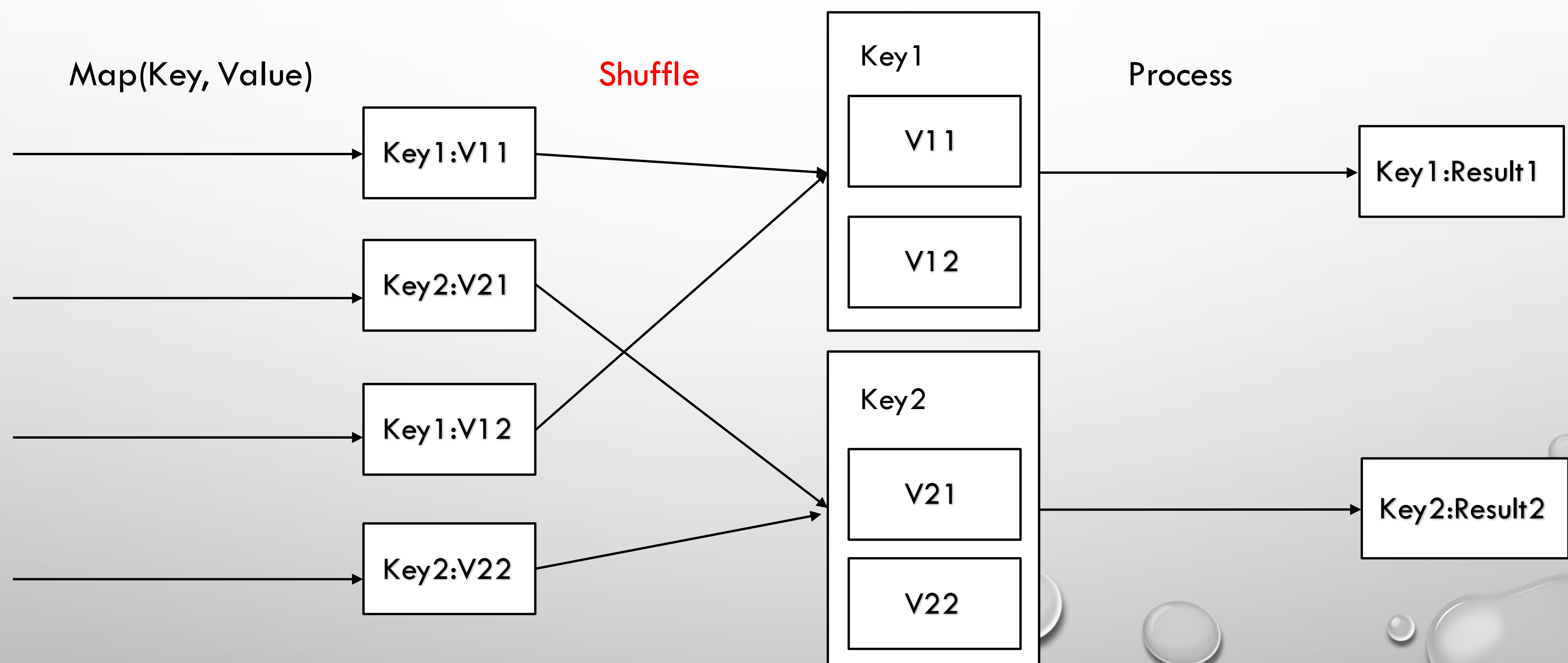
Automating distributed partial aggregation

