The Pennsylvania State University
The Graduate School

A CACHE TOPOLOGY AWARE MULTI-QUERY SCHEDULER FOR MULTICORE ARCHITECTURES

A Thesis in
Computer Science and Engineering
by
Umut Orhan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2011

The thesis of Umut Orhan was reviewed and approved* by the following:

Mahmut Taylan Kandemir
Professor of Computer Science and Engineering
Thesis Advisor

Mary Jane Irwin
Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

As mainstream computer chip architectures are switching from single core machines to multicore ones, it is becoming increasingly important to exploit multicore specific characteristics to extract maximum performance. One of these characteristics is the existence of shared on-chip caches, through which different threads/processes can share data (help each other) or displace each other's data (hurt each other). Most of current commercial multicore systems on the market have on-chip cache hierarchies with multiple layers (typically, in the form of L1, L2 and L3, the last two being either fully or partially shared). In the context of database workloads, exploiting full potential of these caches can be critical. Motivated by this, our main contribution in this work is to present and experimentally evaluate a cache hierarchy-aware query mapping scheme targeting workloads that consist of batch queries to be executed on emerging multicores. Our proposed scheme distributes a given batch of queries across the cores of a target multicore architecture based on the affinity relations among the queries. The primary goal behind this scheme is to maximize the utilization of the underlying on-chip cache hierarchy while keeping the load nearly balanced across affinity domains. Each affinity domain in this context corresponds to a cache structure at a particular level of the cache hierarchy. A graph partitioning-based method is employed for distributing queries across cores, and an integer linear programming (ILP) formulation are employed for addressing locality and load balancing concerns. We evaluate our scheme using the TPC-H benchmarks on two commercial multicore machines with different on-chip cache topologies. Our solution achieves up to 25% improvement in individual query execution times and 15%-19% improvement in throughput over the default Linux-based process scheduler.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to express my appreciation to Prof. Mahmut T. Kandemir for his support throughout this study.

I am deeply grateful to my colleagues, forever friends Emre Kultursay and Mahmut Sami Aktasoglu for their help and support during my graduate study.

I would also thank my dear friends Safakcan Tuncdemir, Julia Woolley, Berkay Celik, Onur Kayiran and Orhan Kislal for their motivating support and cheerful presence.

Dedication

To my family...

Chapter 1

Introduction

Growing performance gap between processors and main memory has made it worthwhile to consider off-chip data accesses in query processing [1, 2, 3]. Especially in multi-query environments, exploiting data-sharing opportunities among concurrent queries can be critical to effectively utilize the underlying shared memory hierarchy. Given a set of queries, there may be several cases where there is a common retrieval operation to the same data. A query can benefit from the data previously loaded in the cache/memory by another query. However, if these queries are scheduled independently, it is very likely that the same data is brought from off-chip memory to on-chip caches multiple times, thereby consuming off-chip bandwidth and slowing down overall execution.

Since conventional database server backends are optimized for addressing the disk-access bottleneck in single core architectures, running these servers on multicore architectures raises an important question from the *data-locality perspective*: how to schedule concurrently-running queries across available cores in order to better utilize the underlying shared memory hierarchy and improve the overall throughput of the system. It is clear that efficient resource utilization and workload scheduling is crucial for maximizing the throughput of any parallel system. Specifically, in the context of multicore systems, a multi-query scheduling method should optimize a given set of queries together by considering the cache/memory hierarchy of the target architecture.

Resource allocation and scheduling in multi-query environments are typically done by the operating system (OS). For example, Linux task scheduler is oriented

towards load balancing and can dynamically change the affinity of running processes (task migration) to utilize each core at its maximum. As it has no in-depth understanding of how database queries are processed individually, an OS scheduler may not exploit potential data sharing opportunities between two or more different queries in a shared cache. Even worse, treating database queries as ordinary processes and, consequently, scheduling them in a traditional manner may penalize concurrently-executing queries at runtime and may lead to degradation in overall system throughput. In shared-memory multicore architectures, on-chip cache hierarchy performance is a major factor as far as workload performance is considered. In fact, application behavior can dramatically change on different on-chip cache hierarchies depending on mapping and scheduling plans [4]. Moreover, cache contention due to hardware resource constraints has already been identified as a challenge that must be addressed in query processing context [5].

Our goal in this study is to make concurrent multi-query execution in conventional relational database systems effectively benefit from hardware-level parallelism provided by emerging multicore architectures and, as a result, improve the overall throughput of the system. Basically, we have two main concerns in optimizing multi-query scheduling: *affinity* and *load balancing*. If we know (i) the execution plan of each query, (ii) an estimated cost for each operator/plan, and (iii) the target multicore platform in advance, we can suggest compile-time assignments of queries to affinity domains (in our case, a set of on-chip caches depending on the underlying cache topology). These assignments can improve data locality on shared caches as opposed to dynamic, OS based scheduling and lead to significantly less cache conflicts as well as reduced number of off-chip data accesses. On the other hand, a simple compile-time multi-query scheduling scheme that relies only on data sharing relations between queries tends to ignore dynamic modulations across workloads of different processors. At runtime, core utilizations can be reduced when a static scheduling scheme is employed and we may even end up with idle cores when queries have diverse execution times. Consequently, we also need to better utilize the available processing units through load balancing.

The techniques discussed in this thesis identify common data retrieval operations in multi-query workloads and build *affinity relations* between queries that represent possible data sharing at runtime. Affinity relations are represented us-

ing an *undirected weighted graph*, where each node represents a query and each edge between two nodes indicates possible data sharing among the corresponding queries. Edge weights are calculated from the query plan estimations provided by the query optimizer. Using this graph, we then invoke a *hierarchical clustering method* to generate *query-to-affinity domain mappings*. An *affinity domain* in this context refers to a particular cache structure at a specific level of the cache hierarchy. It can be a private cache or a cache shared by multiple cores. According to the generated mappings, each query is executed only on the cores that are connected to the corresponding affinity domain.

Our clustering method creates partitions starting from bottom cache level (close to main memory) until it has the same number of partitions as the number of target affinity domain levels. More specifically, the method tries to create the exact number of partitions as requested while maximizing the total edge weight within a partition (i.e., the amount of data sharing) and minimizing the total weight of cutting-edges. When moving to upper levels, the method takes the parent partition and divides it into the same number of available caches in the upper level. We further enhance this scheme by introducing vertex weights to model runtime working memory requirements of queries so that we can balance queries and reduce cache thrashing.

Our proposed clustering method works as expected when the number of queries in the given workload is equal to or less than the available cores in the target architecture. In such a case, a particular core can be dedicated to a single query. However, when we increase the workload size, static affinity domain-query mappings can result in idle cores at runtime, especially when the queries in the workload have diverse execution times. A workload on an affinity domain may be finished before other domains, and consequently, the overall system utilization gets reduced compared to a dynamic OS-based scheduler since static mapping does not consider runtime reassignments. Motivated by this observation, we extend our clustering approach with an integer linear programming (ILP) based load balancing step where we try to balance the loads assigned to different affinity domains.

We implement our scheduling scheme as a middleware in PostgreSQL 8.4 [6], which takes a batch of queries to be executed in parallel and the cache topology information of the target multicore architecture as inputs. As a motivating point,

this kind of batch scheduling schemes can be applied into real-world scenarios where several database users run a fixed set of queries for generating daily reports from a data warehouse. Hence, we evaluate our approach with workloads consisting of OLAP queries provided by the TPC-H benchmarks [7].

There are three main contributions presented in this thesis:

- We present a *cache topology aware* multi-query scheduling scheme for multicore architectures. This approach defines affinity relations between queries and assigns closely related queries into similar affinity domains in order to effectively utilize the on-chip cache hierarchy by exploiting data locality throughout the cache hierarchy.

- We explain how this scheduling strategy can be extended to reduce cache thrashing effects of concurrent queries sharing the same cache structures as well as to tolerate load balancing concerns brought by static affinity domain mappings.

- We report experimental data (obtained by a PostgreSQL based implementation) that show the effectiveness of our proposed mapping scheme.

Our experimental results indicate that the proposed algorithm achieves up to 25% improvement in query execution time and 15%-19% improvement in overall system throughput. To the best of our knowledge, this is the first work that considers the disparities between different on-chip cache topologies for scheduling multiple queries in multicore architectures.

