Staged Database Systems

Stavros Harizopoulos

December 2005
CMU-CS-05-194

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This work was supported in part by NSF through grants CCR-0113660, IIS-0133686, and CCR-0205544, and by IBM and the Lilian Voudouri Foundation through graduate student fellowships. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Database Management Systems (DBMS), query execution, query pipelining, operator parallelism, cache performance, instruction cache misses, thread scheduling, Online Transaction Processing (OLTP), Decision Support Systems (DSS).

Abstract

Advances in computer architecture research yield increasingly powerful processors which can execute code at a much faster pace than they can access data in the memory hierarchy. Database management systems (DBMS), due to their intensive data processing nature, are in the front line of commercial applications which cannot harness the available computing power. To prevent processors from idling, a multitude of hardware mechanisms and software optimizations have been proposed. Their effectiveness, however, is limited by the sheer volume of data accessed and by the unpredictable sequence of memory requests.

This Ph.D. dissertation introduces *Staged Database Systems*, a new software architecture for optimizing data and instruction locality at all levels of the memory hierarchy. The key idea is to break database request execution in stages and process a group of sub-requests at each stage. Group processing at each stage allows for a context-aware execution sequence of requests that promotes reusability of both instructions and data. The Staged Database System design requires only a small number of changes to the existing DBMS codebase and provides a new set of execution primitives that allow software to gain increased control over what data and instructions are accessed, when, and by which requests. The central thesis is the following:

"By organizing and assigning system components into self-contained stages, database systems can exploit instruction and data commonality across concurrent requests thereby improving performance."

for my grandfather

ΓΙΑ ΤΟΝ ΠΑΠΠΟΥ ΜΟΥ

Acknowledgments

I would like to thank my advisor Anastassia Ailamaki; my thesis committee members, Panos Chrysanthis, Christos Faloutsos, Todd Mowry, and Mike Stonebraker; faculty and staff at Carnegie Mellon, especially Sharon Burks, Catherine Copetas, Joan Digney, Babak Falsafi, Garth Gibson, Karen Lindenfelser, and Charlotte Yano; friends, colleagues, and co-workers, especially Umut Acar, Mehmet Bakkaloglu, Paul Bennett, Tina Bennett, Alan Cow, Kun Gao, Christoffer Hall-Frederiksen, Nikos Hardavellas, Hyang-Ah Kim, Yiannis Koutis, Dimitris Margaritis, Spiros Papadimitriou, Stratos Papadomanolakis, Francisco Pereira, Jovan Popovic, Kivanc Sabirli, Melissa Santos, Giorgos Sapunjis, Bianca Schroeder, Vladislav Shkapenyuk, Ryan Williams, Elly Winner, and Weng-Keen Wong; my family, Yannis, Eva, Nikos, and Zaharenia; my partner, Lina Papadaki.

Had the completion of this thesis been a journey, then

Natassa would have been my guiding star.

Panos, Christos, Todd, and Mike would have all been giving me directions at every port.

Paul would have been helping me keep the destination in my mind, Umut would have been telling me which seas to avoid, Stratos and Vlad would have been showing me the good rest places in every city, and Spiros would have taught me the value of ignoring the compass once in a while.

My family would have given me the ship.

Lina would have been my sails.

Contents

List of figures

List of tables

Chapter 1

Overview

Advances in computer architecture research yield increasingly powerful processors which can execute code at a much faster pace than they can access data in the memory hierarchy. Database management systems (DBMS), due to their intensive data processing nature, are in the front line of commercial applications which cannot harness the available computing power. To prevent processors from idling, a multitude of hardware mechanisms and software optimizations have been proposed. Their effectiveness, however, is limited by the sheer volume of data accessed and by the unpredictable sequence of memory requests.

This Ph.D. dissertation introduces *Staged Database Systems*, a new software architecture for optimizing data and instruction locality at all levels of the memory hierarchy. The key idea is to break database request execution in stages and process a group of sub-requests at each stage. Group processing at each stage allows for a context-aware execution sequence of requests that promotes reusability of both instructions and data. The Staged Database System (StagedDB) design requires only a small number of changes to the existing DBMS code base and provides a new set of execution primitives that allow software to gain increased control over what data and instructions are accessed, when, and by which requests.

Chapter 1 motivates the need for a new database system architecture to alleviate the mismatch between DBMS software behavior and modern CPU architectural features. It also serves as a dissertation road map, providing an overview of all chapters. The thesis maintained throughout this document is the following. "By organizing and assigning system components into self-contained stages, database systems can exploit instruction and data commonality across concurrent requests thereby improving performance."

1.1. Introduction

Advances in processor design, storage architectures and communication networks, and the explosion of the Web, have allowed storing and accessing terabytes of information online. Database management systems are responsible for supporting an increasing base of millions of users executing time-critical operations. Although database software architecture has fundamentally remained unchanged over the past two decades, hardware infrastructure has undergone significant changes. Computer processors execute multiple instructions every clock cycle and apply out-of-order and instruction-level parallelism techniques to maximize performance. In the mean time, while main memory sizes have grown along with processor speed, memory speed has fundamentally lagged behind. Today's memory systems incur access latencies that are up to three orders of magnitude larger than the latency of a single arithmetic operation.

Research shows that the performance of database systems on modern hardware is tightly coupled to how efficiently the entire memory hierarchy, from disks to on-chip caches, is utilized. Unfortunately, according to recent studies, 50% to 80% of the execution time in database workloads is spent waiting for instructions or data [Ailamaki et al. 1999; Barroso et al. 1998; Keeton et al. 1998]. Current technology trends call for computing platforms with higher-capacity memory hierarchies, but with each level requiring increasingly more processor cycles to access. At the same time, advances in chip manufacturing process allow the simultaneous execution of multiple programs on the same chip, either through hardware-implemented threads on the same CPU (simultaneous multithreading—SMT), or through multiple CPU cores on the same chip (chip multiprocessing—CMP), or both. With higher levels of hardware-available parallelism, the performance requirements of the memory hierarchy increase.

To improve DBMS performance, it is necessary to engineer software that takes into consideration all features of new microprocessor designs. Database systems have traditionally employed a work-centric multi-threaded (or multi-process) execution model for parallelism. Since typically each database query or transaction is handled by one or more processes (threads), the DBMS essentially relinquishes execution control to the operating system and then to the CPU. The operating system's context switching decisions are oblivious to the state of the request/thread and therefore cause severe context thrashing in the memory hierarchy.

1.2. Pitfalls of thread-based concurrency

Modern database systems adopt a thread-based concurrency model for executing coexisting query streams. To best utilize the available resources, DBMS typically use a pool of threads or processes. Each incoming query is handled by one or more threads, depending on its complexity and the number of available CPUs. Each thread executes until it either blocks on a synchronization condition, an I/O event, or until a predetermined time quantum has elapsed. Then, the CPU switches context and executes a different thread (e.g., IBM's DB2[1]) or the same thread takes on a different task (e.g., Microsoft's SQL Server[2]). Context-switching typically relies on generated events instead of program structure or the query's current state. While this model is intuitive, it has several shortcomings:

- There is no single number of preallocated worker threads that yields optimal performance under changing workloads. Too many threads waste resources and too few threads restrict concurrency.
- Preemption is oblivious to the thread's current execution state. Context-switches that occur in the middle of a logical operation evict a possibly larger working set from the cache. When the suspended thread resumes execution, it wastes time restoring the evicted working set.
- Round-robin thread scheduling does not exploit cache contents that may be common across a set of threads. When selecting the next thread to run, the scheduler ignores that a different thread might benefit from already fetched data.

These three shortcomings are depicted in Figure 1-1. In this hypothetical execution sequence, four concurrent queries handled by four worker threads pass through the optimizer or the parser of a single-CPU database server. The example assumes that no I/O takes place. Whenever the CPU resumes execution on a query, it first spends some time

1. IBM DB2 Universal Database V7 Manuals. "Administration Guide V7.2, Volume 3: Performance"
2. Microsoft SQL Server 2000 Technical Article. Available at: http://msdn.microsoft.com/library

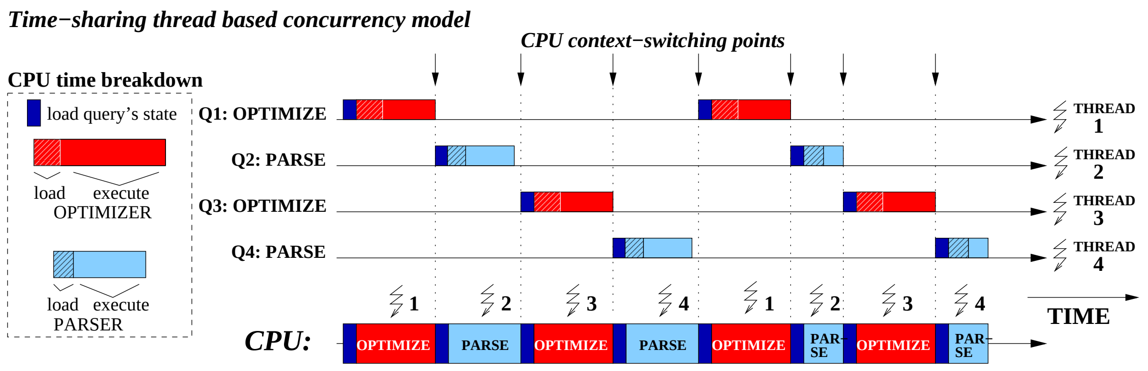


Figure 1-1. Uncontrolled context-switching can lead to poor performance.

loading (fetching from main memory) the thread's private state. Then, during each module's execution, the CPU also spends time loading the data and code that are shared on average between all queries executing in that module (shown as a separate striped box after the context-switch overhead). A subsequent invocation of a different module will likely evict the data structures and instructions of the previous module, to replace them with its own ones.

The performance loss in this example is due to (a) a large number of worker threads: since no I/O takes place, one worker thread would be sufficient, (b) preemptive thread scheduling: optimization and parsing of a single query is interrupted, resulting in unnecessary reloads of its working set, and, (c) round-robin scheduling: optimization and parsing of two different queries are not scheduled together and, thus, the two modules keep replacing each other's data and code in the cache. The solution to these problems is a new DBMS software architecture that allows for context switching on module boundaries, and for group scheduling of requests on a per-module basis.

1.3. Techniques to improve locality

Since each memory hierarchy level trades capacity for lookup speed, research has focused on ways to improve locality at each level. Techniques that increase reusability of a page or a cache block are referred to as *temporal locality* optimizations, while *spatial locality* optimizations are improving the utilization within a single page or cache block. Database researchers propose[1] relational processing algorithms to improve data and instruction temporal locality [Padmanabhan et al. 2001; Shatdal et al. 1994], and also indexing structures and memory page layouts that improve spatial locality [Shao et al. 2004]. Buffer pool replacement policies seek to improve main memory utilization by increasing the reusability of data pages across requests [Chou and DeWitt 1985; Megiddo and Modha 2003].

Computer architecture researchers propose to rearrange binary code layout [Ramirez et al. 2001] to achieve better instruction spatial locality. A complementary approach to improve performance is to overlap memory hierarchy accesses with computation. At the software level, recent techniques prefetch known pages ahead of time [Chen et al. 2001], while at the microarchitecture level, out-of-order processors tolerate cache access latencies by executing neighboring instructions [Hennessy and Patterson 1996]. Although related work identifies memory-related bottlenecks and proposes techniques to boost performance, current DBMS designs do not have the means to exploit commonality across all levels of the memory hierarchy.

1.4. A *staged* approach for DBMS software

Most of the research to date for improving locality examines data accessed and instructions executed by a single query (or transaction) at a time. Database systems, however, typically handle multiple concurrent users. By properly synchronizing and multiplexing the concurrent execution of multiple requests there is a potential of increasing both data and instruction reusability at all levels of the memory hierarchy. Existing DBMS designs, however, pose difficulties in applying such execution optimizations as they abide by the execution primitives provided by the underlying operating system. A new software design is needed to allow for application-induced control of execution.

In this dissertation, we re-engineer database systems to improve utilization of all memory hierarchy levels. Rather than rewriting the entire code of the database system, we provide the support to organize system components into self-contained "stages" and change request execution sequence to perform group-processing at each stage, thus effortlessly

1. A more complete treatment of related work follows in Chapter 2.

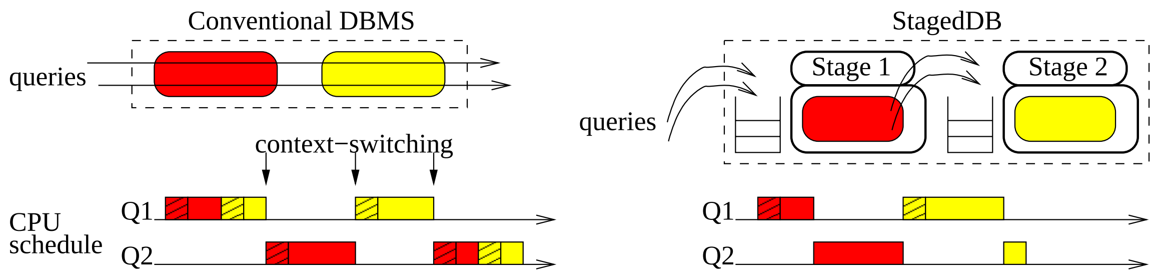


Figure 1-2. Example of conventional and Staged DBMS architecture with two system modules. Striped boxes in the query schedule (assuming single CPU) correspond to time spent accessing instructions and data common to both queries; these need to be reloaded when modules are swapped. StagedDB loads a module only once.

exploiting commonality across queries. A stage acts as an independent server with its own queue for the incoming requests, thread support, and resource management. Each query consists of a number of *packets* that queue up in front of staged operators. The query is essentially the first-class citizen in the system: it carries all the information necessary for its execution, and it visits the stages it needs.