Presenter: Guangzhe Gao

BackGround: MapReduce

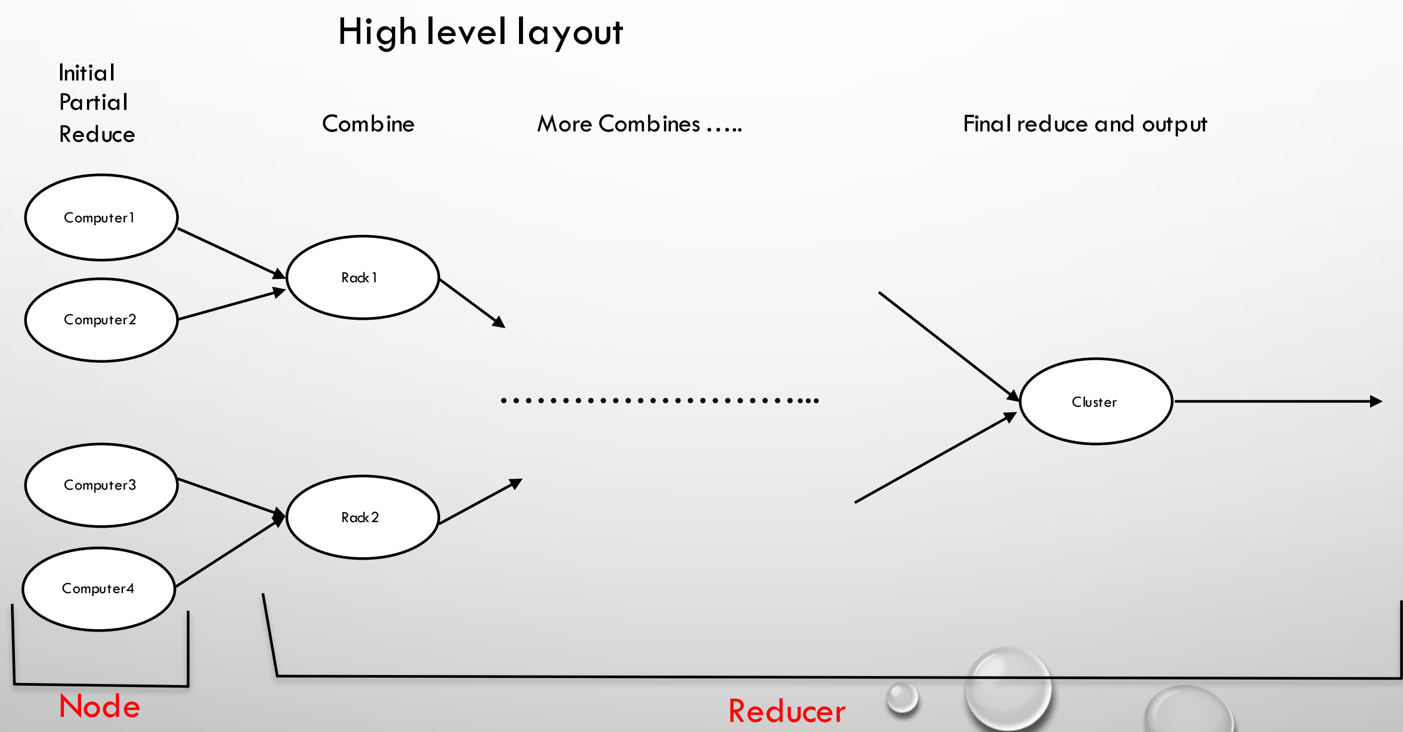


Bottleneck

- Network I/O from data nodes to the aggregate nodes.
- Means high delay when shuffling data to one machine
- Can't avoid if there is aggregation measure: e.g. SUM, COUNT, AVERAGE …
- How about aggregate what we have and only pass the partial aggregate result to the network?

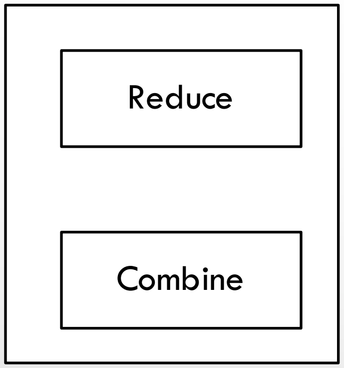
Motivation

- Solution: aggregate intermediate results, then transmit partial result.



