Automating Distributed Partial Aggregation

Chang Liu[1], Jiaxing Zhang[2], Hucheng Zhou[2], Sean McDirmid[2],
Zhenyu Guo[2], Thomas Moscibroda[2]

[1]University of Maryland, College Park liuchang@cs.umd.edu
[2]Microsoft Research {jiaxz,huzho,smcdirm,zhenyug,moscitho}@microsoft.com

Abstract

Partial aggregation is of great importance in many distributed data-parallel systems. Most notably, it is commonly applied by MapReduce programs to optimize I/O by successively aggregating partially reduced results into a final result, as opposed to aggregating all input records at once. In spite of its importance, programmers currently enable partial aggregation by tediously encoding their reduce functionality into separate reduce and combine functions. This is error prone and often leads to missed optimization opportunities.

This paper proposes an algorithm that automatically *verifies* if the original monolithic reduce function of a MapReduce program is eligible for partial aggregation, and if so, *synthesizes* enabling partial aggregation code. The key insight behind this algorithm is a novel necessary and sufficient condition for when partial aggregation is applicable to a reduce function. This insight provides us with a formal foundation for an automaton, which derives a satisfiability problem that can be fed into a standard SMT solver. By doing so, we transform the problem of synthesis into a program inversion problem, which is however nondeterministic. Although such inversion is hard to solve in general, we observe that most reducers in practical distributed computing contexts can be classified into a few categories for which we can design efficient synthesis algorithms. Finally, we build and evaluate a prototype of our method to demonstrate its feasibility in the SCOPE distributed data-parallel system.

1. Introduction

MapReduce [11] is a programming model for the scalable processing of large data sets on machine clusters where grouping and aggregation are encoded as *map* and key-grouped *reduce* functions. MapReduce is the de-facto standard for web-scale distributed computing. Input records are first mapped to intermediate ⟨key, value⟩ pairs by mappers running on multiple machines. Intermediate results are then shuffled across the cluster so that all values for the same key are transmitted to the same machine to be processed "all-at-once" by a single reducer into that key's result.

In most real-world cases, network I/O due to shuffling dominates a program's execution's overall latency. *Partial aggregation* is a fundamental optimization that mitigates this problem. After intermediate results are grouped by key, an *initial-reduce* function is applied over them to get one *partial result* for each key. The system can then transmit partial results whose size is typically much smaller than the original intermediate results that would have been transmitted without partial aggregation. At multiple levels (computer, rack, cluster), partial results can be further combined using a *combine* function. Finally, reducers compute final results using a *final-reduce* function.

Most MapReduce-like systems allow programmers to leverage partial aggregation optimizations by providing or annotating the three functions: initial-reduce, combine, and final-reduce. For example, Hadoop programmers can provide a `Combiner` function, and SCOPE programmers can annotate *recursive* reducers to indicate that they can serve as combine functions [5]. At first glance, it appears that writing or annotating these functions is quite easy. However, doing this correctly and optimally in terms of efficiency is often hard even for expert programmers. Our own empirical study over real MapReduce-style SCOPE programs reveals that among 183 jobs that employ partial aggregation optimizations, 28 of them (15.3%) produce incorrect results. In some cases, partial aggregation is used wrongly, in others it cannot be applied at all.

Given these problems, there should ideally be an automatic tool that could verify the applicability of partial aggregation optimizations. Better yet, given normal reducers written by non-expert programmers, the tool could correctly and efficiently synthesize the initial-reduce, combine, and final-reduce functions. The work in this paper aims to provide such a tool.

Achieving this goal is challenging because there is no existing amenable theory for either verification or synthesis tasks for partial aggregation. A reduce function that supports partial aggregation is called *decomposable* in the distributed systems community. Yu et al. [33] defined the concept of function decomposability to characterize distributed programs that are eligible for partial aggregation. However, these definitions reveal only declarative constraints and cannot be directly used to check a function for decomposability or perform decomposition.

Based on an in-depth examination of real-world SCOPE programs, we identify common properties that can be used to simplify the problem of decomposability. First, all reduce functions have a loop that enumerates all records in a group using an *accumulator*. Second, the initial/final-reduce function of most decomposable reduce functions can be simply constructed by rearranging reduce function code, and so only the combine function needs to be synthesized. Finally, decomposability of a reduce function is determined by the algebraic properties of the combine function and accumulator.

Based on these properties, we solve the problem of verification and synthesis for partial aggregation in three steps:

- We provide a theoretical foundation for our solution by proving a necessary and sufficient condition for decomposability: Namely, a reduce function's decomposability is entirely determined by the accumulator's commutativity, while the combine function can be constructed by finding an inverse function of the accumulator.
- We use a program analysis-based technique to convert the decomposability verification problem into a program verification problem that can be solved using an off-the-shelf Satisfiability Modulo Theories (SMT) solver.
- Finally, we show that the combine function synthesis problem can be reduced into a general version of the program inversion synthesis problem, which unfortunately, is nondeterministic and hard to solve though its deterministic counterpart has been solved [27]. However, we observe that most target functions have inverse functions that conform to at least one of three non-trivial special cases. We can then design combiner synthesis algorithms for each of these practically relevant cases.

We implement our algorithms for SCOPE and evaluate it on real-world production jobs. The results show that our prototype can identify 78 additional reducers that can take advantage of partial aggregation but do not so originally. Our prototype succeeds in generating combiners for 90.9 % of these reducers, and performance experiments show that by enabling partial aggregation, there is a latency improvement of up to 55%, and a shuffling IO reduction of up to 99.98%.

We summarize the contribution of this work as follows.

1. We prove that the commutativity of the accumulator is necessary and sufficient for the decomposability of a reduce function. This is the first known easy-to-verify necessary and sufficient condition for decomposability.
2. We provide a program analysis-based method to verify the reduce function decomposability, as well as novel program synthesis methods to synthesize combine functions.
3. We implement a prototype of our techniques for SCOPE, and conduct empirical studies and evaluations on real-world jobs. The results indicate that our prototype is effective in identifying optimization opportunities and synthesizing combine functions. The optimized programs can dramatically reduce overall latency.

The rest of the paper is organized as follows. Section 2 provides background on MapReduce and partial aggregation. Section 3 formally studies the problem of automated partial aggregation while we develop in Sections 4 and 5 algorithms based on our formal conclusions to solve the verification and synthesis problems. Implementation and evaluation are discussed in Section 6. Section 7 presents related work and Section 8 concludes.

2. Partial Aggregation in Distributed Systems

2.1 Distributed Aggregation

In MapReduce as well as in more generalized distributed settings, the aggregation problem can be seen as one or more aggregate nodes that collect data from a cluster of data nodes. The aggregate nodes apply a `Reduce` function to the collected data in order to get the aggregated result; e.g. `SUM`, `COUNT` and `AVERGAGE`. We can view `Reduce` functions as having the following standard form:

```
   Reduce(Iterable<Value> values) {
1:   s = s0;
2:   foreach (x in values)
3:     s = F(s, x);
4:   emit(O(s));
   }
```

Here s is a set of *solution variables* that store a partial solution and are initialized to constant values represented by s_0 on Line 1 (*initializer*). The main foreach loop (Lines 2–3) enumerates input values and aggregates s according to function $F(s,x)$ (*accumulator*). Finally, Line 4 outputs the solution according to function O (*finalizer*). A `reduce` function is then represented by the triple $R = \langle s_0, F, O \rangle$.