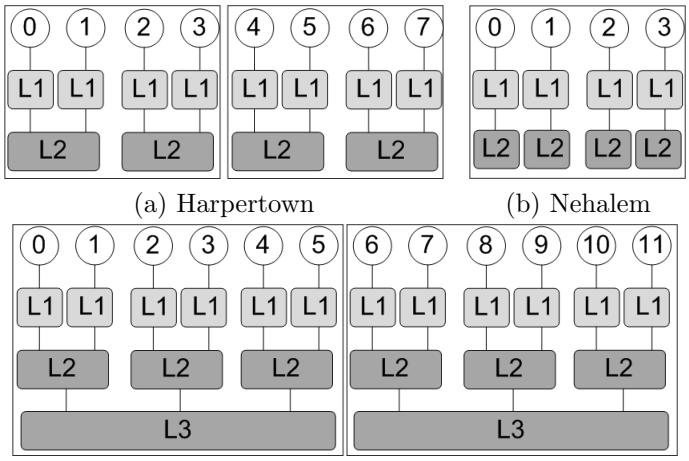
In the next section, a brief background on multicore architectures and data reuse is given. A detailed comparison of proposed approach with the prior related efforts can be found in Chapter 2.2. Chapter 3 presents the details of the proposed multi-query scheduling scheme. In Section 4, an experimental evaluation of this scheme on commercial multicore machines is given. The thesis is concluded in Section 5 with a summary of our major observations and possible future research directions.

Chapter 2

Background

2.1 Multicore Architectures and Data Reuse

Today, we are witnessing an on-chip multiprocessing revolution. The number of cores (processors) packed in a single chip as well as the on-chip cache space shared by these cores are rapidly increasing. In addition, chip multiprocessors paved the way to alternative cache topologies, which means that cache memories can be connected to on-chip cores in a multi-leveled fashion by exhibiting various different patterns. Intel's Dunnington [8] and Harpertown [9] architectures are good examples of such diversity. Dunnington has six on-chip cores whereas Harpertown has four cores. Both architectures have an L1 cache per core and L2 caches are shared by a pair of cores. However, Dunnington adds one more level to Harpertown's cache hierarchy and introduces L3 cache. On the other hand, architectures such as Intel Nehalem [10] can have a completely different topology with private L2 caches. All these three multicore machines have distinct on-chip cache hierarchies which are shared across different number of cores, as illustrated in Figure 2.1. Today, a server rack can contain more than one of these chips, resulting in parallel systems with large number of cores. One of the main distinguishing characteristics of multicore architectures is the existence of *shared on-chip caches*. Shared caches are preferable to their private alternatives especially when we consider (i) efficient utilization of cache space and (ii) avoiding data redundancy across caches. Cache memories play a significant role in determining runtime performance of concurrent applications. In particular, depending on their access patterns, cache sharing



(a) Harpertown (b) Nehalem

(c) Dunnington

Figure 2.1: Different multicore cache topologies. Cores are represented using ovals. On-chip L1, L2 and L3 caches are denoted using labels. Harpertown and Dunnington consist of two homogenous sockets having the same number of processing units and caches. In contrast to other two architectures, each core has a private L2 in Nehalem.

among two processes/threads can be constructive or destructive [11, 12, 13]. Shared caches can lead concurrent applications running on different cores to contest for the available space. In other words, an application, process or thread executing on a particular core can be slowed down by a *co-runner* which uses the same cache space at the same time through a different core. As a result, one can expect that scheduling decisions on multicore architectures can dramatically change the overall system performance. In order to avoid such contentions, one must find an appropriate match of processes. We call this challenge as *the application-to-core mapping* problem throughout the thesis. Recently, cache-aware process/thread scheduling has become an active research area, primarily dealing with the application-to-core mapping problem [11, 14, 15].

Shared caches make use of the property of data reuse in applications. A data reuse is an access to a memory location that has already been accessed previously. The ability of a cache in converting a data reuse into a *cache hit* depends on (i) the capacity of the cache and (ii) the distance at which the reuse occurs, namely, the *reuse distance*. Reuse distance is defined as the number of "unique" memory locations accessed between two contiguous accesses to the same memory location.

In this study, we approach the application-to-core mapping problem from a

different abstraction level. We use the term *affinity domain* for referring to a cache structure and each *affinity domain level* covers all caches at that level, e.g. affinity domain level 2 includes all L2 caches whether they are private to a particular core, pairwise shared by two cores, fully shared or placed in a different socket. By doing this, we can model cache topologies hierarchically and work at different levels of granularity instead of just working on core level.

We should note that for shared memory multiprocessor systems consisting of single-core processors, the application-to-core mapping problem has already been attacked from the data locality perspective [16]. With the advent of many-core architectures, this mapping problem is becoming more and more important and will become vital in the near future, due to increasing core counts and deeper/more complex cache hierarchies.

2.2 Related Work

Batch scheduling and resource allocation problems have been studied in the scope of parallel database systems [17, 18]. In comparison to these prior efforts, our work specifically targets emerging multicore platforms and hence, the problem of optimizing data locality in shared on-chip cache hierarchies.

Historically, query processing is optimized for better utilizing the DBMS's buffer pool in order to reduce the number of I/O operations. However, rising memory-wall concerns forced researchers to focus on off-chip data access too. Several studies are presented for making query processing and operators aware of on-chip cache spaces for both single CPU [19] and contemporary architectures [20, 21, 22]. In addition to query operators and processing algorithms, Stonebraker et al. [23] and Boncz et al. [24] introduce tuple access and storage optimizations in order to cope with the memory access bottleneck.

Various data sharing approaches for concurrent database queries are studied in the literature. Harizopoulos et al. [25] present a pipelined query engine where a single data retrieval operation serves more than one queries in parallel. Alternatively, in [26, 27, 28], work sharing opportunities through exploiting common operators across concurrently running queries are discussed. The goal of our multi-query scheduling scheme is similar to these work sharing approaches from data locality

perspective. However, we focus more on the issues arising due to different on-chip cache topologies. Extending our approach with these expert work sharing based approaches can further improve data locality through all levels of the on-chip cache hierarchies in multicores.

There are a number of studies proposed so far addressing cache partitioning problem from different perspectives [29, 30, 31, 5, 32]. In general, one can take advantage of sharing and/or isolation by explicitly partitioning shared caches, and as a result, minimize possible cache conflicts in multi-threaded environments. Lee et al. [5] specifically target database queries sharing same on-chip cache structures in multicore architectures. They introduce an OS-level cache partitioning scheme which is based on data access patterns and working memory requirements of the given workload queries.

Our approach is complementary to prior studies on data-sharing, storage and query operator optimizations for shared cache architectures. We believe that supporting them with our scheduling scheme can further improve query latencies and overall system throughput. Moreover, cache partitioning schemes can favor our affinity domain-query mapping based scheduling scheme, especially when constructive cache isolation for mappings cannot be provided on a specific topology by default. They can introduce alternative affinity domains to existing hardware directed cache structures and provide them as input to our multi-query scheduler.

Chapter 3

Cache-Aware Query Scheduling

3.1 Motivation

To motivate the need for a cache-aware multi-query scheduler, we analyze a sample query workload consisting of four TPC-H queries that are scheduled on a Dunnington architecture with two different query-to-affinity domain mappings. In both these mappings, all queries use dedicated affinity domains at level 2, which refer to four distinct L2 caches in Figure 2.1c. In the first mapping, queries 1, 2, 3 and 4 execute on cores 0, 2, 4 and 6, thereby the first three queries sharing one of the level 3 affinity domains (L3 cache) and the query 4 being left alone. On the other hand, in the second mapping, queries 1, 2, 3 and 4 are mapped to cores 0, 6, 2 and 8. As a result, queries 1 and 3 share a particular affinity domain at level 3 and queries 2 and 4 share the other.

Initially, what was expected from these two mappings is similar execution time results since we only changed the order of affinity domain assignments at level 2. The load among processors were kept same between the two mappings. However, after the experiments, we observed that these two mappings generate very different execution time results on the Dunnington system. Specifically, Figure 3.1 shows the normalized execution times of these queries under the two mappings. We observe significant performance degradations of 41% and 114% in queries 2 and 4 respectively, which are due to the cache-unaware nature of the second mapping. Moreover, we can see that query 1 ad query 3 were not affected at all.