Partial Aggregation

- Why: saves I/O time than aggregate everything finally in one shot.
- Old Approach to change to partial aggregation:
- Reduce function becomes:



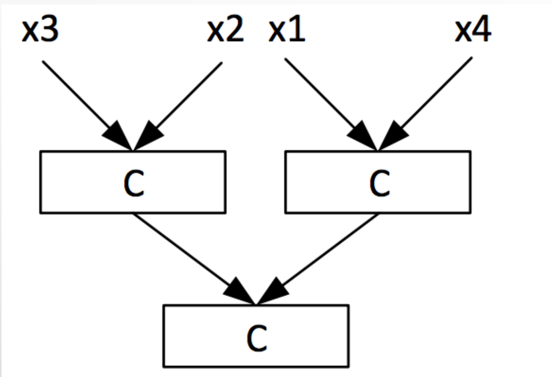
- Old Approach Problem: 1. Error Prone 2. Miss optimization opportunity.
- Need a way to judge partial aggregation feasibility.

Tool for aggregation optimization

- Automatic verifiction of partial aggregation applicability
- Change normal reducer code to more efficient scheme (initial reduce, combine, final-reduce)
- Decomposible: distributed programs eligible for partial aggregation.
- Decomposible Properties:
- 1. Reduce functions loops through records in a group using accumulator.
- 2. Initial/final-reduce function is the rearrangement of original reduce function code. Only combine function need to be made.
- 3. Reduce function decomposability depends on argebraic properties of combine function and accumulator.
- Claim: the necessary and sufficient condition of reduce function's decomposability: is determined by accumulator's commutativity; also, combine function is the inverse function of accumulator.
- Prove and verify (using program) the above claim is the work of the paper.

Combiner Difficulty

- Combining tree may be different, but the result must be same.
- For the graph example below, C(C(x3,x2),C(x1,x4)) or C(C(x1,x4), C(x2,x3)) should yield same result.




```
public IEnumerable<Row>
  Reduce(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);
   yield return outputRow;
}
```

Solution Variable

Accumulator

- The above right picture is an sample reducer. (emits key and summation plus 10)
- Commutativity of accumulator is verified by making sure for two row sets x, y, C(x,y) = C(y,x)

Combiner based on new Initial Reducer

```
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;
}
```

```
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;
}
```

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

Reducer Decomposibility?

- Even if we have comutativity, how does this mean decomposability of reduce function (which means partial aggregatable)?
- Also, above optimization is easy for human, but not for computer. Complex optimization will be infeasible for human.
- Things need to proof:
- Combiner exists <-> accumulator and reduce function commutative.
- General inverse function of accumulator -> constuction of combiner.
- Both will be proved later on.

Accumulator decomposibility Definition Preliminaries

$F(s,\varepsilon) = s$ and $F(s,\langle x\rangle \oplus X) = F(F(s,x),X)$.

- Just means accumulator F can do element-wise accumulation.
- If Solution space is defined as $S = \{s | s = F(s_0,X), X \in I^\star\}$ I is input domain
- We have for all input sequences $X,Y \in I^\star$, then $F(F(s,X),Y) = F(s,X \oplus Y)$.
- Result of reduce function R = Results of P(InitialReduce, Combine, and FinalReduce) -> R is decomposable.

Accumulator decomposibility Definition

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.*

- (1) means accumulator can wave the hand and throw work to combiner. Other parts just follow previous definitions and proof. Note that requirement 2. implies all other requirements (necessary and sufficient). Will be shown later.

Reducer Decomposibility <-> Accumulator commutativity

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.*

- The author only gives proof outline in this paper.
- First proof is to show: **Lemma 1.** *Given a combiner C that satisfies Equation 1, C is commutative and associative if and only if accumulator F is commutative.*

$$F(s_0, X_1 \oplus X_2) = C(F(s_0, X_1), F(s_0, X_2)). \qquad (1)$$

- Recall Definition 1, condition 2: *F is commutative: for any two input sequences* $X_1, X_2 \in I^\star$, $F(s_0, X_1 \oplus X_2) = F(s_0, X_2 \oplus X_1)$;
- consider input as single element rather than sequence, we have C commutative. Similar reasoning for associative.