2.2 Partial Aggregation

In most large-scale distributed data-parallel computations, the network I/O from data nodes to the aggregate nodes is the key bottleneck. *Partial aggregation* aims to reduce network I/O by decomposing the `reduce` function into three functions: `InitialReduce`, `Combine`, and `FinalReduce`. In each data node, `InitialReduce` partially aggregates the data into partial results. Partial results are transmitted to reducers where they are combined via `Combine`. Finally, `FinalReduce` computes the final result. Network I/O is reduced substantially because only partial results, which are usually smaller than lists of values, are transmitted. `InitialReduce` reuses the original `Reduce` function, while

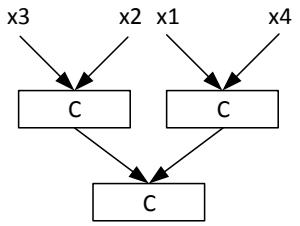

Figure 1. An example of a combining tree. Partial results come in order `x3, x2, x1, x4`. The aggregation combines them as `C(C(x3,x2),C(x1,x4))`.

```
public IEnumerable<Row>
  Reduce(RowSet input, Row outputRow) {
1.  int sum = 0;
2.  bool isFirst = true;
3.  foreach (Row row in input.Rows) {
4.    if (isFirst) {
5.      row[0].CopyTo(outputRow[0]);
6.      isFirst = false;
7.      sum = 10 + row[1].Integer;
8.    } else sum += row[1].Integer;
    }
9.  outputRow[1].Set(sum);
10. yield return outputRow;
}
```

Figure 2. An example reducer.

`FinalReduce` is exactly the finalizer. The only unknown function in a partial aggregation is therefore `Combine`.

To further optimize network I/O and temporary storage, these partial result can be combined hierarchically. However, in this case, multiple partial results from different mappers can arrive in an arbitrary order due to different data node speeds and uncertain network latency. Consequently, the aggregation has an arbitrary combining tree depending on the incoming order of partial results. Figure 1 illustrates an example of hierarchy combining and shows that a combiner should be both commutative and associative.

2.3 Outline of Method

We first provide an intuitive example of our method before delving into technical details. Notice that although we present all examples for SCOPE, the techniques discussed in this paper are applicable to other MapReduce-like distributed systems where partial aggregation is available.

Figure 2 is a SCOPE reducer. The records, `RowSet input`, are grouped by their key, `row[0]`. This reducer computes the summation, `sum`, of all values in each group, and emits the group key as well as the summation plus 10. Therefore, partial aggregation is applicable. To automate partial aggregation with our approach, the first step is to identify the accumulator, the main-loop body (Line 4-8). The partial results, i.e. *solution variables*, are composed by three variables, `isFirst`, `sum`, and `outputRow[0]`.

Based on next section's theoretical result, we must verify the commutativity of the accumulator (Line 4-8). To do so, we use PEX [28]–a white-box software testing tool that employs an SMT-solver Z3 [9]–to try and synthesize two `RowSet` objects x and y such that: i) $x[0] = y[0]$; and ii) applying the reducer over x,y and over y,x will lead to two results differing in at least one of `outputRow[0]`, `isFirst`, or `sum`. PEX reports an impossible result, thus the verifier concludes that partial aggregation is applicable.

For partial aggregation, the `InitialReduce` is easily rewritten from the `Reduce` function in Figure 2 as:

```
public Row InitialReduce(RowSet input,
                         Row outputRow) {
  int sum = 0;
  bool isFirst = true;
  foreach (Row row in input.Rows) {
    if (isFirst) {
      row[0].CopyTo(outputRow[0]);
      isFirst = false;
      sum = 10 + row[1].Integer;
    } else sum += row[1].Integer;
  }
  outputRow[1].Set(sum);
  outputRow[2].Set(isFirst);
  return outputRow;
}
```

Different from the original `reduce` function, all solution variables including `isFirst` are output. The combiner synthesizer then synthesizes for two groups of solution variables separately: (1) `sum` and `isFirst`, and (2) `isFirst` and `outputRow[0]`. Both groups fall into the *Single Input* category (Section 5.3). By employing the techniques from Section 5, the following combine function can be synthesized:

```
public Row Combine(Row x, Row y) {
  if (x[2].Boolean) {
    x[0] = y[0];  // for outputRow[0]
    x[1] = y[1];  // for sum
    x[2] = false; // for isFirst
  } else x[1] += y[1].Integer - 10;
  return x;
}
```

In this function, the partial result `y` (solution variables in `InitialReduce`) is combined into existing partial result `x`. These partial results have three columns corresponding to solution variables `outputRow[0]`, `sum`, and `isFalse` respectively. Finally, `FinalReduce` can simply extract the columns from combined result `x`:

```
public Row FinalReduce(Row x, Row output) {
  output[0] = x[0];
  output[1] = x[1];
  return output;
}
```

3. Decomposability as a Foundation of Partial Aggregation

This section first provides an intuitive reducer example to illustrate the eligibility for partial aggregation. We then discuss the *decomposability* of a reduce function, which is a formal requirement of partial aggregation. To achieve decomposability, it is important that a *combiner* exists, which can be used to combine partial aggregation results to compute the final result on behalf of the original reduce function. To guide and simplify the development of a tool, we next prove that a combiner exists if and only if the *accumulator* of the reduce function is *commutative*. Further, we show that the combiner can be constructed using any *general inverse function* of the accumulator.

3.1 Intuition

We have shown that the example reducer in Figure 2 that computes $10+\Sigma_{\texttt{row}\in\texttt{input}}\texttt{row}[1]$, can be rewritten to take advantage of partial aggregation. Such rewriting of simple programs is easy for a human programmer, but is non-trivial for a computer; or by both human and computer as the program becomes more complicated. We therefore study what properties such programs must have for partial aggregation to be legal. First, row order must not affect output, i.e. the computation should be commutative. Commutativity is crucial for partial aggregation—Xiao et al. [31] reported many bugs due to non-commutative reducers in Microsoft production programs that were running periodically (often daily) for at least three months. Our study in Section 6.2 also finds bugs due to non-commutativity.

Another common property needed to guarantee correctness is that the combiner must be *associative* (i.e. data can reside on different machines), and *commutative* (i.e. data can arrive in any order). The combiner must also produce the same result as the reducer would directly.

To facilitate synthesis of the three functions, we adopt the philosophy that `InitialReduce` and `FinalReduce` should be trivially derived from the original reduce function. In this way, we formally prove that the combiner always exists and can be uniquely determined when the original reduce function is *commutative*. We now define the problem and state the main theorem, providing a sketch of the proof.