One can conclude from these results that in multi-query environments, utiliza-

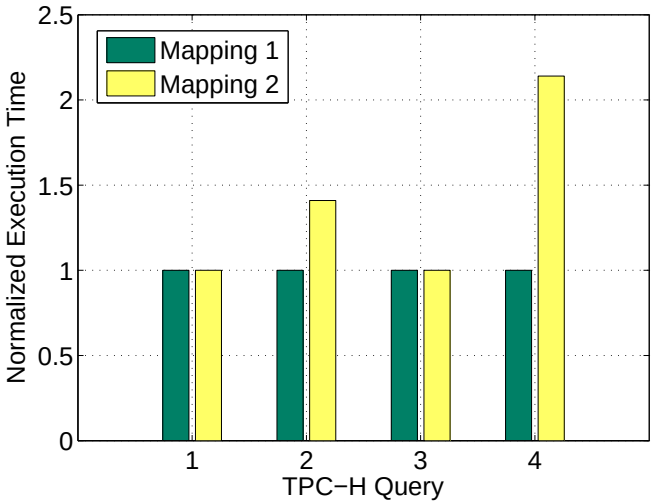


Figure 3.1: Execution times of concurrent queries on Intel Dunnington architecture under two different mappings. For each query, the second bar is normalized with respect to the first one.

tion of the shared caches can be an important factor that shapes overall performance. Therefore, memory access behaviors of concurrent queries must be taken into account in order to avoid possible cache contentions and convert inherent data reuse across queries into locality in the shared cache space (that is, accesses to shared data are performed while the data still resides in the shared on-chip cache space). In the rest of the thesis, we model and estimate memory access behaviors of queries by analyzing the generated query plans and we try to better utilize shared caches through cache aware query scheduling decisions.

3.2 Proposed Scheduler

3.2.1 Problem Definition and High-Level View

Our goal in this study is to present and evaluate a scheduling algorithm which assigns queries of a given batch job to the affinity domains in the target multicore architecture in a cache conscious manner. This cache hierarchy-aware scheduler can reduce possible cache contentions among concurrent queries and improve the overall throughput of the system. We define this query-to-affinity domain mapping problem more formally as follows. A query (q_i) to affinity domain (D_j) mapping

at level L is defined as:

$$M(L) = \{(q_i, D_j) \mid 1 \leqslant i \leqslant n_q, 1 \leqslant j \leqslant n_L\}, \tag{3.1}$$

where n_q denotes the number of queries and n_L denotes the number of caches at level L of the target cache topology.

Our scheduling algorithm takes two inputs; a set of query plans to be executed as well as the target cache topology of the architecture where these queries are processed. The main goal of the algorithm is to decide which query should be executed on which affinity domain. It tries to evenly distribute the queries among available cores while maximizing possible data sharings through shared caches.

In Figure 3.2, we give the high-level description of our automated approach to cache topology aware query scheduling. In the first step, we invoke the PostgreSQL Query Planner and Optimizer. We then analyze the generated query plans to extract possible data sharing opportunities across queries and estimate the amount of memory consumed by each query. In this step, we build a *graph structure* representing the data sharing relationships between queries with respect to cache behavior and then partition this graph considering the target architecture and the affinity domain level. Finally, we try to balance the load on each affinity domain according to the estimated query execution times.

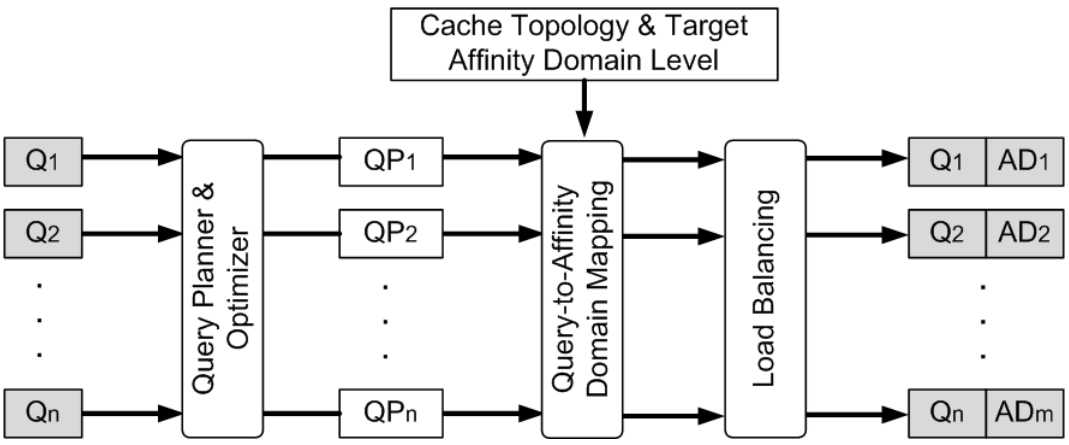


Figure 3.2: High level sketch of our cache topology aware query scheduling approach.

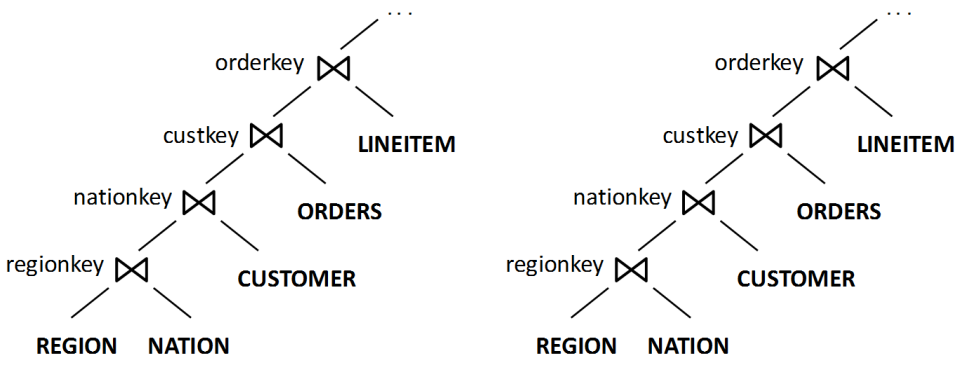
3.2.2 Assumptions

For this study, we employ *hash join* as our default join operator. This is because, instead of using nested loop or sort merge joins, PostgreSQL mostly prefers hash joins for executing TPC-H queries when no indices are introduced to the system. Further, enabling hash join allows us to test our approach in the presence of private data structures generated by queries at run-time such as hash tables. Each hash table belongs to a particular query and is not shared with other queries. These in-memory tables tend to have short reuse distances during join processing, thus, besides aggregations, they can easily jeopardize the benefits of on-chip caches by causing contention especially when the cache is used by other hash joins at the same time [5, 27].

A conventional hash join operation consists of two consecutive steps: *building* and *probing*. In the building phase, a hash table is created from the rows of the smaller relation or from the results of another join. Afterwards, the other relation is scanned and suitable rows are joined with the ones found in the hash table. The building phase is materialized in the classical hash join method, i.e., probing step is started right after finishing the construction of the hash table.

Despite their drawback of extra memory consumption, we can take advantage of hash join operations for join processing in exploring data sharing opportunities. Specifically, with a query optimizer favoring the left-deep query plans, the materialized nature of a hash join operation can be exploited to maximize data reuse between concurrent queries that are working on same relations. In such a case, scan operations within a query are likely to be executed in the reverse order of the cardinalities of the relations that they scan. When the selectivities on shared relations are similar, chances of finding the data in an on-chip cache which was once brought in by another query can be improved. As an example, in Figure 3.3, query plans of TPC-H query 5 and query 8, which are generated by PostgreSQL query optimizer, are given. One can figure out that these two different query plans have same hash join ordering decisions for same relations.

In this study, we statically assign queries to affinity domains and do not handle changes that might happen over time. The proposed technique is dependent on the query optimizer of the database system. The results can be hindered by the wrong selectivity or execution time estimates. Especially in highly concurrent



(a) TPC-H Q5 (b) TPC-H Q8

Figure 3.3: Ordering in hash join chains largely depends on relation cardinalities. In this example, we have $|LINEITEM| > |ORDERS| > |CUSTOMER| > |NATION| > |REGION|$.

environments, not only the selectivities but also the execution frequencies of the queries might change at run-time. As our future work, we are planning to consider the dynamic nature of the query execution as well.

3.2.3 Estimating the Amount of Shared Data between Two Queries

A query reads the data stored on a DBMS through scan operations. For example, a sequential scan operation fetches all tuples of a relation starting with the first tuple. Therefore, we can represent the data that a query reads during its lifetime as a set of relations R:

$$\mathcal{R} = \bigcup r, \tag{3.2}$$

where r denotes a scanned relation. At this point, one can approximate the total amount of data shared between two queries as follows;

$$DataSharing = \sum_{r_i \in \mathcal{R}_1, \mathcal{R}_2} |r_i|\,, \tag{3.3}$$

where $\mathcal{R}_1$ and $\mathcal{R}_2$ represent two set of relations read by distinct queries, and r_i is the relation that scanned by both of these queries. Note that, we calculate the amount of data sharing in terms of tuples, instead of using the actual size of the

stored data in bytes.

