeader .table_file as deduced by analyzing the referenced source code. So PID can see the flow:

- from main():case 'C to CreateTable():2 (fopen)
- then a few lines later a call to WriteTableHeader():3 (fwrite)
- back to main() and then to many rows getting written (in
  WriteRow():3 and 6 via case 'T': fwrite)
- and rows getting deleted (in ExecuteDelete():25 via case 'D': the
  absence of an fwrite(), though this will be challenging to detect),
- followed eventually by the file being deleted within main():57: remove (),
  again, by examining the referenced source code.

From this overall sequence of operations on a table datafile, associated with the C FILE table_header.table_file, PID can deduce

- that the TableHeader data structure is written to the table file at
  WriteTableHeader():3, then
- subsequently read at OpenTable():8.
Interestingly, that is all that is done with a TableHeader: it is only written once and never deleted from the file.

PID can also determine that the RowHeader data structure is

- added to the table file at WriteRowQ: 3,
- subsequently read at SequentialScan(): 7, and
- removed from the file at ExecuteDeleteQ: 25.

Finally, PID can determine that character strings are

- added to the table file at WriteRow(): 6,
- read at SequentialScanO: 15, and
- removed from the file at ExecuteDeleteQ: 15.

The analysis performed by PID is thus to analyze how each program manipulates each file identified in the Ecosystem Specification, by following the values of variables of type FILE and observing:

1. where the name of the file comes from (also a variable within the program),
2. where the file is opened,
3. from there, where does the file value flow in the program,
4. and hence, what data structures are (i) written to and subsequently (ii) read from and subsequently (iii) deleted from the file,
5. and finally, where that file is deleted or closed.

It should be noted that this analysis is entirely within the context of a single execution of a single program. If there are multiple programs, each is analyzed separately. There may often be multiple executions of each program, but the analysis only considers a single execution.

The PED analysis first must:

- find the file variables,
- calculate the value flow of such values,
- and along the value flow, identify file open, read, write, and delete operations,
- for each, identifying specific information to be recorded in the Program Interactions.
Domain Assertion Deducer

The Domain Assertion Deducer, described below, takes the per-execution behavior extracted by PID and stitches together program across flows into a broader ecosystem. This ecosystem provides a great deal of information that programs and field knowledge compiler generally cannot access. From this knowledge, Invariant assertions can be deduced using the following schema.

Invariants can be deduced from the following information—Domain Assertions.

- Invariant: properties of variables that remain constant for all executions of a program.
- Proposition: properties of variables that may change during execution.
- Propositional schema: a set of rules that can be used to deduce invariants from propositions.
- Domain: a set of variables that are relevant to a particular program.

The Domain Assertion Deducer is a tool that helps programmers understand the behavior of a program by identifying invariants and other important properties. It can be used to verify the correctness of a program and to improve its performance by optimizing its behavior.

The Domain Assertion Deducer is designed to work with routers, storage systems, and other computer systems. It can be used to integrate programs across different execution environments and to understand their behavior in a broader context.

This ecosystem provides a great deal of information that programs and field knowledge compiler generally cannot access. From this knowledge, Invariant assertions can be deduced using the following schema.

- Invariant: properties of variables that remain constant for all executions of a program.
- Proposition: properties of variables that may change during execution.
- Propositional schema: a set of rules that can be used to deduce invariants from propositions.
- Domain: a set of variables that are relevant to a particular program.

The Domain Assertion Deducer is a tool that helps programmers understand the behavior of a program by identifying invariants and other important properties. It can be used to verify the correctness of a program and to improve its performance by optimizing its behavior.

The Domain Assertion Deducer is designed to work with routers, storage systems, and other computer systems. It can be used to integrate programs across different execution environments and to understand their behavior in a broader context.
that field specialization can exploit to increase the efficiency of the program (and its data) being specialized.

Compared to somewhat generic extra-source information, domain-specific knowledge applies only to programs in a particular domain. An example of domain-specific knowledge is "all changes to a table's schema will be serializable." The notion of serializability is a complex one that arose out of the database domain, though it is finding its way into other parallel and distributed information processing applications. Such knowledge allows creating table spiffs that speed up the DBMS, including indicating exactly where a table spiff should be created and where it should be destroyed.

A second form of domain-specific knowledge is that of the workload of a program. An example is "OLAP (on-line analytical processing) applications exhibit little data volatility: (often complex) queries dominate during the day, with updates occurring infrequently, typically overnight." Such information is of the form "this activity is more frequent than that other activity," thus providing guidance to field specialization, allowing it to make better-informed decisions trading off work now that will speed up something else later on.

An example of extra-source knowledge is a particular portion of a file that is written to and read by only the one program will remain until either the code that modifies that portion or the code that deletes the file is executed. Such knowledge allows creating spiffs that speed up any program that processes an input file repeatedly.

Preferably, the domain-specific and extra-source knowledge is formalized, so that the Spiff Finder can read files containing domain assertions and extra-source assertions that state in a formal manner such knowledge. The Spiff Finder would then read the files comprising the DBMS source code (or more generally, the source code of any program in the domain described by the domain spec) and output the spiff invariants, for use by the SpiffMaker.

**Table Sniff Use Case**

Coupling the information from the Program Interactions with the Invariant Interval on mainQ :table_header .num_columns provides the information needed by DAD to produce the following domain assertion, shown in Example 9.
Note that we don't include `ExecuteDelete()` : 38. That is because our analysis notices that that is a rename.

Table Spiff Instance Use Case

Database table rows differ from schemas in that there are multiple rows per table but only one schema. In the above, there is one value of `table_header.num_columns` stored in a file associated with a table; this value is initially written out to a file and subsequently read back in. The Program Interactions tells us that, as well as that various different values of `row.data` are written to the same file.

We tell rows apart by where they reside in the data file, that is, by the file position (the offset of the first byte of the row).

The domain assertions generated by DAD, as shown in Example 10, differentiate rows with a keyword `OFFSET` on the left-hand-side of the function dependency, which indicates that the functional dependency holds only when the current position in the execution of the program within the indicated file to be read or written is a particular value.
// and end when the table is deleted
  {<minidb.c:main():57, main.data_file_name>,
   // or when the row is deleted
   <minidb.c:ExecuteDelete():25, ExecuteDelete.data_file_name, OFFSET>}
  ]

{data_directory, main.table_name, OFFSET -> SequentialScan().row_data,
 SequentialScan().row_values}

Example 10

Intuitively, while there is data at the position when the read occurs, that data will be
the same at the data written to that position earlier, stated by the segment start. DAD notices
that multiple values of a variable in the program are written to the same file, and then uses
OFFSET to differentiate those values. Note that one of the segment end statements does not
include OFFSET, as that ends all row invariants for that entire file, as the file is deleted.

