Optimization of Query Processing with Cache Conscious Buffering Operator

Yoshishige Tsuji[1], Hideyuki Kawashima[2,3], and Ikuo Takeuchi[1]

[1] Graduate School of Information Science and Technology, The University of Tokyo
[2] Graduate School of Systems and Information Engineering, University of Tsukuba
[3] Center for Computational Sciences, University of Tsukuba

Abstract. The difference between CPU access costs and memory access costs incurs performance degradations on RDBMS. One of the reason why instruction cache misses occur is the size of footprint on RDBMS operations does not fit into a L1 instruction cache. To solve this problem Zhou proposed the buffering operator which changes the order of operation executions.

Although the buffering operator is effective, it cannot be applied for RDBMS in the real business. It is because Zhou does not show an algorithm for optimizer to select the buffering operator.

Thus we realized the CC-Optimizer which includes an algorithm to appropriately select the buffering operator. Our contributions are the design of new algorithm on an optimizer and its implementation to RDBMS.

For experimental RDBMS, we used PostgreSQL as Zhou did, and our machine environment was Linux Kernel 2.6.15, CPU Intel Pentium 4(2.40GHz). The result of preliminary experiments showed that CC-Optimizer was effective. The performance improvement measured by using OSDL DBT-3, was 73.7% in the greatest result, and 17.5% in all queries.

1 Introduction

The difference between the CPU and memory degrades relational database management system (RDBMS) performance. The traditional approach to this problem has been to reduce L2/L3 data cache misses [1,2,3]. In another approach, Zhou proposed to reduce the number of L1 instruction cache misses [4]. Zhou proposed a buffering operator, which buffers usual operators such as index scan, sequential scan, aggregation, and nested loop join.

The buffering operator reduces the number of L1 instruction cache misses under certain conditions, but it increases misses under other conditions. Beyond that, the positive and negative conditions were not clarified in the original work [4]. Also, [4] does not go far enough in the design of an optimizer with buffering operator. In designing such an optimizer, the positive and negative conditions should be clarified.

This paper investigates the effects of the buffering operator by re-implementing it. We adopted PostgreSQL-7.3.16, Linux Kernel 2.6.15, CPU Intel Pentium 4 (2.4 GHz). We propose a **C**ache **C**onscious query optimizer, the CC-optimizer with algorithms that judge the usage of buffering operators.

The rest of the paper is organized as follows: Section 2 describes related work that reduces the number of CPU cache misses on RDBMS. We introduce the buffering operator paper [4] in detail. We then describe the flaws in [4] and formulate the problem. Section 3 describes re-implementation of the buffering operator and experimental investigations of its behavior. Section 4 describes implementation of the CC-optimizer. Section 5 describes experimental results of the CC-optimizer measured by a standard benchmark tool, OSDL DBT-3[5]. Section 6 concludes this paper.

2 Related Work and Problem Formulation

2.1 CPU Archichitecture and L2 Data Cache

CPU has a hierarchical memory architecture. CPU memory accesses are accelerated by cache memories. Cache memories are incorporated into the CPU die internally, taking into account both spatial locality and time locality. Figure 1 shows the architecture of the Pentium 4 cache memory[1].

The CPU first accesses an L1 cache to look for data or instructions. If it fails, then CPU accesses an L2 cache. If that fails, the CPU then accesses an L3 cache if it exists, or memory if an L3 cache does not exist. The access cost for memory is more than

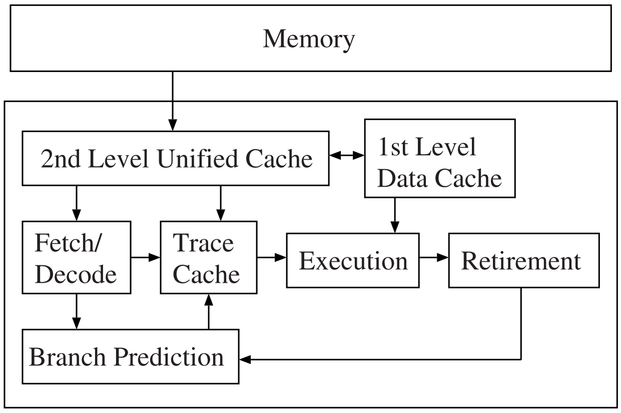

Fig. 1. Pentium 4 Cache Memory Architecture

```
SELECT COUNT(*) FROM item;
```

Fig. 2. A Query with Simple Aggregation

[1] Reference: Zhou, J. and Ross, K.A., Buffering Database Operations for Enhanced Instruction Cache Performance, Figure 2, Proc. SIGMOD'04.