The StagedDB design requires only a small number of changes to the existing DBMS code base and provides a new set of execution primitives that allow software to gain increased control over what data is accessed, when, and by which requests. Figure 1-2 illustrates the high-level idea behind StagedDB. For simplicity, we only show two requests passing through two stages of execution. In conventional DBMS designs, context-switching among requests occurs at random points, possibly evicting instructions and data that are common among requests executing at the same stage (these correspond to the striped boxes in the left part of Figure 1-2). A StagedDB system (right part of Figure 1-2) consists of a number of self contained modules, each encapsulated into a stage. Group processing at each stage allows for a context-aware execution sequence of requests that promotes reusability of instructions and data.

The rest of this section summarizes the key components of StagedDB that were developed and evaluated as part of this dissertation and will be described over the next chapters. First, we explore the design space of staged execution schemes along with the software engineering benefits of staging a database system. Then, we focus on the query execution engine, where most of the query's lifetime is spent, and build a staged engine to enhance locality across concurrent queries. At the microarchitectural level, we apply the principles of StagedDB to address instruction-stream optimization in transaction processing.

1.4.1 StagedDB: design tradeoffs and benefits

Staged software architectures, similar to StagedDB, exhibit the following fundamental scheduling tradeoff. On one hand, all requests executing as a batch in the same software module benefit from improved locality. On the other hand, each completed request suspends its progress until the rest of the batch finishes execution, thereby increasing response time. We investigate this tradeoff by developing a simulated execution environment that is also analytically tractable. We propose and evaluate scheduling policies that are shown to improve query response times in staged execution, when compared to traditional techniques such as first-come first-serve and processor-sharing, for a wide range of workload parameter values.

Looking from a software engineering point of view, years of DBMS software development have lead to monolithic, complicated implementations that are difficult to extend, tune, and evolve. StagedDB eases software development by isolating functions inside self-contained micro-servers (stages), and also allows for an accurate resource allocation based on the needs of each stage. For example, stages that do not perform I/O are assigned fewer threads than stages performing heavy I/O. As a case study, this dissertation describes the benefits, both performance and software engineering ones, of staging PREDATOR, a research prototype DBMS [Seshadri et al. 1997].

1.4.2 Staging a relational query execution engine

In decision-support applications, database systems typically execute concurrent queries independently by invoking a set of relational operator instances for each query. To exploit common data retrievals and computation in concurrent queries, researchers have proposed a wealth of techniques, ranging from buffering disk pages to constructing materialized views (precomputed query results) and optimizing multiple queries. The ideas proposed, however, are inherently limited by the query-centric philosophy of modern query execution engine designs. Ideally, the query engine should proactively coordinate same-operator

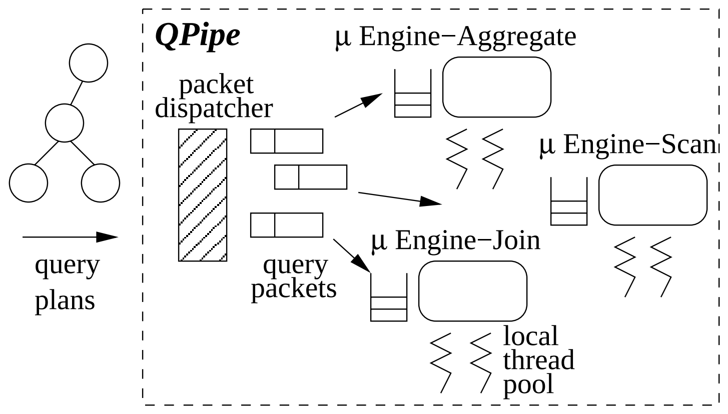


Figure 1-3. QPipe architecture: queries with the same operator queue up at the same micro-engine (for simplicity, only three micro-engines are shown).

execution among concurrent queries, thereby exploiting common accesses to memory and disks as well as common intermediate result computations.

We apply the StagedDB design to a full-blown relational query execution engine, built on top of BerkeleyDB[1], and show how to maximize data and work sharing across concurrent queries at execution time. In the resulting query engine prototype, called *QPipe*, each relational operator becomes a staged micro-engine serving query tasks from a queue, as shown in Figure 1-3. A *packet dispatcher* converts an incoming query plan to a series of query packets. Data flow between micro-engines occurs through dedicated buffers - similar to a parallel database engine [DeWitt et al. 1990]. QPipe achieves better resource utilization than conventional engines by grouping requests of the same nature together, and by having dedicated micro-engines to process each group of similar requests. Every time a new packet queues up in a micro-engine, all existing packets are checked for overlapping work. On a match, each micro-engine employs different mechanisms for data and work sharing, depending on the enclosed relational operator. We show that QPipe achieves a factor of two speedup over a commercial DBMS when running a workload consisting of TPC-H queries (the industry-standard decision-support benchmark). When running QPipe with queries that present no sharing opportunities, we find that the overhead of checking for overlapping work is negligible (less than 1% of the query execution time).

1. BerkeleyDB, http://www.sleepycat.com

1.4.3 Improving instruction cache performance

In examining the efficiency of the highest levels of the memory hierarchy, recent research studies have reported that, when running Online Transaction Processing (OLTP) workloads, instruction-related delays in the memory subsystem account for 25% to 40% of the total execution time [Barroso et al. 1998; Keeton et al. 1998; Shao and Ailamaki 2004]. In contrast to data, instruction misses cannot be overlapped with out-of-order execution, and instruction caches cannot grow as the slower access time directly affects the processor speed. With commercial DBMS exhibiting more than 500KB of OLTP code footprint [Lo et al. 1998] and modern CPUs having 16-64KB instruction caches, transactional code paths are too long to achieve cache-residency. The challenge is to alleviate the instruction-related delays without increasing cache size.

This dissertation proposes and evaluates *STEPS* (Synchronized Transactions through Explicit Processor Scheduling), a technique based on the StagedDB design that minimizes instruction-cache misses by multiplexing concurrent transactions and exploiting common code paths. At a higher level, STEPS applies the StagedDB design to form groups of concurrent transactions that execute the same transactional operation (e.g., traversing an index, updating a record, or performing an insert). Since DBMS typically assign a thread to each transaction, STEPS introduces a thin wrapper around each transactional operation to coordinate threads executing the same operation. Although transactions in a group have a high degree of overlap in executed instructions, a conventional scheduler that executes one thread after the other would still incur new instruction misses for each transaction execution (as shown in the left part of Figure 1-4). The reason is that the code working set of transactional operations typically overwhelms the L1-I cache. To maximize instruction sharing among transactions in the same group, STEPS lets only one transaction incur compulsory instruction misses, while the rest of the transactions “piggyback” onto the first one, finding the instructions they need in the cache. To achieve this, STEPS identifies the points in the code where the cache fills up and performs a quick context-switch, so that other transactions can execute the cache-resident code (right part of Figure 1-4).

We evaluate STEPS inside the *Shore* research prototype database storage manager [Carey et al. 1994]. STEPS yields a 82-96% reduction in instruction cache misses for each

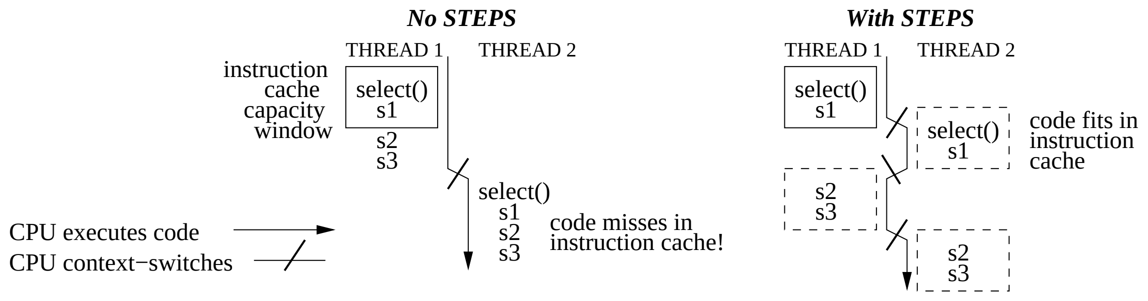


Figure 1-4. Illustration of instruction-cache aware context switching with STEPS.

additional concurrent transaction, and at the same time eliminates 62-64% of mispredicted branches by loading a repeating execution pattern into the CPU. In a full-system evaluation on a real processor using TPC-C, the industry-standard transactional benchmark, STEPS produces a 16-39% speedup.

1.5. Contributions

The dissertation makes the following contributions, organized in four categories.

1. *By examining current database system software designs and by exploring an alternative, staged design, this dissertation:*

- provides an analysis of design shortcomings in modern DBMS software.
- introduces a novel database system design which is shown to exhibit both performance and software engineering benefits over traditional database system designs.
- presents new research opportunities based on the proposed design.

2. *By studying scheduling tradeoffs in staged server architectures, this dissertation:*

- introduces four novel cohort scheduling techniques for staged software servers that follow a “production-line” model of operation; the proposed techniques outperform traditional scheduling policies and justify software staging even when the degree of inter-request locality is low.

- presents a mathematical framework to methodically quantify the performance tradeoffs when using the proposed scheduling policies.

3. *By designing, building, and evaluating a staged relational query engine, and by studying opportunities for data and computation sharing across concurrent queries, this dissertation:*

- introduces QPipe, a novel query execution engine that provides full intra-query parallelism, taking advantage of all available CPUs in multiprocessor servers (both traditional multiprocessors and chip multiprocessors) for evaluating a single query, regardless of the plan's complexity.
- provides a set of query evaluation techniques to maximize data and work sharing across concurrent queries at run time.
- presents an efficient run-time for evaluating plans produced by a multi-query optimizer, by avoiding costly materializations.
- demonstrates an improved design over traditional, tuple-by-tuple query engines (using the *iterator* evaluation model), by saving extraneous procedure calls and by improving temporal locality.

4. *By investigating the poor instruction cache performance exhibited in transaction processing workloads, and by exploring the potential of explicit thread scheduling techniques coupled with a grouped execution discipline, this dissertation:*

- contributes a software approach to address instruction cache performance in transaction processing applications; the proposed techniques enable thread scheduling at very fine granularity to reuse instructions in the cache across concurrent threads. The proposed techniques are implemented and evaluated inside a full-blown research prototype database system running a multi-user transactional benchmark on real hardware.
- provides a tool to automate the deployment of the proposed set of techniques inside a commercial DBMS server.

1.6. Dissertation organization

The dissertation consists of six chapters. Chapter 1 provides an overview of the dissertation and lists its contributions[1]. Chapter 2 presents related work from a broad scope of research in database system architectures, performance studies on recent processors, and research in operating systems. Chapter 3 explores the design space of staged execution schemes and describes performance and software engineering benefits of staging a database system[2]. Chapter 4 describes QPipe, a staged relational query engine that enhances locality across concurrent queries[3]. Chapter 5 describes STEPS, a transaction coordinating mechanism based on the StagedDB design that minimizes instruction cache misses in OLTP workloads[4]. Chapter 6 discusses additional benefits of the StagedDB design, presents promising future research directions, and concludes.

1. Parts of the chapter are published in [Harizopoulos and Ailamaki 2005].
2. Parts of the chapter are published in [Harizopoulos and Ailamaki 2002; Harizopoulos and Ailamaki 2003].
3. Parts of the chapter are published in [Harizopoulos et al. 2005; Shkapenyuk et al. 2005].
4. Parts of the chapter are published in [Harizopoulos and Ailamaki 2004].

Chapter 2
Related work

This chapter presents related work from a broad scope of research in database system architectures, performance studies on recent processors, and research in operating systems. We first discuss both traditional and novel database system architectures, along with mechanisms to improve query processing performance, such as buffer pool management, materialized views, query caching, and multiple query optimization. We then review recent studies pointing out the CPU-memory bottleneck in DBMS performance, along with related work to address cache performance. Lastly, we discuss related research efforts in the operating systems community to improve performance of thread-based servers and reduce the response time of requests by changing the server CPU scheduling policy.

2.1. DBMS architectures and query processing

In the past three decades of database research, several new software designs have been proposed. One of the earliest prototype relational database systems, INGRES [Stonebraker et al. 1976], actually consisted of four "stages" (processes) that enabled pipelining (the reason for breaking up the DBMS software was main memory size limitations). Staging was also known to improve CPU performance in the mid-seventies[1].