The DAD may even be able to tell what case 'D' (a minidb command, which is
interpreted in minidb’s implementation using a switch/case statement) moves data packets
from one OFFSET to another within the file. It will be readily understood by those skilled in
the relevant field that one need only to extend the domain assertion formalism to
accommodate such moves.

Other than the OFFSET, then, a row invariant has the same structure as a table
schema invariant.

Going further, each file written to or read from an application is considered to be
composed of data packets, each of which is an external form of values in local variables of
the program that are written and read as a unit. So minidb.c places in files first a schema
data packet (including the value of table_header .num_columns) and then a sequence
of row data packets (including the value of row_values).

Query Spiff Use Case (Example 11)

There are two cases. The first is when the workload (that is, the queries) come from
stdin, in which case no domain assertion can be deduced, because the user could type
anything in. (Of course, we can still use the VFT to determine the (many) invariants active
during a query, but that was already done by Invariant Finder, in a previous step.) The second is when workload comes from a file, say as a file named in the invocation argument, in which case the domain assertion is essentially the same as that for a row invariant, in that it deals with the OFFSET. For this second case, we use the name of the file to denote the actual file as the source of the workload. These two cases are differentiated by the Ecosystem Specification, which in the second case states that a particular Workload file could be specified. That said, we may not know who created the workload file, and in such a case we cannot yet create query spiffs for that workload. The second case may be used to specialize on a workload. Query spiff ids may then be put into the workload or as an association stored somewhere else, and used when that workload was executed. In the following, we will just consider the first case, where the query is not known until it is read in. Note that a query spiff that only includes invariants in effect during a query should be discoverable by an aggressive optimizing compiler. But more importantly, a query spiff combines such query invariants with schema and row invariants, which cannot be discovered by a compiler, because such schema and row invariants require knowledge about the semantics of file reads and writes. It's that aspect that makes a query spiff a true spiff. Example Algorithm for Domain Assertion Deducer Trace partition elements come from program interactions: where data files or components of data files (here: table header and rows) are created, inserted, or deleted. The
dependencies within domain assertions come from the directory and file name and
optionally an OFFSET within the file. The Invariants tie these together, so that values of
variables within the application can be seen to flow through the application code, out into a
file, and later back into the application, thereby ascertaining long-lived invariants, for which
candidate snippets and ultimately spiffs can be determined.

For the table spiff use case, in which the table_header .num_columns is
categorized in a domain assertion, DAD can determine that:

- main(): Case 'C calls CreateTable()
- which calls WriteTableHeader( )
- which writes the table header to the file.

The table header:
- is read by this and subsequent invocations of the program,
- until the file is deleted, at main(): 57.

This implies that the table header in a table <datafile> is only written once and
never modified by the program. Knowing this, DAD can generate the appropriate trace
partition elements, that of the table <datafile> being created and deleted. DAD can also
create the dependency concerning the table_header .num_columns.

For the table spiff instance use case, the relevant data packets are the row_data and
row_values being added to the table <datafile>, perhaps deleted from that file, and
immediately removed when the file itself is deleted. DAD determines that once the file is
created, the program may store multiple values of row_data into the file, thus each such
packet can be identified by the OFFSET it resides at.

The above observations provide an algorithm for DAD. For each FILE that is created
or opened by a program, DAD figures out where that file was initially created and where the
name for that file comes from, via the VFT. Then, for each data structure that is stored in
that file (these are the <data > elements in the Program Interactions), DAD ascertains the
file operations performed on that data structure (adding that data structure to the file,
possibly changing or removing that data structure, and finally deleting that file). These
operations then imply the appropriate trace partition elements. Finally, from the program
data structures used in these operations (the C or C++ program data structure written to the
file), DAD can inspect the VFT to determine where the values in such program data
structures originate, to imply the dependencies. DAD also determines whether a file contains only one data packet (as in the case of num_columns) or multiple packets (as in the case of row_data) by tracking what is done on each FILE variable as it flows through the program, also determinable via the VFT. Multiple packets require an OFFSET in domain trace partitions and dependencies.

**Correctness**

Correctness dictates that the Domain Assertions that are produced are complete and are consistent with the input PR, Invariants, and Program Interactions.

**Snippet Finder**

As shown in FIG. 1, another tool, Snippet Finder, takes as input:

- One or more Invariants
- An PR
- One or more Execution Summaries
- Domain Assertions Program Interactions, and
- A Cost Model

Snippet Finder outputs one or more `<spiff>` elements, each containing one or more Candidate Snippets, each of which contains:

- an Interval of code identified by PEs
- a set of Invariants
- a set of possible values for each Invariant
- the source location(s) where the value of each invariant was first written to a file,
- the source location(s) where that value was removed from a file,
- the source location(s) where the value is read in each time,
- the appropriate lifetime of the Candidate Snippet, that is, when the associate spiff can be created (and whether at compile-time or run-time), and
- optionally, suggested optimizations to be employed within the Interval.

Each Domain Assertion implies an interval that is probably broader than just one program execution, in contrast to the interval recorded in Invariants, which has a scope within a single program execution.
Snippet Finder uses Domain Assertions to expand the scope of the Invariants and to refine the set of possible values for each Invariant. The interval of each Invariant overlaps (either partially or entirely) the interval of the Candidate Snippet. Furthermore, the interval of each Candidate Snippet is tailored by the Cost Model, to minimize the size of the interval while maximizing the savings, calculated as the cost of executing an optimized version of the snippet times the number of times that snippet was evaluated, drawn from the Execution Summaries, plus the cost of invoking a spiff. To do so, the Snippet Finder must have an idea of the possible optimizations that the Spiff Maker performs and the benefit of each such optimization, the latter from the Cost Model.

**Query Spiff Use Case (Example 12)**

There are two major differences between query spiffs and the table spiffs considered earlier. The first is encountered at this step: Snippet Finder uses the Invariants, not from the Domain Assertions, as queries usually do not persist (though see the discussion above about Workloads being given to stdin). The second will be encountered later: Spiff Maker needs explicit guidance from the Ecosystem Specification as to where the boundary is between compiling spiff code and just instantiating a spiff instance at run time.