With unlimited cache sizes, *reuse distance* of shared data would not be of any concern. Consequently, once a tuple is brought into the on-chip cache, it would not be kicked out due to a capacity miss and, after the very first miss, any read request for this tuple would be a hit. However, in real-world settings, we must consider the distance between two read operations to the same tuple, as cache capacities are limited. If the distance between two scan operations which read tuples of the same relation is significantly large, then leading scan may displace all existing tuples from the cache and replace them with newer ones before the lagging scan can access them. As a result, these queries may not benefit from data sharing. In other words, if the same data is read by two queries at completely different points in time, then the amount of data shared between the queries might be zero.

In order to address this timing issue, we enhance our initial data sharing model by considering the *selectivity* of each scan operation. Selectivity in a scan operation is defined as the percentage of the filtered tuples over the total number tuples in the relation. During our experiments, we observed that the execution time of a scan operation is related to its selectivity and hence, two scan operations having similar selectivities are more likely to share tuples brought into a cache. On the other hand, even if we satisfy that join orderings for the same tables are the same, scan operations may not necessarily occur at the same level of the associated query plan trees. For example, one of the queries might work on a completely different data first, and compute a join among shared relations as the rest. To consider such cases, we calculate vertical differences between scan operations and enable this information for data sharing estimations. Accordingly, we change our data sharing model to:

$$\begin{aligned} &\Delta(LevelDifference) = 1/(1 - |\eta_1 - \eta_2|), \\ &DataSharing(revised) = \sum_{r_i \in \mathcal{R}_1, \mathcal{R}_2} (|r_i| * \Delta * (1 - |\sigma_1 - \sigma_2|)), \end{aligned} \tag{3.4}$$

where σ_1 and σ_2 represent the selectivity of two scan operations, and each η gives the order of a scan operation according to the postordered query plan tree. In our framework, we elicit the selectivity information by parsing the query execution plan of a given query where scan operations are associated with estimated cost

and the number of the resulting tuples.

3.2.4 Estimating the Working Memory Sizes

The overall goal of our affinity-based query clustering method is to reduce the amount of memory stalls experienced due to cache misses. Even when several concurrent queries with data sharing are executed on cores, cache thrashing may hinder the benefits of any data sharing. Cache thrashing occurs when the data structures required by each query, such as aggregations and hash tables, overflow the cache. Thus, any data sharing optimization in concurrent query execution needs to target at reducing thrashing effects of non-shared data structures as much as possible.

In order to minimize the cache thrashing effects of the working memory, we estimate the necessary and sufficient amount of memory space that required by a query during its lifetime. We perform this estimation by exploiting query plan definitions produced by the query optimizer. A node in the PostgreSQL's query plan is associated with the estimated execution time, the number of tuples returned, and the width of a returned tuple in size of bytes. Hence, we can have a general idea about the size of working memory allocated for hash table and aggregation table nodes individually by multiplying the cardinality of the returned tuple set and the corresponding width value. Cache trashing is more likely to occur during the peak memory consumption periods, and therefore, we estimate the upper bounds of the working memory.

We estimate the peak working memory size in the worst case by comparing the largest hash table created during the lifetime of the query with the sum of the estimated working memory sizes of the pipelined stages. In most of our benchmark queries, pipeline stages consist of an aggregation and a generation of the intermediate results that are supplied to this aggregation. These intermediate results are typically generated after a hash join. We can therefore estimate the peak working memory size of a query as:

$$H = max(\bigcup |h|), P = |k_i| + |a|, WMS = max(H, P) \tag{3.5}$$

where P denotes the sum of aggregation table size ($|a|$) and its inputs (k_i), H is

the size of the largest hash table created among all other hash tables (hs), and WMS is the estimated working memory capacity required for this query.

Consider, for example, the query plan given in Figure 3.4, produced by PostgreSQL's query optimizer for TPC-H Q12. In this query, a hash table on *ORDERS* relation is built first. As indicated in the plan node (*Hash* node at line 6), this table has 1.5M rows, each of which is 20 bytes, resulting in a table size of nearly 28.6MB. In the pipelined execution, results fetched in the join operation are provided to the aggregation operation. The sum of the working memory required in the pipelined stages are calculated and found to be less than the size of the hash table generated in the beginning. Consequently, WMS in this case is equal to 28.6MB.

```
                          QUERY PLAN
-------------------------------------------------------------
(1) Sort  (cost=329380.96..329380.97 rows=1 width=27)
     Sort Key: lineitem.l_shipmode
(2)  -> HashAggregate(cost=329380.93..329380.95 rows=1 width=27)
(3)   -> Hash Join(cost=68238.00..329179.33 rows=26879 width=27)
(4)    Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
(5)     -> Seq Scan on lineitem(cost=0.00..252082.75 rows=268...
         Filter: ((l_shipmode = ANY ('{MAIL,RAIL}'::bpchar[])...
(6)       -> Hash(cost=41431.00..41431.00 rows=1500000 width=20)
(7)        -> Seq Scan on orders(cost=0.00..41431.00 rows=150...
```

Figure 3.4: An example query plan for TPC-H Q12.

3.3 Query-to-Affinity Domain Mapping

In this section, we first describe our cache topology aware multi-query scheduling scheme that uses the estimated amount of data shared among queries. We then enhance this scheme through minimizing cache conflicts by considering the working memory sizes of queries and balancing the loads across different affinity domains. Pseudo-code for the proposed scheduler is provided in Algorithm 1.

3.3.1 Exploiting Data Locality and Avoiding Cache Conflicts

We start with building an undirected weighted graph where each query is represented as a vertex. An edge between two vertices has a weight equal to the

Algorithm 1 Cache Topology Aware Query Scheduling

1: $QP = \{q_0, ..., q_n\}$: query plans
2: T: cache hierarchy tree
3: K: number of available cores/affinity domains
4: G: a graph consisting queries as vertices
5: $CS = \{cs_0, ..., cs_{k-1}\}$: cluster set (affinity groups)

6: **procedure** SCHEDULER(QP, T, K) ▷ Main Routine
7: $G \leftarrow BuildGraph(QP)$
8: $CS \leftarrow Partitioner(G, T)$
9: $CS \leftarrow LoadBalancer(CS, QP, K)$ ▷ ILP Solver
return CS
10: **end procedure**

estimated amount of data sharing using the technique presented in Section 3.2.3.

To avoid cache thrashing effects of overflowed working memories on a shared cache as much as possible, one has to consider the total amount of memory space allocated to the concurrent queries assigned to a particular affinity domain. In order to model working memory requirements, we slightly modify our graph structure and introduce *vertex weights* representing the working memory sizes of queries. Pseudo-code for how we build this graph structure is given in Algorithm 2.

After representing queries and the potential data sharing opportunities as a graph, we cluster the vertices/queries based on the cache topology. An on-chip cache topology can be modeled using a *tree* where the last level on-chip cache is the root and the first level caches are the leaves. For a 2-socket system with 2 last level caches such as in Figure 2.1c, a virtual root is used. Our clustering algorithm partitions queries starting from the root level moving towards the leaf level caches. At each level, a *k-way* partitioning takes place where k is equal to the number of child nodes. In other words, the number of generated partitions in each level is equal to the number of child nodes in the cache hierarchy tree. When the algorithm terminates, we have the same number of partitions as the number of domains available at the target affinity level and each query is assigned to a particular partition. A *k-way* graph partitioning problem [33] can be defined more formally as:

> For a given graph $G(V, E)$, find a set of graphs such as
> $P = \{G_0(V_0, E_0), ..., G_{k-1}(V_{k-1}, E_{k-1})\}$, where $\bigcup_0^{k-1} V_i = V$ and

Algorithm 2 Building Graph Structure

```
1: QP = {q_0, ..., q_n}: query plans

2: procedure BuildGraph(QP)
3:     V ← ∅
4:     E ← ∅
5:     for all q ∈ QP do
6:         |V_q| ← Work_Mem(q)
7:         V ← V + {V_q}
8:     end for
9:     for all v_i ∈ V do
10:        for all v_j ∈ V do
11:            sharing ← Data_Shared(v_i, v_j)
12:            if sharing > 0 then
13:                E_k ← AddEdge(v_i, v_j)
14:                |E_k| ← sharing
15:                E ← E + {E_k}
16:            end if
17:        end for
18:    end for
19:    G ← {V, E}
   return G
20: end procedure
```

$$\forall i, j, i \neq j \rightarrow V_i \cap V_j = \emptyset.$$

In general, the *k-way* graph partitioning problem is associated with a cost function. The goal of this partitioner is to minimize this cost function. One common cost function is the sum of inter-partition edge weights that are spanning more than one partitions. In our approach, we try to group queries which are working on the same data more than the others. We achieved our goal through representing the data sharing amount as the edge weight and minimizing the cost function.