Parallel database systems [DeWitt and Gray 1992; Chekuri et al. 1995] exploit the inherent parallelism in a relational query execution plan and apply a dataflow approach for designing high-performance, scalable systems. In the GAMMA database machine project [DeWitt et al. 1990] each relational operator is assigned to a process, and all processes

1. Asynchronous Work Elements, IBM/IMS team. Comments from anonymous reviewer, Oct. 2002.

work in parallel to achieve either pipelined parallelism (operators work in series by streaming their output to the input of the next one), or partitioned parallelism (input data are partitioned among multiple nodes and operators are split into many independent ones working on a part of data). In *extensible* DBMS [Carey and Haas 1990], the goal was to facilitate adding and combining components (e.g., new operator implementations). Both parallel and extensible database systems employ a modular system design with several desirable properties, but there is no notion of cache-related interference across multiple concurrent queries.

2.1.1 Stream processing systems

Recent database research focuses on a data processing model where input data arrives in multiple, continuous, time-varying streams [Babcock et al. 2002]. The relational operators are treated as parts of a chain where the scheduling objective is to minimize queue memory and response times, while providing results at an acceptable rate or sorted by importance [Urhan and Franklin 2001]. Avnur and Hellerstein propose eddies, a query processing mechanism that continuously reorders pipelined operators in a query plan, on a tuple-by-tuple basis, allowing the system to adapt to fluctuations in computing resources, data characteristics, and user preferences [Avnur and Hellerstein 2000]. Operators run as independent threads, using a central queue for scheduling. While the aforementioned architectures optimize the execution engine's throughput by changing the invocation of relational operators, they do not exploit cache-related benefits. For example, eddies may benefit by repeatedly executing different queries at one operator, or by increasing the tuple processing granularity (we discuss similar tradeoffs over the next thesis chapters).

TelegraphCQ (CACQ [Madden et al. 2002] and PSoup [Chandrasekaran and Franklin 2003]) and NiagaraCQ [Chen et al. 2000] describe techniques to share work across different queries in stream management systems, by sharing either physical operators or their state. Although the concept of sharing operators is similar to what we propose in this thesis, the different context creates an entirely different problem. Queries in stream systems always process the most recently received tuples. In traditional DBMS, queries have specific requirements as to which tuples they need and in what order they need to process them.

2.1.2 Buffer pool management

In its simplest form, a buffer pool manager keeps track of disk pages brought in main memory, decides when to write updated pages back to disk, and evicts pages (typically using a LRU policy) when new ones are read. The Hot Set [Sacco and Schkolnick 1986] and DBMIN [Chou and DeWitt 1985] algorithms rely on explicit hints from the query optimizer on query access patterns. Since it is infeasible for the optimizer to predict the query patterns in a multi-query environment, several algorithms base replacement decisions on the observed importance of different pages. LRU-K [O'Neil et al. 1993] and 2Q [Johnson and Shasha 1994], for instance, improve the performance of the traditional LRU eviction policy by tracking multiple past-page references, while ARC [Megiddo and Modha 2003] shows similar performance improvements without relying on tunable parameters.

Since queries interact with the buffer pool manager through a page-level interface, it is difficult to develop generic policies to coordinate current and future accesses from different queries to the same disk pages. The need to efficiently coordinate and share multiple disk scans on the same table has long been recognized [Gray 2004] and several commercial DBMS incorporate various forms of multi-scan optimizations (Teradata, RedBrick [Fernandez 1994], and SQL Server[1]). The challenge is to bypass the restrictions implied by the page-level interface in order to fully exploit the knowledge of query access patterns, even if it requires run-time adjustments to the query evaluation strategy.

2.1.3 Materialized views, query caching, and plan recycling

Materialized view selection [Roussopoulos 1982] is typically applied to workloads known in advance, in order to speed-up queries that contain common subexpressions. The most commonly used technique is to exhaustively search all possible candidate views, while employing various heuristics to prune the search space. It is important to note that materialized views exploit commonality between different queries at the expense of potentially significant view maintenance costs. Modern tools for automatic selection of materialized

1. C. Cook. "Database Architecture: The Storage Engine." Microsoft SQL Server 2000 Technical Article, July 2001. Available at: http://msdn.microsoft.com/library

views [Agrawal et al. 2000] take such costs into account when recommending a set of views to create [Blakeley et al. 1986]. The usefulness of materialized views is limited when the workload is not always known ahead of time or the workload requirements are likely to change over time.

Caching query results can significantly improve response times in a workload that contains repeating instances of the same query or queries that are subsumed by others. A recently proposed cache manager [Shim et al. 1999] dynamically decides on which results to cache, based on result computation costs, sizes, reference frequencies, and maintenance costs due to updates. Semantic data caching [Dar et al. 1996] (as opposed to page or tuple caching) can result in more efficient use of a cache and reduced communication costs in client-server environments. Query plan recycling [Sarda and Haritsa 2004] reduces the query optimization time by exploiting potential similarity in the plans of different queries. The queries are first clustered based on characteristics of their execution plans, and then all queries assigned to a cluster use the same plan generated for the cluster representative query. Both approaches complement any type of run-time optimizations. QPipe improves a query result cache by allowing the run-time detection of exact instances of the same query, thus avoiding extra work when identical queries execute concurrently, with no previous entries in the result cache.

2.1.4 Multi-query optimization

Multiple-query optimization (MQO) [Finkelstein 1982; Sellis 1988; Roy et al. 2000] identifies common subexpressions in query execution plans during optimization and produces globally-optimal plans. Since the detection of common subexpressions is done at optimization time, all queries need to be optimized as a batch. In interactive scenarios where queries may arrive at any time, other queries that share the same computation may be already running (waiting to collect a batch delays the early queries). In addition, to share intermediate results among queries, MQO typically relies on costly materializations. To avoid unnecessary materializations, a recent study [Dalvi et al. 2001] introduces a model that decides at the optimization phase which results can be pipelined and which need to be materialized to ensure continuous progress in the system. In contrast, QPipe identifies and exploits common subexpressions at run time without forcing the optimizer to wait for a sufficient number of queries to arrive before optimizing a batch. Moreover, QPipe can efficiently evaluate plans produced by a multi-query optimizer, since it always pipelines shared intermediate results.

2.2. DBMS performance on modern processors

Computer architecture research addresses the ever-increasing processor-memory speed gap [Hennessy and Patterson 1996] by exploiting data and code locality and by minimizing memory stalls. Modern systems employ mechanisms ranging from larger and deeper memory hierarchies to sophisticated branch predictors and software/hardware prefetching techniques. Prior research [Maynard et al. 1994] indicates that adverse memory access patterns in database workloads result in poor cache locality and overall performance. Recent studies of OLTP workloads and DBMS performance on modern processors [Ailamaki et al. 1999; Keeton et al. 1998] narrow the primary memory-related bottlenecks to L1 instruction and L2 data cache misses. More specifically, Keeton et al. measure an instruction-related stall component of 41% of the total execution time for Informix running TPC-C on a PentiumPro. When running transactional (TPC-B and TPC-C) and decision-support (TPC-H) benchmarks on top of Oracle on Alpha processors, instruction stalls account for 45% and 30% of the execution time, respectively [Barroso et al. 1998; Stets et al. 2002]. A recent study of DB2 7.2 running TPC-C on Pentium III [Shao and Ailamaki 2004] attributes 22% of the execution time to instruction stalls.

Work in "cache-conscious" DBMS optimizes query processing algorithms [Shatdal et al. 1994], index manipulation [Chen et al. 2001; Chilimbi et al. 2000; Graefe and Larson 2001], and data placement schemes [Ailamaki et al. 2001]. Such techniques improve the locality within each request, but have limited effects on the locality *across* requests. Context-switching across concurrent queries is likely to destroy data and instruction locality in the caches. For instance, when running workloads consisting of multiple short transactions, most misses occur due to conflicts between threads whose working sets replace each other in the cache [Jayasimha and Kumar 1999; Rosenblum et al. 1995].

Unfortunately, unlike DSS workloads, transaction processing involves a large code footprint and exhibits irregular data access patterns due to the long and complex code paths of transaction execution [Harizopoulos and Ailamaki 2004]. Finally, OLTP instruction streams have strong data dependencies that limit instruction-level parallelism opportunities, and irregular program control flow that undermines built-in pipeline branch prediction mechanisms and increases instruction stall time [Keeton et al. 1998].

2.2.1 Techniques to address L1-I cache stalls

Most research on cache-conscious database systems has primarily addressed data cache performance [Shatdal et al. 1994; Chen et al. 2001; Graefe and Larson 2001; Ailamaki et al. 2001]. L1-I cache misses, however, and misses occurring when concurrent threads replace each other's working sets [Rosenblum et al. 1995], have received little attention by the database community. Two recent studies [Padmanabhan et al. 2001; Zhou and Ross 2004] propose increasing the number of tuples processed by each relational operator, improving instruction locality when running single-query-at-a-time DSS workloads. Unfortunately, similar techniques cannot apply to OLTP workloads because transactions typically do not form long pipelines of database operators.

Instruction locality can be improved by altering the binary code layout so that run-time code paths are as conflict-free and stored as contiguously as possible [Romer et al. 1997; Ramirez et al. 2001]. Such compiler optimizations are based on static profile data collected when executing a certain targeted workload and therefore, they may hurt performance when executing other workloads. Moreover, such techniques cannot satisfy all conflicting code paths from all different execution threads. While the proposed techniques can enhance instruction-cache performance by increasing *spatial locality* (utilization of instructions contained in a cache block), they cannot increase *temporal locality* (instruction reusability) since that is a direct function of the application nature.

A complementary approach is instruction prefetching in the hardware [Chen et al. 1997]. Call graph prefetching [Annavaram et al. 2003] collects information about the sequence of database functions calls and prefetches the function most likely to be called next. The success of such a scheme depends on the predictability of function call sequences. Unfortunately, OLTP workloads exhibit highly unpredictable instruction streams that challenge even the most sophisticated prediction mechanisms (call graph prefetching is evaluated through relatively simple DSS queries [Annavaram et al. 2003]).

2.3. Related work in operating systems

Recently, OS research introduced a staged server programming paradigm [Larus and Parkes 2002], that divides computation into stages and schedules requests within each stage. The CPU processes the entire stage queue while traversing the stages going first forward and then backward. The authors demonstrate that their approach improves the performance of a simple web server and a publish-subscribe server by reducing the frequency of cache misses in both the application and operating system code. While the experiments were successful, significant scheduling trade-offs remain unsolved. For instance, it is not clear under which circumstances a policy should delay a request before the locality benefit disappears. Welsh et al. propose a staged event-driven architecture (SEDA) for deploying highly concurrent internet services [Welsh et al. 2001]. SEDA decomposes an event-driven application into stages connected by queues, thereby preventing resource overcommitment when demand exceeds service capacity. SEDA does not optimize for memory hierarchy performance, which is the primary bottleneck for data-intensive applications.

Research in operating systems points out limitations in thread scalability when building highly concurrent applications[1] [Pai et al. 1999]. Related work suggests inexpensive implementations for context-switching [Anderson et al. 1991; Banga and Mogul 1998], and also proposes event-driven architectures with limited thread usage, mainly for internet services [Pai et al. 1999].

2.3.1 Scheduling algorithms

Most server architectures adopt processor-sharing (PS) scheduling as a “fair” scheduling policy. The CPU(s) spend a fixed amount of time (typically in the order of a few ms) on each active process and keep switching processes in a round-robin fashion. Recent work argues in favor of SRPT (Shortest Remaining Processing Time first) scheduling for the case of a single server [Bansal and Harchol-Balter 2001]. Crovella et al. have shown that traditional assumptions for workload modeling are not accurate [Crovella et al. 1998]. More specifically, it was shown that UNIX jobs have sizes that follow a Pareto (heavy-

1. J. K. Ousterhout. “Why threads are a bad idea (for most purposes).” Invited talk at 1996 USENIX Tech. Conf. (slides available at http://home.pacbell.net/ouster/), Jan. 1996.

tailed) distribution and not an exponential one. The effects of taking into account this distribution can be counter-intuitive and are discussed in [Crovella et al. 1998]. We experiment with such a distribution for query sizes in the next chapter.

Affinity scheduling explicitly routes tasks to processors with relevant data in their caches [Squillante and Lazowska 1993; Srivastava and Eustace 1994] (the term "affinity scheduling" is widely used in the context of shared-memory multiprocessor systems). Although this type of affinity is similar to the one that the staged programming paradigm tries to leverage on, the latter approaches the problem by restructuring a single application to exploit locality, rather than improve locality for a collection of tasks in a generic setting. Frequent switching between threads of the same program interleaves unrelated memory accesses, thereby affecting locality. This thesis addresses memory performance from a single application's point of view, improving locality across its threads.

2.4. Chapter summary

Research shows that the performance of database management systems on modern hardware is tightly coupled to how efficiently the entire memory hierarchy, from disks to on-chip caches, is utilized. Unfortunately, according to recent studies, 50% to 80% of the execution time in database workloads is spent waiting for instructions or data. Related work in the database community has proposed (a) modular and pipelined database system designs for parallelism, extensibility, or continuous query performance, (b) mechanisms to share and reuse data and work across different queries, and, (c) cache-conscious algorithms to improve cache performance of query execution. The operating systems community has proposed (a) techniques for efficient threading support, (b) event-driven and locality-aware staged server designs for scalability and performance, and, (c) scheduling algorithms for cache performance and low response times.