Snippet Finder infers:

- an Interval of code from `SequentialScan()`: 40 (immediately after unpacking the data packet into `row_values[]`) to `SequentialScan()`: 59 (the end of the method),

- a set of Invariants: `main().query, in particular, query.executor_routine, query.executor_command, query.num_predicates, query.predicate_list and predicates[]` read from stdin and query.schema from the table spiff use case,

- a set of possible values for each Invariant, in this case, `query.executor_routine` is always `SequentialScan()`, `query.executor_command` is always `SCAN_FWD`. For each predicate, `column_id` is an arbitrary int read from stdin (deduced from the assignment to that field in `BuildPredicates()`, constant_operand is an unsigned long read from stdin, and operator_function is either `&EqualInt4or&LessThanInt8`,

```plaintext
#include <iostream>
#include <string>
#include <vector>

int main() {
    int column_id;
    std::cin >> column_id;
    return 0;
}
```
· the source location(s) where the value of each invariant was first determined:
  the value of query is determined by main():32, that is, right after the call to
  BuildAndPlanQueryO.
· the value of query is never written to, removed from, or read from a file,
· the appropriate lifetime of the Candidate Snippet: the lifetime is just within
  the 'S' switch; the Ecosystem Specification tells us that we cannot call the
  compiler there, so we just pre-compile spiffs for executor_routine as
  SequentialScan( ), for executor_command as SCAN_FWD, for
  num_predicates from 1 to 6, and for each such predicate,
  operator_function is either &EqualInt4 or &LessThanInt8,
· unroll the for loop at SequentialScan() :47 as it is terminated by the
  value of the for loop at SequentialScan().

```xml
<?xml version="1.0" encoding="UTF-8"?>
<candidateSnippets xmlns:xsi="..."
  xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createAt="compileTime">

    <invariantIntervalSet variable="query"
      start="main():28">
      <interval stop="main():28"/>
      <generate variable="query->num_predicates" fromValue="1"
        toValue="6"/>
      <possibleValues variable="query->executor_routine">
        <possibleValue value="SequentialScan()"/>
      </possibleValues>
      <possibleValues variable="query->executor_command">
        <possibleValue value="SCAN_FWD"/>
      </possibleValues>
      <possibleValues variable="query->predicate_list[]->operator_function">
        <possibleValue value="&EqualInt4"/>
        <possibleValue value="&LessThanInt8"/>
      </possibleValues>
  </spiff>
</candidateSnippets>
```
It is not clear how fromValue and toValue are determined, but it seems that it is to bound the number of compile-time query spiffs that are generated.

Example Algorithm for Snipper Finder

Snippet Finder first expands invariants to across program executions by tracking which variables are read from files and where those values are put into files. This results in invariant intervals that span multiple executions. The tool also needs to track when the value was first written to the file and when it is deleted.

The other challenge to Snippet Finder is in bounding the snippets, using the Cost Model. In doing so, the tool needs to know what optimizations Spiff Maker can effect, and under what conditions each optimization is feasible.

Correctness

Correctness of this tool dictates that each of the Candidate Snippets produced by this tool be consistent with the input Invariants, PR, Execution Summaries, Domain Assertions, Program Interactions, and Cost Model. The indicated invariants should indeed be invariant over the snippet, the suggested optimizations should be consistent with these invariants and their manipulation within the PR, and the possible values should indeed be possible.

It is desirable though not required that:

- the snippets that are returned are the most desirable, given the cost model,
- the snippets be maximal, in that making them larger would result in a larger cost, from the cost model,
- the snippets be minimal, in that making them smaller would result in a larger cost, from the cost model, and
that the suggested optimizations would be helpful, given the cost model.

Spiff Maker as shown in FIG. 1, another tool, Spiff Maker, takes as input one or more Candidate Snippets and produces as output Specialized Source Code. Specifically, for each input Candidate Snippet, Spiff Maker should perform the following tasks:

1. Create a .h file for the spiff pattern, defining all pattern parameters and spiff pattern functions.
2. Create a .h file for the spiff implementation declarations.
3. Create a .c file for the spiff implementation definitions.
4. Insert code in the appropriate place(s) to create the spiff (for dynamic spiffs), to invoke the spiff, and to destroy the spiff (again, for dynamic spiffs).

Each use case is associated with a specified branch of minidb, for concreteness. Each branch includes the Candidate Snippet that causes the generation of that configuration. It may be convenient to use TXL for the actual transformation, as a PR-to-PR transformation, with the transformed PEs, then converted back into textual source code to create the spiff. TXL includes a parser, but it may be possible to take PEs directly. TXL also includes a syntax tree unparser which may work with our PEs.

For Spiff Maker to function as described, it may require some guidance based on domain knowledge. Specifically, Spiff Maker may need to be given/told:

- Specifications for all static implementations to be produced, (ie, which variables(s) are specialized, and their values if so.)
- Rules for disambiguation, for cases where more than one static implementation applies.
- Rules for creation of dynamic implementations: Are they allowed at all? Are they cached? If so, how? In memory? On disk? What are the size(s) and management rules for the cache(s)? Should a generated dynamic implementation be fully specialized, or would it be better to only partially specialize, and leave some parameters generic?
- Is it acceptable to compile a dynamic implementation on-the-fly as needed, or is it only acceptable to use...
a dynamic implementation if it's already in the cache? Do the answers to
these questions change?
- Should a fully-generic implementation be created (and used as a fallback
  inside)? Or, will some variables always be specialized, one way or another?
  (This will determine which variables will need to have representations in the
data block.)

In general, Spiff Maker will be told all of the above in the input file. It's Snippet
Finder's job to figure out how many static implementations to create, whether it should be
static/dynamic, which variables are specialized and which are not, and so on. There will only
be a single static implementation to create, and that single static implementation is always
the one that should be called.

