Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE

Xuepeng Fan, Zhenyu Guo, Hai Jin, Xiaofei Liao, *Member, IEEE*, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Wei Lin, Jingren Zhou, and Lidong Zhou, *Member, IEEE*

Abstract—To minimize the amount of data-shuffling I/O that occurs between the pipeline stages of a distributed data-parallel program, its procedural code must be optimized with full awareness of the pipeline that it executes in. Unfortunately, neither pipeline optimizers nor traditional compilers examine both the pipeline and procedural code of a data-parallel program so programmers must either hand-optimize their program across pipeline stages or live with poor performance. To resolve this tension between performance and programmability, this paper describes PeriSCOPE, which automatically optimizes a data-parallel program's procedural code in the context of data flow that is reconstructed from the program's pipeline topology. Such optimizations eliminate unnecessary code and data, perform early data filtering, and calculate small derived values (e.g., predicates) earlier in the pipeline, so that less data—sometimes much less data—is transferred between pipeline stages. PeriSCOPE further leverages symbolic execution to enlarge the scope of such optimizations by eliminating dead code. We describe how PeriSCOPE is implemented and evaluate its effectiveness on real production jobs.

Index Terms—Data-parallel, data-shuffling I/O, optimization, static analysis, symbolic execution

1 Introduction

The performance of big data computations improves dramatically when they are parallelized and distributed on a large number of machines to operate on partitioned data [1], [2]. Such data-parallel programs involve pipelines of computation stages where I/O intensive data shuffling between these stages can dominate program performance. Unfortunately, data-shuffling I/O is difficult to optimize automatically because computations at each pipeline stage are encoded as flexible procedural code; current pipeline optimizers treat this code as a black box while compilers treat pipelines as black boxes and so are unaware of the data flow between the procedural code at different computation stages. The programmer must manually perform optimizations that require examining both the program's pipeline and procedural code; e.g., to not propagate unused data or to move the computation of smaller derived values to an earlier stage so less data is transmitted during data shuffling. Performing these optimizations by hand is not only tedious, it also limits code reuse from generic libraries.

- *X. Fan, H. Jin, and X. Liao are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China. E-mail: athrunarthur@gmail.com, {hjin, xfliao}@hust.edu.cn.*
- *Z. Guo, S. McDirmid, H. Zhou, J. Zhang, and L. Zhou are with the Microsoft Research Asia. E-mail: {Zhenyu.Guo, smcdirm, huzho, jiaxz, lidongz}@microsoft.com.*
- *J. Zhou and W. Lin are with the Microsoft Bing. E-mail: {jrzhou, weilin}@microsoft.com.*

Manuscript received 15 Oct. 2013; revised 4 Apr. 2014; accepted 11 May 2014. Date of publication 21 May 2014; date of current version 8 May 2015. Recommended for acceptance by S.-Q. Zheng.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2326416

So that programmers can write data-parallel programs with reasonable performance without sacrificing programmability, automatic optimizations must examine both the pipeline and procedural code of a data-parallel program. Common logical optimizations [3], [4], [5], [6], [7] for data-parallel programs focus on a high-level pipeline topology that is subject to relational query-optimization techniques. Unfortunately, at best relational components are extracted from procedural code into a relational optimization framework [8], which is limited by the inability of the relational framework to match the expressiveness of procedural code. We instead observe that projecting well-understood declarative pipeline properties into more flexible procedural code is intrinsically simpler than extracting declarative properties from procedural code. Such projection can then be used to reconstruct program data flow, enabling automatic optimizations of procedural code across pipeline stages that can improve I/O performance.

This paper presents PeriSCOPE, which automatically optimizes the procedural code of programs that run on SCOPE [3], [9], a production data-parallel computation system. PeriSCOPE connects the data flow of a SCOPE program's procedural code together by examining a high-level declarative encoding of the program's pipeline topology. PeriSCOPE then applies three core compiler-like optimizations to the program. *Column reduction* suppresses unused data in the pipeline based on the program's reconstructed data flow. *Early filtering* moves filtering code earlier in the pipeline to reduce how much data is transmitted downstream. Finally, *smart cut* finds a better boundary between pipeline stages in the data flow graph to minimize cross-stage I/O; e.g., the code that computes a predicate from two string values could be moved to an earlier stage. The result is faster program execution because less data needs to be transferred between pipeline stages.

```
1  t1 = EXTRACT query:string,clicks:long,market:int,
2         name:string, text:string, ...
3       FROM    "/users/foo/click_0342342"
4       USING   DefaultTextExtractor("-s")
5       HAVING  IsValidUrl(url) AND clicks != 0;
6  t2 = REDUCE  t1 ON query
7       PRODUCE query, score, mvalue, cvalue,
8         name, text, ...
9       USING   PScoreReducer("clicks")
10      HAVING  GetLength(query) > 4;
11 t3 = PROCESS t2
12      PRODUCE query, cscore, pbran, pname
13      USING   SigReportProcessor("cvalue")
14      OUTPUT  t3 TO "/users/foo/click/0342342";
```

Fig. 1. Declarative code that defines the pipeline of a sample SCOPE program. Rows of typed columns (line 1) are first extracted from a log file (line 3) using a default text extractor (line 4) and filtered based on certain conditions (line 5). Next, the input rows are *reduced* with a user-defined function `PScoreReducer` (line 9) to produce a table with six columns (line 7) after being filtered (line 10). Finally, the user-defined function `SigReportProcessor` (line 13) is applied to the result as it is emitted (line 14).

The above code transformation moves the code across pipeline stages, and PeriSCOPE introduces several safety rules to ensure its correctness, i.e., the execution results of the job remain unchanged, which bridges our domain knowledge of data-parallel computation to program analysis and transformation. For example, a counting statement is not allowed to be moved before data-shuffling, because the ending count of the total records will be different due to data-shuffling. While the traditional program analysis usually considers *any* value for the input data to ensure the correctness of later code transformation, PeriSCOPE adopts symbolic execution to collect and propagate the constraints applied to the data along the execution flow to further enlarge the optimization scope while the safety rules always hold. For example, a job may have an early filter on the input data, which leads a predicate be always false for a *if* statement, enabling the whole if-body dead code. Removing such code enables more opportunities for all above optimizations, i.e., column reduction, early filtering, and smart cut.

We have implemented PeriSCOPE and evaluated its effectiveness on 33,681 real SCOPE jobs from a large production cluster. We also evaluate end-to-end performance comparisons on several real jobs.

The remainder of this paper is organized as follows. Section 2 presents a sample SCOPE program to show the potential benefits of PeriSCOPE's optimizations. The I/O-reduction optimizations in PeriSCOPE are described in Section 3, including column reduction, early filtering and smart cut. Section 4 discusses how PeriSCOPE ensures the correctness of the optimizations, and how it leverages symbolic execution to expand optimization scope. PeriSCOPE's implementation is covered in Section 5, followed by an evaluation in Section 6. We survey related work in Section 7 and conclude in Section 8.

2 Motivation and Overview

2.1 A Motivated Example

We motivate PeriSCOPE by describing the pipeline-aware optimization opportunities that are found in a sample data-parallel program, which is adapted from a real SCOPE job. SCOPE is a distributed data-parallel computation system that employs a hybrid programming model where declarative SQL-like code describes a program's high-level pipeline

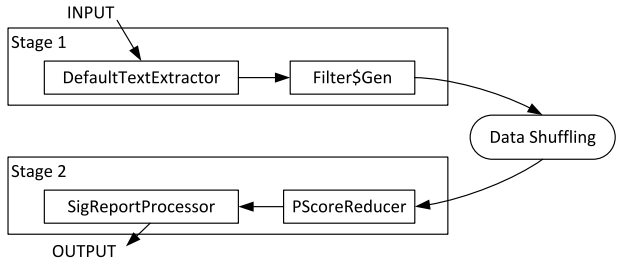


Fig. 2. An illustration of the pipeline defined by the declarative code in Fig. 1. The `Filter$Gen` operator is generated from the `HAVING` clauses on lines 4 and 8 of Fig. 1; other operators refer to user-defined functions. Each directed edge represents the data flow between operators.

structure, like other similar systems such as Hive [10], Pig [11], and DryadLINQ [6]. Fig. 1 shows the declarative code of our sample job that is compiled into an execution pipeline, which we illustrate in Fig. 2.

The operators of a SCOPE pipeline manipulate a data model of *rows* and *columns* and can be encoded as *user-defined functions* of procedural code that are either defined by the user or reused from generic libraries. A *computation stage* consists of one or more chained operators, and runs on a group of machines independently with partitioned data stored locally; *data-shuffling phases* then connect computation stages together by transmitting requisite data between machines. The pipeline in Fig. 2 contains two *computation stages* that are separated by one data-shuffling phase according to the reduce call on line 6 in Fig. 1. SCOPE applies logical optimizations, such as early selection, to programs according to the declarative structure of their pipeline. For example, the filtering clause on line 10 of Fig. 1 can be applied before data shuffling; and so the `Filter$Gen` operator in the first stage of Fig. 2 therefore includes the conditions from line 10 as well as line 5. Such logical optimizations apply only to the declarative code defined in Fig. 1, treating the procedural code of the `DefaultTextExtractor`, `PScoreReducer`, and `SigReportProcessor` as black boxes.