Although related work identifies memory-related bottlenecks and proposes techniques to boost performance, current DBMS designs do not have the means to exploit commonality across all levels of the memory hierarchy. Despite the multitude of proposed optimizations, these are inherently limited by the sheer volume of data that DBMS need to process and by the unpredictable sequence of memory requests. Most of the research to date for improving locality examines data accessed and instructions executed by a single query (or transaction) at a time. Database systems, however, typically handle multiple concurrent users. Request concurrency adds a new dimension for addressing the locality optimization problem. Unfortunately, the operating system's context switching decisions are oblivious to the state of the request/thread and therefore cause severe context thrashing in the memory hierarchy.

By properly synchronizing and multiplexing the concurrent execution of multiple requests there is a potential of increasing both data and instruction reusability at all levels of the memory hierarchy. Existing DBMS designs, however, pose difficulties in applying such execution optimizations. Since typically each database query or transaction is handled by one or more processes (threads), the DBMS essentially relinquishes execution control to the OS and then to the CPU. A new design is needed to allow for application-induced control of execution.

The solution is a new DBMS software architecture that allows for context switching on module boundaries, and for group scheduling of requests on a per-module basis. This thesis applies the principles behind the staged programming paradigm to relational database system design and presents a new DBMS software architecture for optimizing data and instruction locality at all levels of the memory hierarchy. The Staged Database System design breaks database request execution in stages and processes a group of sub-requests at each stage, thus effortlessly exploiting data and work commonality.

Chapter 3

Designing a Staged Database System

Traditional database system architectures face a rapidly evolving operating environment, where millions of users store and access terabytes of data. In order to cope with increasing demands for performance, high-end DBMS employ parallel processing techniques coupled with a plethora of sophisticated features. However, the widely adopted, work-centric, thread-parallel execution model entails several shortcomings that limit server performance when executing workloads with changing requirements. Moreover, the monolithic approach in DBMS software has lead to complex and difficult to extend designs.

In this chapter we introduce the StagedDB design for high-performance, evolvable DBMS that are easy to tune and maintain. StagedDB breaks the DBMS software into multiple modules and encapsulates them into self-contained stages connected to each other through queues. Rather than rewriting the entire code of the database system, we provide the support to organize system components into stages and change request execution sequence to perform group-processing at each stage, thus effortlessly exploiting commonality across queries. This chapter shows how this staged, query-centric approach remedies the weaknesses of modern DBMS by providing solutions at both a hardware and a software engineering level.

3.1. Introduction

Modern commercial DBMS are built as a large piece of software that typically serves multiple requests using a thread-based concurrency model. Queries are handled by one or more threads (or processes) that follow the Query Execution Plan (QEP) up to its completion—hence the term work-centric: the query execution is centered around the work that needs to be done. While widely adopted, the problem with this model is that it implicitly defines a query execution sequence and a resource utilization schedule in the system. Whenever a thread blocks due to an I/O, an ungranted lock request, an internal synchronization condition, or due to an expiring quantum, the thread scheduler assigns the CPU to the next runnable thread of the highest priority. This context-switching mechanism creates a logical gap in the sequence of actions the DBMS performs. While a software developer optimizes the individual steps involved in a single query's execution, she or he typically has no means of applying similar optimization techniques to a collection of multiplexed queries. Therefore, the key to high performance is to take into account all concurrent requests when addressing database performance bottlenecks.

Looking from a software engineering point of view, years of DBMS software development have lead to monolithic, complicated implementations that are difficult to extend, tune, and evolve. While database software developers commonly organize code into separate components, the final product consists of tightly integrated and interdependent software modules, "glued" together to eliminate overheads and increase performance. This practice, however, makes it difficult to profile individual components, fine-tune them, or even trace bugs across components. Moreover, the DBMS is viewed as an "all or nothing" application, with no support for partial services involving only a small subset of the software components. To allow for a flexible database architecture, a new design is needed that can combine the existing system components while at the same time improves performance when compared to a tightly integrated traditional design.

This chapter describes the Staged Database System (StagedDB) design for high-performance, evolvable DBMS. The core idea is to provide the support to organize system components into self-contained stages and change request execution sequence to perform group-processing at each stage. A stage acts as an independent server with its own queue for the incoming requests, thread support, and resource management. This staged, query-centric approach improves current DBMS designs by providing solutions (a) at the hardware level: it optimally exploits the underlying memory hierarchy and takes direct advantage of new architectures such as Chip Multiprocessors (CMP), and (b) at a software engineering level: it aims at a highly flexible platform that is easy to program and tune.

In the remaining of the chapter, we first describe in more detail the pitfalls of thread-based concurrency (Section 3.2) and the problems of monolithic DBMS software (Section 3.3). We then present the main components of StagedDB (Section 3.4), while in Section 3.5 we analyze and provide mechanisms to overcome the main performance tradeoff when staging server software. We describe both performance and software engineering benefits in Section 3.6. The last section draws the chapter's conclusions.

3.2. Pitfalls of thread-based concurrency

To best utilize the available resources, DBMS typically use a pool of threads or processes. Each incoming query is handled by one or more threads, depending on its complexity and the number of available CPUs. Each thread executes until it either blocks on a synchronization condition, an I/O event, or until a predetermined time quantum has elapsed. Then, the CPU switches context and executes a different thread or the same thread takes on a different task. Context-switching typically relies on generated events instead of program structure or the query's current state. While this model is intuitive, it has several shortcomings:

- There is no single number of preallocated worker threads that yields optimal performance under changing workloads.
- Preemption is oblivious to the thread's current execution state.
- Round-robin thread scheduling does not exploit cache contents that may be common across a set of threads.

These shortcomings, along with their tradeoffs and challenges are further discussed over the next paragraphs.

3.2.1 Choosing the right thread pool size

Although multithreading is an efficient way to mask I/O and network latencies and fully exploit multiprocessor platforms, many researchers argue against thread scalability [Pai et al. 1999; Welsh et al. 2001]. Related studies suggest (a) maintaining a thread pool that continuously picks clients from the network queue to avoid the cost of creating a thread per client arrival, and (b) adjusting the pool size to avoid an unnecessarily large number of threads. Modern commercial DBMS typically adopt these guidelines. The database administrator (DBA) is responsible for statically adjusting the thread pool size. The trade-off the DBA faces is that a large number of threads may lead to performance degradation caused by increased cache and TLB misses, and thread scheduling overhead. On the other hand, too few threads may restrict concurrency, since all threads may block while there is work the system could perform. The optimal number of threads depends on workload characteristics which may change over time. This further complicates tuning[1].

To illustrate the problem, we performed the following experiment using PREDATOR [Seshadri et al. 1997], a research prototype DBMS, on a 1GHz Pentium III server with 512MB RAM and Linux 2.4. We created two workloads, A and B, designed after the Wisconsin benchmark [DeWitt 1993]. Workload A consists of short (40-80msec), selection and aggregation queries that almost always incur disk I/O. Workload B consists of longer join queries (up to 2-3 secs) on tables that fit entirely in main memory and the only I/O needed is for logging purposes. We modified the execution engine of PREDATOR and added a queue in front of it. Then we converted the thread-per-client architecture into the following: a pool of threads that picks a client from the queue, works on the client until it exits the execution engine, puts it on an exit queue and picks another client from the input queue. By filling the input queue with already parsed and optimized queries, we could measure the throughput of the execution engine under different thread pool sizes.

Figure 3-1 shows the throughput achieved under both workloads, for different thread pool sizes, as a percentage of the maximum throughput possible under each workload. Workload A's throughput reaches a peak and stays constant for a pool of twenty or more threads. When there are fewer than twenty threads, the I/Os do not completely overlap, and thus there is idle CPU time, resulting in slightly lower throughput[2]. On the other hand,

1. Quoting the DB2 performance tuning manual ("agents" are implemented using threads or processes): "If you run a decision-support environment in which few applications connect concurrently, set {num_pool_agents} to a small value to avoid having an agent pool that is full of idle agents. If you run a transaction-processing environment in which many applications are concurrently connected, increase the value of {num_pool_agents} to avoid the costs associated with the frequent creation and termination of agents."

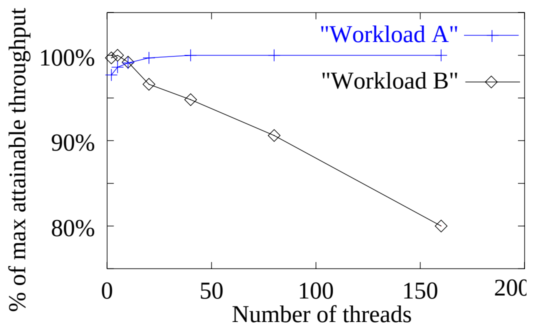


Figure 3-1. Different workloads perform differently as the number of threads changes.

Workload B's throughput severely degrades with more than 5 threads, as there is no I/O to hide and a higher number of longer queries interfere with each other as the pool size increases. The challenge is to discover an adaptive mechanism with low-overhead thread support that performs consistently well under frequently changing workloads.

3.2.2 Preemptive context-switching

A server's code is typically structured as a series of logical operations (or procedure calls). Each procedure (e.g., query parsing) typically includes one or more sub-procedures (e.g., symbol checking, semantic checking, query rewriting). Furthermore, each logical operation typically consists of loops (e.g., iterate over every single token in the SQL query and symbol table look-ups). When the client thread executes, all global procedure and loop variables along with the data structures that are frequently accessed (e.g., symbol table) consist the thread's working set. Context-switches that occur in the middle of an operation evict its working set from the higher levels of the cache hierarchy. As a result, each resumed thread often suffers additional delays while re-populating the caches with its evicted working set.

However, replacing preemption with cooperative scheduling where the CPU yields at the boundaries of logical operations may lead to unfairness and hurt average response time. The challenge is to (a) find the points at which a thread should yield the CPU, (b) build a mechanism that will take advantage of that information, and (c) make sure that no execution path holds the CPU too long, leading to unfairness.

2. We used user-level threads which have a low context-switch cost. In systems that use processes or kernel threads (such as DB2), increased context-switch costs have a greater impact on throughput.

classification	data	code
PRIVATE	Query Execution Plan, client state, intermediate results	NO
SHARED	tables, indices	operator specific code (e.g., nested-loop vs. sort-merge join)
COMMON	catalog, symbol table	rest of DBMS code

Table 3-1. Data and code references across all queries

3.2.3 Round-robin thread scheduling

The thread scheduling policy is another factor that affects memory affinity. Currently, selection of the next thread to run is typically done in a round-robin fashion among equal-priority threads. The scheduler considers thread statistics or properties unrelated to its memory access patterns and needs. There is no way of coordinating accesses to common data structures among different threads in order to increase memory locality.

Table 3-1 shows an intuitive classification of commonality in data and code references in a database server. Private references are those exclusive to a specific instance of a query. Shared references are to data and code accessible by any query, although different queries may access different parts. Lastly, common references are those accessed by the majority of queries. Current schedulers miss an opportunity to exploit shared and common references and increase performance by choosing a thread that will find the largest amount of data and code already fetched in the higher levels of the memory hierarchy.

In order to quantify the performance penalty, we performed the following experiment. We measured the time it takes for two similar, simple selection queries to pass through the parser of PREDATOR under two scenarios: (a) after the first query finishes parsing, the CPU works on different, unrelated operations (e.g., optimize another query, scan a table) before it switches into parsing the second query, and, (b) the second query starts parsing immediately after the first query is parsed (the first query suspends its execution after exiting the parser). Using the same setup as in 3.2.1, we found that Query 2 improves its parsing time by 7% in the second scenario, since it finds part of the parser's code and data structures already in the server's cache. As we show in simulation results in Section 3.5, even such a modest average improvement across all server modules results into more than 40% overall response time improvement when running multiple concurrent queries at high system load.

However, a thread scheduling policy that suspends execution of certain queries in order to make the best use of the memory resources may actually hurt average response times. The trade-off is between decreasing cache misses by scheduling all threads executing the same software module together, while increasing response time of other queries that need to access different modules. The challenge is to find scheduling policies that exploit a module's affinity to memory resources while improving throughput and response time.

3.3. Pitfalls of monolithic DBMS design

Extensibility. Modern DBMS are difficult to extend and evolve. While commercial database software offers a sophisticated platform for efficiently managing large amounts of data, it is rarely used as stand-alone service. Typically, it is deployed in conjunction with other applications and services. Two common usage scenarios are the following: (a) Data streams from different sources and in different form (e.g., XML or web data) pass through "translators" (middleware) which act as an interface to the DBMS. (b) Different applications require different logic which is built by the system programmer on top of the DBMS.

The above-mentioned scenarios may deter administrators from using a DBMS as it may not be necessary for simple purposes, or it may not be worth the time spent in configurations. A compromise is to use plain file servers that will cover most needs but will lack in features. DBMS require the rest of the services and applications to communicate with each other and coordinate their accesses through the database. The overall system performance degrades since there is unnecessary CPU computation and communication latency on the data path. The alternative, extending the DBMS to handle all data conversions and application logic, is a difficult process, since typically there is no well-defined API and the exported functionality is limited due to security concerns.