Query Spiff Use Case

Referring to Example 13, the input is as follows, indicating compile-time query spiff
for executor_command as SCAN_FWD, for num_predicates from 1 to 6, and for each
such predicate, operator_function is either &EqualInt4 or &LessThanInt8, as
specified by Snippet Finder.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<candidateSnippets xmlns:xsi="..."
xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createdAt="compileTime">
    <invariantIntervalSet variable="query"
      start="main():28">
      <interval stop="main():28" />
      <generate variable="query->num_predicates" fromValue="1"
toValue="6" />
      <possibleValues variable="query->executor_routine">
        <possibleValue value="SequentialScan()" />
      </possibleValues>
      <possibleValues variable="query->executor_command">
        <possibleValue value="SCAN_FWD" />
      </possibleValues>
    </invariantIntervalSet>
  </spiff>
</candidateSnippets>
```
As noted above, Spiff Maker needs explicit guidance from the Ecosystem Specification as to where the boundary is between compiling spiff code and just instantiating a spiff instance at runtime. We assume that Ecosystem Specification specifies that that boundary occurs at the 'S', 'Γ, and 'D' cases: that no spiffs can be compiled within anything called by these three cases. (That emphasizes knowledge about what delays users will find tolerable. Note that compiling a new spiff for a particular row might be particularly beneficial for speeding up a collection of Workloads viewed as a whole, but the user may still want to specify that that not be done, say because a particular workload must itself run faster with field specialization.) This is the reason why Spiff Maker includes the Ecosystem Specification as an input. Spiff Maker thus makes a spiff for a portion of SequentialScan( )-each value of num_columns at compile-time, for query executor_command always being SCAN_FWD. For each predicate, column_id is an arbitrary int, constant_operand is an unsigned long read from stdin, and operator_function is either &EqualInt4 or &LessThanInt8, with spiff 0 a non-specialized version that can handle any number of num_columns. The relevant transformations are loop unrolling and constant folding. Spiff Maker will produce a spiff pattern for spiffID of 23, for num_predicates = 2, with the first having column_id = 2 and operator_function = &EqualInt4 and the second...
having column_id = 7 and operator_function = &LessThanInt8, as the following.

Note that query spiffs ID computation is normally associated with the particular values of the spiff pattern parameter. Spiff Maker should utilize an application-specific ID generation mechanism to produce the proper spiffID. In this example however, we are going to assume a computed spiffID of 23.

**Example Algorithm for Spiff Maker**

Spiff Maker decides only one thing: whether to allow the compiler to perform the optimization, after Spiff Maker has indicated what the invariant value(s) are, or to perform the optimization manually, by generating different code.

Spiff Maker then cobbles together the generated files by copying mostly verbatim from the original source to the specialized source, using the file names, line numbers, and column counts within the relevant PEs to determine the extent of what is copied and of what is replaced say with a spiff parameter (e.g., num_columns). Hence, Spiff Maker needs to do a very limited amount of parsing and unparsing, with most of its effort consisting of copying code from the appropriate place in the original source to the appropriate place in the specialized source.

**Correctness**

Correctness dictates that the code compiles and runs and is identical in semantics to the original code it replaces, while being consistent with the input information.

The following discussion provides additional examples demonstrating the creation of a MiniDB Table Spiff and a MiniDB Query Spiff.

**MiniDB Table Spiff**

The following example demonstrates the creation of a table spiff from the invariant schema->num_columns == CONSTANT in the SequentialScan() function, shown in Example 4.

**Invariant Finder**

In the above example, Invariant Finder should identify the following Invariant Interval Sets for the SequentialScan():

- Invariant Interval Set #1: Starts on line 52, with 1 Invariant Intervals:
  - Invariant Interval #1.1: Ends on line 114
Invariant Finder should also produce a VFT to show where the variable

```
SequentialScan()::schema->num_columns
```

got its value:

- `SequentialScan()::schema->num_columns`
- `ExecuteQuery()::query->schema->num_columns`
- `ExecuteQuery()::query->schema->num_columns`
- `main():query->schema->num_columns`
- `main():table_header->num_columns`
- `main():table_header->num_columns`

Thus, the value of `SequentialScan()::schema->num_columns` eventually comes from a call to `fread()` in `OpenTable()`.

Invariant Checker

Invariant Checker would verify that once `main():table_header->num_columns` was assigned to on line 634, and that the value of this variable was never changed through the particular end node(s) taken by that execution of the given Workload.

If the Domain Assertion Deducer executed before Invariant Finder, the Invariant Checker could check to ensure that the actual value was included in the Possible Values. That might be able to reduce the scope of the Invariant Finder, by focusing that analysis on particular values or variables.

Snippet Finder

Snippet Finder should determine, through an analysis of the Execution Summaries in conjunction with the Cost Model, that the 'C', 'L', and 'D' cases are too expensive to create spiffs, but that the computation time within the 'S' case is sufficient to suggest that that case be specialized.

Let's start with a simple Cost Model that just states that a PE (or other equivalent embodiment) that is executed less than a fixed or percentage of times will not be specialized.

In such a case, Snippet Finder should then infer from the domain assertion that `schema->num_columns` is invariant across the body of `SequentialScan()` with a scope of when the data file was created to when that file was removed, thus indicating that the
value of that variable was first written to the file when the number of columns is stored, in
`WriteTableHeader() : 3`, which is executed shortly after `minidbc.c:553`. The value was
never removed from the file, but the file itself was removed at `main() : 57`. This indicates
that the spiff can be created at compile-time. The snippet should extend from
`ExecuteTable() : 20` through `ExecuteTable() : 23`. That is the scope of the snippet,
specialized on `num_columns`, which is determined from seeing what other statements can
be specialized on `num_columns`. (Essentially, `num_columns` is used infrequently, and thus
far away from this specialization opportunity.) However, doing an extra indirect call is
expensive, so Snippet Finder expands this snippet to the entire body of `ExecuteQuery/`,
which has another use of this invariant on line 7. Finally, Snippet Finder should suggest loop
unrolling for this Candidate Snippet.

In this case, the spiff will have but one spiff function, indicated by the `<snippet>`,
as shown in Example 14.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<candidateSnippets xmlns:xsi="..."
xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createAt="compileTime" valueRead="OpenTable() : 8">
    <invariantIntervalSet variable="ExecuteQuery.query->schema->num.num_columns">
      <interval start="ExecuteQuery() : 1" value=">0"></interval>
    </invariantIntervalSet>

    <snippet existsFrom="WriteTableHeader() : 3" existTo="main() : 57" replaceFunction="ExecuteTable()">
      <suggest opt="Constant fold" on="num_columns" at="ExecuteQuery() : 7" />
      <suggest opt="Unroll loop" at="ExecuteQuery() : 21" />
    </snippet>

  </spiff>
</candidateSnippets>
```
Example 14

Snippet Finder could infer from the domain assertion that data packets are created in cases 'Γ' and 'D' of main() and removed is cases 'P' and 'D'. More specifically, Snippet Finder infers:

- an Interval of code from SequentialScan():16 (immediately after the data packet is read in) to SequentialScan():38 (the end of the unpacking for loop),
- a set of Invariants: values from the row_data and the schema, from the invariant analysis above,
- a set of possible values for each Invariant, in this case, that the value of num_columns is 3, that the value of the first column is a hard-coded int, and schema, that the type of the first column is int, of the second column, long, and of the third column, int, an array of arbitrary char,
- the source location(s) where the value of each invariant was first written to a file: WriteRow():18 and WriteRow():25,
- the source location(s) where that value was removed from a file: main():57 and ExecuteDelete():25,
- the source location(s) where the value is read in each time: main():SequentialScan():3,
- the appropriate lifetime of the Candidate Snippet: built at table definition time using the query.schema invariants, with the row_values provided when the spiff is instantiated at run-time, as this involves data packets that might be inserted, a few queries run, then the data packet removed, so must be fast, and also because the number of possibilities is very great for the row-values, and
- unroll the for loop at SequentialScan():16 as it is terminated by the value of schema->num_columns and includes use of row_data and schema values.
As shown in Example 15, note that the analysis combines the broader scope of the
table invariant with the narrower scope of the row invariant, and employs different strategies
for each: the former allows generating code when the table spiff is defined, whereas the
latter involves instantiating the spiff at runtime by providing value(s) for the row_values
array. In field specializing DBMSes, the schema invariants will play a large roll in table
spiff instances and query spiffs, which involve invariants of successively narrower scope.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<candidateSnippets xmlns:xsi="" xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createAt="WriteRow():19" removeAt="main():57"
       getSpiffID="SequentialScan():16">
    <invariantIntervalSet variable=" CreateTable.table_header"
                           start=" CreateTable():30">
      <interval stop="main():57" />
      <interval stop="ExecuteDelete():25" />
    </invariantIntervalSet>
    <invariantIntervalSet variable=" main.values"
                           start=" main():42">
      <interval stop="main():42" />
    </invariantIntervalSet>
    <instanceParameter value="row_values[1]" for="i=1" />
    <snippet existsFrom="SequentialScan():16"
              existsTo="SequentialScan():38">
      <suggest opt="Unroll loop" at="SequentialScan():19" />
    </snippet>
  </spiff>
</candidateSnippets>
```

Example 15
SpiffMaker

This is the simplest use case, as there are no instances. We explore four variants of this case.

Variant 1: A single static implementation:

Consider the following input Candidate Snippet, corresponding to a single invariant in minidb that should result in a static spiff implementation, shown in Example 16.