Reducer Decomposibility continued

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

- Lamma 2 is more like definition so not proof given.
- *F is commutative: for any two input sequences $X_1, X_2 \in I^\star$, $F(s_0, X_1 \oplus X_2) = F(s_0, X_2 \oplus X_1)$;* implies exist combiner C satisfies
$$F(s_0, X_1 \oplus X_2) = C(F(s_0, X_1), F(s_0, X_2)). \qquad (1)$$
- Proof: given $s_1 = F(s_0, X)$, $s_2 = F(s_0, Y)$ $C(s_1, s_2)$ can be defined as $F(s_0, X \oplus Y) = F(s_1, Y)$
- How to do mapping?

General Inverse Function

- Inverse function $\mathscr{H}_F = \{H | \forall s \in S . F(s_0, H(s)) = s\}$
- H need not to be one-on-one according to the construction.
- We define derived combiner $C_H(s_1, s_2) = F(s_1, H(s_2))$ for each H. $
- Now need to show if F commutative, derived containers produce same result.
- The intuition in paper: If same aggregate output for two input sequences with initial solution value,
- The two input sequences can always generate same out put for arbitrary 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 $s \in S$.*

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

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

- Finally: the proof is done: for any commutative accumulator F, the combiner C can be generated using any general inverse function H of F.
- Formal restatement of everything:

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.*

Decomposability Verification

- From above lemmas and theorems, only need to show F(s, xy) = F(s, yx) for all s, x, y.
- But F is a program?
- Parse the language code.

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

- f: function name, F….F: nested function definition,
- e: expression (variable x, constant n, binary operator)

$$
\begin{array}{ll}
F_l = F(s,xy) & f_l[s,x,y].F.s_1 := f(s,x); s_2 := f(s_1,y); \mathbf{return}\ s_2 \\
F_r = F(s,yx) & f_r[s,x,y].F.s_1 := f(s,y); s_2 := f(s_1,x); \mathbf{return}\ s_2
\end{array}
$$

Path Formula

Output variables

Predicate

Expression

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

Index Set (of input and output)

Path Formula is true if {X1 … Xn, O1 … Om} evaluate to true. Meaning F(X1 … Xn) = O1 … Om

Convert program to formulae

- Symbolic execution.
- Look a lot like compiler parsers. If interested refer to extra slides.
- Now decompasability verification becomes SMT satisfiability problem:

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

- i.e. not output different case.

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

Separate by cases (isFirst) and output

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

Dilemma again

- Given decomposable accumulator F, combiner C that satisfies $C_H(s_1, s_2) = F(s_1, H(s_2))$ is difficult to construct.
- H is general inverse function of F.
- Greedy approach:
- most accumulators of decomposable reducer are in thre categories:

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.

Counting category

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 $X, Y \in I^\star$, $|X| = |Y|$.*

- In other words only size matters

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.*

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

r accumulates |s2| - |s0| zeros, so set H be of this number zeros

State Machine category

- Finite state machine defined by accumulator?
- BFS with depth(states #) threshold T.
- Explore all possible states and store transition table in Sol, so H can computed based on lookup
- C(S1, S2) has T^2 combinations

$F_{sm} = sm[s,x]$.

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

Sol:

$(0,-1,101)$, $(2,-1,100)$, and $(1,0,100)$.

Combiner

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

Single Input aggregation

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 $F(s_0, xy) = F(s_0, z)$.*

- Novel idea: Eliminate the need of z.
- Can eliminate if satisfies the partial function that requires z.

$$\forall x_0, x_1, y_0, y_1.$$
$$\exists z_0, z_1. z_0 = x_0 \wedge 10 + z_1 = 10 + x_1 + y_1 \wedge 0 = 0$$

- Can be rewritten as :

$$\forall x_0, x_1, y_0, y_1.$$
$$\exists z_0, z_1. z_0 = x_0 \wedge z_1 = 10 + x_1 + y_1 - 10 \wedge 0 = 0$$