10 times that for the L2/L3 cache [4]. That is why considerable work has focused on reducing L2/L3 cache misses [1][6][7][3].

2.2 L1 Instruction Cache: Buffering Operator

As shown above, some work focused on reducing the number of L2/L3 cache misses. On the other hand, a few work focused on reducing the number of L1 cache misses [2,4].

To the best of our knowledge, the [4] only focused on reducing L1 cache misses for general relational query processing. Although the penalty for an L1 cache miss is smaller than for an L2 cache miss, L1 cache misses outnumber L2 cache misses. It is therefore important to reduce L1 cache misses from the standpoint of DBMS performance.

[4] proposes a method that reduces L1 instruction cache misses on a real DBMS, PostgreSQL. The method is referred to as "buffering operator." Although the advantage of the buffering operator is clear, it should not be directly applied to real DBMSs. This is because the buffering operator improves performance in some situations, but it degrades performance in other situations. The following section describes the buffering operator in detail.

The total size of operators required for a query processing sometimes exceed capacity of an L1 instruction cache. When that happens, an L1 instruction cache miss occurs, degrading performance of the RDBMS. Let us look at the mechanism of an L1 instruction cache miss in detail.

First, an operator is a processing unit with a specific meaning in an RDBMS. PostgreSQL implements many operators including sort, join, aggregation, etc. We take into account the behavior of operators when processing the simple query shown in Figure 2. To process the query, a scan operator that reads the item relation and an aggregation operator that aggregates the result of the scan are necessary. On PostgreSQL, these operators are implemented as the TableScan operator and Agg operator. They construct a plan tree as shown in Figure 3.

On the plan tree, Agg is denoted as a parent operator (P) and TableScan is denoted as a child operator (C). For each aggregate processing on P, P fetches a tuple from C followed by the pipeline execution manner. This mechanism contains the sporadic L1

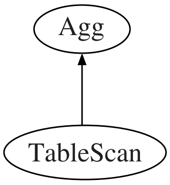

Fig. 3. Simple Plan Tree

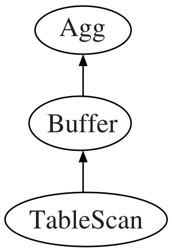

Fig. 4. Buffering Operator for a Simple Plan Tree

cache miss occurrences. Assuming that cardinality of the item relation in Figure 3 is 8, operators are executed as follows:

PCPCPCPCPCPCPCPC

If the total footprint of P and C is larger than the L1 instruction cache, an execution of both operators causes an L1 instruction cache miss. This occurs because only one of P or C can fit into the L1 instruction cache; the other will be excluded from the cache.

To solve this problem, [4] proposed a novel technique, the buffering operator. The buffering operator buffers a pointer to data that is generated by child operator. Buffering operator can be implemented with a small footprint. With the buffering operator, usual operators can be executed continually, thereby reducing the number of L1 instruction cache misses. Let us look at an example. By applying buffering operator to the simple plan tree in Figure 3, a buffering operator is inserted into a link between two nodes, and a new plan tree is constructed as shown in Figure 4. If the buffering size of the buffering operator is 8, the execution order of parent/child operators can be changed as follows using the buffering operator.

PCCCCCCCCPPPPPPP

The same operators are executed continually (C. . .C or P. . .P), thereby reducing the number of L1 cache misses. However, please note that the reduction is achieved only when the total footprint of P and C is larger than L1 instruction size.

[4] evaluated the buffering operator, reporting that the buffering operator reduced the number of L1 instruction cache misses by about 80%, and the buffering operator improved query performance by about 15%.

2.3 Problem Formulation

As Section 2.2 described, the buffering operator reduces the number of L1 instruction cache misses if total size of the parent/child operators is larger than the L1 cache. Otherwise, the buffering operator increases the number of L1 instruction cache misses because of the overhead produced by the buffering operator itself [4]. Therefore, the technique should be applied only when it is effective. In the final stage of query processing in a usual RDBMS, after several plan trees are generated by a planner module, a query optimizer chooses the best plan from among them. For the chosen plan tree, with the buffering operator, the optimizer should find links where the buffering operator should be inserted to improve performance further.

To realize such a query optimizer, the following two problems should be addressed.