The SCOPE program of Fig. 1 is easily written by reusing two functions (`DefaultTextExtractor` and `SigReportProcessor` in Fig. 4) from generic libraries while the encoding of the custom `PScoreReducer` function, shown in Fig. 3, is straightforward. However, the program contains four serious I/O inefficiencies that need to be eliminated before it is "fast enough." First, the `if` statement on line 7 of Fig. 3 is actually a procedural filter that discards rows. Such rows can be filtered out early so that they are not transmitted during the data-shuffling phase, which can be accomplished by splitting `PScoreReducer` into two parts as encoded in Fig. 6: a `PScoreReducerPre` function that executes the computations of lines 5-7 in Fig. 3 before data-shuffling; and a `PScoreReducerPost` that executes the rest of the computations from the original `PScoreReducer` function after data-shuffling. Our sample program's declarative SCOPE code is updated in Fig. 5 to reflect this split, whose pipeline is illustrated in Fig. 7.

Next, the `alteredQuery` column is transmitted only for computing a simple predicate on line 9 of Fig. 3; the predicate computation can be done before the shuffling phase so that smaller Boolean values are transmitted instead of

```
1  public class PScoreReducer : Reducer {
2  List<Row> Reduce(List<Row> input, string[] args){
3    maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
4    foreach (Row row in input) {
5     int impr = SmoothImpr(row[args[0]].Long());
6     bool incl = row["ctrls"].Contains(args[0]);
7     if (!incl && impr < 0) continue;
8     string[] keys = row["query"].Split(',');
9     bool p = row["alteredQuery"].ContainsAny(keys);
10     if (p)
11       score += ...;
12     if (impr > maxImpr)
13       maxImpr = impr;
14     if (impr * IMPR_RATIO > maxImpr) continue;
15     ... cvalue += ...
16     ... mvalue += ... row["market"] ...
17   }
18   outRow[1] = Normalize(score, ...);
19   outRow["mvalue"] = mvalue;
20   outRow["cvalue"] = cvalue;
21   ...
22   yield return outRow;
23 }}
```

Fig. 3. The procedural code of the `PScoreReducer` user-defined function. Because `PScoreReducer` is a reduce operator, the preceding data shuffling ensures that rows having the same shuffling key are grouped together. For each group (`input`) of rows sharing the same shuffling key (line 2), this reduce operator iterates on each row in that group using a loop (lines 4-17) and outputs a single row as outRow for that group (line 22). The `impr` variable of line 5 represents an "improvement" that regulates accumulation of `mvalue` and `cvalue`.

strings. This is accomplished by computing the predicate in **`PScoreReducerPre`** on line 16 of Fig. 6 and propagating its result as a column to **`PScoreReducerPost`** where it is used on line 29.

Third, the **`SigReportProcessor`** function called on line 13 of Fig. 1 uses the **`cvalue`** column, bound to its input parameter, that is computed by the **`PScoreReducer`** function; in contrast the **`mvalue`** column computed on lines 16 and 19 of Fig. 3 is unused and therefore does not need to be computed and propagated in the **`PScoreReducerPost`** function of Fig. 6. As a consequent, the programmer can define their own specialized **`MyTextExtractor`** function (top of Fig. 6) that does not extract and propagate the **`market`** column.

Finally, the predicate for the **`if`** statement on line 6-7 in Fig. 4 is always `False` because all input rows are guarded by the previous filter **`GetLength(query)>4`** (line 10 in Fig. 1). Consequently, lines 6-9 in Fig. 4 are dead and can be

```
1  public class SigReportProcessor : Processor {
2  List<Row> Process(List<Row> input, string[] args){
3  foreach (Row row in input) {
4     string text = row["text"];
5     kwstring name = row["name"];
6     if(row["query"].IsNullOrEmpty() ||
7        row["query"].Length <= 2){
8       outRow["pbran"] = text;
9       outRow["pname"] = name;
10    }else if(!text.IsNullOrEmpty()) {
11      Product p = new Product(text +","+row[args[0]]);
12      outRow["pbran"] = p.GetProdBrand();
13      outRow["pname"] = p.GetProdName();
14      outRow["cscore"] = p.GetCScore();
15    }
16    ...
17    yield return outRow;
18 }}
```

Fig. 4. The procedural code of the `SigReportProcessor` user-defined function; strike-through text is original code that is eliminated. Lines 6-9 are eliminated because they are unreachable, line 5 is eliminated as it doesn't contribute to `outRow`, and line 10 is eliminated as it's forwarded by early filtering.

```
1  t1 = EXTRACT query:string,clicks:long,market:int,
2         name:string, text:string, ...
3       FROM    "/users/foo/click_0342342"
4       USING   MyTextExtractor
5       HAVING  IsValidUrl(url) AND clicks != 0
6                       AND GetLength(query) > 4
7                       AND !String.IsNullOrEmpty(text);
8  t2 = PROCESS t1 PRODUCE query, impr,
9         name, text, ...
10      USING   PScoreReducerPre;
11 t3 = REDUCE  t2 ON query
12      PRODUCE query, score, mvalue, cvalue,
13        name, text
14      USING   PScoreReducerPost
15 t4 = PROCESS t3 PRODUCE query, cscore, pbran, pname
16      USING   SigReportProcessor("cvalue")
17      OUTPUT  t4 TO "/users/foo/click/0342342";
```

Fig. 5. Optimized declarative code of our sample program; strike-through text is original code that is eliminated.

eliminated, which further makes *name* an unused column because line 9 is the only place to use the value derived from *name*.

2.2 Overview of PeriSCOPE

The optimized sample program in Figs. 5 and 6 executes more efficiently at the expense of programmability: it can no longer reuse the **`DefaultTextExtractor`** function, the logic for the **`PScoreReducer`** function is now distributed into two sections that execute in different pipeline stages, while the optimizations are tedious as the programmer must carefully move code across pipeline stages.

PeriSCOPE automates such optimizations by considering both user-defined functions and the pipeline of a data-parallel program. In particular, PeriSCOPE reconstructs the data

```
1  public class MyTextExtractor : Extractor {
2  List<Row> Extract(StreamReader reader, string[] args){
3    ...
4    string[] columns = line.Split(',');
5    int market = int.Parse(columns[2]);
6    outRow[2].Set(market);
7    ...
8  }}
9  public class PScoreReducerPre : Processor {
10 List<Row> Process(List<Row> input, string[] args){
11   foreach (Row row in input) {
12     int impr = SmoothImpr(row["clicks"].Long());
13     bool incl = row["ctrls"].Contains("clicks");
14     if (!incl && impr < 0) continue;
15     string[] keys = row["query"].Split(',');
16     outRow["p"] =
17       row["alteredQuery"].ContainsAny(keys);
18     outRow["impr"] = impr;
19     ...
20     yield return outRow;
21 }}
22 public class PScoreReducerPost : Reducer {
23 List<Row> Reduce(List<Row> input, string[] args){
24   maxImpr = 0; score = 0; mvalue = 0; cvalue = 0;
25   foreach (Row row in input) {
26     int impr = row["impr"].Int();
27     bool incl = row["ctrls"].Contains("clicks");
28     if (!incl && impr < 0) continue;
29     if (row["p"].Boolean()) score += ...;
30     if (impr > maxImpr) maxImpr = impr;
31     if (impr * IMPR_RATIO > maxImpr) continue;
32     ... cvalue += ...
33     ... mvalue += ... row["market"] ...
34   }
35   outRow["score"] = Normalize(score, ...);
36   outRow["mvalue"] = mvalue;
37   outRow["cvalue"] = cvalue;
38   ...
39   yield return outRow;
40 }}
```

Fig. 6. Optimized procedural code of our sample program.

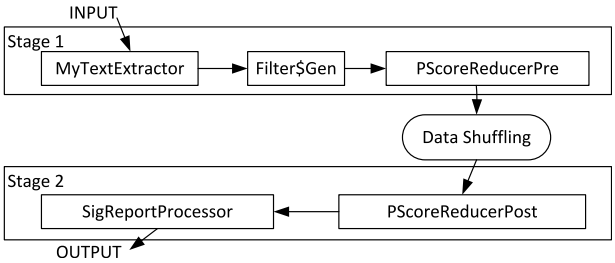


Fig. 7. Resulting pipeline for the optimized job.