Tuning. Database software complexity makes it difficult to identify resource bottlenecks and properly tune the DBMS in heavy load conditions. A DBA relies on statistics and system reports to tune the DBMS, but has no clear view of how the different modules and resources are used. For example, the optimizer may need separate tuning (e.g., to reduce search space), or the disk read-ahead mechanism may need adjustment. Current database software can only monitor resource or component utilization at a coarse granularity (e.g., total disk traffic or table accesses, but not concurrent demand to the lock table). Based solely on this information it is difficult to build automatic tuning tools to ease DBMS administration. Furthermore, when client requests exceed the database server's capacity (overload conditions) then new clients are either rejected or they experience significant delays. Yet, some of them could still receive fast service (e.g., if they only need a cached tuple).

Maintainability. An often desirable property of a software system is the ability to improve its performance or extend its functionality by releasing software updates. New versions of the software may include, for example, faster implementations of some algorithms. For a complex piece of software, such as a DBMS, it is a challenging process to isolate and replace an entire software module. This becomes more difficult when the programmer has no previous knowledge of the specific module implementation. The difficulty may also rise from a non-modular coding style, extended use of global variables, and module interdependencies.

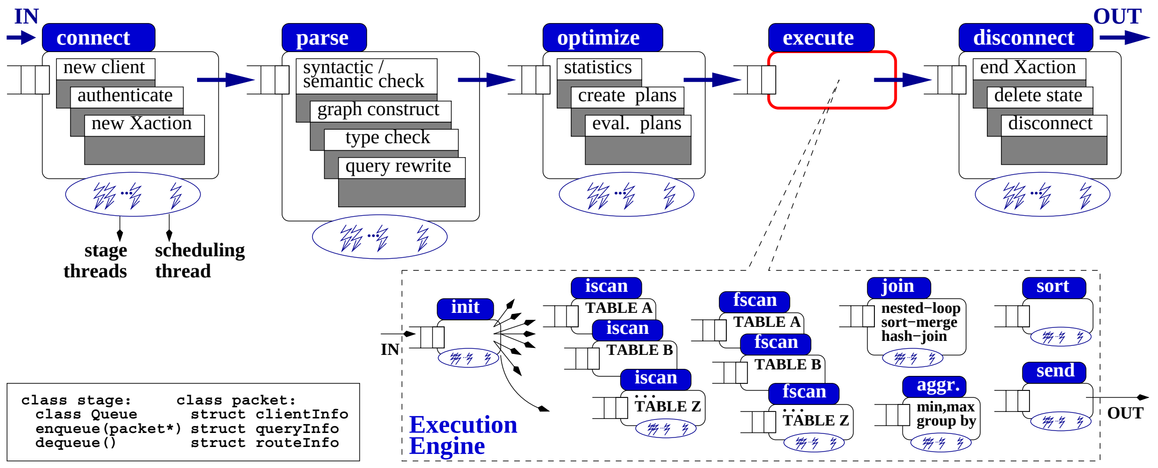
Testing and debugging. Large software systems are inherently difficult to test and debug. The test case combinations of all different software components and all possible inputs are practically countless. Once errors are detected, it is difficult to trace bugs through millions of lines of code. Furthermore, multithreaded programs may exhibit race conditions (when there is need for concurrent access to the same resource) that may lead to deadlocks. Although there are tools that automatically search program structure at run time to expose possible race conditions [Savage et al. 1997], they may slow down the executable or increase the time to software release. A monolithic software design makes it even more difficult to develop code that is deadlock-free since accesses to shared resources may not be contained within a single module.

3.4. A staged approach for DBMS software

A staged database system consists of a number of self-contained modules, each encapsulated into a stage. A stage is an independent server with its own queue, thread support, and resource management that communicates and interacts with the other stages through a well-defined interface. Stages accept packets, each carrying a query's state and private data (the query's *backpack*), perform work on the packets, and may enqueue the same or newly created packets to other stages. The first-class citizen is the query, which enters stages according to its needs. Each stage is centered around exclusively owned (to the degree possible) server code and data. There are two levels of CPU scheduling: local thread scheduling within a stage and global scheduling across stages. This design promotes stage autonomy, data and instruction locality, and minimizes the usage of global variables.

We divide at the top level the actions the database server performs into five query execution stages (see Figure 3-2): connect, parse, optimize, execute, and disconnect. The execute stage (*query execution engine*) typically represents the largest part of a query's lifetime and is further decomposed into several stages (described in Section 3.4.2). The break-up objective is (a) to keep accesses to the same data structures together, (b) to keep instruction loops within a single stage, and (c) to minimize the query's backpack. For example, connect and disconnect execute common code related to client-server communication: they update the server's statistics, and create/destroy the client's state and private data. Likewise, while the parser operates on a string containing the client's query, it performs frequent lookups into a common symbol table.

The design in Figure 3-2 is general enough to apply to any modern relational DBMS, with minor adjustments. For example, commercial DBMS support precompiled queries that bypass the parser and the optimizer. In our design the query can route itself from the connect stage directly to the execute stage. Figure 3-2 also shows certain operations performed inside each stage. Depending on each module's data footprint and code size, a stage may be further divided into smaller stages that encapsulate operation subsets (to better match the cache sizes).



```
class stage:      class packet:
  class Queue       struct clientInfo
  enqueue(packet*)  struct queryInfo
  dequeue()         struct routeInfo
```

Figure 3-2. The Staged Database System design: Each stage has its own queue and thread support. New queries queue up in the first stage, they are encapsulated into a “packet”, and pass through the five stages shown on the top of the figure. A packet carries the query’s “backpack”: its state and private data. Inside the execution engine, queries issue multiple packets to increase parallelism.

There are two key elements in the proposed system: (a) the stage definition along with the capabilities of stage communication and data exchange, and (b) the redesign of the relational execution engine to incorporate a staged execution scheme. These are discussed next.

3.4.1 Stage definition

A stage provides two basic operations, *enqueue* and *dequeue*, and a queue for the incoming packets. The stage-specific server code is contained within dequeue. The proposed system works through the exchange of *packets* between stages. A packet represents work that the server must perform for a specific query at a given stage. It first enters the stage’s queue through the enqueue operation and waits until a dequeue operation removes it. Then, once the query’s current state is restored, the stage specific code is executed. Depending on the stage and the query, new packets may be created and enqueued at other stages. Eventually, the stage code returns by either (i) destroying the packet (if done with that query at the specific stage), (ii) forwarding the packet to the next stage (i.e., from parse to optimize), or by (iii) enqueueing the packet back into the stage’s queue (if there is more work but the client needs to wait on some condition). Queries use packets to carry their state and private data. Each stage is responsible for assigning memory resources to a query. As an optimization, in a shared-memory system, packets can carry only pointers to the query's state and data structures (which are kept in a single copy).

Each stage employs a pool of worker threads (the stage threads) that continuously call dequeue on the stage's queue, and one thread reserved for scheduling purposes (the scheduling thread). More than one threads per stage help mask I/O events while still executing in the same stage (when there are more than one packets in the queue). If all threads happen to suspend for I/O, or the stage has used its time quantum, then a stage-level scheduling policy specifies the next stage to execute. Whenever enqueue causes the next stage's queue to overflow we apply back-pressure flow control by suspending the enqueue operation (and subsequently freeze the query's execution thread in that stage). The rest of the queries that do not output to the blocked stage will continue to run.

3.4.2 A staged relational execution engine

In a staged query engine each relational operator is assigned to a stage. This assignment is based on the operator's physical implementation and functionality. We group together operators which use a small portion of the common or shared data and code (to avoid stage and scheduling overhead), and separate operators that access a large common code base or common data (to take advantage of a stage's affinity to the processor caches). The dashed box in Figure 3-2 shows the individual execution engine stages.

Although control flow amongst operators/stages still uses packets as in the top-level DBMS stages, data exchange within the execution unit exhibits significant peculiarities. Firstly, stages do not execute sequentially anymore. Secondly, multiple packets (as many as the different operators involved) are issued per each active query. Finally, control flow through packet enqueueing happens only once per query per stage, when the operators/stages are activated. This activation occurs in a bottom-up fashion with respect to the operator tree, after the *init* stage enqueues packets to the leaf node stages (similarly to the "push-based" model [Graefe 1996] that aims at avoiding early thread invocations). Dataflow takes place through the use of intermediate result buffers and page-based data exchange using a producer-consumer type of operator/stage communication.

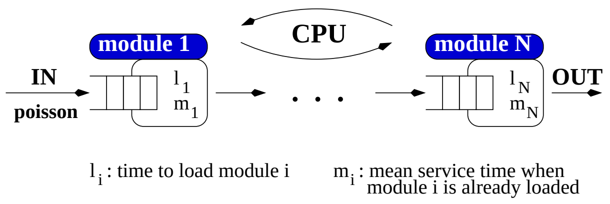

CPU
module 1
module N
IN
poisson
l_1
m_1
l_N
m_N
OUT
l_i : time to load module i
m_i : mean service time when module i is already loaded

Figure 3-3. A production-line model for staged servers.

Based on the above-mentioned design guidelines, Chapter 4 describes in detail and evaluates *QPipe*, a staged relational query engine, that can enhance locality across concurrent queries. Before we describe the benefits of StagedDB, the next section addresses the main performance tradeoff in staging server software.

3.5. The scheduling tradeoff

Any staged server architecture, including StagedDB, exhibits a fundamental scheduling tradeoff: On one hand, all requests executing as a batch in the same module benefit from enhanced cross-request locality. On the other hand, each completed request suspends its progress until the rest of the batch finishes execution, thereby increasing response time. In this section we investigate this tradeoff and propose scheduling policies that offer a significant performance advantage for the staged DBMS design. To compare alternative strategies for forming and scheduling query batches at various degrees of inter-query locality, we develop a simple simulated execution environment, based on a “production-line” operation model (i.e., requests go through a series of stages only once and always in the same order), that is also analytically tractable.

3.5.1 Problem formulation

We consider the case of a single-CPU database server with a memory-resident workload. Each query submitted to the server passes through several stages of execution, and each one of those stages corresponds to a server *module* (see also Figure 3-3). A module is defined as an autonomous, in terms of the data structures owned and accessed, part of the work that the database server performs in response to a request. An example of such a module is the parser or the optimizer of the database. Data structures referenced and code executed by two different queries during the execution of a single module can overlap by a variable amount of instructions and data. For instance, two different queries may use different execution operators, produce their own plans, and access different tuples; however, they will reference common code such as buffer pool code or index look-up, and common data such as the symbol table or the database catalog.

Once the common data structures and instructions of a module are accessed and loaded in the cache, subsequent executions of different requests within the same module will significantly reduce memory delays. To model this behavior, we charge a query with an additional CPU demand (quantity l_i in Figure 3-3) whenever the query starts execution at a certain module and the common data structures are not already in the cache (that is, the CPU was previously working on a different module). This extra CPU demand represents the time spent on average in memory stalls attributed to data and code shared by all queries. The model assumes, without loss of generality, that the entire set of a module's data structures that are shared on average by all requests can fit in the higher levels of the memory hierarchy, and that a total eviction of that set takes place when the CPU switches to a different module.

The prevailing scheduling discipline, Processor-sharing (PS), fails to reuse cache contents, since it switches from query to query in a random way with respect to the query’s current execution module. To exploit the data locality across requests, we propose techniques for scheduling queries between different modules, and evaluate them against PS. In order to compare the different scheduling policies we assume a Poisson stream of queries, whereas the query service time is based on several analytical distributions.

Clearly, several of the assumptions related to the system architecture (single CPU, in-memory DBMS, context switch costs, common data structure sizes and cache behavior) are simplifications used to understand the problem and to allow for a meaningful evaluation of the scheduling policies. Relaxing these assumptions does not affect the generality of the results, as we briefly discuss below.

High-end commercial database system installations typically run on multiprocessor systems. Although this essentially adds a degree of freedom to the space of possible scheduling policies, it is straightforward to extend the proposed approach to include execution on multiple CPUs: we divide the modules into groups and assign each group to a processor. The analysis in this section then holds for each individual CPU. The results drawn using this model may easily be transferred from the processor/cache/memory to the CPU/memory/disk hierarchy. I/O interrupts cause the suspension of a thread execution, before the end of the time slice assigned to that thread. The assumption of an in-memory DBMS removes the need of premature thread preemption (except for synchronization purposes), and thus, allows for a simpler execution model. Nevertheless, blocking threads do not affect the way the proposed policies work since in our implementation each stage is served by a pool of threads, allowing this way the CPU to overlap I/O events among the queries consisting the processed batch.

We do not model context-switch overhead, since this cost applies to all policies and thus it does not affect the relative performance gains. We also chose to model the locality benefits of staged execution with a single parameter per module. This parameter is a combination of all the code/data overlaps between different queries executing in a given module. Although the execution of the first query in a batch will not always fetch all common data and code of a module, and a context-switch may not always cause the eviction of all the common data and code, this single parameter nevertheless captures the average behavior. For instance, in Chapter 5, we show how to stage code appropriately to exploit common instructions in the cache across queries, and find that a single parameter can describe the amount of overlap.

The assumption of exponentiality with respect to query interarrival times (Poisson arrivals) does not match workloads universally, but it provides a tractable model for comparing the different policies. For the specific problem under consideration, bursty arrivals will only increase the potential of locality-aware scheduling, and thus, the use of Poisson arrivals is adequate for qualitatively comparing the different scheduling policies. Given the assumptions mentioned, the exact problem definition follows.