1. Deep analysis of buffering operator behavior.
   The behavior of the buffering operator is not analyzed in depth by [4]. The paper does not present enough data for a query optimizer to find links in a plan tree to insert the buffering operator. To obtain the data, detailed investigation should include cases in which the buffering operator is appropriate and when it is inappropriate; and the boundaries should be clarified with several parameters.
2. Realization of a query optimizer with buffering operator.

After finishing the in-depth analysis of buffering operator behavior, an algorithm to find links, where the buffering operator is inserted, should be constructed based on the analysis.

We describe approaches to these problems in the remainder of this paper.

3 Analysis of Buffering Operator

3.1 Reimplementation of Buffering Operator

To analyze the behavior of the buffering operator, we re-implement it reading [4]. For implementation, we used PostgreSQL-7.3.16 since [4] used PostgreSQL-7.3.4; their basic architectures are the same.

We implemented the buffering operator as a new operator. Therefore, conventional operators are not modified with this implementation. In our implementation, the buffering operator is invoked by the executor module. The executor module indicates the position of the buffering operator in a plan tree. An example of the indication is shown in Figure 4. To achieve it, we modified the executor module so that it can deal with the buffering operator similarly to conventional operators.

We adopted the open-next-close interface used in PostgreSQL. The open function and close function allocate and release pointers to tuples generated by child operators, and the arrays that hold them. However, our implementation of arrays differs from [4]. The difference is that [4] used only fixed length arrays, while we allow variable length arrays. The length of an array is specified by our optimizer module. The next function performs as follows: (1) If the array is empty, then it executes child operators until the array is fulfilled or the final tuple is given; it then stores pointers to tuples, generated by a child operator, into the array. (2) If the array is not empty, then it provides pointers in the array to a parent operator. When a buffering operator is inserted, (1) if the array is empty, then a child operator is continually executed and thus the L1 instruction cache miss does not occur, and (2) if the array is not empty, then a parent operator is continually executed; thus the L1 instruction cache miss does not occur either.

As we described in Section 2.2, the buffering operator should be small enough to fit into the L1 instruction cache with another operator. The footprint of the buffering operator we implemented was about 700 Bytes while large types of PostgreSQL operators are more than 10 KBytes. In measuring the footprint, we used Intel VTune.

3.2 Experimental Analysis Outline

Hardware. To analyze the experiment, we used hardware with the specs given in Table 1. The hardware for [4] and our hard are the same, which is Pentium 4. Please note that “trace cache” is the same as the usual L1 instruction cache.

Relations. In our experiment, we used two simple relations, points and names. The points relation has two integer attributes, id and point. The names relation has two integer attributes, id and name.

Table 1. Hardware Environment

CPU	Pentium(R) 4 2.40GHz
OS	Fedora Core 5 (Linux 2.6.15)
RAM	1GB
Trace Cache	12K μops (8KB–16KB)
L1 Data Cache	8KB
L2 Cache	512KB
Compiler	GNU gcc 4.1.0

Investigation Parameters. Using the buffering operator, we observed its behavior through experiments. This subsubsection describes the result of experiments. On the experiments, we thought the following three parameters should be investigated.

1. Total footprint of parent and child operators
   When total foot print size of the parent and child operators is smaller than the L1 instruction cache size, then the cache miss does not occur. Otherwise it occurs. Therefore, the footprint of each operator should be investigated to know the pairs that cause cache misses.
2. Number of buffering operators in the L2 cache
   The L2 cache stores both buffering operators and child operators buffered by the buffering operators. When there are too many buffering operators, L2 cache capacity is exceeded and an L2 cache miss results. Therefore, the relationship between the number of buffering operators and L2 cache misses should be investigated.
3. Number of buffering operator executions
   Execution of the buffering operator requires excess cost. The cost, however, is small and its execution may reduce L1 cache misses. Therefore, the buffering operator improves the performance of query processing if the number of execution times of buffering operators crosses a specified threshold. Execution times are directly related to the cardinality of a relation. Therefore, the relationship between improving query processing performance and the cardinality of a relation should be investigated.

3.3 Result with (1) Total Footprint Size of Parent and Child Operators

Footprint size of each operator is investigated by [4], and this paper follows it. It should be noted that simply adding the sizes of two operators is not enough to estimate the total size of two operators. This is true because (1) operator sizes after compiling differ depending on a machine environment, and (2) even if the architectures are the same, different modules may share the same functions. Therefore, we executed a variety of queries and investigated conditions under which the buffering operator is appropriate.