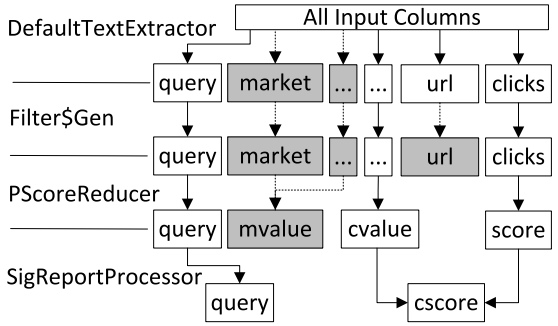


Fig. 8. A simplified column-dependency graph for column reduction. Columns and computation in the shaded areas are removed by the column reduction optimization.

flow across the user-defined functions according to the pipeline topology and applies *column reduction* to remove unused columns along with the computations to compute them from user-defined functions; e.g., PeriSCOPE can eliminate the unused **mvalue**, **market** and **name** columns of our sample program. PeriSCOPE next identifies filtering conditions in a user-defined function and moves them earlier in the pipeline through *early filtering*; e.g., the **if** condition on line 7 of Fig. 3 is moved earlier to reduce row propagation in the pipeline. Finally, PeriSCOPE applies *smart cut* that finds better boundaries between two stages to minimize data-shuffling I/O by moving size-reducing transformation upstream and size-enlarging transformation downstream in the pipeline. We describe how PeriSCOPE automates these optimizations in Sections 3.

PeriSCOPE also adopts *symbolic execution* to enlarge optimization opportunities by reasoning that the code is executed with *what* conditions of the input data instead of *any* value, e.g., lines 11-14 and line 17 are executed with a constraint (line 10) on input data, with which early filtering is applied for constraint so that less data is transferred. We describe how symbolic execution helps PeriSCOPE to expand optimization scope in Section 4.

3 I/O Reduction

This section describes three I/O reduction optimizations in PeriSCOPE. *Column reduction* suppresses unused data in the pipeline. *Early filtering* moves filtering code earlier in the pipeline to reduce the number of rows transmitted downstream. *Smart cut* finds a better boundary between pipeline stages in the data flow graph to minimize cross-stage I/O.

3.1 Column Reduction

A user-defined function might not use a particular input column that is available to it in a calling pipeline. For example, the **SigReportProcessor** function of Section 2's sample program does not use the **mvalue** column of the pipeline encoded in Fig. 1. In distributed data-parallel programs, transferring unused columns during data-shuffling can consume a significant amount of network I/O. As we discuss in Section 6, this problem commonly arises from the reuse of user-defined functions that we observe in production SCOPE jobs.

Column reduction is an optimization in PeriSCOPE that leverages information about how operators are connected together in a pipeline to eliminate unused columns from the program, removing their associated computation and I/O costs. The optimization analyzes the dependency information between the input and output columns of all operators in the pipeline; Fig. 8 shows part of the column dependency graph for the example in Fig. 1. An input or output column of an operator is represented as a vertex while an edge from a source column s to a destination column d indicates that d has either a data or control dependency on s. Only data dependency edges are shown in Fig. 8 as control dependency edges are too numerous to illustrate clearly. Because SCOPE allows a column to be accessed by name (e.g., line 6 in Fig. 3) or index (e.g., line 18), a column read or write operation may be *unresolved* during compilation, where PeriSCOPE considers that this column could be any column that is visible to the user-defined function, leaving no opportunity for column reduction. Fortunately, as we discuss in Section 6, column accesses that cannot be resolved through simple optimization techniques are relatively rare—at only a 13.4 percent occurrence in our survey of real SCOPE jobs.

PeriSCOPE applies column reduction by computing a set of "used" output columns for each operator that are used as the input columns of succeeding operators in the pipeline topology. If the operator immediately precedes a data-shuffling phase, the shuffling-key columns are also required as used output columns. Any unused output columns of an operator are removed and, if the operator is a user-defined function, PeriSCOPE also rewrites it to remove all code that only contributes to computing the removed output columns. If any columns were removed, PeriSCOPE removes any input columns that are no longer used because of removed code and repeats column reduction again.

For example, the column **mvalue** is removed from the **PScoreReducer** function because it is not used by the **SigReportProcessor** function listed in Fig. 3. This causes the code that computes **mvalue** to be removed (lines 36 and 33 in Fig. 6), which further causes the output column **market** to be removed from the **DefaultTextExtractor** function. Finally, PeriSCOPE creates, through specialization that eliminates code, function whose code is semantically similar to the **MyTextExtractor** function of Fig. 6.

3.2 Early Filter

Early filtering is applied to the first user-defined function in a computation stage that executes after a data shuffling phase. PeriSCOPE first identifies filtering statements in the user-defined function's main loop, which are statements that branch back to the beginning of the main loop. Fig. 9 shows the control flow graph for the loop body of

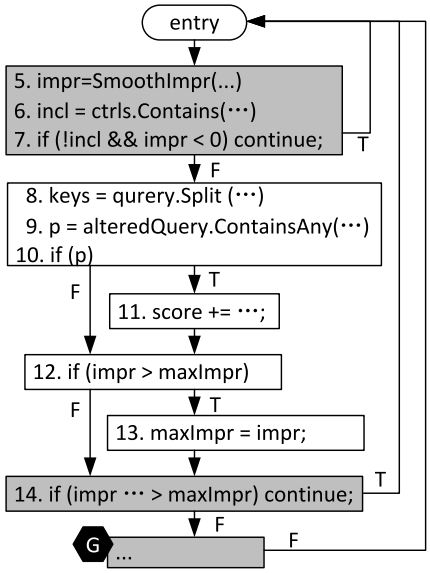


Fig. 9. Control flow graph for the loop body in `PScoreReducer` in Fig. 3. Edges marked T and F are branches that are taken when the last predicate in the source basic block evaluates to true and false, respectively. The vertices in gray are the basic blocks that contain filtering statements.

`PScoreReducer` in Fig. 3; statements 7, 14 and the end of basic block `G` are identified as filtering statements.

Earlier filtering will reduce the number of rows that are iterated on later. Since the data-shuffling phase can transparently change the number and order of the rows processed on each machine through re-partitioning and grouping, PeriSCOPE must ensure that moving a filtering statement does not change the cumulative effect. Specifically, we define an computation that relies on the number or order of the rows *stateful*, and such stateful computation (e.g., lines 11-16 on Fig. 4) must be carefully taken care of to ensure correctness; further details are discussed in Section 4.

PeriSCOPE next identifies code that computes the filtering condition by applying backward slicing [12], which starts from the identified filtering statement and collects, as its *backward slice*, the statements that can affect it. The backward slice of statement 7 in Fig. 9 includes statements 5-7. PeriSCOPE then copies the entire backward slice upward causing rows to be filtered out before data shuffling occurs. Finally, the conditions of the moved filter can now be assumed in the original user-defined function, enabling the removal of code through dead code elimination. For the code in Fig. 9, statement 7 is removed because (`!incl && impr` < 0) is always false; no row otherwise is permitted past the data-shuffling phase due to early filtering. Statement 6 is then removed because `incl` is not used anymore, causing `ctrls` to become unused in the user-defined function. As a result, early filtering not only reduces the number of rows that are transferred across a data shuffling phase, but can also trigger column reduction (e.g., on `ctrls`).

3.3 Smart Cut

The cross-stage flow of data across the network in a data-parallel program is significantly more expensive than a traditional program whose data flows only through memory. PeriSCOPE therefore aims at re-partitioning the code by finding *smart cuts* as shuffling I/O boundaries that minimize cross-stage data flow. Finding smart-cuts can be formulated as a compiler-like instruction scheduling problem. However, while a compiler usually rearranges instructions to improve instruction-level parallelism on a specific CPU architecture, smart cut reorders statements to reduce the amount of data transmitted across the network.

```
5   (T)  impr = SmoothImpr(row["clicks"].Long());
8   (T)  keys = row["query"].String().Split(',');
9   (T)  p = row["alteredQuery"].ContainsAny(keys);
11  (p)  score += ...;
12  (T)  p1 = impr > maxImpr;
13  (p1) maxImpr = impr;
14  (T)  p2 = !(impr * IMPR_RATIO > maxImpr);
15  (p2) ... cvalue += ...
```

Fig. 10. Simplified if-conversion result for lines 5-15 in Fig. 3. `T` stands for `True` which means that the statement always executes.

Smart cut is applied to user-defined functions that are immediately adjacent to data-shuffling phases. PeriSCOPE first applies if-conversion [13] to the body of the main loop for a given user-defined function so that the loop body becomes a single basic block, which is necessary because instruction scheduling can only be applied to blocks of non-branching instructions. Fig. 10 shows the simplified result for the code segment on lines 5-15 of Fig. 3, after lines 6 and 7 are removed according to early filtering. Every statement is now guarded with a predicate that specifies the path condition of its execution; e.g., the statement on line 13 is guarded with predicate `p1` because it is executed only when `p1` is true.

PeriSCOPE then builds a data dependency graph for this basic block using the SSA [14] format. Vertices in the data dependency graph are instructions, while directed edges represent read-after-write (RAW) data dependencies where sink instructions use variables defined in the source instructions. PeriSCOPE labels the edges with the name and byte size of the dependent variables, which are either columns or local variables. Fig. 11 shows part of the labeled data dependency graph for our example; PeriSCOPE further adds two vertices `S` and `T` to represent the overall input and output of this code snippet, respectively. PeriSCOPE also adds an edge labeled `query` from `S` to `T` as `query` is used as the shuffling key and should always be transmitted.