- Two Z's in the left, so requiment is $\forall x_0, x_1, y_0, y_1. 0 = 0$

Evaluation

- It is a prototype

- Total jobs 4,429
- Baseline of auto partial aggregation: 183 Jobs use partial aggregation, 28 of them (15.3%) may be incorrect.
- Remaining 4246 jobs, 261 are true positives. (manually checked)

- Performance gain(in reduction):
- Time (61.6%) 165 sec -> 64 sec
- Space: (99.98%) 7.99GB -> 1.22MB
- 62.4% latency, 76% in network IO

Time cost and failure (not very useful)

#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$	#	$\forall sxy$	$\forall xy$
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.

Thank you!

Extra slides

- Symbolic Execution:

$$\boxed{\sigma(e) = e}$$

$$\text{S-Var}\ \frac{x \in \mathbf{Vars}}{\sigma(x) = \sigma(x)}$$

$$\text{S-Cnst}\ \frac{}{\sigma(n) = n}$$

$$\text{S-Aop}\ \frac{\sigma(e_i) = e_i', i = 1,2 \quad e = e_1'\ op_a\ e_2'}{\sigma(e_1\ op_a\ e_2) = e}$$

$$\boxed{\sigma(p) = \phi}$$

$$\text{S-TrFls}\ \frac{v = \mathbf{true} \text{ or } v = \mathbf{false}}{\sigma(v) = v}$$

$$\text{S-Rop}\ \frac{\sigma(e_i) = e_i', i = 1,2 \quad \phi = e_1'\ op_r\ e_2'}{\sigma(e_1\ op_r\ e_2) = \phi}$$

$$\boxed{\Upsilon \vdash \langle \mathcal{M}, s\rangle \to \mathcal{M}'}$$

$$\text{S-Seq}\ \frac{\Upsilon \vdash \langle \mathcal{M}, s_1\rangle \to \mathcal{M}' \quad \Upsilon \vdash \langle \mathcal{M}', s_2\rangle \to \mathcal{M}''}{\Upsilon \vdash \langle \mathcal{M}, s_1; s_2\rangle \to \mathcal{M}''}$$

$$\text{S-Ass}\ \frac{\mathcal{M}' = \{(\phi, \sigma') | (\phi, \sigma) \in \mathcal{M} \wedge \sigma' = \sigma[x \mapsto \sigma(e)]\}}{\Upsilon \vdash \langle \mathcal{M}, x := e\rangle \to \mathcal{M}'}$$

$$\text{S-Skip}\ \frac{}{\Upsilon \vdash \langle \mathcal{M}, \mathbf{skip}\rangle \to \mathcal{M}}$$

$$\text{S-If}\ \frac{\begin{array}{c}\Upsilon \vdash \langle \mathcal{M}, s_i\rangle \to \mathcal{M}_i, i = 1,2 \\ \mathcal{M}' = \{(\phi \wedge \sigma(p), \sigma) | (\phi, \sigma) \in \mathcal{M}_1\} \\ \cup \{(\phi \wedge \sigma(\neg p), \sigma) | (\phi, \sigma) \in \mathcal{M}_2\}\end{array}}{\Upsilon \vdash \langle \mathcal{M}, \mathbf{if}(p)\mathbf{then}\ s_1\ \mathbf{else}\ s_2\rangle \to \mathcal{M}'}$$

$$\text{S-Func}\ \frac{\begin{array}{c}\Upsilon(f) = f[x_1', \ldots, x_m'].F_1, \ldots, F_l.s; \mathbf{return}\ e_1', \ldots, e_n' \\ \mathcal{M}^\star = \{(\mathbf{true}, \{x_j' \mapsto x_j' | j = 1, \ldots, m\})\} \\ \Upsilon^\star = \{name(F_j) \mapsto F_j | j = 1, \ldots, l\} \quad \Upsilon^\star \vdash \langle \mathcal{M}^\star, s\rangle \to \mathcal{M}^{\star\star} \\ \mathcal{M}' = \{(\phi_1 \wedge \phi_2, \sigma') | (\phi_1, \sigma_1) \in \mathcal{M} \wedge (\phi_2, \sigma_2) \in \mathcal{M}^{\star\star} \\ \wedge \sigma' = \sigma_1[x_1 \mapsto \sigma_1(\sigma_2(e_1'))]\}\end{array}}{\Upsilon \vdash \langle \mathcal{M}, x_1, \ldots, x_n := f(e_1, \ldots, e_m)\rangle \to \mathcal{M}'}$$