Large Size Opetators. As described above, when total size of the parent operator and child operator exceeds L1 cache capacity, an L1 cache miss occurs. To confirm, we evaluated the performance of queries with large operators. As representatives, we selected the aggregation operator and sort operator. The following descrbes the result.

Table 2. Queries varying Aggregation Complexity

ID	SQL
A	SELECT COUNT(*) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
B	SELECT COUNT(*), AVG(point) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
C	SELECT COUNT(*), AVG(point), SUM(id) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
D	SELECT COUNT(*), AVG(point), SUM(id), SUM(id/2) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
E	SELECT COUNT(*), AVG(point), AVG(id), SUM(id/2) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
F	SELECT COUNT(*), AVG(point), max(point/2), SUM(id/2) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
G	SELECT COUNT(*), AVG(point), AVG(id), SUM(id/2), max(point/2) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0
H	SELECT COUNT(*), AVG(point), AVG(id), AVG(id/2), AVG(point/2), AVG(id/3) FROM points WHERE id $<$ 5000000 AND point $\geq$ 0

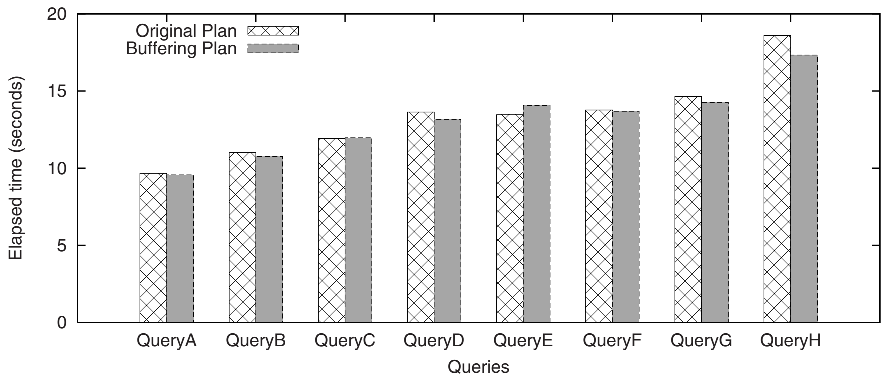

Fig. 5. Result of Aggregate Queries

Aggregation Operator. The aggregation operator comprises a base operator and each functional operator [4]. The size of the aggregation operator differs depending on query complexity. Varying the complexity of aggregation operators, we measured query execution time. The queries are shown in Figure 2. The result of experiments is shown in Figure 5. The following summarizes the result of experiments. Please note that all the queries included buffering operators. (1) When more than three kinds of aggregations, including AVG and SUM, were used, performance improved. (2) When two kinds of aggregations, including AVG and SUM, were used, performance sometimes improved and sometimes degraded. (3) When only AVG was used and the number of AVGs was more than 5, performance improved. (4) When only SUM was used and the number of SUMs was more than 5, performance improved.

From the above result, we judged that the buffering operator should be applied when more than three kinds of aggregations, including AVG and SUM, were used.

Sort Operator. Next, we investigated effectiveness of the buffering operator with the sort operator through experiments. For the experiments, we used queries shown in Figure 6.

```
SELECT * FROM points WHERE id < 5000000 AND point ≥ 0 ORDER BY point DESC
```

Fig. 6. Query with Sort

Table 3. Execution Time with Sort

Original Plan	Buffered Plan	Improvement by Buffering
91.63 sec	90.93 sec	0.76%

Table 3 shows the results of experiments. When a sort operator was parent, the buffered plan slightly (0.76%)improved performance. On the other hand, when a sort operator was child, the buffered plan degrades performance. This occurs because the sort operator simply generates sorted tuples; therefore, its footprint is small.

In summary, the buffering operator performed effectively when the total size of operators was too large to fit into the L1 instruction cache.

Small Size Operators. Third and last, we measured the execution times of queries when buffering operators are applied to small-sized operators. The query is shown in Figure 7. The buffering operator was applied only to sub queries.

The results of experiments are given in Table 4. The result shows that the buffering operator slightly degrades performance (1.25%).

3.4 Result with (2) the Number of Buffering Operators in L2 Cache

As the number of buffering operators increases, the number of L1 cache misses decreases. However, if the buffering operators do not fit in the L2 cache, L2 cache misses occur. We investigated appropriate sizes for buffering operators through the following two experiments.