PeriSCOPE adds directed edges from `S` to any statement that is either stateful or generates shuffling keys before the data-shuffling phase, and adds directed edges from any stateful statement after the data-shuffling phase to `T`; all of these edges have an infinite weight to ensure

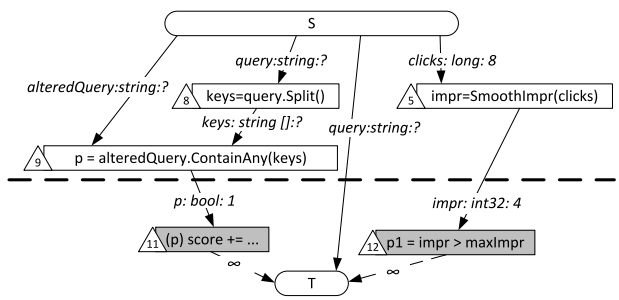


Fig. 11. Labeled data dependency graph with a smart cut. Statements 13-15 are omitted. Statements in gray are stateful.

that those statements are never moved across the data-shuffling phase.

The smart-cut problem is now reduced to one of finding an edge cut between S and T in the data dependency graph that minimizes the total byte size of all dependent variables on edges across the cut.

Computing an optimal edge cut statically is difficult because the precise weights of some edges depend on dynamic data. In practice, PeriSCOPE resorts to a simple heuristic-based technique to identify opportunities to move code across data-shuffling phases. Specifically, PeriSCOPE looks for a simple pattern with a variable computed from one or more input columns. If the total size of the input columns that are used only for computing this variable is larger than the size of this variable, this computation should be moved to an earlier stage. Similarly, PeriSCOPE also looks for a reverse pattern where a variable is used to generate one or more output columns. In Fig. 11, the input columns **alteredQuery** and **query** from Fig. 3 are used to compute variable **p** in the optimize function of Fig. 6. Although the **alteredQuery** column is never used elsewhere, the **query** column is used in a later stage. Because the byte size of a string type (**alteredQuery**) is always larger than that of a Boolean variable (p), the cut should cross the edges labeled with p, instead of those labeled with **alteredQuery**. In the end, edges between statements 9 and 11, and between statements 5 and 12, are selected for the smart cut.

4 Defining Optimization Scope

This section describes how PeriSCOPE defines the scope for the above optimizations. In particular, it focuses on two questions: first, how to ensure the code transformation to be safe in that the execution results of a job remains the same; second, how to enlarge the optimization scope without breaking the safety guarantee? We answer the first question by introducing three safety rules, which bridges our domain knowledge about data-parallel computation to program analysis, and the second by adopting symbolic execution to collect and propagate certain constraints on the input data values along the execution flow to expose further opportunities.

4.1 Safety Rules

Our stated techniques, *early filtering* and *smart cut*, move code from user-defined functions across pipeline stages. Such code motion must be done only if it's safe, i.e., the results of the program are unchanged. We describe three correctness conditions of code motion using the example in Fig. 3, with a focus on identifying the domain knowledge that is needed to define correctness. The idea is to model the dependencies between the data shuffling code and the code that before and after data shuffling, without really analyzing the data shuffling code.

First, any computation which relies on the number or order of the rows cannot be moved across the data-shuffling phase, i.e.,

Rule 1. *PeriSCOPE must not move a stateful statement across the data shuffling phase.*

For example, for the stateful statements on lines 12-14 in Fig. 3, the value of **maxImpr** depends on the processing order of the input rows. Because data shuffling re-orders rows based on a shuffling key, computing **maxImpr** before and after data shuffling would yield different results. So PeriSCOPE cannot move the computation of **maxImpr** before data shuffling.

Second, early filtering can only eliminate rows which donnot affect other downstream rows, or contribute to stateful variables; i.e.,

Rule 2. *PeriSCOPE must not move a filtering statement before a data shuffling phase if the statement is, or is reachable from, a stateful statement.*

This rule excludes statement 14 from early filtering in Fig. 9 because it is stateful, and excludes the last statement in basic block G because it is reachable from statement 14.

Finally, the data-shuffling phase reads the shuffling-key columns of each row, leaving other columns untouched; i.e.,

Rule 3. *PeriSCOPE must not move a statement after data shuffling if it generates shuffling-key columns.*

Since our safety rules largely depend on stateful statements, care must be taken in identifying stateful statements. PeriSCOPE applies loop dependency analysis [15] to the body of the main loop for each user-defined function to identify stateful statements as those that have loop-carried dependencies. A loop-carried dependency indicates that the destination statement relies on the execution of the source statement from an earlier iteration.

Although all the three safety rules depend on stateful statements, they are largely orthogonal and provide different ensurance to correctness. Rule 1 strictly forbidden moving stateful statements across data shuffling. Rule 2 shows the condition of forwarding a filtering statement, while Rule 3 shows the condition of moving a statement after data shuffling.

4.2 Enlarging Scope with Symbolic Execution

PeriSCOPE further adopts symbolic execution to expand optimization scope, which is inspired by the observation that there are redundant or contradicted if-predicates in many nested *if* statements, resulting dead code be the predicate itself and those in one of the branches. For example, lines 6-9 in Fig. 4 is considered dead code because the input data is guarded by a previous if predicate (**GetLength**(**query**) $>$ **4** on line 6 in Fig. 2) and the predicate on line 7 will be always **False** (contradicted to the above predicate). Removing the dead code usually exposes more opportunities to our I/O reduction techniques above. In this case, eliminating the code further makes name a dead column therefore column reduction becomes applicable.

We formally define this optimization as follows. Consider an if-predicate p (called branch condition), and all constraints S applied to the input data that make the data reach p (called path condition). When

$$\Omega(S, p) = \emptyset,$$

where $\Omega(X)$ contains all data satisfy X, p is always `False` which results the if-block dead code. Similarly, when

$$\Omega(S, \neg p) = \emptyset,$$

p is always `True` which makes the $else$-block dead.

To realize this opportunity, PeriSCOPE slices a program into *paths* with symbolic execution by traversing all instructions in a topological order with respect to their data and control dependencies: (1) PeriSCOPE starts from the entry-point of the program (a PeriSCOPE job in this case) with an empty path constraint S; (2) when it encounters a branch instruction with predicate p, it forks the current traverse process into two with different path constraints (S, p) and $(S, \neg p)$. PeriSCOPE then checks the two path constraints to see whether $\Omega(S, p) = \emptyset$ (or $\Omega(S, \neg p) = \emptyset$). If it is true, PeriSCOPE terminates the traverse of the correspondent path as an *unreachable path*; (3) when it comes across an instruction which yields output data for the job, the current path is terminated as a *reachable path*; (4) for other instructions, PeriSCOPE generates symbolic value for all defined operands, which may be referenced in later branch conditions.

As a result, PeriSCOPE slices a program into many reachable and unreachable paths with each labeled with different path constraints. For instructions that do not appear in any paths, PeriSCOPE eliminates them because they are dead. After that, PeriSCOPE applies the I/O reduction optimization again to see whether the code elimination exposes more opportunities. Note we have more opportunities here by doing path-specific optimization with the labeled path constraints as the filtering condition for the input data, which is left as a future work.

4.3 Solving Constraints

This section describes how we solve $\Omega(S, p) = \emptyset$ given path condition S and branch condition p. According to how PeriSCOPE generates the path conditions described above, we know $S = \{c_1, c_2, \ldots, c_k\}$ where $c_1, c_2, \ldots c_k$ are the branch conditions from the entry-point of the job to the place where this path condition is generated. Because PeriSCOPE terminates the traverse when $\Omega(S, p) = \emptyset$, we have $\Omega(c_i, c_j) \neq \emptyset$ where $c_i, c_j \in S$ (otherwise $\Omega(S) = \emptyset$). We therefore have

Lemma 1.

$$\exists c_i \in S : \Omega(c_i, p) = \emptyset \quad \Rightarrow \quad \Omega(S, p) = \emptyset.$$

As a result, instead of checking S against p holistically, which is required if we really want to find the data satisfying both S and p, PeriSCOPE only checks each c_i against p, which simplifies the problem. Intuitively, we can use a solver (e.g., Z3 [16]) to answer the question: is there a possible data value that satisfies both c_i and p? If the answer is no, we know $\Omega(S, p) = \emptyset$. If the answer is yes or timeout due to the solver capability or time constraint, PeriSCOPE conservatively thinks that there are possible satisfiable data values. This approach only produces false negatives, ensuring our optimization is safe.

The challenge relies on the expressiveness of the supported language by a solver: state-of-the-art solvers like Z3 do not support non-arithmetic types and their operations

```
1 if (name.StartWith("ABCD"))
2    if (!name.Contains("XYZ")) {
3       len = name.GetLength();
4       if (len < 4){
5       ...
6       }
7    }
```

Fig. 12. Example for symbolic execution. Line 5 is dead.