3.2 Formalization

Consider an accumulator F that maps from a solution and input domain $S\times I$ to solution domain S. For input value sequences $I^\star$, we generalize the accumulator F to $F(s,\varepsilon)=s$ and $F(s,\langle x\rangle\oplus X)=F(F(s,x),X)$. Here ε is the empty sequence, $\oplus$ is the concatenation operator of two sequences, $x\in I$, $X\in I^\star$, and $\langle x\rangle$ is a list containing only one element x. The solution space S is defined by $S=\{s|s=F(s_0,X),X\in I^\star\}$. Hereby for all input sequences $X,Y\in I^\star$, then $F(F(s,X),Y)=F(s,X\oplus Y)$.

Partial aggregation decomposes the reduce function $R=\langle s_0;F;O\rangle$ into `InitialReduce`, `Combine`, and `FinalReduce`. It is eligible if and only if it yields the same result as R, in which case we say R is *decomposable*. Inspired by Yu et al. [33], we define decomposability as follows.

Definition 1 (Decomposability)**.** *An accumulator F in reduce function R with the initial solution s_0 is decomposable, if and only if there exists a function C such that the following four requirements are satisfied:*

1. *For any two input sequences $X_1,X_2\in I^\star$,*
$$F(s_0,X_1\oplus X_2)=C(F(s_0,X_1),F(s_0,X_2)). \quad (1)$$
2. *F is commutative: for any two input sequences $X_1,X_2\in I^\star$, $F(s_0,X_1\oplus X_2)=F(s_0,X_2\oplus X_1)$;*
3. *C is commutative: for any two solutions $s_1,s_2\in S$, i.e., $C(s_1,s_2)=C(s_2,s_1)$;*
4. *C is associative: for any three solutions $s_1,s_2,s_3\in S$, i.e., $C(C(s_1,s_2),s_3)=C(s_1,C(s_2,s_3))$.*

*We say that C is the **decomposed combiner** of F.*

Requirement 1 guarantees the combiner semantic of $C: S\times S\to S$ that returns the combined result of two partial aggregated results $F(s_0,X_1)$ and $F(s_0,X_2)$. Requirements 2, 3 and 4 ensure the same partial aggregation result even with arbitrary combining order.

However, there is no techniques to verify these properties over an infinite length of inputs. Clearly, the accumulator F and the initial value s_0 are much more interesting than the finalizer O. We focus on F and s_0 first and ignore O in the following discussion. Our theory can be easily extended to consider O.

Our main finding is that Requirement 2 implies all other requirements, i.e. the commutativity of accumulator F is the necessary and sufficient condition of decomposability.

Theorem 1 (Informally)**.** *Reducer R is decomposable if and only if the corresponding accumulator F is commutative. The decomposed combiner C is uniquely determined by F.*

3.3 An Overview of the Proof

In this subsection, we sketch the main idea to prove Theorem 1 by providing a list of lemmas. The full proof can be found in our online technical report [21]. Our first step is to show that Requirements 1 and 2 imply Requirements 3 and 4. In fact, we prove the following lemma.

Lemma 1. *Given a combiner C that satisfies Equation 1, C is commutative and associative if and only if accumulator F is commutative.*

We further simplify Requirement 2 to avoid arbitrary sequences: we consider two inputs values x and y, rather than two input sequence X and Y. As a result, commutativity can be verified by tools such as symbolic execution engines and constraint solvers as described in Section 4.

Lemma 2. *F is commutative if and only if for any solutions $s \in S$, and any two input values $x, y \in I$, $F(s,xy) = F(s,yx)$.*

We then show that the commutativity of an accumulator (i.e. Requirement 2) implies that there exists a combiner C that satisfies Requirement 1. The intuition is that, given two solutions $s_1, s_2 \in S$, where we know $s_1 = F(s_0, X)$ and $s_2 = F(s_0, Y)$, the combiner $C(s_1, s_2)$ can be computed as $F(s_0, X \oplus Y) = F(s_1, Y)$. We define the class of *general inverse functions*, $\mathscr{H}_F = \{H | \forall s \in S. F(s_0, H(s)) = s\}$.

Notice that $\mathscr{H}_F$ can contain more than one element. For instance, the following reducer

```
int F(int s, int x) {
    if(s == 0)
        return 2;
    else return s+x;
}
```

with an initial value s_0 of 0, has at least two general inverse functions H_1, H_2 where $H_1(2) = \langle 0 \rangle$, and $H_2(2) = \langle 1 \rangle$. That means, since F might not be a one-to-one function, it is almost impossible to pick out the exact Y that generates s_2. Given each possible $H \in \mathscr{H}_F$, a **derived combiner** C_H related to H is defined as $C_H(s_1, s_2) = F(s_1, H(s_2))$. Our next goal is to show that if F is commutative, then all derived combiners produce the same outputs. The observation is that when two input sequences generate the same aggregated output for a given initial solution value, they always generate the same output for any solution value.

Lemma 3. *Given an accumulator F that is commutative and two input sequences $X, Y \in I^\star$, if there is a s_0 such that $F(s_0, X) = F(s_0, Y)$, then $F(s, X) = F(s, Y)$ holds true for any $s \in S$.*

Proof. As this conclusion is less obvious, we present the proof here. For any value $s \in S$, we know there is an input sequence Z such that $F(s_0, Z) = s$. So we have

$$\begin{aligned} & F(s,X) = F(F(s_0,Z),X) = F(s_0, Z \oplus X) \\ = \; & F(s_0, X \oplus Z) = F(F(s_0,X) \oplus Z) = F(F(s_0,Y) \oplus Z) \\ = \; & F(s_0, Y \oplus Z) = F(s_0, Z \oplus Y) = F(F(s_0,Z),Y) = F(s,Y) \end{aligned}$$

□

Consequently, if F is commutative, for two possible input sequences (or values) Y_1 and Y_2 of s_2, the corresponding computed combined result based on Y_1 and Y_2 are the same; i.e. if $F(s_0, Y_1) = F(s_0, Y_2) = s_2$, then $C(s_1, s_2) = F(s_1, Y_1) = F(s_1, Y_2)$.

Lemma 4. *If an accumulator F is commutative, then for any $H, H' \in \mathscr{H}_F$, $C_H \equiv C_{H'}$.*

Based on the above lemmas, we know that for any commutative accumulator F, the combiner C can be generated using any general inverse function H of F, implying Requirement 1. In conclusion, we have

$$\begin{array}{rcl} F & ::= & f[x,...,x].F,...,F;s;\textbf{return } e,...,e \\ e & ::= & x \mid e\ op_a\ e \mid n \\ s & ::= & x := e \mid x,...,x := f(e,...,e) \mid s;s \\ & & \mid \textbf{if } (p) \textbf{ then } s \textbf{ else } s \mid \textbf{skip} \\ p & ::= & e\ op_r\ e \mid \textbf{true} \mid \textbf{false} \end{array}$$

Figure 3. Language syntax.