Single Buffering Operator into a Plan Tree. In the first experiment, we inserted only one buffering operator into a plan tree. For the query, we included four aggregations so that operators exceed the L1 cache size. Results of the experiment are shown in Figure 8. The results show that execution times improve when buffering size is from 100 to

```
SELECT * FROM names n, (SELECT COUNT(*) FROM points GROUP BY point) AS t
WHERE t.count = n.id
```

Fig. 7. Small Size Operator

Table 4. Execution Time with Small Size Operator

Original Plan	Buffered Plan	Improvement by Buffering
184.6 sec	186.9 sec	-1.25%

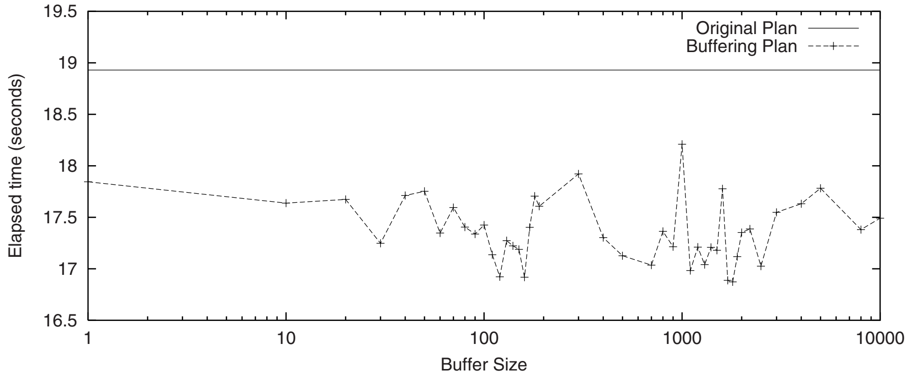

Fig. 8. Execution Time with Single Buffering Operator

2500. In particular, when buffering size is 1800, the execution time improved 10.86% comparing with the original plan.

Further, we measured the number of L1 cache misses and L2 cache misses. The results are shown in Figure 9. With the L1 cache, as buffering size increases, the number of cache misses decreases. With the L2 cache, when buffering size is from 10 to 1000, the number of L2 cache misses is relatively small, and when buffering size is more than 1000, the number of cache misses rapidly increases.

From the above results, with this query, appropriate buffering sizes are from 100 to 1800.

Multiple Buffering Operators into a Plan Tree. Second, we conducted an experiment with multiple buffering operators. For the experiment, we used a query with two join operators. The result of the experiment is shown in Figure 10. When buffering size is 700, performance improvemes 19.7%. Similar to aggregation operators, when the buffering size of a buffering operator was more than 100, large improvements were observed.

3.5 Result with (3) Number of Buffering Operator Executions

Even if the buffering operator is effective, the invocation of buffering operator requires excess cost. Thus, for a query processing to improve performance, the reduction in L1 cache misses should at least complement the excess cost. We measured the relationship