(e.g., string and string-based operations). However, they are commonly used in our targeted SCOPE job programs. PeriSCOPE addresses this challenge as follows. Considering $\Omega(c_i, p)$, if both c_i and p are arithmetic constraints, they can be fed to solvers directly. If both of them are non-arithmetic constraints, PeriSCOPE involves specific symbolic execution engine to solve it, described later. Otherwise, PeriSCOPE conservatively thinks there are possible satisfiable data values, which only introduces false negative (ensuring safety) and it is usually true because these two kinds of constraints are in most cases orthogonal.

4.3.1 Handling Non-Arithmetic Constraints

In PeriSCOPE, most non-arithmetic constraints come from containers, like list, array, dictionary and string. We here use string, which is the most challenging container type we encountered in SCOPE, as an example to illustrate how we model and solve non-arithmetic constraints; other containers are handled similarly.

We use the example in Fig. 12, and we assume the symbolic execution process is now at line 1 where we want to check whether the following if-block at lines 2-6 is dead or not. At this point, we have $S = \{\texttt{name.StartWith}(\text{“}\texttt{ABCD}\text{”})\}$ and $p = \texttt{!name.Contains}(\text{“}\texttt{XYZ}\text{”})$. To make the constraint solving process operation independent, PeriSCOPE models the constraint imposed by a string operation based on the observation that the constraint is usually applied to a sub string with its value equal to (or not equal to) some input value, defined as follows:

$$C = (host, value, location, isEqual)$$

where $host$ is the host string the operation is bound to, $location$ tells where the sub string is, $value$ is a constant string or a symbolic value computed before, and $isEqual$ indicates whether the sub string is equal to the given value or not. Instead of precisely capturing the location, PeriSCOPE defines the following four kinds of location as an approximation to ease the reasoning later:

- $start$: $value$ is at the head of $host$.
- end: $value$ is at the tail of $host$.
- any: there are one or more $value$s contained in $host$.
- all: $value$ is equal to $host$.

We therefore get the above S and p defined as follows:

$$S = \{(\texttt{name}, \text{“}\texttt{ABCD}\text{”}, start, \texttt{True})\}$$
$$p = (\texttt{name}, \text{“}\texttt{XYZ}\text{”}, any, \texttt{False})$$

Evaluating $\Omega(c_i, p) = \emptyset$ or not where $c_i \in S$ is non-trivial. However, due to this limited model, we are able to enumerate all the possible combinations (regarding to $location$ and $isEqual$) and write decision function for each. For the above example, the function return **`False`**, i.e., it is possible that

both the constraints are satisfied. We put an complete implementation of these decision functions at [17].

4.3.2 Extracting Arithmetic Constraints

While PeriSCOPE in general considers there are always satisfiable data points regarding to an arithmetic constraint and a non-arithmetic constraint because they are largely orthogonal, one meaningful exception we see in SCOPE jobs is that the program extracts the `Length` property from `string`, which anticipates further arithmetic computation. PeriSCOPE therefore tries to minimize false negatives introduced by this phenomena by adding a simple constraint: the length of a string is equal to or greater than the sum of each $value$'s length appeared in all constraints with their $host$ the current string and $isEqual$ true, i.e.,

$$len(h_i) \geq \sum len(c_i.value),$$
$$\text{where} \quad c_i.host = h_i, c_i.location \neq all, c_i.isEqual = \texttt{True}$$

$$len(h_i) = len(c_i.value),$$
$$\text{where} \quad c_i.host = h_i, c_i.location = all, c_i.isEqual = \texttt{True}.$$

Consequently, the path condition at line 3 in Fig. 12 contains an additional constraint: $len(name) >= len(\text{“}ABCD\text{”})$, which contradicts to the succeeding constraint $len(name) < 4$ (imposed by line 4), making line 5 an unreachable statement.

Note this additional constraint has a precondition that $value$s from different constraints are non-overlapping. This is not always true and PeriSCOPE has to check it statically. If they are overlapping or PeriSCOPE cannot decide it statically, PeriSCOPE conservatively considers overlapping possible, and drops this additional constraint at the cost of higher false negative ratio. In reality, this rarely happens: our study of more than 20K SCOPE jobs reveals that only less than 1.5 percent of the jobs contain overlapped $value$s or cannot be decided statically.

5 Implementation

PeriSCOPE examines a SCOPE program's operators, the definition of the rows used by the operators, and the program's pipeline topology represented as a directed acyclic graph (DAG) in the program's execution plan. In PeriSCOPE's targeting large scale data parallelism, a program is modeled as a dataflow graph: a directed acyclic graph with vertices representing processes and edges representing data flows [3]. Furthermore, the DAG part is organized with declarative SQL-like code, while the vertices, which include `if-else` branches and loops, are organized with procedural code, like C#. As if-else branches or loops are in each vertices, instead of between execution stages, we only represent the execution stages as DAG, instead of the whole program.

The operators and row definitions are extracted from .NET binary executables, while the pipeline topology is represented as an XML file. PeriSCOPE extends ILSpy [18], a de-compiler from .NET byte-code to C# code, and Cecil [19], a library to generate and inspect .NET assemblies, to implement PeriSCOPE as two components. PeriSCOPE's *optimizer* is built on top of ILSpy to specialize all operators

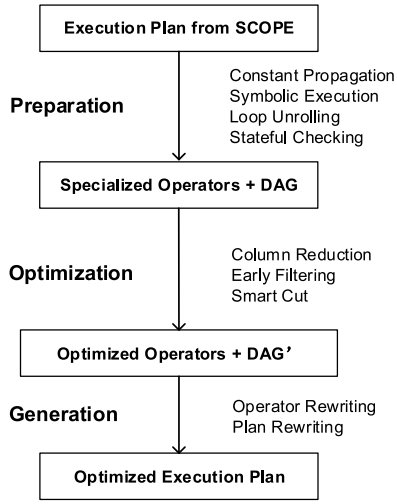


Fig. 13. Optimization flow in PeriSCOPE.

in the input execution plan, applying all PeriSCOPE's optimizations to operators (as user-defined functions) as found at the intermediate representation (IR) level. The *generator* emits new bytecode for user-defined functions and generates all utility code for the program, such as new row schemas and their related serialization routines, as well as the new SCOPE description file for the execution plan.

The optimizer and generator components are both implemented in C# with 7,334 and 2,350 lines of code, respectively. The reason we choose C# is most of our existing tools are written with C#, including ILSpy and Cecil. As PeriSCOPE's optimizations work at the level of byte-code operators and pipeline descriptions, our implementation does not limits to C# as our discussion shows in Section 7. Fig. 13 illustrates PeriSCOPE's optimization flow with three major tasks, each containing several steps, where the optimizer performs the first two tasks, while the generator performs the last. Plan rewriting updates the original DAG XML file that describes pipeline topology because some original operators are now split into different computation stages.

The symbolic execution engine is implemented in C# with about 3,130 lines of code, which uses Mathematica [20] as the SAT solver. Inside the engine, we built support for the most commonly used non-arithmetic types: string, list and dictionary, with 53 decision functions in total. To enable commonly used non-arithmetic types, we consider an object as a set of properties generated by corresponding operations instead of representing it with possible input values. All operations which involve non-algorithmic types, are converted to one of our decision functions.

Another challenge of implementing symbolic execution is we have to leverage heuristics to limit the time of PeriSCOPE's symbolically executing a loop. Usually, PeriSCOPE can't decide the loop time as the loop bound is a variable, and symbolic execution may be trapped by *infinite* loop. We explore a loop until all branches in the loop are tested instead of testing the symbolic value. This means we cannot cover all paths in loop but it's sufficient for PeriSCOPE's detecting unreachable paths and path conditions. Special loops also leverage some domain knowledge. For example, the main loop which iterates on input data is

TABLE 1
Optimization Coverage Statistics which Lists the Number and the Percentage of the Jobs that Are Eligible for the Given Optimization

optimization	eligible jobs (BSE)	eligible jobs (ASE)
column reduction	5,289 (15.70%)	5,963 (17.70%)
early filtering	1,843 (5.47%)	3,630 (10.78%)
smart cut	1,818 (5.40%)	2,060 (6.12%)
Total	6,989 (20.75%)	9,002 (26.73%)

BSE is short for before symbolic execution while ASE is short for after symbolic execution.

symbolically executed only once, the loop with known bound is also checked if the loop bound is too large to suffer.

Instead of directly rewriting operator code, PeriSCOPE copies operator code when it needs to be written because a user-defined function can be reused multiple times in a job, each reuse requiring different code transformations. Likewise, row type schema definitions and serialization code are copied and transformed as columns are eliminated from different points in the pipeline.

6 Evaluation

Our evaluation focuses on first assessing the overall potential for these optimizations and second evaluating in detail the effectiveness of these optimizations on the end-to-end performance of several real production jobs. We use a real trace of 33,681 jobs from a 10,000 machine SCOPE production cluster to evaluate PeriSCOPE's core I/O reduction optimizations of column reduction, early filtering, and smart cut. The jobs are randomly collected without filtering or sampling to demonstrate the overall potential for these optimizations and several typical cases are selected to show PeriSCOPE's effectiveness in detail.