Theorem 2. *Reducer R is decomposable if and only if the corresponding accumulator F is commutative. The decomposed combiner C is uniquely determined by F, and takes the form $C(s_1, s_2) = F(s_1, H(s_2))$ where H is any inverse function of F.*

4. Verifying Decomposability

This section discusses how to verify the decomposability of a given accumulator F. By Theorem 2, it is sufficient if $F(s,xy) = F(s,yx)$ for all s, x, and y. However, this cannot be easily verified directly because F is a program. Prior studies [14, 26, 27] have shown that certain programs can be represented as formulae. A straightforward idea then is to transform the accumulator program into a formula that can be used to convert the condition $F(s,xy) = F(s,yx)$ into a formula. The satisfiability of this condition can then be checked by an SMT solver.

We first introduce the syntax of a programming language and show that $F(s,xy)$ and $F(s,yx)$ can be represented in this language. We then show how to use *path formula* to reduce the decomposability verification problem into an SMT satisfiability solving problem. We also show how to convert a program into a path formula. Different from prior work, our newly defined path formula can be easily converted back into a program, which is leveraged to solve our combiner synthesis problem in the next section.

4.1 Language

This work considers programs in the language whose syntax is defined in Figure 3. A program in this language is a function of the form:

$$F = f[x,...,x].F,...,F.s;\textbf{return } e,...,e$$

where f is the function name, $x,...,x$ are input variables, $F,...,F$ are nested function definitions, s is the body of the function, and $e,...,e$ are returned values. This language can encode assign, if, skip, sequence, and function call statements as in most imperative language. We exclude while-loop statements because most loops in real accumulators can be unrolled statically. A function call statement returns a list of expressions, and assigns their values to a list of variables respectively. An expression e is either a variable x, a constant n, or a binary operation between two expressions. We assume the variables only take integer or real values. The formal semantics of this language is defined and discussed in our online technical report [21].

$F_{acc} = acc[row_0, sum, isFirst, x_0, x_1].$
if $(isFirst = 1)$ **then** {
$row_0 := x_0;$
$isFirst := 0;$
$sum := 10 + x_1;$
} **else** {
$sum := sum + x_1;$
};
return $row_0, sum, isFirst$

Figure 4. The accumulator in Figure 2.

Note that our language is not Turing-complete; i.e. unbounded loops or recursion cannot be expressed. Regardless, we argue that most aggregation functions that are eligible for partial aggregation can be expressed in our language.

Given an accumulator F with name f, $F(s,xy)$ and $F(s,yx)$ can be represented as the following two programs:

$F_l = F(s,xy)$ $\quad f_l[s,x,y].F.s_1 := f(s,x); s_2 := f(s_1,y);$ **return** s_2
$F_r = F(s,yx)$ $\quad f_r[s,x,y].F.s_1 := f(s,y); s_2 := f(s_1,x);$ **return** s_2

Example 1. We rewrite the accumulator *acc* of our example reducer in Figure 2 into our language as shown in Figure 4. This function takes five inputs, including three solution variables row_0, *sum*, and *isFirst*, corresponding to `outputRow[0]`, `isFirst`, and `sum` in Figure 2, and two input variables x_0 and x_1 corresponding to `row[0]` and `row[1]` in Figure 2. Since our language supports only integer values and real number values, without loss of generality, we assume row_0, x_0, and *isFirst* are all integers. Further *isFirst* takes values of either 0 or 1, where $isFirst = 1$ corresponds to `isFirst==true` in Figure 2. The initial solution is $row_0 = 0$, $isFirst = 1$, and $sum = 0$.

4.2 Path formula and verifying decomposability

Given a function $F = f[x_1,...,x_n].F_1,...,F_l.s;$ **return** $e_1,...,e_m$, we are interested in a formula for F of the form:

$$\phi_F^{o_1,...,o_n} = \bigvee_{i\in I}\Big(\bigwedge_{j\in J} p_{ij} \wedge \bigwedge_{j=1}^{n} o_j = e_{ij}\Big) \tag{2}$$

where I and J are two index sets, p_{ij} is a predicate, e_{ij} is an expression, and p_{ij} and e_{ij} contains only variables in $\{x_1,...,x_n,o_1,...,o_m\}$. Here $x_1,...,x_n$ are input variables, and $o_1,...,o_m$ are output variables. Intuitively, a path formula ϕ is "correct" if an assignment to $\{x_1,...,x_n,o_1,...,o_m\}$ makes ϕ be true if and only if evaluating $F(x_1,...,x_n)$ returns $o_1,...,o_m$. Then the satisfiability of ϕ exactly implies that the function will compute the same output.

Existing work [27] has shown that symbolic execution can convert programs of various imperative languages into formulae. For our language, we also leverage symbolic execution to convert a program into a path formula. The formal definition of a path formula and the details about the conversion can be found in [21].

We put two further restrictions on a valid path formula: (1) $\bigvee_{i\in I}(\bigwedge_{j\in J} p_{ij})$ is a tautology; and (2) $\forall i_1 \neq i_2 \in I. \bigwedge_{j\in J} p_{i_1 j} \wedge \bigwedge_{j\in J} p_{i_2 j}$. Intuitively, I is the set of indices of all possible paths. For each $i \in I$, $\bigwedge_{j\in J} p_{ij}$ is the pre-condition of the path, i.e. when it is true, the program will compute the output as $o_j = e_{ij}$ $(j = 1,...,n)$. These two restrictions actually guarantee that no two paths will be taken simultaneously, and all possible paths will be covered by the path formula, providing us with a convenient way to convert a path formula back into a program:

if $(p_{11}$ **and** ... **and** $p_{1|J|})$ **then return** $e_{11},...e_{1n}$
else if $(p_{21}$ **and** ... **and** $p_{2|J|})$ **then return** $e_{21},...e_{2n}$
else ...

Based on the above discussion, the decomposability verification problem can be reduced to an SMT satisfiability problem via the following theorem.

Theorem 3. *$\forall sxy.F(s,xy) = F(s,yx)$ if and only if $\phi_{F_l}^{o_1,...,o_n} \wedge \phi_{F_r}^{o'_1,...,o'_n} \wedge (\bigvee_{i=1}^{n} o_i \neq o'_i)$ is not satisfiable.*

The proof can be found in our online technical report [21].

Concretely, we convert our example accumulator F_{acc} in Figure 4 into the following path formula:

$$\phi_{F_{acc}}^{o_1,o_2,o_3} = (isFirst = 1 \wedge o_1 = x_0 \wedge o_2 = 10 + x_1 \wedge o_3 = 0)$$
$$\bigvee (isFirst \neq 1 \wedge o_1 = row_0 \wedge o_2 = sum + x_1 \wedge o_3 = isFirst)$$

5. Combiner Synthesis Algorithms

As discussed in Section 3, given a decomposable accumulator F, the combiner C can be constructed as $C(s_1,s_2) = F(s_1,H(s_2))$, where H is any general inverse function of F. Unfortunately, constructing H is a nondeterministic program inversion problem whose input cannot be uniquely determined by its output. As far as we know, no existing work considers this problem.