3.5.1.1 Problem definition

Queries arrive at a DBMS server according to a Poisson process with rate λ. Each query passes through a series of M modules, always in the same order. Each module has its own

symbol	explanation	value
M	number of modules	to be set at the experimentation section
$\alpha_i, \sum_{i=1}^{M} \alpha_i = 1$	fraction of a query's total execution time spent at module *i*	
λ	query arrival rate	
m	mean query service time, when all common data+code is found in cache	
l	total time a query spends loading in cache common data+code	
m_i	mean query service time, at module *i,* when common data+code is found in cache	$\alpha_i \times m$
l_i	time a query spends at module *i*, loading in cache common data+code	$\alpha_i \times l$
ρ	system load	$\lambda \times (m + l)$

Table 3-2. Symbol definitions

separate queue. There is only one CPU in the system. Whenever a query arrives at the server, its service time is drawn from an analytical distribution with mean *m*; when that query executes at a given module *i*, it spends time m_i (see Table 3-2 for symbol definitions). If the immediately previous query execution happened at a different module, then the CPU also spends time l_i (either fixed or drawn from a distribution) to load module *i* common contents in the cache. The goal is to devise a scheduling policy that minimizes the average query response time.

The baseline scheduling policies are First Come First Serve (FCFS) and the prevailing one, PS. Under FCFS, whenever a query arrives at module *1,* it executes and then continues with module *2*, until it has executed all *M* modules. New queries that arrive at module *1* will have to wait until the current query exits the system. Since there is only one CPU, the only queue that actually accumulates input is the one at the entrance of module *1*. Like PS, FCFS also fails to reuse cache contents (each newly loaded module wipes the previous one from the cache).

3.5.2 Proposed scheduling policies

The execution flow in the model is purely sequential, thereby reducing the search space for scheduling policies into combinations of the following parameters: the number of queries that form a batch at a given module (one, several, all), the time they receive service (until completion or up to a cutoff value), and the module visiting order. Following are the four proposed scheduling policies.

D-gated *(Dynamic-gated)*. This policy dynamically imposes a gate on the incoming queries, and executes the admitted group of queries as a batch at each module, until their completion. Execution takes place in a first-come first-served basis at the queue of each module. When the first query in the queue of module *1* starts execution, it fetches the common data structures of the module in the cache. The rest of the queries that form the current batch pass through module *1* without paying the penalty of loading the common data structures in the cache. When the last query of the batch finishes execution at module *1*, the CPU shifts to the queue of module *2* and again processes the whole batch in a FCFS fashion. Meanwhile, incoming queries to the database server are queued up, at the first module's queue. Eventually, the current batch moves to the last module, and each query leaves the system immediately after its execution. Then, the CPU shifts to the first module and marks the new batch of admitted queries. These are the queries that have accumulated so far in the first module's queue. From that point of time and on, a gate is imposed to all incoming queries, for the duration of the next batch's execution. Since the gate defines a batch size each time module *1* resumes execution, we call this policy Dynamic-gated, or *D-gated*. Whenever the CPU shifts to module *1* and the queue is empty, D-gated reduces to plain FCFS. Note that D-gated is a cache-conscious scheduling policy since it only pays the penalty of loading a module in the cache once per batch of queries.

T-gated(*N*) *(Threshold-gated)*. This policy works similarly to D-gated, except for the way it specifies the size of the admitted batch of queries. T-gated explicitly defines an upper threshold *N* for the number of queries that will pass through module *1* and form a batch of maximum size *N*. If more than *N* queries have queued up in module *1* when the CPU finishes with the previous batch, T-gated will admit just *N* queries while the rest will be considered for the next batch. For *N*=1, this policy reduces to FCFS.

non-gated. This policy admits all queries queued up in module *1*, and works on module *1* until the queue becomes empty. At that point the CPU moves to the next module and proceeds in the same fashion as D-gated and T-gated. Once the current batch exits the system, the CPU shifts to module *1* and keeps admitting queries until there is no more work to be done at module *1*. *Non-gated* is more sensitive to starvation, since a continuous stream of queries might cause the server to work indefinitely on module *1*. While the analytic workloads we used did not produce this behavior, the use of a time-out mechanism is necessary in a real implementation. This policy is similar to the one described in [Larus and Parkes 2002] (when applied to our model).

C-gated *(Cutoff-gated)*. One possible issue with the previous policies is that a very large query can essentially block the way to other, smaller ones, and thus lead momentarily to higher response times. With an exponential distribution of query sizes, this issue does not have an impact on the average response time. This is because the majority of the system load is attributed to relatively small query sizes (close to the mean). A heavy-tailed distribution on the other hand, typically involves half the system load to be made up by infrequent but very large queries. In a scenario like that, FCFS based policies could lead to unreasonably high response times, compared to processor-sharing. C-gated tries to bridge the cache-awareness that D-gated and T-gated exhibit with the fairness in the presence of large queries that PS shows. Under C-gated, apart from the imposed gate to the incoming queries at the first module (either dynamically or as a predefined threshold), an additional cutoff value applies to the time the CPU spends on a given query at a given module. Whenever the CPU exceeds that cutoff value, it switches execution to the rest of the queries in the queue and to the next module, leaving the large query unfinished. That query will rejoin the next batch, and eventually resume execution. Whenever its remaining CPU demand for the current module drops below the cutoff value, it will advance to the next module. This way, both small and large queries make progress while still benefiting from increased data locality. The cutoff technique resembles the *Foreground-Background* scheduling policy used in Unix, where large jobs, when identified, are pushed in a separate queue and receive service only when there are no jobs in the default queue. The difference is that under C-gated large queries receive service, even at the presence of small queries, once per batch passing.

The analysis of PS and FCFS under the proposed model, as well as the analysis of staged locality-aware policies, is in the Appendix of the dissertation (page 141). Next, we evaluate and compare the proposed policies against PS and FCFS.

3.5.3 Evaluation

In all of our experiments, we set *M*, the number of modules, equal to five. For simplicity, an equal percentage of service time breakdown is assigned to the five different modules (that is, a query spends equal time in all modules or $a_i = 0.2$, for all *i*). PS and FCFS are not affected by the number of modules neither the service time breakdown. The gated algorithms can actually benefit by a biased assignment of service times to the different modules. This happens when queries spend a significant amount of time at the last module. On average, those queries will leave the server faster since they execute mostly in a FCFS fashion (and thus, are not delayed by all queries in the batch) and benefit from module locality at the same time. Since this is not typically the case in a real DBMS, the number of modules and service time breakdown remain the same through all experiments.

Whenever a query arrives at the server, its service time is drawn from two distributions: exponential and bounded Pareto (a highly variable distribution with a cutoff value for the maximum service time; see Section 3.5.3.3 for details), both with mean *m*. Note that this is the minimum amount of time that the CPU needs to work on the query, and reflects code execution and accesses to data that are private to the given query (and may be carried through successive stages). A query may also spend on average time l_i at each module *i* (a total time of *l* for all modules), loading the common data structures and instructions of that module.We choose a deterministic value for the module loading time, *l*, that is equal for all queries and varies in the experiments as a percentage of the total expected execution time of a query in a default, non-staged configuration (*m*+*l*). While in both cases all policies performed almost the same (with the exception of FCFS which is discussed later; PS performed exactly the same), a deterministic module loading time fits better in the context of a database server, since all queries need to execute and access a fairly predictable common set of instructions and data structures at all modules. The gated family of policies performed almost the same in both cases because it is the average value of the common code and data size that affects the locality benefits and not the distribution of it.

parameter	variance	value range
M, number of modules	fixed	5
a_i, fraction of execution time at module i	fixed	0.2
λ, query arrival rate	Poisson	0-12 queries/sec
$m+l$, query service time, no module loaded	see below for m, l	mean = 100ms
m, query service time, all modules loaded	exponential, bounded pareto	mean = 0-100% of 100ms
l, common data+code loading time	equal for all queries	0-100% of 100ms

Table 3-3. simulation parameters.

The total expected query execution time, $m+l$, (or mean CPU demand) refers to the case where all common instructions and data structures need to be loaded in the cache (and which always is the case for FCFS and PS), and is set to 100ms. Since other values for the total mean CPU demand resulted in the same relative differences in response times among all policies tested, all performance graphs are based only on that value. The results for both PS and FCFS are derived from analytical formulas for the M/G/1 queue, as those are described in the Appendix of the dissertation. The formula for PS reflects a best-case scenario where there is no penalty for the more frequent (when compared to the other policies) context-switches. FCFS is actually the same as T-gated(1) and so the formula was also validated against the simulation scripts. The confidence intervals were very tight and so they are omitted, for better graph readability. Table 3-3 shows all the experimentation parameters, along with their value range.

3.5.3.1 *Effect of various degrees of data locality*

In the first experiment, the variable component of the mean query CPU demand is drawn from an exponential distribution with mean m in the range of 40-100ms. Initially, the query arrival rate is set to produce a system load of 80%. This experiment compares the mean response time for a query under PS, FCFS, D-gated, non-gated and T-gated(2), for various module loading times (the time it takes all modules to fetch the common data structures and code in the cache, l). This time varies as a percentage of the mean query CPU demand, from 0% to 60% (the mean query service time that corresponds to private

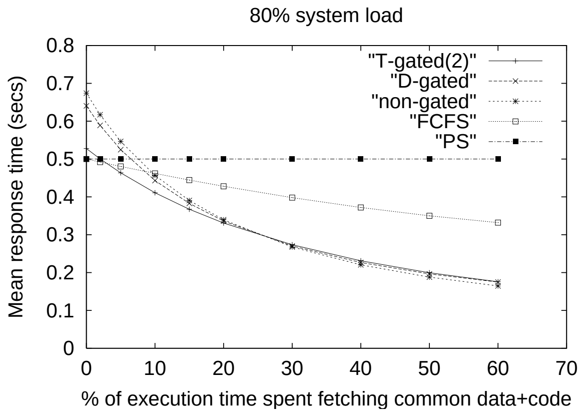


Figure 3-4. Mean response times for 80% system load.

data and instructions, m, is adjusted accordingly so that $m+l=100$ms). This value (l) can also be viewed as the percentage of execution time spent servicing cache misses, attributed to common instructions and data, under the default server configuration (e.g., using PS). The results are in Figure 3-4.

The graph of Figure 3-4 shows that the gated family of algorithms performs better than PS for module loading times that account for more than 8% of the query execution time. Response times are up to twice as fast and improve as module load time becomes more significant. On the other hand, for module loading times that correspond to less than 8% of the execution time, non-gated and D-gated policies show up to 20% worse response times. Among those two gated policies, D-gated performs best. T-gated(2) performs consistently well, outperforming PS in almost all configurations. The reason that T-gated(2) performs better than D-gated or non-gated is because it closer approximates FCFS than D-gated does and thus, the first query of every batch of two queries is delayed by only one other query. When the benefits of cache hits are reduced, it is more important for a query not to be delayed by other queries. T-gated performed better in this scenario for a threshold value of $N = 2$. Note that FCFS shows better response times than PS as the percentage of module loading time increases. This is because of the deterministic component (module loading time) in the mean CPU demand. A fixed part in the service time reduces variability and thus, queueing time delays. As an example, consider the case where the module loading time is 100ms (that is, no variability in the service time since it is always 100ms). Then,

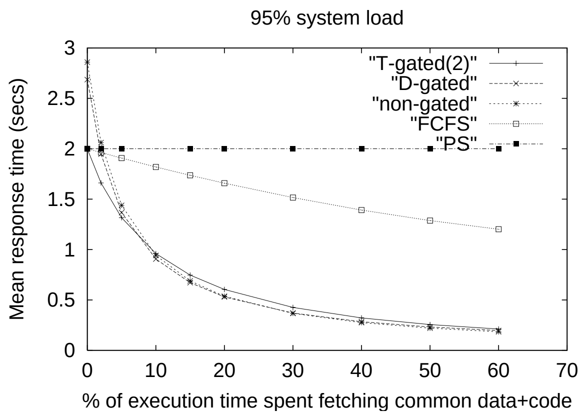


Figure 3-5. Mean response times for 95% system load.

two requests that arrive together will have an average of 150ms response time under FCFS, and 200ms under PS. In fact, it would always be better to use FCFS rather that PS.

For the graph in Figure 3-5, the same experimental setup is used, but the arrival rate is set to create a system load of 95%. As the system load increases, the trends in the gated family of policies do not change. The performance of PS and FCFS though drops significantly and as a result, gated policies perform better for almost all values of module loading time percentages (> 2%). The gains of the gated policies now increase to up to 7 times reduced response times. T-gated(2) is still the policy of choice, since it never loses to PS. Moreover, its ability of improving the performance (when compared to PS), even when the common memory references are low, outweigh its slightly worse performance than D-gated and non-gated for high percentages of common memory references.