```xml
<candidateSnippets xmlns:xsi=""
  xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createAt="compileTime" valueRead="OpenTable():8">
    <invariantIntervalSet variable="ExecuteQuery().query->schema->num_columns"
      start="ExecuteQuery():1" value="3">
      <interval stop="ExecuteQuery():33"/>
    </invariantIntervalSet>

    <snippet existsFrom="WriteTableHeader():3" existTo="main():57"
      replaceFunction="ExecuteQuery()">
      <suggest opt="Constant fold" on="num_columns"
        at="ExecuteQuery():7" />
      <suggest opt="Unroll loop" at="ExecuteQuery():21" />
    </snippet>
  </spiff>
</candidateSnippets>
```

Example 16

- `createAt="compileTime"` states that this spiff should have a static implementation.
- `valueRead="OpenTable():8"` states where the variable is read from the outside world, thus indicating where a spiff might be selected.
This input is telling Spiff Maker to make a spiff pattern, with one spiff pattern function based on \texttt{ExecuteQuery()}, and to make a static implementation that specializes the variable \texttt{\texttt{\_\_query-}\texttt{-schema->num\_columns}} to the single literal value $3$.

Variant 2: Specialization at a finer granularity:

The above example showed a spiff applied to replace an entire function \texttt{\texttt{\_\_query}}. In fact, we can observe that only a small code segment in \texttt{\texttt{\_\_query}} function involves an invariant. Thereafter, we can convert just that small code segment into a spiff, as shown in the following snippet.

The candidate snippet shown below (shown in Example 17) is very similar to before, with the interval much smaller than the entire \texttt{\texttt{\_\_query}} function, rather just three lines of the for loop. The \texttt{replaceFunction} attribute thus disappears. Finally, the constant fold suggestion for line 21 is omitted, because that line is not in the interval to be specialized. (We could have left it in, to be ignored.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<candidateSnippets xmlns:xsi="..."
  xsi:noNamespaceSchemaLocation="candidateSnippets.xsd">
  <spiff createAt="compileTime" valueRead="OpenTable()[:8]">
```
Variant 3: Employing fixed array of implementations:

In our particular example, we decide to identify the spiff implementations with a one-byte integer. Therefore, we can have a total of 255 implementations from the same spiff pattern, with the `num_columns` variable acting as the spiff-pattern parameter, varying from 1 to 255 (0 is chosen to represent the invalid value). So the candidate snippet shown below in Example 42 doesn't have a value just of 3, but all values from 1-255 (that is, the `fromValue` and `toValue` attributes of the `invariantIntervalSet` element). We also revert back to a full function replacement.
Variant 4: Dynamic spiff:

In reality, each column in a table can be of a particular datatype. Assuming there are eight datatypes (int2, int4, char, varchar, etc), a static table spiff for a three-column table requires $3^8$ possible implementations. Hence, a dynamic table spiff is more suitable in this scenario.

The candidate snippet given below (see Example 19) states this with the createAt attribute, which here specifies where in the application the spiff is created, that is, within the CreateTable() function (the createAt attribute, which in the previous example was compileTime) as well as where the spiff is to be instantiated, that is, within the OpenTable() function (the instantiateAt attribute). There are no fromValue or toValue attributes in the invariantIntervalSet element, as the num_columns value is supplied when the snippet is instantiated. The one other important difference from the previous example is the additional optimization suggestion to constant fold on column_definitions.
Unlike creating a static spiff, a dynamic spiff is created at runtime by inserting a call to compile the spiff into `CreateTable()`, for a table spiff.

The Design of Various Types of Spiffs

(Here we use examples from the open-source Postgres DBMS.)

Predicate Query Spiff

This spiff is utilized by evaluating both regular predicates within queries, such as:

- `o_orderdate >= date '19940801'`, and join predicates, such as `o_orderkey = l_orderkey`. 
These predicates are evaluated by the `ExecQual()` function (in Postgres). Specifically, predicates are normally represented in a linked list. `ExecQual()` iterates through this list and invoke particular evaluation function corresponding to each individual predicate. The code excerpt presented in Example 20 (from PG 9.3 stock, src/backend/executor/execQual.c:5125) shows such logic.

```c
bool ExecQual(List *qual, ExprContext *econtext, bool resultForNull) {
    ...
    foreach(l, qual) //line 4157
    {
        ExprState *clause = (ExprState *) lfirst(l);
        Datum expr_value;
        bool isNull;
        expr_value =ExecEvalExpr(clause, econtext, &isNull, NULL);
        ...
    }
    ...
}
```

**Example 20**

The per-predicate evaluation function is stored within the clause variable. For each predicate, in the form of `a > b`, there are three components, operand #1, an operator, and operand #2. In Postgres, the operator is evaluated by function `ExecEvalOper`. This function (see Example 21) essentially performs a look up according to the type of operator and fetches the address of the actual type-specific comparison function. `ExecEvalOper()` also requires that the operands to be stored in another linked list. In many cases, the length of this list is two. Below is an example to specialize this function on those cases.

```c
Datum ExecEvalOper(FuncExprState *fcache,
                     ExprContext *econtext,
                     bool *isNull,
                     ExprDoneCond *isDone)
{
    ...
    /* Initialize function lookup info */
    init_fcache(op->opfuncid, op->inputcollid,
    ...
```
fcache->xprstate.evalfunc =
  (ExprStateEvalFunc)ExecMakeFunctionResultNoSets;
return ExecMakeFunctionResultNoSets(fcache, econtext, isNull, isDone);
}

Example 21

Note that an optimization to ExecEvalOperator() is that it is executed just once to
perform the comparison function look up. It then stores into xprstate.evalfunc a
different function. It also calls that function once to do the predicate. Subsequent evaluations
of the operator is done by ExecMakeFunctionResultNoSet(s) (for scalar predicates
considered within our current specialization scope).