Figure 6. Symbolic Execution for Path Formula

If interested, refer to

http://www.cs.umd.edu/~liuchang/paper/pa-socc2014-tr.pdf

See the section after citations: Proofs for Decomposability Theory

$$\boxed{\langle M,e\rangle \Downarrow v}$$

$$\text{Var}\ \frac{M(x)=v}{\langle M,x\rangle \Downarrow v}$$

$$\text{Cnst}\ \frac{}{\langle M,n\rangle \Downarrow n}$$

$$\text{Aop}\ \frac{\begin{array}{c}\langle M,e_i\rangle \Downarrow v_i, i=1,2 \\ v = v_1\ op_a\ v_2\end{array}}{\langle M,e_1\ op_a\ e_2\rangle \Downarrow v}$$

$$\boxed{\langle M,p\rangle \Downarrow v}$$

$$\text{TrFls}\ \frac{v=\textbf{true} \text{ or } v=\textbf{false}}{\langle M,v\rangle \Downarrow v}$$

$$\text{Rop}\ \frac{\begin{array}{c}\langle M,e_i\rangle \Downarrow v_i, i=1,2 \\ v = op_r(v_1,v_2)\end{array}}{\langle M,e_1\ op_r\ e_2\rangle \Downarrow v}$$

$$\boxed{\Upsilon \vdash \langle M,s\rangle \to M'}$$

$$\text{Seq}\ \frac{\Upsilon \vdash \langle M,s_1\rangle \to M' \qquad \Upsilon \vdash \langle M',s_2\rangle \to M''}{\Upsilon \vdash \langle M,s_1;s_2\rangle \to M''}$$

$$\text{Ass}\ \frac{\langle M,e\rangle \Downarrow v \qquad M' = M[x \mapsto v]}{\Upsilon \vdash \langle M,x := e\rangle \to M'} \qquad \text{Skip}\ \frac{}{\Upsilon \vdash \langle M,\textbf{skip}\rangle \to M}$$

$$\text{If}\ \frac{\begin{array}{cc}\Upsilon \vdash \langle M,s_i\rangle \to M_i, i=1,2 & \langle M,p\rangle \Downarrow v \\ v \Rightarrow M' = M_1 & \neg v \Rightarrow M' = M_2\end{array}}{\Upsilon \vdash \langle M,\textbf{if}(p)\textbf{then}\ s_1\ \textbf{else}\ s_2\rangle \to M'}$$

$$\text{Func}\ \frac{\begin{array}{c}\Upsilon(f) = f[x'_1,\ldots,x'_m].F_1,\ldots,F_l.s;\textbf{return}\ e'_1,\ldots,e'_n \\ \langle M,e_i\rangle \Downarrow v_i, i=1,\ldots,m \qquad M^\star = \{x'_j \mapsto v_j \mid j=1,\ldots,m\} \\ \Upsilon^\star = \{name(F_j) \mapsto F_j \mid j=1,\ldots,l\} \\ \Upsilon^\star \vdash \langle M^\star,s\rangle \to M^{\star\star} \qquad \langle M^{\star\star},e'_k\rangle \Downarrow v'_k, k=1,\ldots,n \\ M' = M[x_1 \mapsto v'_1,\ldots,x_n \mapsto v'_n]\end{array}}{\Upsilon \vdash \langle M,x_1,\ldots,x_n := f(e_1,\ldots,e_m)\rangle \to M'}$$

Figure 5. Operational Semantics