Solving the general nondeterministic program inversion problem is probably too difficult. However, we observe that most accumulators of a decomposable reducer can be classified into three categories according to how they aggregate:

1. *Counting* aggregation over an input sequence that is only determined by the length of the sequence;
2. *State machine* aggregation that essentially simulates a state machine with a limited number of states; and
3. *Single input* aggregation over an input sequence that can be simulated by aggregating over one input record.

For each of the first two categories, we develop criteria that can determine if a decomposable accumulator belongs to it, and then develop an algorithm to synthesize a combiner. For the third category, we also provide a criteria, but show that the general problem to synthesize a general inverse function

is hard. Instead of solving it completely, we provide two heuristics that work well for real-world accumulators.

5.1 Counting Aggregation

The prototypical accumulator belonging to this category is the `COUNT` function built into most DBMSs. We formally define the criterion of this category as follows:

Definition 2 (Counting category)**.** *An accumulator F belongs to the* counting category *if and only if, $F(s_0,X) = F(s_0,Y)$ holds for any two input sequences $X,Y \in I^\star$, $|X| = |Y|$.*

This criterion can be transformed into an amenable form via the following lemma:

Lemma 5. *An commutative accumulator F belongs to the counting category if and only if, for any two input records $x,y \in I$, $F(s_0,x) = F(s_0,y)$ holds true.*

With this lemma, we can use an SMT solver as described earlier to verify that $\forall x,y.F(s_0,x) = F(s_0,y)$. For the combiner synthesis problem, the combiner can be computed using the following pseudo-code:

```
input(s1, s2);
s = s0; r = s1;
while (s<>s2) {
  s = F(s, 0);
  r = F(r, 0);
}
return r;
```

Here we use `s0`, `s1`, and `s2` to represent the initial solution s_0, and the two input solutions s_1 and s_2 respectively. We use s and r to represent two local variables. As an argument for correctness, suppose that the loop body is executed n times and X is an input sequence of n zeros. We notice that when the loop stops, `r` stores the value of $F(s_1,X)$, and $F(s_0,X) = s_2$. We can then build a general inverse function H of F such that $H(s_2) = X$, and then $r = C(s_1,s_2) = F(s_1,H(s_2))$.

Notice that although we restrict the input functions of our synthesizing algorithm to be in our language, the output combiners can leverage any language features, such as loops, that are supported by MapReduce-style systems. It is clear that verifying if an accumulator belongs to the counting category calls the SMT solver only once The combiner synthesis algorithm is constant time.

As an optimization, if we know that the accumulator is exactly the `COUNT` program, i.e. $\forall sx.F(s,x) = s+1$, the combiner can be synthesized as $C(s_1,s_2) = s_1 + s_2 - s_0$.

5.2 State Machine Aggregation

Every accumulator can be treated as a transition function of a state machine. If this state machine has a small size, then it is easy to synthesize a combiner as we will show. Given an explicit finite state machine, a general inverse function is equivalent to finding a sequence of inputs that will transform the state machine to a given state. However, the finite state machine defined by an accumulator is implicit and so (1) we do not know the size of the state machine; and (2) we do not know its transition table.

Algorithm 1 Detecting finite state machine and generating its combiner function.

```
1:  Q.clear(); Q.add(s0); Sol.add(s0, null, null);
2:  while not Q.isEmpty() do
3:    ps = Q.poll();
4:    while (ns, x) = find_new(ps, Sol) do
5:      if Sol.size() ≥ T then
6:        return (false, null);
7:      end if
8:      Sol.add(ns, ps, x);
9:      Q.add(ns);
10:   end while
11: end while
12: return (true, generate_combiner(Sol));
```

To tackle these problems, we design a breadth first search algorithm (Algorithm 1) to identify all states by using a set data structure *Sol* to store all explored states as a set of triples (s, ps, x). Intuitively, (s,ps,x) encodes that ps can be transformed into s by taking an input x and so $s = F(ps,x)$. If we identify that F is a state machine containing no more than a threshold of T states, the algorithm succeeds along with the code for the combiner; otherwise, if the algorithm has found more than T states, the algorithm will immediately fail. The key parts of this algorithm are the *find_new* call (Line 4) that finds a new state, and the *generate_combiner* call at the end that generates the code for the combiner.

Function *find_new* finds a new state ns starting from a state ps by taking x as input so:

$$ns = F(ps,x) \wedge \bigwedge_{(a,b,c)\in Sol} a \neq ns$$

We transform this into the following query that can be submitted to a SMT solver:

$$\bigvee_{i\in I}(\Phi_i(ps,x) \wedge ns = \phi_i(ps,x) \wedge \bigwedge_{(a,b,c)\in Sol} a \neq ns)$$

Find_new returns (ns,x) as the answer if the SMT solver returns a solution; otherwise, it returns an empty answer.

When all possible states have been explored, they are all stored in *Sol*. *Sol* contains all information of the transition table of the finite machine so computing H can be done by a table lookup. To synthesize the combiner, since there are at most T different states, we can materialize all these up to T^2 combinations of (s_1,s_2) and the results of $C(s_1,s_2)$ can be computed in constant time. The complexity to decide if an accumulator belongs to the state machine category requires calling an SMT solver at most T times, and the combiner synthesizer runs in $O(T^2)$ time and space.

Example 2. Let us consider the following code as the accumulator, and $s_0 = -1$:

$F_{sm} = sm[s,x]$.
  if $(s = -1)$ **then**
    if $(x > 100)$ **then** $s := 0$; **else** $s := 2$;
  else if $(s = 0)$ **then**
    if $(x > 100)$ **then** $s := 0$; **else** $s := 1$;
  else if $(s = 2)$ **then**
    if $(x > 100)$ **then** $s := 1$; **else** $s := 2$;
  return s

By running our algorithm, a *Sol* is calculated that includes three triples: $(0,-1,101)$, $(2,-1,100)$, and $(1,0,100)$. The synthesized combiner looks as follows:

```
input(s1, s2)
if (s1 == -1 and s2 == -1) then return -1;
...
else if (s1 == 2 and s2 == 0) then return 1;
...
```

5.3 Single Input Aggregation

Most standard accumulators such as `SUM` and `MAX` belong to this category, which is formally defined as follows:

Definition 3 (Single input category)**.** *An accumulator F belongs to the* single input category *if and only if, for any input sequence $X \in I^\star$ and $|X| > 1$, there is an input record $x \in I$ such that $F(s_0,X) = F(s_0,x)$.*

The following lemma can be used to produce a more amenable criterion:

Lemma 6. *An accumulator F belongs to single input category if and only if, for any two input records $x,y \in I$, there exists an input record z such that $F(s_0,xy) = F(s_0,z)$.*

We then need to verify that $\forall x,y.\exists z.F(s_0,xy) = F(s_0,z)$. However, the techniques that we have discussed so far cannot eliminate the quantifier for z. Most existing methods used by SMT solvers to handle quantifiers rely on the E-match algorithm [8], which cannot deal very well with our queries. In fact, our use of Z3 [9] results in a timeout to check if `MAX` belongs to the single input category.