3.5.3.2 Direct comparison of T-gated(2) and PS

This experiment directly compares PS and T-gated(2) (T-gated with *N* set to 2) by plotting the area that each of those policies results in lower response times. The same exponential distribution is used for the query sizes. The graph in Figure 3-6 is produced by varying both the system load and the module loading times. It shows the relative speedup of T-gated(2) over PS, for a wide range of different locality scenarios. The x-axis is the percentage of execution time that is eliminated for a query that finds the common data structures of a module in the cache. This value varies from 1% to 70%. The y-axis is the server load;

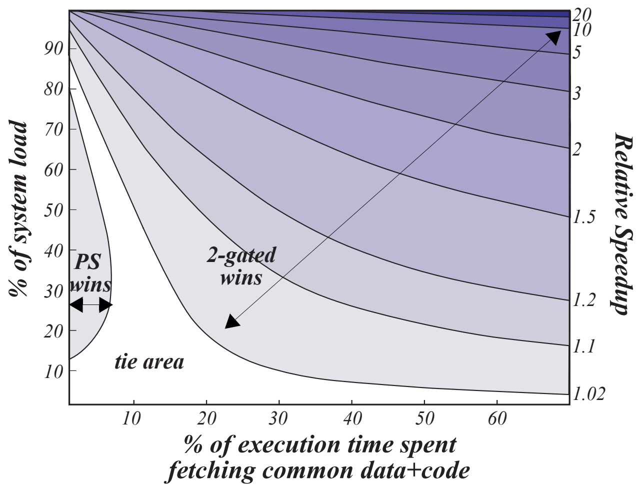


Figure 3-6. Direct comparison of PS and T-gated.

we varied the arrival rate to achieve server loads between 1% and 98%. On the right of the y-axis we denote the areas where the relative speedup of T-gated(2) over PS is within a certain range. Areas with darker color correspond to higher speedup, while the white area corresponds to those combinations that both policies perform almost the same. PS is only able to perform better than T-gated(2) in a small area on the left of the graph; the relative speedup of PS in that area does not exceed 1.1.

3.5.3.3 Effect of very large query sizes

For the last experiment, we study the effect of a highly variable distribution. The exponential distribution assumes that the remaining query service times are independent of the CPU time used so far. This is not true in many real workloads, where the few large queries are increasingly more likely to take more of the CPU time. This property exactly (known as decreasing failure rate) is characteristic of a heavy-tailed distribution, under which, a very small fraction of the largest queries can comprise as much as half of the system load. While PS, as a fair policy, is insensitive to different distributions, the rest of the policies are not. A really large query can block many smaller ones and incur higher mean response times. This experiment tests how well C-gated can push small queries out of the system fast, while still exploiting data locality.

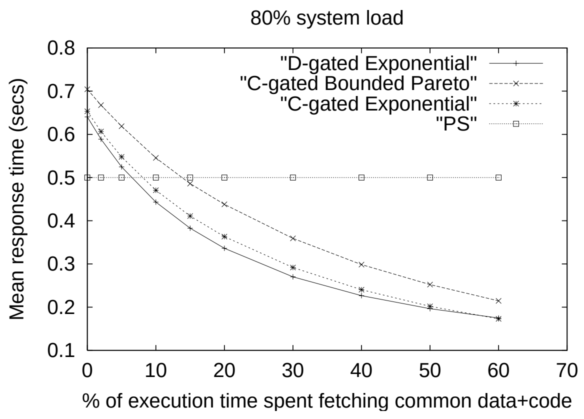


Figure 3-7. Highly variable distribution: Bounded Pareto.

We use the same distribution as in [Crovella et al. 1998], Bounded Pareto, for the variable component of the total query size, with fixed mean in the range of 40-100ms (depending on the choice for *l*, so that the mean query CPU demand remains 100ms), and a maximum query size of 10,000sec. Bounded Pareto has been shown to closely model task size distributions in real world workloads [Crovella et al. 1998]. For the C-gated policy we manually set a fixed cutoff value of 100ms (same as the mean CPU demand). In a real environment, this can be implemented by monitoring the query sizes and using the average value as the cutoff. We repeated the first experiment (Figure 3-4) but this time we tested C-gated under both the exponential and the Bounded Pareto distributions. FCFS, non-gated, D-gated, and T-gated explode under Bounded Pareto (they result into very large response times) and thus they do not appear in the plot. Instead, we show again D-gated response times under the exponential distribution. PS behaves exactly the same under both distributions (a well known result from queueing theory).

The results in Figure 3-7 show that C-gated under Bounded Pareto outperforms PS for module loading times of 15% or more of the mean query execution time. While C-gated under Bounded Pareto is worse when compared to D-gated under the exponential distribution, it nevertheless manages to exploit data locality through locality-aware scheduling and avoid the pitfall of working almost indefinitely on very large queries. Note that C-gated under the exponential distribution is only slightly worse than D-gated. This means that C-gated is the policy of choice when there is no apriori knowledge of the workload characteristics. By increasing the system load, the same trends as in the graph of Figure 3-5 are observed. That is, the gains of the gated family over PS increase.

3.6. Benefits of the StagedDB design

To assess the benefits of StagedDB we built a staged mechanism on top of PREDATOR [Seshadri et al. 1997], a single-CPU, multi-user, client-server, object-relational database system that uses the Shore [Carey et al. 1994] storage manager. The first decision we faced is how to break the system into stages, i.e., what the granularity of computation should be at each stage. We initially partitioned the database system into the same five high-level stages shown in Figure 3-2: a connection manager, parser, optimizer, execution engine, and a final stage to handle query results. Since the majority of query time is spent in the execution engine, we further map the different operators into five distinct stages (see also Figure 3-2): file scan (*fscan*) and index scan (*iscan*), for accessing stored data sequentially or with an index, respectively, *sort*, *join* which includes three join algorithms, and a fifth stage that includes the *aggregate* operators (min-max, average, etc.).

The rationale for the above-mentioned high-level partitioning is that existing prototypes already have well-defined boundaries for the five high-level components (commercial DBMS have more modules but the boundaries are often cleaner). Our approach into staging PREDATOR included three steps: (1) identifying stage boundaries in the base system and modifying the code to follow the staged paradigm (these were relatively straightforward changes that transformed the base code into a series of procedure calls and are not further discussed here), (2) adding support for stage-aware thread scheduling techniques, and (3) implementing page-based dataflow and queue-based control-flow schemes inside the execution engine.

A similar approach can apply to the process of staging a commercial DBMS which is far more complicated than our prototype. Each of the high-level stages will need to further break into smaller ones, following a recursive, top-down approach. The strategy will be largely guided by the functionality and performance requirements of each stage. The optimizer, for instance, is a computation-intensive module that consists of three easily identifiable components: candidate plan construction, access to statistics, and plan cost computation. A database software developer will first need to assign each of those components to a different stage and then recursively componentize until computation is evenly divided across stages (while keeping overhead low). For stages that include accesses to data structures common to different queries, accesses will need to be clustered separately into stages. The wizards, tools, and statistic collection mechanisms need to be assigned to separate stages.

3.6.1 Evaluation of StagedDB prototype implementation

The StagedDB prototype consists of 2,500 new lines of code (PREDATOR is 60,000 lines of C++) and supports most of the database functionality of the original system. Shore provides non-preemptive user-level threads that typically restrict the degree of concurrency in the system since a client thread yields only upon I/O events. We use this behavior to explicitly control the points at which the CPU switches thread execution. We incorporate the proposed affinity scheduling schemes (from Section 3.5) into the system's thread scheduling mechanism by rotating the thread group priorities among stages. For example, whenever the CPU shifts to the *parse* stage, the stage threads receive higher priority and keep executing dequeue until either (a) the queue is empty, or (b) the global scheduler imposes a gate on the queue, or (c) all working threads are blocked on a I/O event. The thread scheduler tries to overlap I/O events as much as possible within the same stage.

Queueing overhead. We conducted an experiment to examine the staged system's overhead due to additional enqueue-dequeue operations. We pick simple selection, aggregate, and 2- or 3-way join queries from the Wisconsin Benchmark [DeWitt 1993] on a small, memory-resident database, and send them one at a time to the original system and the staged one. Note that we expect the staged system to perform about the same or slightly worse, since we only submit one query at a time (no opportunity for enhancing cross-query locality). Moreover, the staged system uses multiple threads per query without being able to exploit inter-query parallelism (because of the single CPU and the memory-resident database). We measured a 0-2% performance degradation for the staged system. Note that this was the pure overhead, without including the benefits of staging. The staged system actually ran faster, even for a single query, by batching tasks at each stage and avoiding extra procedure calls.

Performance and software engineering benefits. Since each stage inside the execution engine processes multiple tuples (with a granularity of a page) on behalf of each query, the staged system improved performance over the original system which uses a tuple-by-tuple iterator evaluation model. Since each relational operator processes a batch of tuples, the staged system avoids extraneous procedure calls and achieves better instruction temporal locality (we address instruction cache optimizations in Chapter 5). Furthermore, the staged system naturally provides intra-query parallelism by employing as many stages as the nodes in a query's plan. When compared to the original system, the staged system was able to run faster queries containing I/O, since it overlaps I/O with computation of other stages. Lastly, in developing and testing code in both the original and the staged system, we were able to trace and isolate software bugs more easily in the staged system.

Our initial implementation of StagedDB on top of PREDATOR confirmed the intuition regarding the benefits of staging database software and contributed several promising research directions. In this dissertation, we further investigate how the staged execution engine can improve cross-query locality (Chapter 4) and how StagedDB can improve instruction cache performance for large, commercial-grade systems processing multiple concurrent transactions (Chapter 5). For the rest of the dissertation, the prototype DBMS used so far (PREDATOR) is discarded in favor of two alternative implementations that better suit the needs of each research direction.

For the staged execution engine (Chapter 4), we transfer our implementation of stages on top of the BerkeleyDB database storage manager. BerkeleyDB allows building applications that utilize native OS threads and can also run on multi-processor servers (PREDATOR runs on single-CPU systems and uses non-preemptive user-level threads). Having a staged implementation based on native OS threads is important because it provides portability, flexibility in constructing scheduling policies, and high performance.

For studying the instruction cache bottleneck in transaction processing (Chapter 5), we use our own high-performance runtime built directly on top of Shore. We chose to bypass PREDATOR (which itself is built on top of Shore), since it contains elaborate code to handle a wide range of data types that contributes a high overall overhead, and also does not support pre-compiled transactions (which is the norm in typical transaction processing installations). Finally, we chose Shore instead of BerkeleyDB for two reasons. First, Shore provides more advanced transaction processing features (e.g., it allows for record-level locking whereas BerkeleyDB currently offers only page-level locking). Second, modifying a user-level thread package instead of a native one, was a less demanding task. Moreover, instruction-cache results obtained from a single-CPU server are easy to generalize to multi-processors, since instruction caches are private to each CPU.

In the remaining of this section we discuss how the StagedDB design addresses the shortcomings of conventional DBMS architectures, as those presented in Sections 3.2 and 3.3.

3.6.2 Solutions to thread-based problems

The StagedDB design avoids the pitfalls of the traditional threaded execution model as those were described in Section 3.2 through the following mechanisms:

- Each stage allocates worker threads based on its functionality and the I/O frequency, and not on the number of concurrent clients. This way there is a well-targeted thread assignment to the various database execution tasks at a much finer granularity than just choosing a thread pool size for the whole system.
- A stage contains DBMS code with one or more logical operations. Instead of preempting the current execution thread at a random point of the code (whenever its time quantum elapses), a stage thread voluntarily yields the CPU at the end of the stage code execution. This way the thread's working set is evicted from the cache at its shrinking phase and the time to restore it is greatly reduced. This technique can also apply to existing database architectures.
- The thread scheduler repeatedly executes tasks queued up in the same stage, thereby exploiting stage affinity to the processor caches. The first task's execution fetches the common data structures and code into the higher levels of the memory hierarchy while subsequent task executions experience fewer cache misses. This type of scheduling cannot easily apply to existing systems since it would require annotating threads with detailed application logic.

3.6.3 Solutions to software-complexity problems

Stages provide a well-defined API and thus make it easy to:

- Replace a module with a new one (e.g., a faster algorithm), or develop and plug modules with new functionality. The programmer needs to know only the stage API and the limited list of global variables.
- Debug the code and build robust software. Independent teams can test and correct the code of a single stage without looking at the rest of the code. While existing systems offer sophisticated development facilities, a staged system allows building more intuitive and easier to use development tools.
- Encapsulate external wrappers or "translators" into stages and integrate them into the DBMS. This way we can avoid the communication latency and exploit commonality in the software architecture of the external components. For example, a unified buffer manager can avoid the cost of subsequent look-ups into each component's cache. A well-defined stage interface enables the DBMS to control distribution of security privileges.

3.6.4 Additional benefits of StagedDB

Multi-processor and multi-core systems. High-end DBMS typically run on multi-processor or multi-core (Chip Multiprocessors—CMP) systems. A staged system naturally maps one or more stages to a dedicated CPU or processor core, providing fine-grain intra-query parallelism. Stages may also migrate to different processors to match the workload requirements. Data and code locality benefits are even higher than in the single-CPU server, since fewer stages are exclusively using a single processor's cache. We discuss extensions to StagedDB for multi-core systems in the last chapter of the dissertation.

