Efficient Execution of Multiple Queries on Deep Memory Hierarchy*

Yan Zhang¹ (张岩), Zhi-Feng Chen², **(陈智峰), and Yuan-Yuan Zhou³ (周源泉)

¹National Laboratory on Machine Perception, Peking University, Beijing 100871, China
²Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, U.S.A.
³Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.

E-mail: zhy@cis.pku.edu.cn; zhifeng.chen@gmail.com; yyyzhou@cs.uiuc.edu

Received May 1, 2006; revised December 19, 2006.

Abstract This paper proposes a complementary novel idea, called MiniTasking to further reduce the number of cache misses by improving the data temporal locality for multiple concurrent queries. Our idea is based on the observation that, in many workloads such as decision support systems (DSS), there is usually significant amount of data sharing among different concurrent queries. MiniTasking exploits such data sharing to improve data temporal locality by scheduling query execution at three levels: query level batching, operator level grouping and mini-task level scheduling. The experimental results with various types of concurrent TPC-H query workloads show that, with the traditional N-ary Storage Model (NSM) layout, MiniTasking significantly reduces the L2 cache misses by up to 83%, and thereby achieves 24% reduction in execution time. With the Partition Attributes Across (PAX) layout, MiniTasking further reduces the cache misses by 65% and the execution time by 9%. For the TPC-H throughput test workload, MiniTasking improves the end performance up to 20%.

Keywords cache performance, temporal locality, mini-task scheduling, concurrent queries

1 Introduction

As the size of main memory is dramatically increasing, most of query processing working sets can fit into main memory for many database workloads. As a result, the main memory latency is becoming a major performance bottleneck for many database applications, such as DSS (Decision Support System) applications[1-3]. This problem is getting worse as the processor-memory speed gap increases. Previous work demonstrates that the L2 data stall time is one of the most significant components of the query execution time[1]. We conducted similar measurements using IBM DB2 with DSS workloads. As shown in Fig. 1, our results demonstrate that on Pentium 4, the L2 cache misses contribute 18%-56% of CPIs (cycle per instructions) for most TPC-H queries. Therefore, improving the L2 cache hit ratio is critical to reduce the number of expensive memory accesses and improve the end performance for database applications.

An effective method for improving the L2 data cache hit ratio is to increase data locality, which includes spatial locality and temporal locality. Spatial locality says that after a memory location is accessed, locations nearby (within the same cache line) are highly likely to be accessed. Temporal locality means that recently accessed memory locations are very likely to be accessed again. Computation with better temporal or spatial locality has fewer cache misses since data in the cache can be reused, which is called spatial reuse or temporal reuse, respectively.

Many previous studies have proposed to improve the data spatial locality of a single query by using cache-conscious data layout[4-6]. These layout schemes place data that are likely to be accessed together consecutively so that servicing one cache miss can “fetch” other data into the cache to avoid subsequent cache misses. Although these techniques are very effective in reducing the number of cache misses, the memory latency still remains a significant bottleneck of the query execution time. For example, our experiment shows that, with the PAX layout[4], the L1 and L2 cache misses still contribute around 20% of CPIs for TPC-H queries.

Another potential technique to reduce cache misses is to improve data temporal reuse by reordering computation. Examples of this approach include compiler-directed tiling or loop transformations[7], fine-grained thread scheduling[8,9], just to name a few. While these techniques are very useful for regular, array-based applications, it is difficult to apply them to database applications that usually have complex pointer-based data structures, and whose structure information is known

---

*Short Paper
*Please contact us if you want a full version.
**Dr. Chen contributed to this work during his Ph.D. study at University of Illinois.
only at run-time after the database schema is loaded into the main memory. So far few studies have been conducted to improve the temporal cache reuse for database applications.