With an average analysis time of 3.9 seconds for each job, our current implementation successfully analyzes 26,109 (78 percent) of the 33,681 jobs. PeriSCOPE fails on the rest of these jobs given limitations in our implementation primarily relating to inconsistent SCOPE versions (6.1 percent) or outright ILSpy de-compilation failures (2.3 percent), but a minority involve code that cannot be analyzed in general due to unresolved column indices (13.4 percent) or for reasons that we have yet been unable to determine (0.3 percent).

Table 1 shows that before applying symbolic execution 15.70 percent of the jobs are eligible for column reduction optimization, 5.47 percent for early filtering, and 5.40 percent for smart cut. It also shows that the optimizing scope is enlarged after applying symbolic execution, 17.70 percent for column reduction optimization, 10.78 percent for early filtering, and 6.12 percent for smart cut. Some jobs are eligible for multiple types of optimizations, and so the total percentage (20.75 percent before and 26.73 percent after applying symbolic execution) of jobs that are eligible for those optimizations is lower than the sum of the three.

We next examine the user-defined functions of these jobs. We found that these jobs used only 1,716 unique user-defined functions, meaning many jobs are encoded purely in declarative code that leverages pre-existing user-defined functions. About 20.0 percent of the user-defined functions are reused more than ten times, where the most popular user-defined function is reused 1,263 times. We suppose that the heavy reuse of user-defined functions creates more opportunities for PeriSCOPE's optimizations. And in fact, about 30.0 percent of the user-defined functions in jobs eligible for column reduction were reused at least 10 times, confirming our speculation that generic library functions contain a lot of redundancies that can be optimized away. On the other hand, no such correlation is observed for early filtering or smart cut, whose eligibility appear to be unrelated to reuse. Finally, 30.2 percent unique user-defined functions used in these jobs have arguments in their function bodies that are used as branch conditions or column names, while 79.1 percent of the user-defined function invocations in the job scripts contain constant parameters. Specialization of such user-defined functions is a necessary pre-processing step to resolve columns and apply PeriSCOPE's optimizations.

6.1 Case Study for I/O Reduction

To understand the overall effectiveness of PeriSCOPE's optimizations, we compare the performance of the jobs before and after our optimization (w/o symbolic execution) in terms of both execution time and the amount of I/O used during the data-shuffling phase; effectiveness of symbolic execution is present next. Ideally, we would carry out this experiment with representative benchmarks, which unfortunately do not exist. We therefore select eight real and typical SCOPE jobs that are eligible for at least one of PeriSCOPE's optimizations and whose input data is still available on the cluster. The selected jobs are mostly related to web-search scenarios that process crawler traces, search query histories, search clicks, user information, and product bidding logs. Our experiment executes these real production jobs (cases 1-8 in Fig. 14) on various number of machines. Specifically, cases 1, 2, and 4 use 1,000 machines, case 3 uses 10 machines, cases 5-7 use 192 machines, while case 8 uses 100 machines.

Fig. 14 shows the performance-gain breakdown for our chosen eight production jobs in terms of a reduction in both execution time and data-shuffling I/O. The unoptimized and optimized versions of each job are executed three times; we report the average. Due to the nature of our shared computing environment we are using, we see high relative standard deviations (7.3 to 23.0 percent) in our latency experiments, while the reduction numbers in data-shuffling I/O is a more reliable indicator. In particular, highest standard deviations are seen for cases 5 (23.0 and 22.6 percent) and 6 (18.0 and 14.9 percent), indicating that the reductions are insignificant statistically in those cases. The execution time reduction for case 8 (10 percent) is also statistically insignificant with standard deviations of 13.4 and 7.3 percent. Case 1 benefits from all three of PeriSCOPE's optimizations, cases 2-3 are eligible for two, while cases 4-8 are only eligible for one each. PeriSCOPE reduces data-shuffling I/O in all cases but the last by between 9 and 99 percent; the last case incurs no benefit for reasons discussed below. Execution time is reduced by between 7

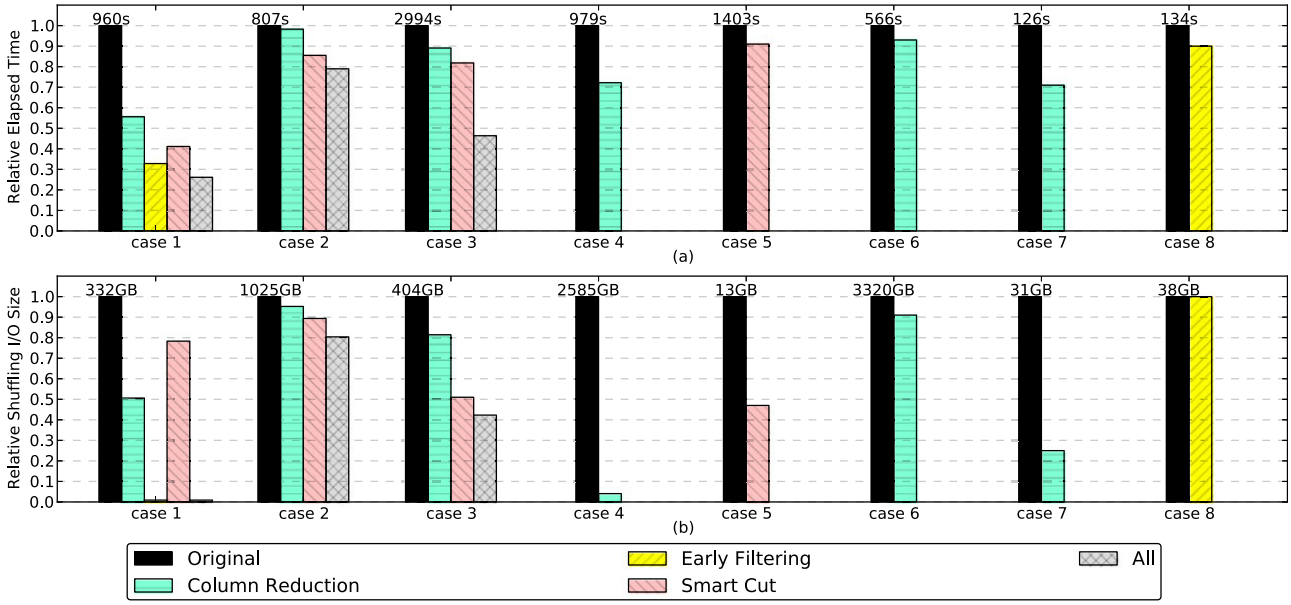


Fig. 14. Performance gains with PeriSCOPE's column reduction, early filtering, and smart cut optimizations (w/o symbolic execution); chart (a) labels unoptimized job time in seconds while chart (b) labels total unoptimized job shuffling I/O size in GB; the bars in each case represent the effectiveness of each optimization relative to unoptimized execution time (a) or shuffling I/O (b); shorter bars indicate more reduction; the "All" bar is only shown for cases that are eligible for more than one PeriSCOPE's optimization; both the execution time and the shuffling I/O are average values with a relative standard deviation (RSD) ranging from 7.3 to 23.0 percent due to the nature of our shared computing environment.

to 74 percent, which, beyond data-shuffling I/O, includes other tasks such as executing data-processing code, and reading and writing data to and from storage. Case 4 is particularly sensitive to storage overhead as this job extracts data from a 2.26 TB log file.

Column reduction can be applied six of the eight jobs, yielding I/O reductions ranging from 4.8 up to 96 percent that depend on how many columns are removed compared to the total byte size of all columns. Column reduction on case 4 removes 18 columns out of 22; the reducer that executes immediately after an extractor uses only four of the columns extracted. For case 7, only 2 out of 31 columns are used by its reducer; other columns are consumed by other operators and are not transmitted across the data-shuffling phase.

The effectiveness of early filtering depends highly on the goal of filtering. We have found that filtering conditions simply exclude rows whose columns have invalid values. While such case is rare, early filtering leads only to a negligible I/O reduction; case 8 is exactly this case. The execution time of case 8 is still reduced because PeriSCOPE moved the filtering computation to before the data-shuffling phase, improving the parallelism because more resource (136 CPU cores) are allocated to the stage before shuffling than after (42). When the filtering does not check for invalid values, they usually exclude a large number of rows and early filtering is quite effective. As an extreme case, data-shuffling I/O is reduced by 99 percent in case 1 because the vast majority of the rows in this job are filtered out and so do not need to be transmitted in the pipeline. The opportunity for early filtering discovered by PeriSCOPE was not obvious: 7 **if** conditions, some of them deeply nested, select desired rows for various computations, and manually writing a single filtering condition to replicate these **if** conditions is not trivial for a developer.

