Indian Institute of Technology, Kanpur

Department of Computer Science and Engineering

Project Report

CS738: Advanced Compiler Optimizations

Academic Year 2020 - 2021

Ankita Dey (20111013)

Tamal Deep Maity (20111068)

Tanisha Rastogi (20111069)

Table of Contents

Introduction

We have tried to reimplement the paper:
'Improving database query performance with automatic fusion'
(link: https://dl.acm.org/doi/abs/10.1145/3377555.3377892)
by Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren.

We think this project is relevant to CS738A since we have used a data dependence graph generated from an array based Intermediate Representation called HorseIR to optimize SQL query and generate fused and optimised C++ code.

Implementation Overview

Since the HyPer's execution engine is no longer publicly available (also declared in the paper itself) and we do not have access to HorseIR, we have translated the SQL query to a C++ program.

We start by writing a C++ program as equivalent as possible to the test SQL query.

We create a data dependence graph from the HorseIR and give it as an input to the algorithm mentioned in the paper to obtain the optimized C++ program as our final output.

Translating SQL Query to C++

Taking an idea from the example SQL query given in paper:

- Created 10000000 X 3 table in C++
- 1st row -> Item Price
- 2nd row -> Item Discount
- 3rd row -> Item Date
- We used >=33% and <=66% in our Where clause

```
1 SELECT
2    SUM(item_price *
        item_discount) AS saving
3 FROM
4    store_items
5 WHERE
6    item_date >=2010.09.01 AND
7    item_date <=2010.09.30;
```

Translating SQL Query to C++ cont.

- Core part of HorseIR code for SQL query shown.
- Each of the statements S0 to S6 has been implemened in the C++ code.

```
// ... load columns from table
(S0) t0:bool = @geq(c0,2010-09-01:date);
(S1) t1:bool = @leq(c0,2010-09-30:date);
(S2) t2:bool = @and(t0, t1);
(S3) t3:f64  = @compress(t2, c1);
(S4) t4:f64  = @compress(t2, c2);
(S5) t5:f64  = @mul(t3, t4);
(S6) t6:f64  = @sum(t5);
// ... return result as a table
```

Shapes in HorseIR

- HorseIR defines two kinds of shape: Vectors and Lists
- Vector is of interest to us
- A vector shape describes a fixed-length one-dimensional array of homogeneous data
- Shapes tell us what will be the form of result on executing HorseIR statement

Shape	Description
V(1)	Vector of constant size 1 (i.e. scalar)
V(c)	Vector of constant size c where $c \neq 1$
V(d)	Vector of unknown static size (unique ID d)
V_s(a)	Vector from boolean selection a

Shape propagation and Conforming rules

F_B(x,y)	x			
y	V(1)	V(c_0)	V(d_0)	V_s(a_0)
V(1)	V(1)	V(c_0)	V(d_0)	V_s(a_0)
V(c_1)	V(c_1)	V(c_0)[1]	I	I
V(d_1)	V(d_1)	I	V(d_0)[2]	I
V_s(a_1)	V_s(a_1)	I	I	V_s(a_0)[3]
[1]: if c_0==c_1 otherwise error [2]: if d_0==d_1 otherwise I [3]: if a_0==a_1 otherwise I				

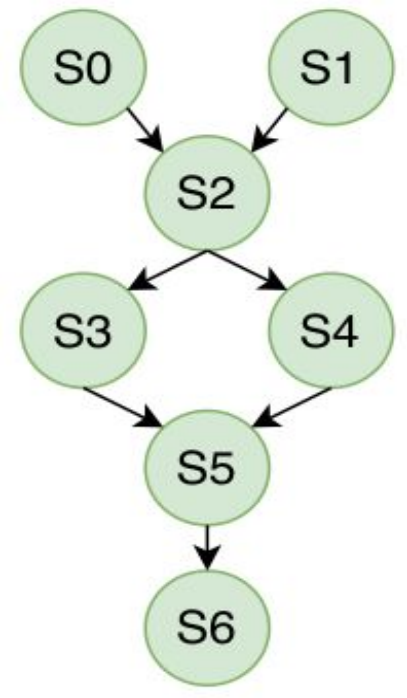
Shape propagation rules

	V(1)	V(c_0)	V(d_0)	V_s(a_0)
V(1)	✓	×	×	×
V(c_1)	×	c_0==c_1	×	cond(a_0,c_1)
V(d_1)	×	×	d_0==d_1	cond(a_0,d_1)
V_s(a_1)	×	cond(a_1,c_0)	cond(a_1,d_0)	a_0==a_1
cond(a,y) is ✓ if a.size == y else ×				