1.2 Our Contributions

In this paper, we propose a technique called MiniTasking to improve data temporal locality for concurrent query execution. Our idea is based on the observation that, in a large scale decision support system, it is very common for multiple users with complex queries to hit the same data set concurrently[10], even though these queries may not be identical. MiniTasking exploits such data sharing characteristics to improve temporal locality by scheduling query execution at three levels: 1) it batches queries based on their data sharing characteristics and the cache configuration; 2) it groups operators that share certain data; 3) it schedules mini-tasks which are small fractions of operators according to their data locality without violating their execution dependencies.

MiniTasking is complementary to previously proposed solutions. First, it works well with cache conscious data layouts such as PAX[4] and data morphing[5], because MiniTasking improves temporal locality while cache conscious layouts improve spatial locality. MiniTasking is also complementary to multiple query optimization (MQO) techniques that produce a global query plan for them[11–14].

We implemented MiniTasking in the Shore storage manager[15]. Our experimental results with various DSS workloads using the TPC-H benchmark suite show that, with the traditional NSM layout, MiniTasking significantly reduces the L2 cache misses by up to 83%, and thereby achieves 24% reduction in execution time. With the PAX layout, MiniTasking reduces the L2 cache misses by 65% and the execution time of concurrent queries by 9%. On a real compound workload running TPC-H throughput testing streams, MiniTasking improves the end performance up to 20%.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 introduces data temporal locality. Section 4 describes MiniTasking in detail. Section 5 discusses the evaluation methodology and experimental results. Finally, we show our conclusions and future work in Section 6.

2 Related Work

Multiple Query Processing is very common in databases[12,14,16]. For example, in a database system enhanced with inference capabilities, a simple query involving a rule with multiple definitions may be expanded to multiple queries that have to be run over the database. As these queries may share common data, it is necessary to perform such an inter-query optimization. This problem becomes more important because of the increasing use of applications such as complex decision support queries on large sets of data, and the frequent queries in a CRM (Customer Relationship Management) system or a web query system.

Multiple Query Optimization endeavors to reduce the execution time of multiple queries by reducing duplicated computation and reusing the computation results. Previous work proposes to extract common sub-expressions from plans of multiple queries and reuse their intermediate results in all queries[11–13,16,17]. Early work shows that the multiple query optimization is an NP-hard problem and proposes heuristics for query ordering and common sub-expressions detection and selection[11,13,16]. Roy et al. propose to materialize certain common sub-expressions into transient tables so that later queries can reuse the results[18]. Instead of materializing the results of common sub-expressions, Dalvi et al. focus on pipelining the intermediate tuples simultaneously to several queries so as to avoid the prohibitive cost of materializing and reading the intermediate results[17]. Harizopoulos et al. propose an operator-centric engine Qpipe to support on-demand simultaneous pipelining[19]. O’Gorman et al. propose to reduce disk I/O by scheduling queries with the same table scans at the same time and therefore achieve significant speedups[20]. However, reusing intermediate results requires exactly the same common sub-expressions. For example, a little change in the selection predicate of one query will render previous results not usable.

Improving Data Locality is another important technique to improve performance of multiple queries, especially when the memory latency becomes a new bottleneck for DSS workload on modern processors. Ailamaki et al. show that the primary memory-related bottleneck is mainly contributed by L1 instruction and L2 data cache misses[1].

Many recent studies have focused on improving data spatial locality to reduce the cache misses in database systems[4,21–26]. Cache-conscious algorithms change data access pattern of table scan[23] and index scan[26] so that consecutive data accesses will hit in the same cache lines. Shatdal et al. demonstrate that several basic database operator algorithms can be redesigned to make better use of the cache, including several join operators[27]. Cache-conscious index structures pack more keys in one cache line to reduce cache misses during lookup in an index tree[22,23,26]. Cache-conscious data storage models partition tables vertically so that one cache line can store the same fields from several records[4,24]. Although these techniques effectively reduce cache misses within a single query, data fetched into processor caches are not reused across multiple queries.

Fine-Grained Thread Scheduling is originally proposed by Philbin et al. to improve data locality for caches[8]. Their main idea is to decompose a program into fine-grained threads and schedule these threads so