In contrast to early filtering, smart cut will always deliver I/O reductions when it can be applied. Computations that trigger smart cut typically involve one column that is mapped to a column of a smaller size, usually via the conversion from string to some arithmetic types, or size-reduction operations such as *Trim* and *Substring*. Binary operations (e.g., $+, *, ==, >$) between two input columns can also trigger smart cut. For example, case 5 contains two string-typed columns as start and end event timestamps; the job parses the two as integer timestamps and computes their delta for the elapsed time of the event, where smart cut causes the delta to be precomputed.

6.2 Case Study for Symbolic Execution

Fig. 15 shows further I/O reduction after PeriSCOPE adopts symbolic execution. The four cases are randomly selected from the job set in which the jobs benefit from symbolic execution.

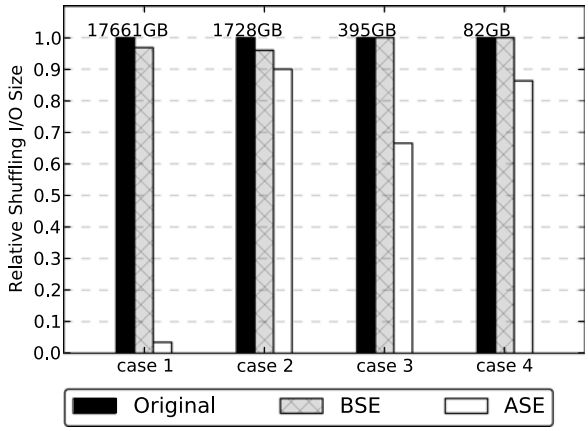


Fig. 15. I/O reduction before and after applying symbolic execution. BSE means applying all stated optimizations before symbolic execution while ASE means applying all optimizations after symbolic execution.

Before applying symbolic execution, PeriSCOPE optimizes case 1 and 2 by column reduction, and PeriSCOPE does not see any opportunities in cases 3 and 4. After applying symbolic execution, dead code is eliminated and further opportunities are exposed for PeriSCOPE's I/O reduction optimizations. In case 1, PeriSCOPE determines that a filtering statement applied on joined data is always false, which further enables early filtering on other branches of joined data, resulting few data joined and data shuffling I/O reduced by 96 percent. In cases 2, an user-defined function is found to generate part of the outputs on an unreachable path, enabling certain code eliminated and early filtering opportunities. In case 3 and 4, symbolic execution helps get all column access index resolved which exposes multiple unused columns.

7 Related Work

PeriSCOPE is closely related to a large body of research in the areas of data-parallel computation, distributed database systems [21] and compiler optimizations [15]. Instead of attempting to cover those areas thoroughly, we focus on the most related research that lies in the intersection of those two areas.

7.1 Distributed Data-Parallel Systems

MapReduce [2] has inspired a lot of follow-up research on large-scale distributed data-parallel computation, including Hadoop [1] and Dryad [22]. The MapReduce model has been extended [23] with Merge to support joins and adapted [24] to support pipelining. High-level languages for data-parallel computation have also been proposed for ease of programming. Examples include Sawzall [25], Pig Latin [11], [26], SCOPE [3], Hive [5], [10], and DryadLINQ [6]. In addition, FlumeJava [27] is a Java library for programming and managing MapReduce pipelines that proposes new parallel-collection abstractions, does deferred evaluation, and optimizes the data flow graph of an execution plan internally before executing. Nova [28] is a work-flow manager with support for stateful incremental processing which pushes continually arriving data through graphs of Pig programs executing on Hadoop clusters. Cascading [29] is a Java library built on top of Hadoop for defining and executing complex, scale-free, and fault tolerant data processing work-flows. Bu et al. [30] shows how recursive SQL queries may be translated into iterative Hadoop jobs. Programs in those systems go through a compilation and optimization process to generate code for a low-level execution engine, such as MapReduce and Dryad. All of them support user-defined functions that are treated as black boxes during optimization of the program's pipeline.

PeriSCOPE's optimizations work at the level of byte-code operators and pipeline descriptions, which are typically the result of the existing compilation and optimization process. Conceptually, the approaches taken by the PeriSCOPE's optimizations can be applied to data-parallel systems other than SCOPE, because almost all systems produce a pipeline with operators that call user-defined functions. The coverage and the effectiveness of the concrete optimizations, however, vary due to their different programming models and language runtime implementation. We show two cases where the differences in those systems matter. First, the data models differ, ranging from a relational data model (e.g., SCOPE) or its variations (e.g., Hive, Pig), to the object model (e.g., FlumeJava and DryadLINQ), which introduces different opportunities and difficulties for PeriSCOPE's optimizations. For example, with an object model, PeriSCOPE does not need to resolve the column access index any more, because all fields are accessed explicitly. Also, in an object model, declaring a new schema requires explicit class/object definitions. The resulting inconvenience often cause developers to reuse existing object definitions that contains unneeded fields, offering more opportunities for column reduction. Developers sometimes write custom (de-)serialization functions for an object to achieve better performance, which would pose challenges to PeriSCOPE's optimizations that cause schema changes: those functions must be modified accordingly.

Second, different systems might define different interfaces to their user-defined functions; those interfaces represent different trade offs between expressiveness and ease of analysis. For example, SCOPE exposes a collection of records to a mapper while others usually take a single record as the input to a mapper (e.g., in the *MapReduce* framework in Hadoop). Other examples include the reducer interface in SCOPE versus the UDAF (user-defined aggregation function) interface in Hive, where the former exposes the record collection and the latter only receives a single value, and is usually applied to a single column. The more restricted the interface and the less expressive the language, the easier it is to analyze. The interface definition also influences where the optimization opportunities lie. For example, if a user-defined function is defined to take a single column as its input, cross-column relationships are now explicitly expressed, reducing the need for program analysis and optimizations.

7.2 Program Analysis and Optimizations

The need to analyze user-defined functions, by means of techniques such as data flow analysis [15], abstract interpretation [31], and symbolic execution [32], has already been recognized. Ke et al. [33] focuses on data statistics and computational complexity of user-defined functions to cope with data skew. Agarwal et al. [34] concludes that certain data and code properties can improve performance of data-parallel jobs, and presents the RoPE system that adaptively re-optimizes jobs by collecting statistics on such code and data properties in a distributed context. Scooby [35] analyzes the data flow relationships of SCOPE's user-defined functions between input and output tables, such as column independence and column equality, by extending the Clousot analysis infrastructure [36]. Yu et al. [37] define the *associative-decomposable* property of a reducer function to enable partial aggregation automatically after analysis on the reducer functions. Sudo [38] identifies a set of interesting user-defined functions, such as pass-through, one-to-one, and monotonicity, and develops a framework to reason about data-partition properties, functional properties, and data shuffling in order to eliminate unnecessary data shuffling. Sudo analyzes user-defined functions to infer their properties, but never rewrites any user-defined functions.

Compilation of declarative language has huge impact on the efficiency of a high-performance and high-throughput environment. Steno [39] can translate code for declarative

LINQ [40], [41] queries both in serial C# programs and DryadLINQ programs to type-specialized, inlined, and loop-based procedural code that is as efficient as hand-optimized code. PeriSCOPE similarly applies those optimizations in program specialization as a preparation step, although differences in the language designs between SCOPE and LINQ lead to different challenges and approaches. Steno can automatically generate code for operators expressed in LINQ, but has to treat external functions called inside operators as black boxes. PeriSCOPE instead works with compiled user-defined functions, which include such external functions.

As a promising approach to improve precision of program analysis, there has been many related works about symbolic execution. Most of them target program testing or finding bugs, such as Pex [42] and KLEE [43], Christoph et al. [44] propose an idea to find bugs MapReduce style programs by leverage symbolic execution. It generates test cases by encoding MapReduce correctness conditions as symbolic program constraints by dynamically symbolic executing a MapReduce program. Only few works target optimizing large scale data parallel programs. HadoopToSQL [45] uses symbolic execution to derive preconditions and postconditions for the map and reduce functions in MapReduce and transform them to equivalent SQL queries to apply restrictions to input data set. The approach of generating preconditions and postconditions is like what PeriSCOPE does on early filtering.

8 Conclusion

Optimizing distributed data-parallel computation benefits from an inter-disciplinary approach that involves database systems, distributed systems, and program languages. In particular, PeriSCOPE has demonstrated performance gains on real production jobs by applying compiler optimizations and symbolic execution in the context of the pipelines that these jobs execute in. Much more can be done. We can explore how to enhance the reliability and predictability of PeriSCOPE's optimizations so programmers can reuse existing code as well as write straightforward code without much guilt that performance is being sacrificed. Going further, we can explore how the programming model itself can be enhanced with more guarantees about program behavior, allowing for even more aggressive optimizations that further improve performance.

Acknowledgments

The authors thank Rishan Chen and Chang Liu for their work on PeriSCOPE's algorithms of program analysis and optimization, and Qihan Li for creating the fancy teaser image for this paper. This paper was supported by China National Natural Science Foundation under Grant No. 61272408, 61322210, National High-tech Research and Development Program of China (863 Program) under Grant No. 2012AA010905, and Doctoral Fund of Ministry of Education of China under Grant No. 20130142110048.