Shape Conforming rules

Data dependence graph

- Data dependence graph created from the HorseIR as shown.
- Fusible sections are determined from DDG using the algorithm given in next page



Algorithm for finding Fusible Sections

```
Input: Data dependence graph G
Output: A list of fusible sections
let ∅ be an empty vector;
allStmts ← reversed topological order of the graph G;
foreach stmt A in allStmts do
    if isNotVisited(A) then
        if getOp(A) is a reduction function then
            section ← findFromReduction(A);
        else
            section ← findFusibleSection(A);
```

```
Function findFusibleSection(A):
    if isNotVisited(A) then
        setVisited(A);
        if isGroupE_Binary(A) or isGroupS(A) then
            list ← fetchFusibleStmts(A, A.first.parent);
            list.append(fetchFusibleStmts(A,
              A.second.parent));
        else if isGroupE_Unary(A) or isGroupB(A) then
            list ← fetchFusibleStmts(A, A.first.parent);
        else if isGroupX(A) then
            list ← fetchFusibleStmts(A, A.second.parent);
        else
            list ← ∅;
        return {A}.append(list)
    return ∅;
Function fetchFusibleStmts((A,P)):
    if isConforming(A, P) then /* Rule 2 */
        return findFusibleSection(P);
    return ∅;
Function findFromReduction(A):
    setVisited(A);
    return {A}.append(findFusibleSection(A.first.parent));
```

Fusible Sections

Finally the following fusible sections found:

- S0 and S1
- S3 and S4

```
S0 (E):  t0::V(d)
S1 (E):  t1::V(d)

S2 (E):  t2::V(d)

S3 (S):  t3::Vs(t2)
S4 (S):  t4::Vs(t2)

S5 (E):  t5::Vs(t2)

S6 (R):  t6::V(1)
```

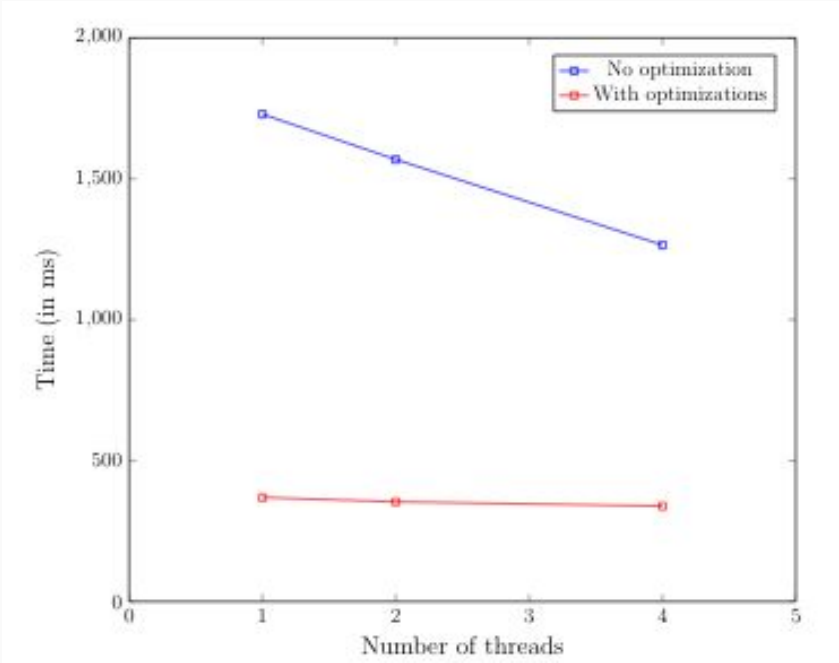
Optimized Code Generation

- Optimized C++ code written after finding fusible sections.
- Same table with 3 columns and 10000000 rows created and processed as shown.

```
1 //... load columns c0,c1,c2
2 t6 = 0;
3 for(int i=0; i<n; i++){
4   if(c0[i] >= 20100901
5       && c0[i] <= 20100930){
6     t6 += c1[i] * c2[i];
7   }
8 }
9 //... return t6, a scalar
```

Evaluation

On running the unoptimized c++ code and this optimized C++ code we saw the following difference in time taken:



Conclusion and Future Work

- Optimizations implemented in this paper improve run time of column based database systems.
- It exploits areas of IR that low level optimizations can’t do.
- It improves performance of both single and multiprocessor programs.
- Future Works include scope of expanding the rules of fusing array based propagation

THANK YOU