ExecMakeFunctionResultNoSet(s) then iterates through the list of operands by
calling the argument-extraction function for each operand.

ExecEvalExpr is a macro, defined in src/include/executor/executor.h:72 as:

  #define ExecEvalExpr(expr, econtext, isNull, isDone) \
   (*((expr)>evalfunc)) (expr, econtext, isNull, isDone))

So if the operand is a constant, the ExecEvalConst() would be called, and finally,
invokes the comparison function.

The bottlenecks observed in predicate evaluation are, first, the loop that iterates
through just two elements in the operand list, and second, the extraction of individual
operands. Specifically, we observe that for a regular predicate, one operand is normally a
table column and the other operand is a constant. In such a case, the constant's value (or
address) can be directly "stored" in the code rather than having to invoke multiple functions
to fetch it. In addition, the original implementation requires multiple function calls to extract
the column ID for the table-originated operand. Similarly, this column ID can be stored
directly into the specialized code.

For a join predicate, both operands are non-constant. The origin of an operand can be
one of three types, that of INNER_VAR (I), OUTER_VAR (O), and scantuple (S).

The origin of the operand is also an invariant given a query. By knowing this invariant, we
can further simplify the routine that extracts the actual operand's value. Note that although
theoretically, there are 9 possible combinations for the origins of the two operands, in
reality, only the following combinations fit


In function `ExecHashJoin()`, defined in file `src/backend/executor/nodeHashJoin.c`. The variable `node->js.jointype` is invariant for a given query. Depending on the query, it will take on one value from the set `{JOIN_ANTI, JOIN_SEMI, JOIN_LEFT, JOIN_INNER}`.

In the same file and function, the variable `List *joinqual` is also invariant for a given query.

Hashjoin Query Spiff eliminates entire branches in the code, thereby reducing the number of if statements and, more importantly, the size of the code.

The analysis has to be able to handle complex data structures involving pointers and heap-allocated structs. For example, to eliminate if statements in the body of the for loop in `ExecHashJoin()`, we have to be able to reason about expressions like the following (Example 22).

```c
#define HJ_FILL_INNER(hjstate) ((hjstate)->hj_NullOuterTupleSlot !≡ NULL)
#define HJ_FILL_OUTER(hjstate) ((hjstate)->hj_NullInnerTupleSlot !≡ NULL)
#define TupleIsNull(slot) ((slot) == NULL || (slot)->tts_isempty)
if (HJ_FILL_INNER(node))
    ...
else if (HJ_FILL_OUTER(node)) |
    (outerNode->plan->startup_cost < hashNode->ps.plan->total_cost &&
     !node->hjOuterNotEmpty)
    ...
if (hashtable->totalTuples == 0 && !HJ_FILL_OUTER(node))
    ...
if (TupleIsNull(outerTupleSlot))
    ...
if (batchno != hashtable->curbatch &&
```
A page spiff utilizes invariant(s) within a disk/memory page with which the DBMS manages its storage of data. Often, such invariants could include the number of rows stored on the page, the free space remained, and whether the page is empty or full. In postgres' page-scan routines, there are additional invariants, such as the scan direction and scan mode (pageatatime).

More interestingly, page spiffs can enable more aggressive optimizations. For instance, a page spiff can reorganize the data layout once the page is read into memory to optimize data locality. In addition, once data layout is changed, instead of following the existing function-call sequence in further process, the page spiff can invoke these calls in a block-at-a-time manner, thereby improving instruction locality as well.

A page spiff is possible to specialize a long sequence of calls that eventually access data, passing up the data a ways, where it is possible to specialize out a lot of code in the called functions.

The chief benefit of the page spiff is to inline the called functions to produce a single specialized function mat can fit in the instruction cache. Once mis transformation is done, three other mutually exclusive transformations become available.

1. Eager invocation of the GetColumnsToLongs() speccode routine: as soon as the packed tuple is extracted from the page, convert to a unpacked tupletableslot, then store it in the array manipulated by the speccode routine.

2. Eager partial unpacking: have the code that calls the specialized code compute the maximum column that will be needed, and only unpack the columns up to there.

3. Lazy unpacking: do multiple unpacking operations, wherever the original code called GetColumnsToLong().

A variant of this uses the selectivity of the select to decide. If the selectivity is high, meaning that only a few rows will be referenced, use lazy unpacking before applying the predicate.
In general, it is best to place the call to `GetColumn` to long( ) ... ( ) . The result is then returned back to `advance_transition_function` within `nodeAgg.c`, which copies this expression given all supported (NUMERIC_MAX_PRECISION) beyond overheads.

Aggregate allocation and deallocation are expensive. This is especially the case for numeric_add which has been using a per-row approach. Nevertheless, this is the default Postgres numeric_add representation. The aggregate_sum of numeric_add is the product of SUM and AVG. The numeric_add function is then reused across rows. This allows the number of digits to be maximized, thereby minimizing memory allocation and deallocation. The number of rows that can be accumulated and reused is limited by numeric_add functionality which has been using a per-row approach.
returned value into pergroupstate->transValue and then frees the returned value. The
next time advance_transition_f unction () is executed to process the next row, the
transvalue is copied to the first input value of numeric_add () via the following
snippet.
fcinfo->arg [0] = pergroupstate->transValue;
finfo->argnull [0] = pergroupstate->transValueIsNull;

This logic indicates that transValue can in fact be shared without being freed
across all the rows. Therefore, for the data section of the EvaluateNumericAdd spiff, at
the beginning of the aggregate evaluation, the necessary variables are allocated, namely
agg_temp_values->result_value and agg_temp_values->result_arg, by using
AllocateAggTempValues (). (Note that these two variables represent the same value but
Postgres requires two such variables as the return value and as a temporary computation
argument, respectively.)

Evaluating Expression
As discussed earlier, another usage of numeric_add () is in computing arithmetic
expression, such as a + b. In this case, the variable that stores the evaluation result of the
expression are reused, which is previously allocated by make_result (). This variable is
added to the spiffs data section as agg_temp_values->expr_result_arg.

Unlike in the case of evaluating SUM (), where the first input comes directly from
agg_temp_values->result_value within the spiffs data section, both inputs in
evaluating a + b are regular variables that are required to be obtained using existing
Postgres' implementation. In fact, in evaluating a + b, numeric_add () is invoked from
ExecEvalOper () within execQualc. So similar to the predicate spiff, a spiff
(EvaluateAggregateExpression) is created that specializes the
ExecMakeFunctionResultNoSet s () function. This spiff then invokes the expression-
evaluation version of the EvaluateNumericAdd spiff.