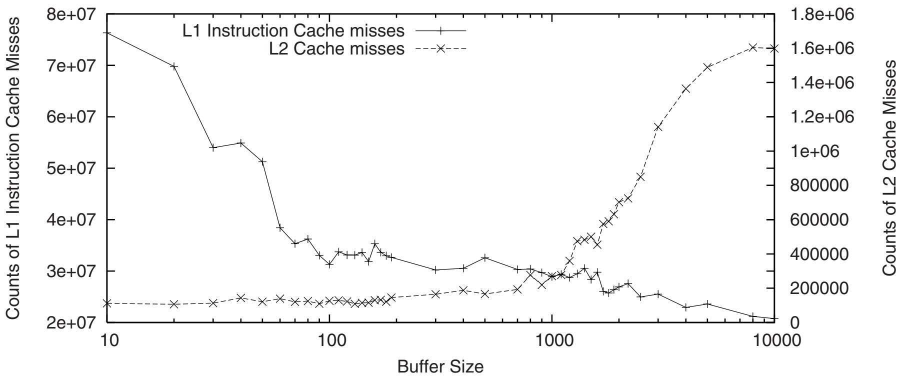

Fig. 9. L1 Cache Misses with Single Buffering Operator

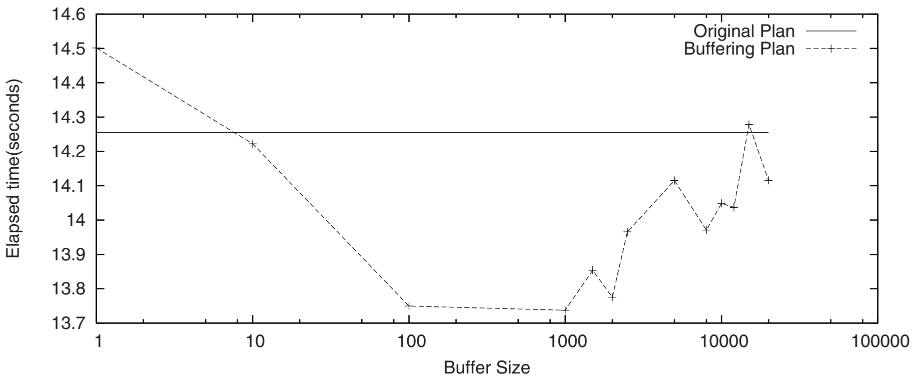

Fig. 10. Execution Time with Multiple Buffering Operators

between the complement and the cardinality of a relation. For the experiment, we used the queries in Figure 2.

The result of experiments are shown in Figure 11 and 12. The results show that when cardinality is about 600, the original plan lost stability and degraded performance.

We also measured the relationship between cardinality and the number of cache misses. The result is shown in Figure 12. For the L1 cache, when cardinality is more than 300, the buffered plan showed better performance than the original plan. For the L2 cache, when cardinality is more than 500, the buffered plan showed better performance than the original plan. Therefore, when cardinality is more than 500, the buffered plan is better than the original plan with both the L1 cache and the L2 cache.

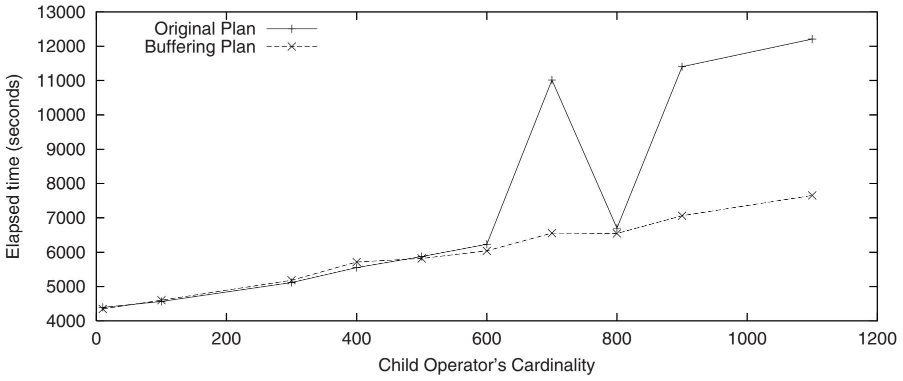

Fig. 11. Execution Time with Cardinality

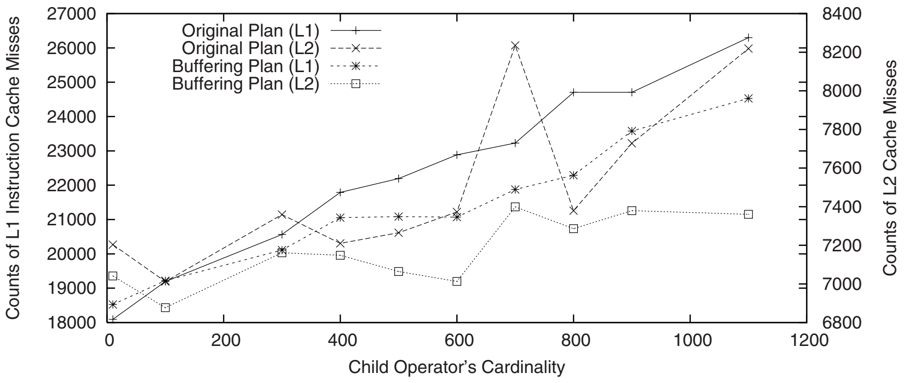

Fig. 12. The Number of Cache Misses with Cardinality

4 Proposal of a Cache Conscious Query Optimizer

This section proposes a cache conscious query optimizer, the CC-optimizer. This section describes algorithms to judge buffering operator insertion in the CC-optimizer, and implementation of the CC-optimizer.

4.1 Algorithm to Insert Buffering Operator

From the experimental results described in Section 3, we summarize conditions under which the buffering operator improves performance in Figure 13. Only when the two conditions in Figure 13 are satisfied, buffering operator is invoked. Since the conditions are simple, execution cost for them is almost for free, which does not incur performance degradation with query optimization processing.