We must eliminate the quantifier on z. First, we write the condition $\forall x,y.\exists z.F(s_0,xy) = F(s_0,z)$ in the following form:

$$\forall xy. \bigvee_{(i,i')\in I\times I'} \exists z. \bigwedge_{(j,j')\in J\times J'} p_{ij} \wedge p'_{i'j'} \wedge e_{ij} = e'_{i'j'},$$

where p_{ij} and e_{ij} are from the path formula of $F(s_0,xy)$, and $p'_{i'j'}$ and $e'_{i'j'}$ are from the path formula of $F(s_0,z)$. It is sufficient to show that for all x and y, the following formula is true for one assignment to i and i':

$$\exists z. \bigwedge_{(j,j')\in J\times J'} p_{ij} \wedge p'_{i'j'} \wedge e_{ij} = e'_{i'j'} \tag{3}$$

We develop two heuristics to eliminate the quantifier in formula (3). Intuitively, if the formula has the form of $\exists z.z = e \wedge \phi(z)$, where e does not contain z, then the quantifier $\exists z.$ along with the clause $z = e$ can be eliminated together, and the formula is equivalent to $\phi(e)$. The first heuristic then works to translate one clause in the formula into the equivalent form of $z = e$. For the second heuristic, if the formula has the form of $\exists z.\phi(z) \wedge \phi'$, where $\phi(z)$ contains only z, and ϕ' does not contain z, then the quantifier along with $\phi(z)$ can be eliminated if $\phi(z)$ is satisfiable. On the other hand, if $\phi(z)$ is not satisfiable, then neither is this formula.

Notice that we do not guarantee these two heuristics will successfully eliminate the quantifier: we can only deal with programs where these two heuristics eliminate all existence quantifiers successfully.

Example 3. We consider the accumulator F_{acc} in Figure 4. The (simplified) condition to verify is

$$\begin{aligned}&\forall x_0,x_1,y_0,y_1.\\ &(\exists z_0,z_1.isFirst = 1 \wedge z_0 = x_0\\ &\quad \wedge 10 + z_1 = 10 + x_1 + y_1 \wedge 0 = 0)\\ &\bigvee(\exists z_0,z_1.isFirst \neq 1 \wedge z_0 = row_0\\ &\quad \wedge sum + z_1 = sum + x_1 + x_2 \wedge isFirst = isFirst)\end{aligned}$$

The initial solution is $isFirst = 1$, $row_0 = 0$, and $sum = 0$. Then the formula

$$\begin{aligned}&(\exists z_0,z_1.isFirst \neq 1 \wedge z_0 = row_0\\ &\quad \wedge sum + z_1 = sum + x_1 + x_2 \wedge isFirst = isFirst)\end{aligned}$$

is vacuously false, since $isFirst = 1$ is true. The above formula can be revised to

$$\begin{aligned}&\forall x_0,x_1,y_0,y_1.\\ &\exists z_0,z_1.z_0 = x_0 \wedge 10 + z_1 = 10 + x_1 + y_1 \wedge 0 = 0\end{aligned}$$

We apply the heuristic to translate the clause, $10 + z_1 = 10 + x_1 + y_1$, into $z_1 = 10 + x_1 + y_1 - 10$. Then the above formula is further rewritten into

$$\begin{aligned}&\forall x_0,x_1,y_0,y_1.\\ &\exists z_0,z_1.z_0 = x_0 \wedge z_1 = 10 + x_1 + y_1 - 10 \wedge 0 = 0\end{aligned}$$

Since each of z_0 and z_1 appears only once in one clause in which the two variables themselves are on the left hand side, we can drop the clauses along with the existential quantification in front of them. Therefore, the formula is translated into

$$\forall x_0,x_1,y_0,y_1.0 = 0$$

which is satisfiable.

It is appealing to develop an algorithm synthesizing the general inverse function for those accumulators falling into this category, since it covers a majority of accumulators. In fact, many one-way functions (e.g. the discrete exponential function) fall into this category, but their inverse functions (e.g. the discrete logarithmic function) are commonly believed hard to compute, and thus automatically synthesizing such (polynomial time) functions are probably impossible.

We notice that the above mentioned heuristics also work for synthesizing the general inverse function. We first convert F into its path formula. Then we employ the above mentioned heuristics to this formula as follows: in each sub-formula Φ_i, which is a conjunction of clauses, (1) if we can transform it into $x = e \wedge \phi(x)$, where e does not contain x, then we will convert it into $x = e \wedge \phi(e)$; and (2) if we can transform it into $\phi_1(x) \wedge \phi_2$, where ϕ_1 contains only x and ϕ_2 does not contain x, then we check the satisfiability of $\phi_1(x)$. If it is satisfiable, and $\mathtt{x}_0$ is a model, then we convert that formula into $x = \mathtt{x}_0 \wedge \phi_2$. Otherwise, if $\phi_1(x)$ is not satisfiable, then we remove Φ_i from $\phi_F^{o_1,\ldots,o_n}$. If all sub-formulas are either converted into the form of $x = e_i \wedge \phi_i$ (where ϕ contains only $\{o_1,\ldots,o_n\}$) or are removed. Then this function is a path formula with respect to input variables $o_1,\ldots,o_n$, and thus we can convert it back into a program. In our evaluation, such a heuristic approach works well in many cases, and produces efficient code (Section 6.2).

Example 4. We consider the accumulator F_{acc} in Figure 4. The path formula is

$$\phi_{F_{acc}}^{o_1,o_2,o_3} = (isFirst = 1 \wedge o_1 = x_0 \wedge o_2 = 10 + x_1 \wedge o_3 = 0) \\ \bigvee (isFirst \neq 1 \wedge o_1 = row_0 \wedge o_2 = sum + x_1 \wedge o_3 = isFirst)$$

Since $isFirst = 1$, then the second conjunction of the clauses is vacuously false. We apply the heuristic to translate it into

$$\phi_{F_{acc}}^{o_1,o_2,o_3} = (isFirst = 1 \wedge x_0 = o_1 \wedge x_1 = o_2 - 10 \wedge o_3 = 0)$$

In this case, the combiner can be synthesized as

$C = comb[row_0, sum, isFirst, row_0', sum', isFirst'].F_{acc}.$
  if $(isFirst' = 0)$ **then** $\{$
    $x_0 := row_0'$;
    $x_1 := sum' - 10$;
    $row_0, sum, isFirst := F_{acc}(row_0, sum, isFirst, x_0, x_1)$
  $\}$.
  return $row_0, sum, isFirst$;

As to its complexity, we notice that only the second heuristic rule involves a call to the SMT solver so there will be at most $|I| \times |I'| \times m$ calls, where m is the number of input variables. As analogous rules are applied, the combiner synthesizer makes the same number of calls to the SMT solver.