For the implementation of *k-way* partitioning, we employ a well-known graph library based on the multi-level recursive bisectioning algorithm presented in [34]. In brief, a multi-level partitioning algorithm can be divided into three distinct phases. The first phase, called *coarsening*, groups the connected vertices of the graph into a bigger vertex to form a coarser graph which contains a smaller number of vertices than the original graph. Coarsening is performed iteratively at multiple levels and the graph is shrinked at each level. At each level, coarsening is done by finding

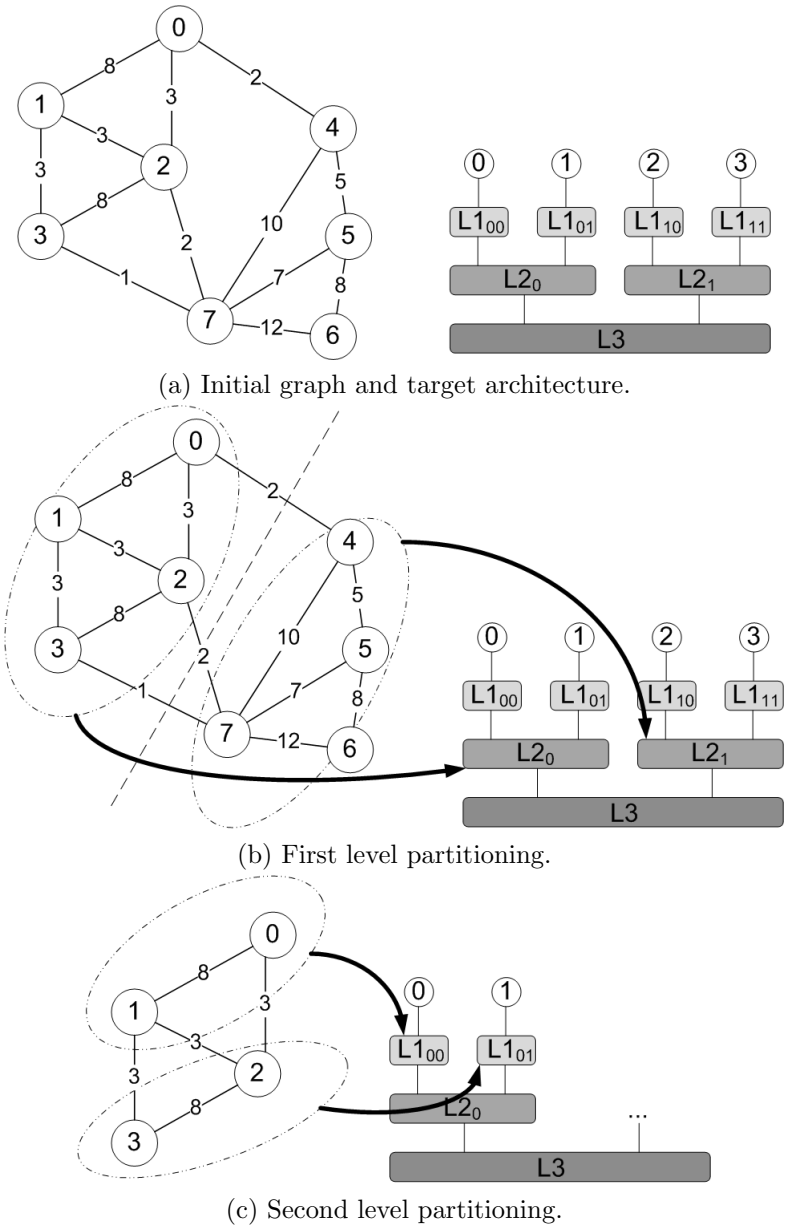
Algorithm 3 Graph Partitioning

```
1: T: cache hierarchy tree
2: G: a graph consisting queries as vertices
3: CS = {cs_0, ..., cs_{k-1}}: cluster set (affinity groups)

4: procedure PARTITIONER(G, T)
5:     CS ← ∅
6:     if isLeaf(T) then
7:         V ← Vertices(G)
8:         for all v ∈ V do
9:             CS ← CS + {v, T}
10:        end for
11:    else
12:        k ← NumClusters(T)
13:        Partitions ← MultiLevelPartitioning(G, k)
14:        for all p ∈ Partitions do
15:            t ← LevelUp(T)
16:            CS ← CS + Partitioner(p, t)
17:        end for
18:    end if
   return CS
19: end procedure
```

a maximal matching through using *heavy-edge matching* algorithm. Coarsening is finished when it reaches the smallest graph, called the top-level graph. In the second phase, a *2-way partitioning* is applied to this top-level graph. Finally, starting from the top-level, each partition is *projected* to upper levels. Coarsening, top-level partitioning, and refinement are all tunable and can be performed using different strategies.

After associating weights with vertices in our original graph along with the edge weights, we modify our cost function in order to minimize the cutsizes of the partitions and balance the sum of the vertex weights in each partition. The multi-level recursive bisection algorithm handles weighted vertices as a balancing constraint in the top-level partitioning phase. Vertices are ordered according to their weights and assigned to partitions satisfying the balancing constraint. Next, the partitioner tries to obtain roughly equal partitions according to the sum of vertex weights while minimizing the edge-cut. Pseudo-code for the proposed graph partitioning approach is given in Algorithm 3.



(a) Initial graph and target architecture.

(b) First level partitioning.

(c) Second level partitioning.

Figure 3.5: Example application of our scheme.

3.3.2 Load Balancing

Although *k-way* partitioning heuristics are able to produce k nonempty partitions, they cannot guarantee balanced query workloads across different affinity domains after applying only the partitioning algorithm. Thus, we need to balance the loads (i.e., the average number of cycles to process queries assigned to each partition)

	Harpertown
Number of Cores	8 cores (2 sockets)
Clock Frequency	3.0GHz
L1	32KB, 8-way,64-byte line, 3 cycle latency
L2	6MB, 24-way, 64-byte line, 15 cycle latency
L3	-
Off-Chip Latency	∼320 cycle

Table 3.1: Important features of our Harpertown machine.

	Dunnington
Number of Cores	12 cores (2 sockets)
Clock Frequency	2.4GHz
L1	32KB, 8-way, 64-byte line, 4 cycle latency
L2	3MB, 12-way, 64-byte line, 10 cycle latency
L3	12MB, 16-way, 64-byte line, 32-40 cycle latency
Off-Chip Latency	∼120 cycle

Table 3.2: Important features of our Dunnington machine.

explicitly across affinity domains. For this, we adopt a 0-1 ILP formulation to balance the query loads mapped onto affinity domains. Table 3.3 gives the constant terms used in our ILP formulation. Note that, the loads given in this table are normalized using the minimum amount of processing load that can be allocated to an affinity domain.

For each query, we define 0-1 variables to specify the assignment of a query to an affinity domain. Specifically, we define:

- $X_{t,q,m}$: to indicate whether affinity domain t and instance q of that domain is assigned to core m.

Constant	Definition
M	Number of cores
T	Number of affinity domain
I	Number of queries for affinity domain
$L_{t,q}$	Load for a given affinity domain t and query q
Q	Load imbalance coefficient

Table 3.3: The constant terms used in our ILP formulation. These are either architecture specific or workload specific.

We use a variable for each one of the possible assignments. If this 0-1 variable is 1, this indicates that the corresponding query can be assigned to core m. If this size is 0 then we conclude that this assignment does not exist.

We use another 0-1 variable for checking whether two different queries of the same affinity domain is assigned to the same core:

- S_{t,q_1,q_2} : indicates whether query q_1 and q_2 of affinity domain t can be assigned to the same core.

We use AL, a non 0-1 variable, to express the total assigned query load onto each core:

- AL_m : indicates the amount of load assigned to core m.

After defining the variables in our ILP formulation, now we explain the necessary constraints.

Each query must be assigned to a particular core, captured by the constraint:

$$\sum_{k=1}^{M} X_{t,q,k} = 1, \quad \forall t, q. \tag{3.6}$$

Two queries are said to be assigned to the same core if the following constraint holds.

$$S_{t,i_1,i_2} >= X_{t,i_1,m} + X_{t,i_2,m} - 1, \quad \forall t, i_1, i_2, m, \text{where } i_1 \neq i_2. \tag{3.7}$$

If both i_1 and i_2 queries(affinity domain t) are assigned to the same core (m), then 0-1 variable S_{t,i_1,i_2} will be forced to have a 1 value.