1. At least one of the following should be satisfied.
   (a) Parent node includes more than three kinds of aggregations, and child node is a main module except for sort.
   (b) Parent node is a join operator and child nodes are main modules except for sort.
   (c) Parent node is a sort operator and child node is a main module.
2. The number of tuples to be processed is more than 500.

Fig. 13. Conditions to Invoke Buffering Operators

4.2 Distribution of Buffer into Multiple Links

When multiple links in a plan tree satisfy conditions in Figure 13, an optimizer should coordinate the number of buffering operators for each link. Two things should be considered. The detailed procedure is described in Section 4.3.

- The L2 cache size limit of buffering operators in all links
  The total size of buffering operators in the L2 cache is computed as: (*the size of a tuple obtained from child operator*) × (*the buffer size of each buffering operator*). This size should be smaller than the L2 cache size. Please note that the L2 cache is not dedicated for buffering operators, so the usable area is smaller than the total L2 cache size. In our experiments, we set the usable area as 14 KBytes of 512 KBytes.
- Minimum number of buffering operators in a link
  From Figure 11 and 12, the number of tuples to be processed should be more than 100 to improve query processing performance when the buffering operator is applied.

4.3 Implementation of Optimizer

In PostgreSQL, plan trees are implemented using list structures that correspond to operators. The judgment of buffering operator insertion is conducted at the final stage of optimizer execution. For the judgment, the conventional optimizer module generates a plan tree with the best cost, and the root node of the plan tree is provided for the judgment. We implemented CC-optimizer through the following three steps.

1. Searching places to which buffering operators can be inserted
   In the first step, the CC-optimizer searches places in which The buffering operators can be inserted. A PostgreSQL server generates primitive logical plan trees when a query arrives. The conventional PostgreSQL optimizer recursively reads the plan trees, assigns minimum cost operators, and generates the execution plan. For example, when join conditions are included in a query, PostgreSQL chooses the fastest operator from nested loop join, hash join, and merge join.
   Based on the algorithm, the CC-optimizer judges whether a buffering operator can be inserted when aggregation, join, and sort operators are selected following the conditions in Figure 13.

2. Calculating size for each buffering operator insertion point
   In the second step after finishing the first step, buffering size for each buffering operator insertion point is calculated following Figure 13.
   The calculation is as follows: First, for each insertion candidate point, a buffering size of 100 is given because the buffering operator is effective as shown in Figure 9 and 10. Second, if L2 cache capacity remains, then the remaining capacity is equally divided and assigned to each insertion candidate point.
3. Insertion of buffering operator
   Finally, buffering operators are inserted into candidate points.

5 Evaluation

5.1 Measurement of Cache Misses with Performance Counter

Most recent CPUs, including Pentium 4, hold registers dedicated to collect specific events for performance analysis. We obtained the events to evaluate performance of the CC-optimizer. The detailed hardware environment for experiments is shown in Table 1.

To obtain the events we used Perfctr, a monitor driver. To use Perfctr, we reconstructed the Linux-2.6 kernel after applying Perfctr patches. We then used PAPI (Performance Application Programming Interface) to conduct the measurement easily. PAPI provides an integrated interface to view events collected by Perfctr. PAPI provides the C language interface. So we implemented the measurement procedures in PostgreSQL internal. The procedures are start point and end point. The start point is where the PostgreSQL server receives a query. The end point is where the socket connection is closed.

We collected four events. They are (1) Trace cache misses, (2) L2 total cache misses, (3) Conditional branch instructions miss-predicted, and (4) Total cycles. We used Pentium 4 in our experiment. The Pentium 4 cpu implements a trace cache instead of an L1 instruction cache. Therefore, we treated trace cache misses as L1 instruction cache misses.

5.2 Benchmark

For benchmarking, we used OSDL DBT-3 [5]. DBT-3 is a database benchmark software based on the TPC-H benchmark. DBT-3 provides 22 complex queries to measure the performance of decision support systems.

In our experiment, we set two parameters of DBT-3 as follows: The first parameter was the scale factor to express the scale of data for the experiment. When the scale factor is 1, then 1 GByte text is generated, and more than a 4 GByte database cluster is generated. We set the scale factor as 1. The second parameter was the number of streams to express the number of concurrently executed transactions on the throughput measurement experiment. We set the parameter as 1 so that context switches will not likely occur. The remaining parameters were default values.