6. Implementation And Evaluation

6.1 Implementation

We prototyped both decomposability verification and combiner synthesis for production SCOPE jobs in Microsoft, where map and reduce functions are written in C#. Our implementation includes 2,326 lines of C# code that use Z3 [9] as the SMT solver. We discuss several of the implementation challenges in the rest of this section.

Solution Variable Identification. Simply marking all variables in the initializer as solution variables can incorrectly include local variables. Instead, solution variables are marked only (1) in the statement `emit(O(s));` to generate output; and (2) in the statement `s = F(s, x);` to update the partial solution. The first condition is checked by examining a reducer's finalizer. The second condition identifies variables in the loop that depend recursively on themselves; i.e. they are *loop-carried dependencies* that can be identified by standard static program analysis techniques [12].

Independent Solution Variables. Some decomposable accumulators with multiple solution variables do not belong to any of the three categories that have only one solution variable. For instance, accumulator AVERAGE includes `sum` and `count`. However, these solution variables can be calculated independently so for each solution variable s, we only consider those variables that s depends on; e.g. we consider:

$F = f[isFirst, s, c, x].$
  if $(isFirst == 1)$ **then** $\{$
    $isFirst := 0;\ s := 10 + x;\ c := 2;$
  $\}$ **else** $\{$
    $s := s + x;\ c := c + 1;$
  $\}$;
  return $isFirst, s, c$;

We first synthesize three combiners for f, s, and c, using algorithms for counting, state machine, and single input categories, which are then combined together.

Arrays and Loops. Arrays and loops with constant lengths can be handled easily: a loop can be unrolled; and an array with length n corresponds to n new variables where $A[i]$ can be converted into the following if-statement:

```
if      (i==0) then return x0;
else if (i==1) then return x1;
...
```

Many seemingly unfixed lengths are often specified by arguments in MapReduce (SCOPE) programs, and so can be made constant via constant propagation. Beyond this, arrays and loops of unknown length cannot be handled.

Approximating Decomposability Criteria. Notice that checking the condition $F(s,xy) = F(s,yx)$ for all $s \in S, x, y \in I$ in Theorem 2 requires the verifier to consider all elements in S. When S cannot be efficiently computed, it is impractical to verify the exact condition. Luckily, it is usually possible to sacrifice a degree of accuracy to consider S to be either $\{s_0\}$, or whole elements of s's type; e.g. if s is a 32-bit integer, then treat S to be the set of all 32-bit integers. The former can report a non-commutative accumulator to be commutative (false-positive), while the later can report a commutative accumulator to be non-commutative (false negative). Our empirical study finds that the first approximation identifies no false-positives, while the later rejects only 19 out of 261 valid partial aggregation jobs. Neither of these two approximations involves computing S to decide decomposability, and so both are tractable in real applications. What approximation to use depends on the application

scenario, and systems can provide both verification results to programmers.

6.2 Evaluation

We conducted an empirical study on a trace of 4,429 SCOPE jobs from production clusters in Microsoft. 183 of these jobs already employ partial aggregation via manually provided combiners. There are 497 unique reducers in the trace.

Problematic Jobs. We developed automatic tools to verify commutativity of the 183 jobs employing partial aggregation, and found 28 of them (15.3 %) to be suspicious. We manually investigated these jobs, contacting their authors to confirm that they indeed had bugs in them. An example suspicious reducer is the following:

```
String key1 = "", key2 = "";
int sum = 0;
foreach (Row row in input) {
    key1 = row[0].String;
    key2 = row[1].String;
    sum += row[2].Integer;
}
output[0] = key1;
output[1] = key2;
output[2] = sum;
yield return output;
```

Here, `row[0]` and `row[1]` are assumed to be the same for the same group of input, which, however, is not ensured by the computation preceding this reducer. By constructing two rows of input that have different values stored in `row[1]`, we easily verify that commutativity, and thus decomposability, is not satisfied. Note that [31] reported similar bugs as here.

Decomposability Verification. We investigated the remaining 4,246 jobs where partial aggregation had not been manually applied. We used PEX [28] to automatically verify these jobs using both criteria $\forall x,y.F(s_0,xy) = F(s_0,yx)$ and $\forall s,x,y,F(s,xy) = F(s,yx)$ to verify decomposability (Section 6.1). There are 261 jobs (containing 22 unique reducers) satisfying $\forall x,y.F(s_0,xy) = F(s_0,yx)$. The results were manually double-checked to ensure that all were true positives.

261 over 4,246 (6%) seems to be a small fraction. Our verification is very restrictive and does not consider reducer pre- and post-condition in the context of entire jobs. The study from [31] shows that some implicit properties are assumed by the programmers but not hinted in the program. These properties could guarantee that the reducers are essentially commutative. This phenomenon inspires us to either allow programmers to annotate or develop automatic tools to discover these implicit properties. Although this is beyond the scope of this paper, we believe it to be a good future direction. Further, our prototype simply checks concrete commutativity. For example, the insertion operations over a set are semantically commutative, but cannot pass our tool's verification. Semantic commutative verification techniques (e.g. [20]) can alleviate this problem.

Table 6.2 reports the running time of our prototype over the 22 commutative reducers of the identified jobs, which is conducted on a PC with an 4-core 3.0 GHz Intel Core i7-870 and 8 GB memory. All execution times are under one second except for reducers 7 and 8, which issue far more SMT queries than other reducers. Reducer 7 contains 44 state variables; and reducer 8 is a four state finite state machine. Three reducers (20, 21, and 22) cannot pass $\forall s,x,y.F(s,xy) = F(s,yx)$.

Combiner Synthesis. We ran our synthesizer prototype over these 22 reducers on the same machine and succeed in synthesizing combiners for 20 out of 22 (91.0%) reducers. Reducers 21 and 22 exhibit patterns outside of our three categories and thus fail in combiner synthesis. Both of these two jobs exhibit the same pattern illustrated as follows:

```
foreach (Row current in input.Rows) {
    if (num == 0) {
        sum = current[1]
    } else {
        sum += current[1];
    }
    num++;
}
```

This example contains two `num` and `sum` state variables. While `num` computes a typical `COUNT` aggregation, `sum`, which depends on `num`, belongs to the single input category. To synthesize the combiner for `sum`, however, the synthesizer must also synthesize `num`'s computation, which belongs to a different category and our prototype cannot deal with.

Performance Gain. We finally evaluate the performance gain of our partial aggregation optimization. To avoid interference with production job execution, we randomly picked one job (contains Reducer 2) with tens of TBs input data for evaluation (some of the jobs are out-of-date or their input data is missing). The performance gains are substantial: overall latency is reduced from 165 seconds to 64 (61.6% reduction), and the data volume transmitted during shuffling decreases from 7.99 GB to 1.22 MB (99.98%). Three manually written jobs with real production data are further evaluated, which calculate SUM, COUNT, and MAX, respectively. In these cases, we also observe an average of reduction of 62.4% in latency and 76.0% in network I/O.