A necessary constraint is related to the load balancing in the query mapping between affinity domains which will prevent overloading of a processor pair with running related queries. To capture this, we use variable AL_m to indicate the total assigned query load onto the core m. The estimated load of a particular query can be extracted from associated query plan derived by the query optimizer.

$$AL_m = \sum_{i=1}^{T} \sum_{j=1}^{I} X_{i,j,m} \times L_{i,j}, \quad \forall m. \tag{3.8}$$

This equation essentially collects all the assigned query loads to generate the total

load of the core. This variable is then used for limiting the discrepancies between different cores. More specifically;

$$AL_{m1} - AL_{m2} < AL_{m2} \times Q, \;\; \forall m1, m2, \text{where } AL_{m1} > AL_{m2}. \tag{3.9}$$

Note that, Q is given as a percentage in Table 3.3.

Having specified the necessary constraints in our ILP formulation, we next give our objective function.

$$C = \sum_{i=1}^{T} \sum_{j=1}^{I} \sum_{k=1}^{I} S_{i,j,k}, \text{ where } j \neq k. \tag{3.10}$$

Based on this formulation, our 0-1 ILP problem can formally be defined as one of "maximizing C under constraints (3.6) through (3.10)."

After the clustering phase, we analyze the query-to-affinity domain mappings and check whether there is an overloaded affinity domain or not. *Overload* simply refers to the case when the difference between the amount of loads assigned to two affinity domains are greater than the fixed load balance threshold. To calculate the load on an affinity domain, we use the sum of query execution time estimations extracted from the corresponding query plans. When we detect an overloaded affinity domain, we try to group it with the affinity domain that has the minimum amount of load in order to exchange queries between domains. If these grouped affinity domains are not overloaded after query transfers/exchanges then load balancing is considered successful and we update query-to-affinity domain mappings according to these new assignments. Otherwise, we leave the overloaded affinity domain as it is and move to the next overloaded affinity domain to try to apply the same logic. Pseudo-code for the load balancer is provided in Algorithm 4.

3.3.3 Example

Consider the query graph shown in Figure 3.5. For illustration purposes, we take only affinities into consideration and have no weights assigned to the vertices. Edges are weighted according to the amount of data sharing between queries. If two queries, such as 2 and 4, do not work on common records, we have no edge between them. It is to be noted that, in this example, our target affinity level

Algorithm 4 Load Balancer

```
1: QP = {q_0, ..., q_n}: query plans
2: K: number of available cores/affinity domains
3: CS = {cs_0, ..., cs_{k-1}}: cluster set (affinity groups)

4: procedure LoadBalancer(CS, QP, K)
5:     results ← runILPSolver(CS, QP, K)
6:     CS ← updateAssignments(CS, results)
7:     for all cs_i ∈ CS do
8:         if isOverloaded(cs_i, K) > 0 then
9:             cs_min ← getMinimum(CS)
10:            grouped ← group(cs_i, cs_min)
11:            trial ← runILPSolver(grouped, QP, K)
12:            if hasSolution(grouped, QP, K) > 0 then
13:                CS ← updateAssignments(CS, trial)
14:            end if
15:        end if
16:    end for
   return CS
17: end procedure
```

corresponds to L1 caches thus, we need to carry out a two-level partitioning.

We now go over our hierarchical query clustering scheme. Since the L3 cache is shared by all cores and is the root of the cache hierarchy tree, the first step is to cluster the query groups for the L2 cache. The graph and the assignment after the first level of clustering and query mapping are shown in Figure 3.5b. Next, the query distribution is applied to each of the two clusters formed in the previous step. After this second and final level of clustering and load balancing, the query clusters are assigned to target affinity domains, as depicted in Figure 3.5c.

Chapter 4

Experiment Set Up and Results

4.1 Experimental Evaluation

4.1.1 Setup

For our experiments, we used two commercial multicore machines from Intel. Important architectural features of these machines are given in Table 3.1 and Table 3.2; their on-chip cache hierarchies are already shown in Figure 2.1.

In order to perform our experiments, we prepared two query workloads listed in Table 4.1. Single-core performance statistics of these queries are given in Table 4.2. They were compiled from TPC-H benchmarks and each of which exhibits distinct workload characteristics based on the results from the TPC-H benchmarks analysis [35]. We ran these workloads on a data set size of 1GB by various number of clients depending on the target experiment goal. All experiments are repeated five times and the presented results represent the average values of these multiple runs. In our experimental setting, each client fires only one query from the corresponding workload. Workloads are provided to the system in batches. In a closed-queueing network, system requests a new workload after each job in the previous batch is terminated. Since we know that increasing data set size does not require any architectural performance changes [3], we omit this parameter from our sensitivity analysis. We used PostgreSQL 8.4 installed in Linux 2.6 kernel without changing its default configuration. Since PostgreSQL heavily relies both on its buffer pool and the underlying operating system's file I/O cache, we warmed up these buffers before

ID	Queries (TPC-H)	Description
WL-1	2,3,4,5,7,8,9,10,11,12,13,14	Join bound
WL-2	1,2,3,4,5,6,7,8,9,10,11,12	Scan - Join Mixed

Table 4.1: Our workloads (query mixes).

Query	L2 Misses	L3 Misses	Time
Q1	665733307	1499043	54.0 sec
Q2	1786521801	74727320	470.1 sec
Q3	76269478	4053419	8.1 sec
Q4	64830065	4282042	19.1 sec
Q5	107732160	8138113	22.3 sec
Q6	34973583	437269	25.5 sec
Q7	92209232	4393086	29.7 sec
Q8	146264773	22806660	9.1 sec
Q9	253157708	13392932	22.0 sec
Q10	98513105	5249866	35.5 sec
Q11	17113140	533485	39.9 sec
Q12	46310994	2413886	23.7 sec
Q13	87072585	1180138	12.1 sec
Q14	27223301	425294	3.2 sec

Table 4.2: Performance evaluation parameters of the benchmark queries. Each column gives the absolute values collected when the query is executed on a single core of the Dunnington machine from start to end.

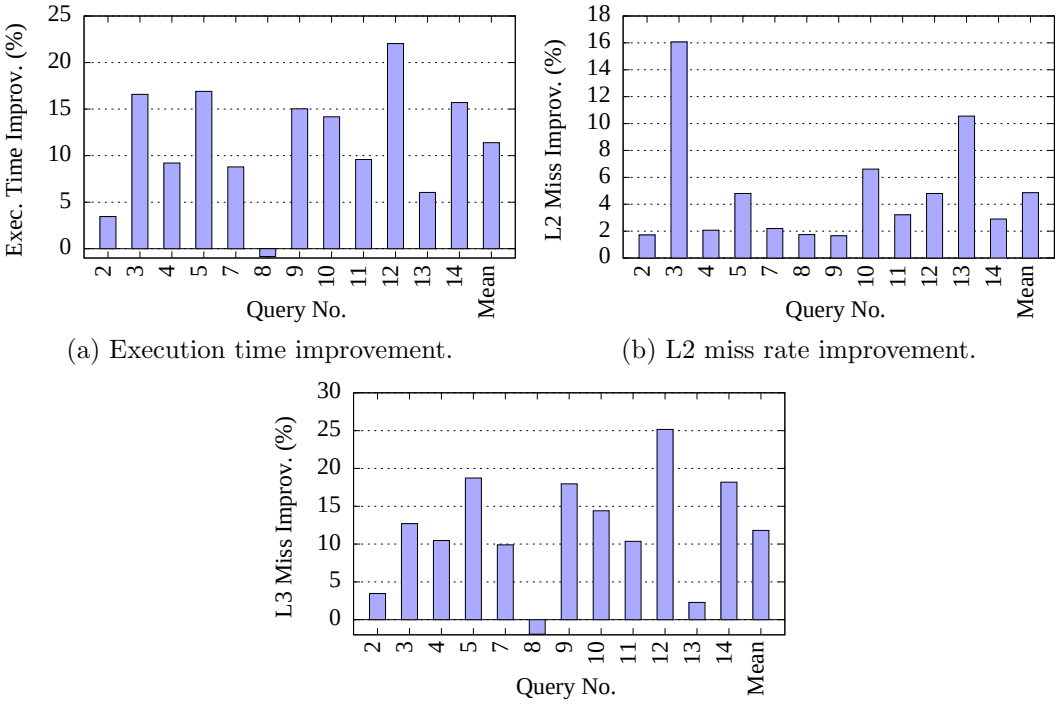
starting to collect results. PostgreSQL handles each client as a separate process, and therefore, we compare our cache topology aware scheduler against the Linux process scheduler. In our experiments, the load imbalance coefficient mentioned in the given ILP formulation is set to 10%. Architectural results are collected by perfmon2 [36] from hardware counters. Since we use a database system running a process per query, query mappings onto affinity domains are forced through *taskset()* system call on particular processes.