In addition to +, an expression can include other operations such as -, *, and /. The
functions that evaluate these operations are also specialized in the same fashion as
numeric_add ().

In summary of the EvaluateNumericAdd spiff, the following invariants are
considered.
1) The caller/execution path where numeric_add() is invoked. This can be either
from evaluating an expression in execQual.c or from evaluating the SUM() function in
nodeAgg.c.

2) In evaluating expressions, the result value's memory location can be invariant.

3) In evaluating SUM(), both the result value's memory location and the first input's
memory location can be invariant. In addition, these two variables can even share the same
memory location.

4) Sharing a common memory segment across all the rows is permitted by the
constant that bounds the maximal precision of the numeric datatype.

String Matching Spiff

Say we have a C function, match, which matches a string x with another string
pattern (containing wildcards and other special characters), y. If we know string y in
advance of query execution (maybe it is a query constant), then we can create a specialized
function for matching arbitrary strings to this particular string pattern.

One approach for specialization would be to first create the following query
specialization code (speccodes):

- one each for constant string of length 1-32
- one for the '%' query string character

We could then string various combinations of these speccodes together to make a
specialized function for matching a string to y. For example, say that we have the pattern
"%abc%def%g%", and we want to create a specialized function for matching arbitrary strings
to it. We would string together the following speccodes:

- a % speccode
- a 3-character speccode, to match "abc"
- a % speccode (could be the same as the first one)
- a 4-character speccode, to match "defg"
- an ending % speccode

Each of these speccodes would assume that there are more characters in the string left to
match after it has completed. Once one of the speccodes has finished matching, it would
pass the rest of the string on to the next speccode in the sequence to continue the matching
process.
Matching the constant portions of the string would be accomplished using a combination of longlong, long, short, and char combos.

Given an arbitrary query string, it would be easy to instantiate a sequence of query spiff function pointers, each one of which except the last invoked the next stage by a query spiff invocation using the spiff id stored as a local variable.

The following (Example 23) shows an example of how this would be implemented for the string "%abc%defg%" (using pseudocode).

```c
/* For matching "abc" */
length3match(char *s, char *t) {
    match first 3 characters of s and t
    remove first 3 characters from s and t
    return
}

/* For matching "defg" */
length4match(char *s, char *t) {
    match first 4 characters of s and t
    remove first 4 characters from s and t
    return
}

/* For matching an arbitrary "%" */
match%(char *s) {
    match n characters in s to '%'
    remove first n characters from s
    return
}

/* For matching a "%" at the end of a pattern */
match%end(char *s) {
```
Once we have these speccode-routines created, we will construct a sequence of functions calls as an array to match a string to this pattern. The array would look like in Example 24.

<table>
<thead>
<tr>
<th>function pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>match%</td>
</tr>
<tr>
<td>length3match</td>
</tr>
<tr>
<td>match%</td>
</tr>
<tr>
<td>length4match</td>
</tr>
<tr>
<td>match%end</td>
</tr>
</tbody>
</table>

These functions would then be called in sequence to match the string. Constant portions of length greater than 32 could be broken up into segments, so a string of length would require three instantiated speccodes, of 32, 32, and 1 character.

More generally, we have a method with a subset of arguments that are invariant. These invariants render some of the if statements, including in recursive calls and within loops, to be deterministic. We unroll this sequence through a series of speccodes that invoke one another. So this appears to be a general transformation that works on loops and on recursive and non-recursive calls.

Since the actual sequence of speccodes that get invoked isn't known until runtime (when the actual pattern being matched is available), we could have the spiff instantiator fill in an array of function pointers that specify the sequence of speccodes to invoke.

**Per-Query Spiff Sequencing**

Per-Query Spiff Sequencing uses a nieta-spiff to traverse an interpreted data structure and hot swapping to convert existing speccode into something similar that would be emitted by a compiler. In some embodiments, the hot-swapping mechanism is used
Specialized code that can be stitched together into runtime according to machine through the 1930's, data was differentiated from code. The data was what was manipulated and the...
field

A field is a unit of information that can be used to hold a piece of data in a record. It is defined by its name, type, and a value. Fields are typically used to store and manipulate data to perform computations.

Programming code was the instructions for how to manipulate data to effect a computation. This is represented in a field, passed around between applications, and invoked by the target application when appropriate. The field values, such as those represented in a tuple, could be stored as code (as in Lisp) or as Postscript code (Microsoft Word). They are processed as they are retrieved from memory, which takes place directly on the device. Some applications, such as those in the 1940s, had a data-represented-as-code quadrant, but the image was stored in a document or in a file.

For example, a snippet of hardware that produced a small set of instructions to perform a particular task, which was then combined with a snippet of code to create a larger program. This program could be executed by a computer, but it was also possible to manipulate the data directly with a drawing program, which could produce a data field in a document, such as an architectural drawing, or in a file.
specialization technology provides the means for identifying when such speccode are
effective in increasing performance, when they should be created, with which invariants
should they be specialized upon, how they can be communicated across applications, and
when they should be invoked.

The implication is that for any coherent region within a data file, it is possible to
ascertain the invariant values within that region, follow those values into areas of the
application code, and then make speccodes out of those areas, then associate those
speccodes back to their regions. This perspective thus focuses on the originating data, rather
than starting with the code and specializing it.

It should be emphasized that the above-described embodiments of the present
disclosure, particularly, any "preferred" embodiments, are merely possible examples of
implementations, merely set forth for a clear understanding of the principles of the
disclosure. Many variations and modifications may be made to the above-described
embodiment(s) of the disclosure without departing substantially from the spirit and
principles of the disclosure. All such modifications and variations are intended to be
included herein within the scope of this disclosure and the present disclosure and protected
by the following claims.
CLAIMS

What is claimed is:

1. A computer-implemented method for improving the performance of a computer program, comprising:
   - identifying, based on a Program Representation (PR), that is, a natural syntax tree or other embodiment of the computer program code, invariant intervals for variables in the computer program code;
   - deducing, based on the PR and an Ecosystem Specification for the computer program, program interactions within the computer program;
   - deducing, based on the PR, the identified invariant intervals for variables in the computer program code and the deduced program interactions, the domain assertions;
   - identifying, based on the invariant intervals for variables in the computer program code, the PR, one or more execution summaries associated with the computer program, the deduced program interactions and the deduced domain assertions, one or more candidate snippets;
   - generating specialized computer program code, based on the one or more candidate snippets;
   - and modifying the computer program code based on the generated specialized computer program code; and
   - concealing the specialized computer program code.

2. The computer-implemented method of claim 1 characterized by one or both of the following features:
   - wherein the identified invariant intervals span multiple executions;
   - wherein the identified invariant intervals comprises a least set of invariant intervals for a particular variable, where all invariant intervals in the set share the same starting node.