7. Related Work

Partial aggregation is closely related to data aggregation techniques used in functional and declarative parallel programming languages [6, 29]. Incoop [4] makes use of Hadoop's *Combiner* to memorize the result of partial aggregation and avoid re-computation.

There are different interfaces for programmers to specify the decomposition for partial aggregation in different systems [33]. Distributed databases typically adopt accumulator-based user-defined aggregators [24] where programmers must supply four methods *Initialize*, *Iterate*,

#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$
1	0.08	0.04	7	0.31	0.29	13	0.17	0.07	19	0.17	0.17
2	0.11	0.04	8	2.54	0.31	14	0.04	0.04	20	0.16*	0.05
3	0.18	0.06	9	0.06	0.03	15	0.05	0.02	21	0.70*	0.07
4	0.20	0.10	10	0.05	0.02	16	0.05	0.02	22	0.22*	0.04
5	0.09	0.04	11	0.02	0.02	17	0.17	0.17			
6	0.10	0.08	12	0.08	0.04	18	0.17	0.17			

Table 1. Performance of our prototype's decomposability verification. The $\forall sxy$ and $\forall xy$ columns show running time to verify the decomposability using $\forall sxy.F(s,yx) = F(s,xy)$ and $\forall xy.F(s_0,xy) = F(s_0,yx)$ respectively. All times are reported in seconds with stars meaning that verification failed.

#	Type	Time	#	Type	Time	#	Type	Time	#	Type	Time
1	C	0.03	7	SI	1.10	13	C+SI	0.15	19	SI	0.09
2	C	0.04	8	SM	0.77	14	SI	0.14	20	SI	0.07
3	C	0.06	9	C	0.02	15	C	0.02	21		0.14*
4	C+SI	0.31	10	C	0.02	16	C	0.02	22		0.08*
5	SI	0.08	11	SI	0.05	17	SI	0.09			
6	C+SI	0.09	12	C	0.03	18	SI	0.09			

Table 2. Performance of our prototype's combiner synthesis. The type column shows which kinds of techniques are used to synthesize the combiner; the time column shows the running time in seconds, including both the times to check technique validity and to generate combiner code; and the stars after Reducer 21 and Reducer 22 mean that combiner synthesis failed.

Merge and *Final* [25]. Similarly, Pig Latin [13], a Hadoop layer, requires *InitialReduce*, *Combine* and *FinalReduce*; while in DryadLINQ [32, 33], programmers annotate the *Decomposable* function and define three functions *Initialize*, *Iterate* and *Merge* for partial aggregation.

There is also limited support for automatic partial aggregation for a bounded number of built-in aggregators in parallel databases and data-parallel computing systems. SCOPE [5] and Pig Latin automate partial aggregation for some built-in algebraic aggregators [15] such as *COUNT*, *SUM*, *MAX*, and *AVERAGE*. DryadLINQ [33] also automates combiner generation for aggregation expressions composed by associative-decomposable sub-expressions. However, they cannot handle user-defined aggregation functions written in imperative languages such as C# or Java.

Kim and Rinard [20] showed how to semantically detect operation commutativity and how to inverse operations. We can benefit from this work to semantically extend decomposability checking. For inverse operations, [20] focused on how to restore to a previous state, while we focus on synthesizing operations given initial and final states.

Saurabh et al. [26] proposed a template-base method to interpret program synthesis as generalized program verification, which brings verification tools to inverse function synthesis. Based on this idea, PINS [27] proposed an approach to deterministic program inversion. Instead of attempting to reason about the program under all environment conditions, it reasons about a small set of carefully chosen paths, which is much simpler. However, those approaches do not fit our non-deterministic program inversion problem; instead, we leverage the high coverage of three accumulator categories and develop specific solutions. Advances in SMT solvers [10] helps both decomposability verification and inverse function synthesis. Many solvers, including Z3 [9, 34], CVC3 [3], and OpenSMT [23], have built-in theories like arithmetic, bitverctors, arrays as well as quantifiers.

There is a line of research for distributed computing scenarios that studies static analysis techniques for user defined functions (UDFs), including data-flow analysis [1, 2, 22], abstract interpretation [7], and symbolic execution [17, 18]. Ke *et al.* [19] deals with data skew using data statistics and computational complexity of UDFs. Scooby [30] studies the dataflow relations of SCOPE UDFs between input and output tables. To optimize I/O, Sudo [35] identifies useful functional properties of UDFs, reasoning about data-partition properties to eliminate unnecessary data shuffling. PeriSCOPE [16] also aims to reduce I/O, but it focuses on automatic global optimizations enabled by observing the full pipeline of the computation. In comparison, our work studies how to automatically enable partial aggregation specifically in distributed computing scenarios.

8. Conclusion

This work considers the problem of partial aggregation that is central to distributed data-parallel computations. We model distributed partial aggregation as a reducer decomposability problem, and formulate a novel necessary and sufficient condition for decomposability. These conditions provide the theoretical foundation for automating partial aggregation, which until now required tedious manual effort. We have implemented a tool for automatic decomposability

verification and partial aggregation synthesis, and evaluate it using production SCOPE jobs to demonstrate feasibility.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN 1-55860-286-0.

[3] C. Barrett and C. Tinell. CVC3. In *CAV*, 2007.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. SOCC, 2011.

[5] 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. *PVLDB*, 2008.

[6] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-53086-4.

[7] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28, June 1996.

[8] L. de Moura and N. Bjøner. Efficient e-matching for smt solvers. In *CADE-21*, volume 4603, pages 183–198. 2007.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[10] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, pages 319–349, 1987.

[13] 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. *PVLDB*, 2009.

[14] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, pages 29–53, 1997.

[16] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. OSDI, 2012.

[17] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In *RV*, 2009.

[18] R. H. H. Jr. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 1985.

[19] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.

[20] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. 2011.

[21] C. Liu, J. Zhang, H. Zhou, S. McDirmid, Z. Guo, and T. Moscibroda. Automating distributed partial aggregation. In *Technical Report*. URL `http://www.cs.umd.edu/~liuchang/paper/pa-socc2014-tr.pdf`.

[22] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

[23] OpenSMT. `http://code.google.com/p/opensmt/`.

[24] L. A. Rowe and M. Stonebraker. The postgres data model. VLDB, pages 83–96, San Francisco, CA, USA, 1987.

[25] J. Russell. *Oracle9i Application Developer's Guide-Fundamentals*. Oracle Corporation, 2002.

[26] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[27] S. Srivastava, S. Gulwani, J. S. Foster, and S. Chaudhuri. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.

[28] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *TAP*, pages 134–153, 2008.

[29] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed haskells. *J. Funct. Program.*, 12(4&5):469–510, 2002.

[30] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *SAS*, 2009.

[31] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou. Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *ICSE Companion*, 2014.

[32] 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 *OSDI*, 2008.

[33] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 2009.

[34] Z3. `http://research.microsoft.com/en-us/um/redmond/projects/z3/documentation.html`.

[35] 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 *NSDI*, 2012.