References

[1] Apache. (2012, Mar.). Hadoop: Open-source implementation of MapReduce [Online]. Available: http://hadoop.apache.org

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Oper. Syst. Des. Implementation*, 2004, pp. 107–113.

[3] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endowment*, vol. 1, pp. 1265–1276, 2008.

[4] C. Olston, B. Reed, A. Silberstein, and U. Srivastava,, "Automatic optimization of parallel dataflow programs," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 267–273.

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - A petabyte scale data warehouse using Hadoop," in *Proc. 26th Int. Conf. Data Eng.*, 2010, pp. 996–1005.

[6] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. 8th Symp. Oper. Syst. Des. Implementation*, 2008, pp. 1–14.

[7] J. Zhou, P.-Å. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the SCOPE optimizer," in *Proc. 26th Int. Conf. Data Eng.*, 2010, pp. 1060–1071.

[8] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for MapReduce programs," *Proc. VLDB Endowment*, vol. 4, pp. 385–396, 2011.

[9] J. Zhou, N. Bruno, M. chuan Wu, P.-Å. Larson, R. Chaiken, and D. Shakib, "SCOPE: Parallel databases meet MapReduce," in *VLDB J.*, vol. 21, pp. 611–636, 2012.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - A warehousing solution over a MapReduce framework," *Proc. VLDB Endowment*, vol. 2, pp. 1626–1629, 2009.

[11] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a highlevel dataflow system on top of MapReduce: The Pig experience," *Proc. VLDB Endowment*, vol. 2, pp. 1414–1425, 2009.

[12] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.

[13] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th SIGACT-SIGPLAN Symp. Principles Program. Language*, 1983, pp. 177–189.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Languages Syst.*, vol. 13, pp. 451–490, 1991.

[15] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2001.

[16] Z3. (2012, Jun.). [Online]. Available: http://z3.codeplex.com

[17] Solving Non-Arithmetic Constraints [Online]. Available: https://github.com/AthrunArthur/se_constraints

[18] SharpDevelop. (2012, Jul.). ILSpy. [Online]. Available: http://wiki.sharpdevelop.net/ilspy.ashx

[19] Xamarin. Mono Cecil. (2012, Jun.). [Online]. Available: http://www.mono-project.com/Cecil

[20] Wolfram Mathematica. (2012, Jun.). [Online]. Available: http://www.wolfram.com/mathematica

[21] G. Graefe, "The cascades framework for query optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, Sep. 1995.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[23] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: Simplified relational data processing on large clusters," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2007, pp. 1029–1040.

[24] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, "MapReduce online," in *Proc. 7th Symp. Netw. Syst. Des. Implementation*, 2010, p. 20.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Sci. Program.*, vol. 13, pp. 277–298, 2005.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.

[27] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, efficient data-parallel pipelines," in *Proc. SIGPLAN Conf. Program. Language Des. Implementation*, 2010, pp. 363–375.

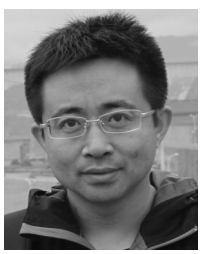
[28] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, "Nova: continuous Pig/Hadoop workflows," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2011, pp. 1081–1090.
[29] Cascading. (2012, Mar.). http://www.cascading.org/
[30] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, pp. 285–296, 2010.
[31] P. Cousot, "Abstract interpretation," *ACM Comput. Surv.*, vol. 28, pp. 238–252, Jun. 1996.
[32] R. H. Halstead, "Multilisp: A language for concurrent symbolic computation," *ACM Trans. Program. Languages Syst.*, vol. 7, pp. 501–538, 1985.
[33] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang, "Optimizing data partitioning for data-parallel computing," in *Proc. 13th Workshop Hot Topics Oper. Syst.*, 2011, p. 13.
[34] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing data parallel computing," in *Proc. 7th Symp. Netw. Syst. Des. Implementation* 2012, pp. 281–294.
[35] S. Xia, M. Fähndrich, and F. Logozzo, "Inferring dataflow properties of user defined table processors," in *Proc. 16th Int. Static Anal. Symp.*, 2009, pp. 19–35.
[36] F. Logozzo and M. Fähndrich, "On the relative completeness of bytecode analysis versus source code analysis," in *Proc. Joint Eur. Conf. Theory Practice Software 17th Int. Conf. Compiler Construction*, 2008, pp. 197–212.
[37] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *Proc. SIGOPS 22nd Symp. Oper. Syst. Principles*, 2009, pp. 247–260.
[38] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y.Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing data shuffling in data-parallel computation by understanding user-defined functions," in *Proc. 7th Symp. Netw. Syst. Des. Implementation* 2012, p. 22.
[39] D. G. Murray, M. Isard, and Y. Yu, "Steno: automatic optimization of declarative queries," in *Proc. SIGPLAN Conf. Programming Lang. Des. Implementation*, 2011, pp. 121–131.
[40] Microsoft. (2007, Feb.). LINQ [Online]. Available: http://msdn.microsoft.com/en-us/library/bb308959.aspx
[41] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling object, relations and XML in the .NET framework," in *Proc. SIGMOD Int. Conf. Manage. Data*, 2006, pp. 706–706.
[42] N. Tillmann and J. De Halleux, "Pex–white box test generation for net," in *Proc. 2nd Int. Conf. Tests Proofs*, 2008, pp. 134–153.
[43] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 209–224.
[44] C. Csallner, L. Fegaras, and C. Li, "New ideas track: Testing mapreduce-style programs," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 504–507.
[45] M.-Y. Iu and W. Zwaenepoel, "HadoopToSQL: A mapReduce query optimizer," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 251–264.

Xuepeng Fan received the BS degree from the Huazhong University of Science and Technology (HUST), China, in 2009, and where he is currently working toward the PhD degree in computer science. His research interests include performance issues and building parallel computing systems, including multicore system, and distributed system.

Zhenyu Guo received the BS degree from Zhejiang University in 2003 and the MS degree from Tsinghua University in 2006. He is a researcher in System Research Group, Microsoft Research Asia. His research interests include distributed systems and related tools, with an emphasis on robustness.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST) in 1994, where he is a Cheung Kung scholars chair professor of computer science and engineering. He is currently the dean of the School of Computer Science and Technology at HUST. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He was at the University of Hong Kong between 1998 and 2000, and was a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System.

Xiaofei Liao received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently a professor in the School of Computer Science and Engineering at HUST. He has served as a reviewer for many conferences and journal papers. His research interests include the areas of system software, P2P system, cluster computing, and streaming services. He is a member of the IEEE and the IEEE Computer society.



Jiaxing Zhang received the PhD degree in electronics from Peking University, China, in 2006. He is currently a researcher at Microsoft Research Asia. He worked in the areas of computational quantum mechanism, distributed key-value storage system, and distributed computing. His current interests include deep learning, compressive sensing, and algorithms for system.

Hucheng Zhou received the PhD degree from Tsinghua University at June, 2011. He joined System Research Group, Microsoft Research Aisa (MSRA). His research interests include design, development, analysis, optimization, diagnosis, and debugging of distributed system, especially of distributed data-parallel computing infrastructure. Before joining MSRA, his research area also included compiler development and optimization, as an active developer both in Open64, Pathscale, and GCC open-source community.

Sean McDirmid received the BS degree from the University of Washington, with a postdoc at EPFL working on Scala, and the PhD degree from the University of Utah (advisor Wilson Hsieh). He is a programming language researcher at System Research Group, Microsoft Research Asia, who focuses on design, objects, IDEs, live programming, and how to program with touch.

Wei Lin received the master's degree from the Institute of Computing Technology Chinese Academy of Sciences. After that, he joined System Research Group, Microsoft Research Asia. His main research interests include distributed system, semistructured storage system, replication, and debugging tools. Currently, he works on the infrastructure of Bing large-scale distribution system: SCOPE & COSMOS, query optimizer, structure stream and execution system.

Jingren Zhou received the PhD degree in computer science from Columbia University. He is a partner development manager at Microsoft Cloud and Enterprise Division. He and his team developed a cloud-scale distributed computation system, called SCOPE, targeted for massive data analysis over tens of thousands of machines at Microsoft. He was a researcher in the Database Group at Microsoft Research. His research interests include database, in particular large-scale distributed computing, query processing, query optimization, and architecture-conscious database systems. He has published many articles in premier database conferences and journals.

Lidong Zhou received the PhD degree in computer science from Cornell University, Ithaca, New York, in 2001. He is currently a principal researcher at Microsoft Research Asia; previously, he was a researcher at Microsoft Research Silicon Valley. His current research interests include distributed systems, storage systems, reliability, and security. He is on the editorial board for the *ACM Transactions on Storage* and served on the PC for conferences such as SOSP, OSDI, PODC, and DISC. He was the PC cochair for the 1st ACM Asia-Pacific Workshop on Systems (APSys) in 2010, and for the 7th Workshop on Large-Scale Distributed Systems and Middleware (LADIS) in 2013. He is a member of the IEEE.