3. The computer-implemented method of claim 1 or claim 2, wherein each of the one or more candidate snippets comprises
   - an interval of code identified by the PR, or
   - a set of invariants and a set of possible values for each variant.

4. The computer-implemented method of any of claims 1-3, wherein each of the one or more candidate snippets comprises an appropriate lifetime of the Candidate Snippet, and...
wherein each of the one or more candidate snippets preferably comprises suggested
optimizations to be employed within the appropriate lifetime of the Candidate Snippet.

5. The computer-implemented method of any of claims 1-4, wherein generating
specialized computer program code comprises (a) inserting codes in places within the
computer program to create the specialized computer program code, to invoke the
specialized computer program code and to destroy the specialized computer program code,
or (b) creating a specialized function for matching arbitrary strings to a given string pattern,
or comprises using a meta-specializer to traverse an interpreted data structure and hot
swapping to convert existing specialized computer program code, or involves query, or
comprises eliminating branches in the computer program code and thus reducing the size of
the computer program code, or comprises utilizing a numeric value to slab-allocate a
specializer-in-the-field (Spiff) data section, wherein the Spiff data section is then reused by
computing a corresponding aggregate function across all input rows within the computer
program code to eliminate memory allocation per-row, wherein the numeric value is defined
by max number of digits that can be supported per row, or comprises utilizing invariant
within a disk or memory page with which the computer program is stored, reorganize the
data layout once the page is read, and optimize data locality.

6. The computer-implemented method of any one of claims 1-5, wherein the
specialized computer program code is created at a run time and invoked at a later time., and
optionally, further comprising determining whether any violations of the identified
invariable intervals occur in a given execution.

7. A system configured for improving the performance of a computer program,
comprising:
   an Invariant Finder identifying invariant intervals for variables in the computer
   program, based on a Program Representation (PR) of the computer program;
   an Interaction Deducer deducing program interactions within the computer program
   based on the PR and an Ecosystem Specification for the computer program,
   a Domain Assertion Deducer deducing domain assertions based on the PR, the
   identified invariant intervals for variables in the computer program and the deduced program
   interactions;
the analysis of the candidate summaries. For example, using the same specialized code generator and the same specializer-in-the-field, the execution time for the program was reduced from 500 milliseconds to 100 milliseconds, thus reducing the size of the program by a factor of 5.

The analysis also showed that the candidate summaries were more accurate and more specific than the original summaries. This improvement in accuracy is due to the use of the specializer-in-the-field, which is able to identify the important parts of the program and to accurately deduce the invariants associated with those parts.

In conclusion, the use of the candidate summaries and the specializer-in-the-field is an effective way to reduce the size of the program and to improve the accuracy of the analysis. The results of this study suggest that this approach could be useful in other applications where the size of the program is a concern and where the accuracy of the analysis is important.
of the computer program code, or to utilize a numeric value to slab-allocate a specializer-in-the-field (Spiff) data section, wherein the Spiff data section is then reused by computing a corresponding aggregate function across all input rows within the computer program code to eliminate memory allocation per-row, wherein the numeric value is defined by max number of digits that can be supported per row, or to utilize invariant within a disk or memory page with which the computer program is stored, reorganize the data layout once the page is read, and optimize data locality.

14. A non-transitory computer readable medium comprising computer executable instructions that, when executed by a processor of a computing device, cause the computing device to:

   determine a variable in a computer program whose value is invariant within an identified invariant interval via static analysis on a Program Representation (PR) of the computer program;

   produce codes in places within the computer program to create a specialized computer program code, to invoke the specialized computer program code and to destroy the specialized computer program code, the a specialized computer program code being created based at least on the determined variable; and

   modify at least a part of the computer program with the specialized computer program code when the specialized computer program is invoked.

15. The non-transitory computer readable medium of claim 14, wherein producing the specialized computer program code is further based on at least one of domain-specific knowledge associated with the computer program and extra-source knowledge.
FIG. 2
What

Represented as Data

\[ \text{Data} \]

\[ \text{Binary rep} \]

310

How Represented?

Represented as Code

\[ \text{Postscript} \]

\[ \text{Lisp continuations} \]

330

\[ \text{relation bees, tuple bees, O/S bees, router bees} \]

Code

\[ \text{von Neumann architecture} \]

320

\[ \text{Source code} \]

340

\[ \text{proto-bees} \]

FIG. 3
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER

IPC(8) - G06F 9/45; G06F 9/445; G06F 17/30; G06F 12/00 (2016.01)
CPC - G06F 8/447; G06F 8/443; G06F 9/44521; G06F 9/45516 (2016.02)

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC(8) - G06F 9/45; G06F 9/445; G06F 17/30; G06F 12/00 (2016.01)
CPC - G06F 8/447; G06F 8/443; G06F 9/44521; G06F 9/45516 (2016.02)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

US: 707/803; 717/140; 717/151; 717/154 (keyword delimited)

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

PatBase, Google Patents, ProQuest

Search terms used: specialized, code, improving, program, interactions, domain, assertions, invariant

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>US 2008/0189277 A1 (MEIJER et al) 07 August 2008 (07.08.2008) entire document</td>
<td>1-3, 7-9, 14, 15</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of Box C. See patent family annex.

Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search
02 June 2016

Date of mailing of the international search report
29 JUL 2016

Name and mailing address of the ISA/
Mail Stop PCT, Attn: ISA-US, Commissioner for Patents
P.O. Box 1450, Alexandria, VA 22313-1450
Facsimile No. 571-273-8300

Authorized officer
Blaine R. Copenhagen
PCT Helpdesk 571-272-4300
PCT DSP 571-272-7774

Form PCT/ISA/210 (second sheet) (January 2015)
<table>
<thead>
<tr>
<th>Box No. II</th>
<th>Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:</td>
<td></td>
</tr>
<tr>
<td>1. □ Claims Nos.: because they relate to subject matter not required to be searched by this Authority, namely:</td>
<td></td>
</tr>
<tr>
<td>2. □ Claims Nos.: because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically.</td>
<td></td>
</tr>
<tr>
<td>3. □ Claims Nos.: 4-6, 10-13 because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Box No. III</th>
<th>Observations where unity of invention is lacking (Continuation of item 3 of first sheet)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This International Searching Authority found multiple inventions in this international application, as follows:</td>
<td></td>
</tr>
<tr>
<td>1. □ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.</td>
<td></td>
</tr>
<tr>
<td>2. □ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.</td>
<td></td>
</tr>
<tr>
<td>3. □ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:</td>
<td></td>
</tr>
<tr>
<td>4. □ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:</td>
<td></td>
</tr>
</tbody>
</table>

**Remark on Protest**

□ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

□ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

□ No protest accompanied the payment of additional search fees.