4.1.2 Results

Results presented in this section are all *normalized* with respect to the results obtained by using the standard Linux scheduler on each multicore architecture. Absolute performance counter values are given in Table 4.2 for Dunnington ma-

chine as an example.

In the Dunnington architecture, we map 12 queries of the WL-1 workload to L1 cache affinity domains. Figure 4.1a shows the improvement in execution times for each query along with L2 and L3 cache miss improvements. For this workload, the performance improvement per query is 11.38% on average. This is due to the fact that our proposed mapping scheme reduces L2 and L3 cache misses on average by 4.8% and 12.4%, respectively (see Figure 4.1b and Figure 4.1c for improvements in L2 and L3 miss rates). We repeated similar performance analysis experiments for the WL-2 workload as well. Recall from Table 4.1, as opposed to WL-1, this workload includes both join and scan bound queries. Our approach improved execution times, L2 cache misses and L3 cache misses on average by 10.1%, 5.9% and 9.6% respectively (see Figure 4.2). As can be observed, in this workload, our approach improves both L2 and L3 miss rates for all queries up to 22%. Overall query execution time improvement is ranging from 4% to 20%.

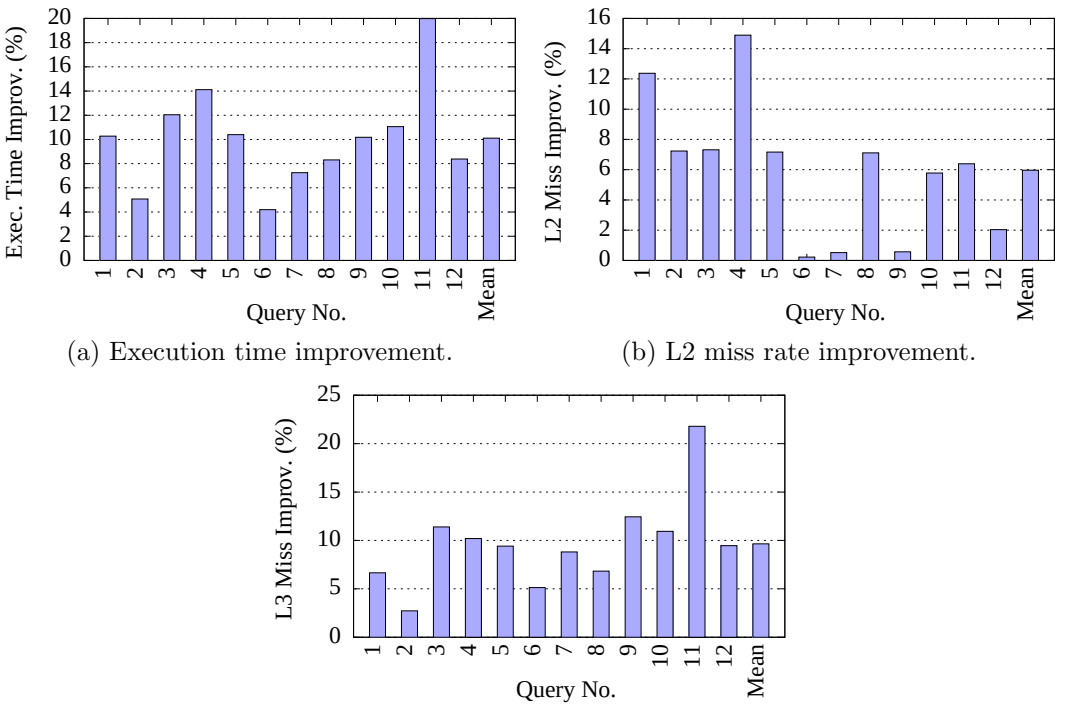


(a) Execution time improvement.

(b) L2 miss rate improvement.

(c) L3 miss rate improvement.

Figure 4.1: WL-1 on Dunnington with 12 clients.



(a) Execution time improvement.

(b) L2 miss rate improvement.

(c) L3 miss rate improvement.

Figure 4.2: WL-2 on Dunnington with 12 clients.

We ran the same set of experiments in Harpertown too and achieved similar performance gains as in Dunnington. Performance results for WL-1 and WL-2 on Harpertown machine can be found at Figure 4.3 and Figure 4.4, where the number of clients is equal to the number of available cores.

We next double the number of queries per core in each of our multicore machines. In doubling the number of queries, we replicated the original workload. The goal behind these experiments is to measure the performance of our load balancing algorithm (see Algorithm 4). In Dunnington, when we double the number of clients (i.e., we move from 12 clients to 24 clients), the corresponding mean execution time saving per query is 17.8%, as presented in Figure 4.5a. Also, we ran 16 clients on 8 cores of the Harpertown and compared the performance results of our approach with the default scheduler in Figure 4.5.

In order to compare the throughput performance of our approach with the default scheduler, we devised a closed-queueing network where the system requests

a new batch of queries whenever all queries from the previous batch are finished. Every new batch is randomly composed of queries from WL-1 and WL-2 with different number of clients. As can be observed from Figure 4.6, when we cumulatively process 60, 84, 108 and 132 clients, the overall throughput improves 15-19% over the default OS scheduler.

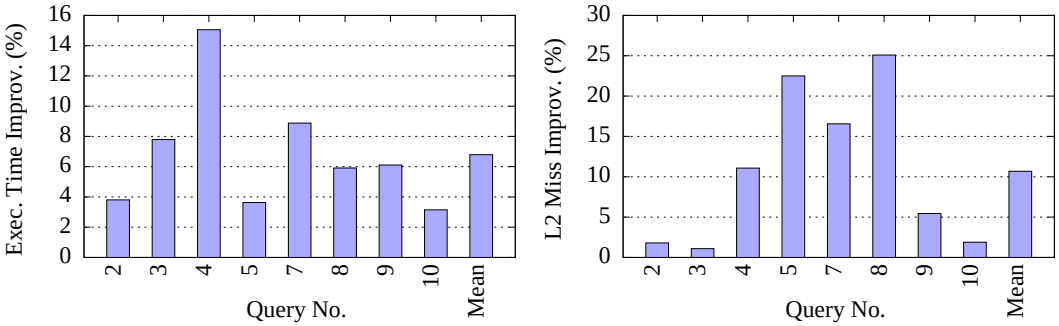
When we look at the performance improvement values per query basis, we can see peak elapsed time improvements around 20-25%. Although most of the queries are managed to benefit from our approach, there are few queries existing which indeed have degraded performance. For example, query 8 of WL-1 in Dunnington and query 2 of WL-2 in Harpertown slightly degraded in last level cache misses and in elapsed time. On the other hand, in Harpertown, query 1 took more time to complete even its L2 cache misses improved. These results are mostly due to other processes, tasks which are controlled by the Linux kernel. They interfere with affinity domains without our control and bring unshared data into affinity domains along with additional load to be processed, causing extra context-switches.

One can see that not every query performs in a similar fashion. This is because, query plans can exhibit different characteristics at run-time and especially, when it is not possible to build appropriate neighborhood (i.e. tandems) among queries, then mutual benefits between these queries become limited. Instead of scheduling each query plan as a whole, it may be more beneficial to split query plans into fine-grained stages and schedule these stages individually, as proposed in [37].

Focusing on the architectural features, we can observe that the difference between our approach and the dynamic scheduler is greater in Dunnington architecture than the Harpertown. This is mainly due to the existence of a more complex cache topology in Dunnington which our cache topology aware scheduling method can optimize data locality at every affinity domain.

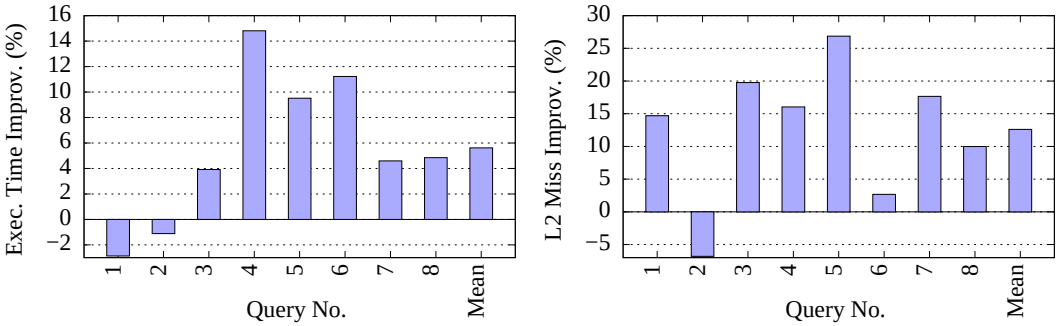
4.1.3 Regression Analysis

In this experiment, we run our throughput analysis on Harpertown machine for various different load imbalance coefficients. We figure out that increasing load imbalance coefficient causes idle cores and our static mapping scheme is suppressed by dynamic OS scheduler. Results are provided in Figure 4.7. As a takeaway



(a) Execution time improvement. (b) L2 miss improvement.