Table 5. Experimental Result (Cardinality=500)

Query	Total Execution Time Improvement (%)	Process Execution Time Improvement (%)	Trace Cache Miss Improvement (%)	L2 Cache Misses Improvement (%)
Q4	73.65	8.036	7.607	25.52
Q18	39.26	0.03162	6.942	76.26
Q20	34.74	5.274	1.771	-2.464
Q10	20.88	2.096	33.39	-14.74
Q7	17.47	2.326	17.39	16.79
Q6	11.36	2.547	36.87	-2.266
Q13	6.356	-3.848	-83.95	-7.666
Q16	5.255	4.637	40.96	-7.072
Q2	5.066	5.094	23.27	18.54
Q14	2.528	4.147	37.87	8.395
Q11	0.7313	-0.7235	36.00	-35.76
Q12	0.1540	-0.9972	28.05	9.089
Q22	-0.05150	-0.04429	-27.34	-3.397
Q19	-0.6108	4.600	2.716	28.22
Q1	-0.7540	1.457	-13.68	-16.69
Q9	-1.042	3.326	4.562	4.664
Q21	-1.084	2.330	-2.256	10.35
Q17	-1.482	-1.542	12.31	6.670
Q3	-5.073	2.434	15.47	-1.632
Q15	-13.22	3.220	38.07	25.58
Q5	-13.43	4.871	18.03	17.22
Q8	-24.57	2.577	19.13	8.761

5.3 Result

When L2 cache capacity is set to 14 KB and the cardinality threshold is 500, we obtained the result of the DBT-3 benchmark as given in Table 5. In the experiment, the CC-optimizer applied the buffering operator to 16 of 22 queries.

Table 5 shows that for 9 of 16 queries, execution times were improved. Q4 improved most, and the ratio was 73.6%. Further, most queries that achieved large execution time improvement also improved trace cache misses (L1 cache misses).

On the other hand, 7 of 16 queries degraded performance. The maximum degradation ratio was 24.5% by Q8.

The buffering operator was not applied to Q2, Q6, Q9, Q17, Q19, and Q20. For these cases, excess cost of the CC-optimizer potentially degrades performance. However, the degradation was observed only in Q17, and it was just 1.5%. Therefore, the implementation cost of CC-optimizer algorithms is considered small.

In all, a 17.5% execution time improvement was achieved by the CC-optimizer.

It should be noted that interesting results are also observed in Table 5. Negative correlation between cache misses and execution time is observed in Queries 13, 19, 9, 17, 15, 5, and 8. It may be caused by disk accesses, but the detailed analysis is left in the future work.

6 Conclusions and Future Work

This paper proposes algorithms that apply buffering operators to query processing. To construct the algorithm, we re-implemented the buffering operator proposed in [4], and conducted several experiments and analyzed behavior of the buffering operator. We added a new module to a conventional query optimizer module so that buffering operators can appropriately be inserted at the final stage of query optimization. We proposed the CC-optimizer, a modified cache conscious query optimizer. The CC-optimizer was implemented on PostgreSQL-7.3.16 and evaluated using the OSDL DBT-3 benchmark suite. The result of experiments showed that the CC-optimizer improved execution time of a query 73.6% in the maximum case; it improved total execution time 17.5%. We therefore conclude that this work provides algorithms that appropriately apply the buffering operator to the query optimizer module.

The algorithms we presented can be applied only for a specific machine. This is because parameters in algorithms must be investigated with a lot of human intervention. In future work, we will investigate the behavior of PostgreSQL more in detail to analyze the reason of negative correlation between cache misses and execution time, and sophisticate the proposed algorithms.

Acknowledgements

This work is partially supported by the Exploratory Software Project Youth Program of the Information-Technology Promotion Agency, Japan (IPA), and by the Grant-in-Aid for Scientific Research from JSPS (#20700078).

References

1. Boncz, P., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: Memory access. In: VLDB (1999)
2. Harizopoulos, S., Ailamaki, A.: Improving instruction cache performance in oltp. ACM Trans. Database Syst. 31(3), 887–920 (2006)
3. Cieslewicz, J., Mee, W., Ross, K.A.: Cache-conscious buffering for database operators with state. In: DaMoN, pp. 43–51 (2009)
4. Zhou, J., Ross, K.A.: Buffering database operations for enhanced instruction cache performance. In: SIGMOD, pp. 191–202 (2004)
5. OSDL DBT-3, `http://osdldbt.sourceforge.net`
6. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: VLDB (2001)
7. Liu, L., Li, E., Zhang, Y., Tang, Z.: Optimization of frequent itemset mining on multiple-core processor. In: VLDB, pp. 1275–1285 (2007)