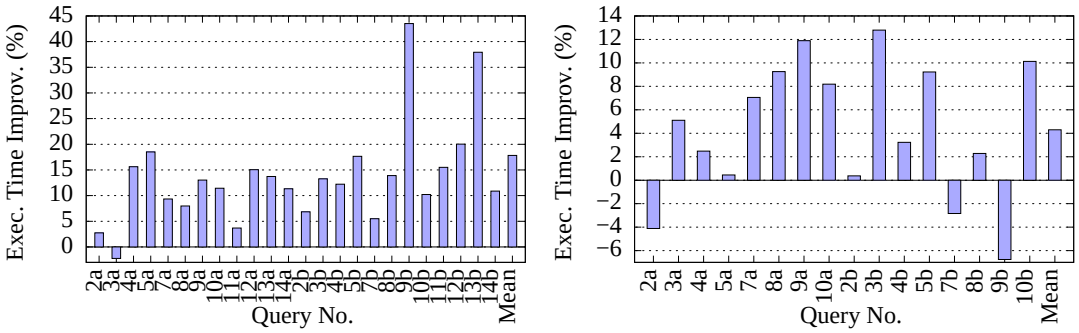
Figure 4.3: WL-1 on Harpertown with 8 clients.



(a) Execution time improvement. (b) L2 miss improvement.

Figure 4.4: WL-2 on Harpertown with 8 clients.

point from this experiment, load balancing is proved itself as a prominent run-time performance factor in multi-query environments.



(a) WL-1 on Dunnington with 24 clients. (b) WL-1 on Harpertown with 16 clients.

Figure 4.5: WL-1 on Dunnington and Harpertown with doubled clients.

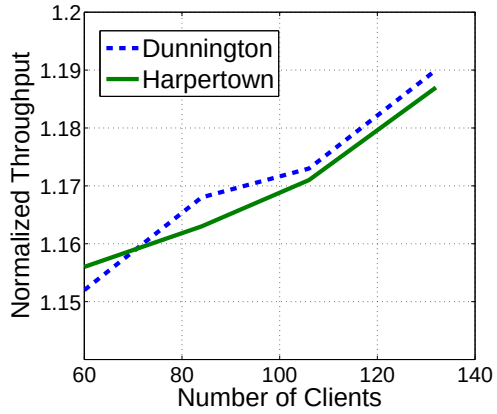


Figure 4.6: Throughput with different number of clients.

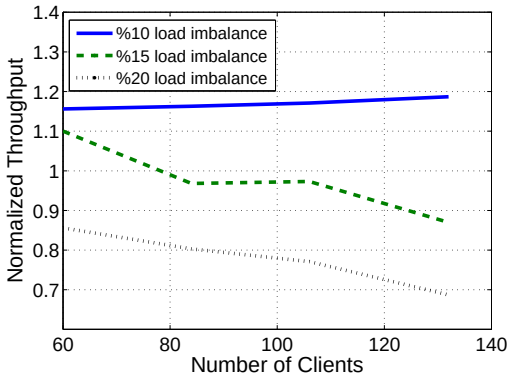


Figure 4.7: Regression Analysis on Load Imbalance Coefficient.

Chapter 5

Conclusion

Despite their intrinsic support towards throughput-oriented computing, moving from single core architectures to shared cache based multicore systems introduces unique challenges for conventional database servers. A complete re-design of database system architecture may be necessary to take advantage of chip multiprocessors; however, we should find ways to overcome emerging deficiencies in existing systems until we have industry-proven alternatives. In this thesis, we address one of these problems, namely, multi-query scheduling on multicore architectures. We show that singularities across on-chip cache topologies designed for different multicore architectures further complicates scheduling decisions beyond the traditional resource allocation and load balancing concerns. Eventually, how a scheduler utilizes on-chip cache topology becomes an important factor of runtime performance. In order to manage and exploit hardware design differences, we propose an architecture aware multi-query scheduling scheme. Initial implementation of this scheme provides encouraging results as demonstrated by our experiments, which were conducted on two different multicore machines. As future work, we are planning to extend our scheduler to cover open-queueing network models as well.

Bibliography

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a modern processor: Where does time go?" in *Proceedings of the VLDB '99*.

[2] R. Agrawal *et al.*, "The claremont report on database research," *Commun. ACM*, vol. 52, June 2009.

[3] N. Hardavellas *et al.*, "Database servers on chip multiprocessors: Limitations and opportunities," in *CIDR*, 2007.

[4] M. Kandemir *et al.*, "Cache topology aware computation mapping for multicores," in *Proceedings of the PLDI '10*.

[5] R. Lee *et al.*, "MCC-DB: minimizing cache conflicts in multi-core processors for databases," *Proc. VLDB Endow.*, vol. 2, August 2009.

[6] PostgreSQL. (2010) http://www.postgresql.org/. [Online]. Available: http://www.postgresql.org/

[7] TPC-H. (2010) http://www.tpc.org/tpch/. [Online]. Available: http://www.tpc.org/tpch/

[8] Intel-Dunnington. (2010) http://ark.intel.com/product.aspx?id=36941. [Online]. Available: http://ark.intel.com/Product.aspx?id=36941

[9] Intel-Harpertown. (2010) http://ark.intel.com/product.aspx?id=33085. [Online]. Available: http://ark.intel.com/Product.aspx?id=33085

[10] Intel-Nehalem. (2010) http://ark.intel.com/product.aspx?spec=slbf5. [Online]. Available: http://ark.intel.com/Product.aspx?spec=slbf5

[11] S. Chen *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *Proceedings of the SPAA '07*.

[12] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 34, May 2006.

[13] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proceedings of the MICRO 41*.

[14] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design," in *Proceedings of the ATEC '05*.

[15] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the EuroSys '07*.

[16] E. P. Markatos and T. J. Leblanc, "Locality-based scheduling in shared-memory multiprocessors," Parallel Computing: Paradigms and Applications, Tech. Rep., 1993.

[17] P. S. Yu, D. W. Cornell, D. M. Dias, and B. R. Iyer, "On affinity based routing in multi-system data sharing," in *Proceedings of the VLDB '86*.

[18] M. Mehta, V. Soloviev, and D. J. DeWitt, "Batch scheduling in parallel database systems," in *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[19] A. Shatdal, C. Kant, and J. F. Naughton, "Cache conscious algorithms for relational query processing," in *Proceedings of the VLDB '94*.

[20] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *Proceedings of the VLDB '07*.

[21] J. Cieslewicz, W. Mee, and K. A. Ross, "Cache-conscious buffering for database operators with state," in *Proceedings of the DaMoN '09*.

[22] D. Yadan *et al.*, "Hash join optimization based on shared cache chip multiprocessor," in *Proceedings of the DASFAA '09*.

[23] M. Stonebraker *et al.*, "C-store: a column-oriented DBMS," in *Proceedings of the VLDB '05*.

[24] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in MonetDB," *Commun. ACM*, vol. 51, December 2008.

[25] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: a simultaneously pipelined relational query engine," in *Proceedings of the SIGMOD '05*.

[26] K. A. Ross, "Optimizing read convoys in main-memory query processing," in *Proceedings of the DaMoN '10*.

[27] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-memory scan sharing for multi-core CPUs," *Proc. VLDB Endow.*, vol. 1, August 2008.

[28] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: dynamic bandwidth sharing in a DBMS," in *Proceedings of the VLDB '07*.

[29] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the PACT '04*.

[30] Q. Lu *et al.*, "Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[31] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven CMP cache management," in *Proceedings of the PACT '06*.

[32] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proceedings of the ICS '07*.

[33] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proceedings of the Supercomputing '98*.

[34] METIS. (2010) http://glaros.dtc.umn.edu/gkhome/views/metis. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis

[35] M. Shao, A. Ailamaki, and B. Falsafi, "DBmbench: fast and accurate database workload representation on modern microarchitecture," in *Proceedings of the CASCON '05*.

[36] Perfmon2. (2010) http://perfmon2.sourceforge.net/. [Online]. Available: http://perfmon2.sourceforge.net/

[37] S. Harizopoulos and A. Anastassia, "StagedDB: designing database servers for modern hardware," in *Proceedings of the ICDE '05*.