Multiple query optimization. Multiple query optimization [Sellis 1988] has been extensively studied over the past fifteen years. The objective is to identify and exploit subexpression commonality in a batch of queries during optimization and reduce execution time by reusing already fetched or computed input tuples. Commercial systems typically do not include a multi-query optimizer, since most applications process queries interactively and not in batches (e.g., e-commerce applications, reservation systems, ad-hoc decision support querying systems). A staged system, however, provides a window of time before executing the optimizer stage, during which incoming queries may queue up and form a batch. Furthermore, the optimizer can spend less time waiting for a sufficient number of incoming queries with common subexpressions, since newly arrived queries can still exploit common data from other queries already inside the execution engine.

3.7. Chapter summary

In this chapter we first motivated the need for a new database system architecture by analyzing the shortcomings of DBMS software. Modern database systems are built as monolithic pieces of software and force the use of *non-synergetic* execution primitives for handling concurrent requests: independent threads are used to multiplex request execution and maximize processor utilization. While database software architecture has fundamentally remained unchanged over the past two decades, hardware infrastructure has undergone significant changes. Good memory hierarchy utilization is essential in today's applications. No matter how well a single execution thread is optimized, there is always interference between non-synergetic threads. For instance, in Chapter 5 we show how eliminating instruction interference in the caches across concurrent transactions yields a 16-39% throughput improvement. With the advent of thread-parallel architectures (CMP and SMT), cross-thread interference will hurt performance even more.

The monolithic nature of DBMS design makes it difficult to find bottlenecks and in turn, optimize resource utilization. It makes it difficult to enforce scheduling policies and also makes it difficult for the hardware designer to isolate and benchmark individual components of the system. Furthermore, DBMS today ship as an “all or nothing” software. While database software complexity and code size have grown to meet application demands, the design philosophy has remained the same. The core system components are tightly glued together and new components are added on top of them. It is economically infeasible for a vendor to ship systems with custom components. As a result, DBMS is frequently considered an “overkill” for a number of applications that would otherwise benefit from the usage of database technology.

As a solution to the above-mentioned problems, this chapter introduced StagedDB, a new design for high-performance, evolvable database systems. The core idea of StagedDB is to encapsulate system components in self-contained stages and replace existing glue with queues and sub-requests. We detailed techniques to define and build individual stages out of a prototype DBMS along with query scheduling policies for staged architectures. To prove the feasibility of staged execution for a database server, we studied the performance tradeoff of delaying sub-requests to execute them in groups and found scheduling disciplines that can outperform traditional architectures. We then evaluated our initial implementation of a staged database system on top of PREDATOR, a research prototype DBMS, and described both performance and software engineering benefits of the case study.

Over the next two chapters, we further investigate how the staged execution engine can improve cross-query locality (Chapter 4) and how StagedDB can improve instruction cache performance for large, commercial-grade systems processing multiple concurrent transactions (Chapter 5).

Chapter 4

Implementing a staged query engine

Relational DBMS typically execute concurrent queries independently by invoking a set of operator instances for each query. To exploit common data retrievals and computation in concurrent queries, researchers have proposed a wealth of techniques, ranging from buffering disk pages to constructing materialized views and optimizing multiple queries. The ideas proposed, however, are inherently limited by the query-centric philosophy of modern engine designs. Ideally, the query engine should proactively coordinate same-operator execution among concurrent queries, thereby exploiting common accesses to memory and disks as well as common intermediate result computation.

This chapter describes *QPipe*, a staged, operator-centric relational engine that enhances locality across concurrent queries. Each relational operator is encapsulated in a micro-engine serving query tasks from a queue. QPipe implements *on-demand simultaneous pipelining* (OSP), a novel query evaluation paradigm for maximizing data and work sharing across concurrent queries at execution time. OSP enables proactive, dynamic operator sharing by pipelining the operator's output simultaneously to multiple parent nodes. Evaluation of QPipe built on top of BerkeleyDB shows that QPipe achieves a 2x speedup over a commercial DBMS when running a workload consisting of TPC-H queries.

4.1. Introduction

Modern decision-support systems (DSS) and scientific database applications operate on massive datasets and are characterized by complex queries accessing large portions of the database. Although high concurrency is predominantly studied in transactional workloads

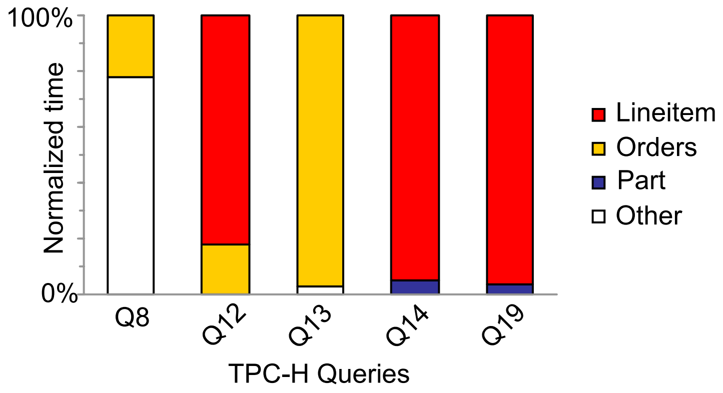


Figure 4-1. Time breakdown for five TPC-H queries. Each component shows time spent reading a TPC-H table.

due to intensive updates, decision-support systems often run queries concurrently (hence the throughput metric suggested in the specification of TPC-H, the prevailing DSS benchmark). In a typical data warehousing installation, new data is periodically bulk loaded into the database, followed by a period where multiple users issue read-only (or read-heavy) queries. Concurrent queries often exhibit high data and computation overlap, e.g., they access the same relations on disk, compute similar aggregates, or share intermediate results. Unfortunately, run-time sharing in modern execution engines is limited by the paradigm of invoking an independent set of operator instances per query, potentially missing sharing opportunities if the caches and buffer pool evict data pages early.

4.1.1 Sharing limitations in modern DBMS

Modern query execution engines are designed to execute queries following the "one-query, many-operators" model. A query enters the engine as an optimized plan and is executed as if it were alone in the system. The means for sharing common data across concurrent queries is provided by the buffer pool, which keeps information in main memory according to a replacement policy. The degree of sharing the buffer pool provides, however, is extremely sensitive to timing; in order to share data pages the queries must arrive simultaneously to the system and must execute in lockstep, which is highly unlikely. To illustrate the limitations of sharing through the buffer pool, we run TPC-H on X, a major

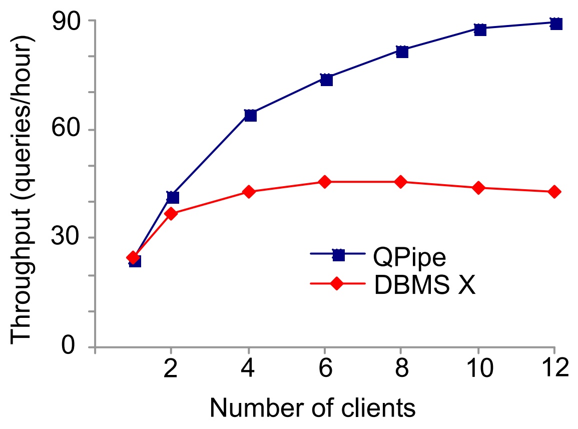


Figure 4-2. Throughput for one to twelve concurrent clients running TPC-H queries on DBMS X and QPipe.

commercial system[1] running on a 4-disk Pentium 4 server (experimental setup details are in Section 4.5). Although different TPC-H queries do not exhibit overlapping computation by design, all queries operate on the same eight tables, and therefore there often exist data page sharing opportunities. The overlap is visible in Figure 4-1 which shows a detailed time breakdown for five representative TPC-H queries with respect to the tables they read during execution[2].

Figure 4-2 shows the throughput achieved for one to twelve concurrent clients submitting requests from a pool of eight representative TPC-H queries, for DBMS X and QPipe, our proposed query engine. Both systems were configured with locking and logging disabled. QPipe achieves up to 2x speedup over X, with the throughput difference becoming more pronounced as more clients are added. The reason QPipe exhibits higher TPC-H throughput than X is that QPipe proactively shares the disk pages one query brings into memory with all other concurrent queries. Ideally, a query execution engine should be able to always detect such sharing opportunities across concurrent queries at run time, for all operators (not just for table scans) and be able to pipeline data from a single query node to multiple parent nodes at the same time. We call this ability *on-demand simultaneous pipelining* (*OSP*). The challenge is to design a query execution engine that supports OSP without incurring additional overhead.

1.Licensing restrictions prevent us from revealing the vendor.
2.Table specifications and TPC-H queries are available from the TPC website: `http://www.tpc.org`

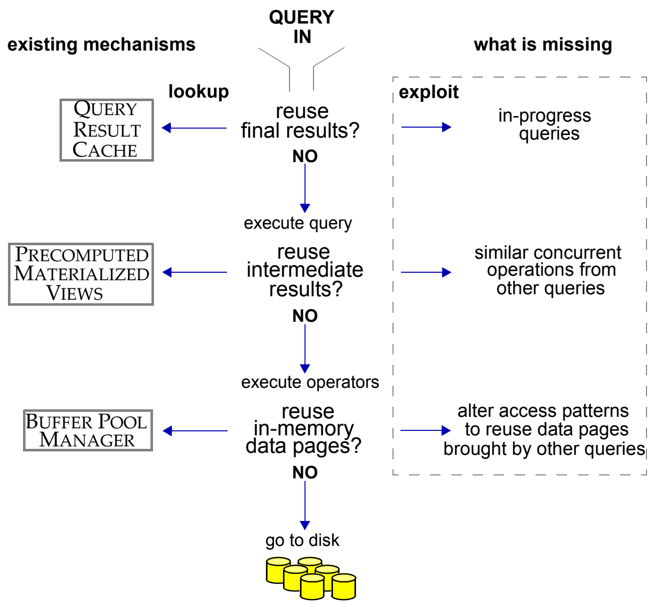

QUERY IN
existing mechanisms
what is missing
lookup
exploit
QUERY RESULT CACHE
reuse final results?
in-progress queries
NO
execute query
PRECOMPUTED MATERIALIZED VIEWS
reuse intermediate results?
similar concurrent operations from other queries
NO
execute operators
BUFFER POOL MANAGER
reuse in-memory data pages?
alter access patterns to reuse data pages brought by other queries
NO
go to disk

Figure 4-3. Existing and desired mechanisms for sharing data and work across queries.

4.1.2 State-of-the-art in data and work sharing

Modern database systems employ a multitude of techniques to share data and work across queries. The leftmost part of Figure 4-3 shows those mechanisms and the center column shows the order in which they are invoked, depending on the high-level phases of query execution. Once a query is submitted to the system, it first performs a lookup to a cache of recently completed queries. On a match, the query returns the stored results and avoids execution altogether. Once inside the execution engine, a query may reuse precomputed intermediate results, if the administrator has created any matching materialized views. To our knowledge, modern engines do not detect and exploit overlapping computation among concurrent queries. When an operator consumes tuples, it first performs a buffer pool lookup, and, on a miss, it fetches the tuples from disk. Buffer pool management techniques only control the eviction policy for data pages; they cannot instruct queries to dynamically alter their access patterns to maximize data sharing in main memory.

The rightmost part of Figure 4-3 shows that during each of the three basic mechanisms for data and work sharing there is a missed opportunity in not examining concurrent queries for potential overlap. Often, it is the case that a query computes the same intermediate result that another, current query also needs. Or, an in-progress scan may be of use to another query, either by reading the file in a different order or by making a minor change in the query plan. It would be unrealistic, however, to keep all intermediate results around indefinitely, just in case a future query needs it. Instead, what we need is a query engine design philosophy that exploits sharing opportunities naturally, without incurring additional management or performance overhead.

4.1.3 On-demand simultaneous pipelining

To maximize data and work sharing at execution time, we propose to monitor each relational operator for every active query in order to detect overlaps. For example, one query may have already sorted a file that another query is about to start sorting; by monitoring the sort operator we can detect this overlap and reuse the sorted file. Once an overlapping computation is detected, the results are *simultaneously pipelined* to all participating parent nodes, thereby avoiding materialization costs. There are several challenges in embedding such an evaluation model inside a traditional query engine: (a) how to efficiently detect overlapping operators and decide on sharing eligibility, (b) how to cope with different consuming/producing speeds of the participating queries, and, (c) how to overcome the optimizer's restrictions on the query evaluation order to allow for more sharing opportunities. The overhead to meet these challenges using a "one-query, many-operators" query engine would offset any performance benefits.

To support simultaneous pipelining, we introduce **QPipe**, a new query engine architecture, based on the principles of the StagedDB design. QPipe follows a "*one-operator, many-queries*" design philosophy. Each relational operator is promoted to an independent micro-engine which manages a set of threads and serves queries from a queue. Data flow between micro-engines occurs through dedicated buffers — similar to a parallel database engine [DeWitt et al. 1990]. By grouping similar tasks together, QPipe can naturally exploit any type of overlapping operation. We implement QPipe on top of the BerkeleyDB storage manager, using native OS threads. The resulting prototype is a versatile engine, naturally parallel, running on a wide range of multi-processor servers (tested on IA-64, IA-32, Linux and Windows). We demonstrate the effectiveness of our techniques through experimentation with microbenchmarks and the TPC-H benchmark. QPipe can efficiently

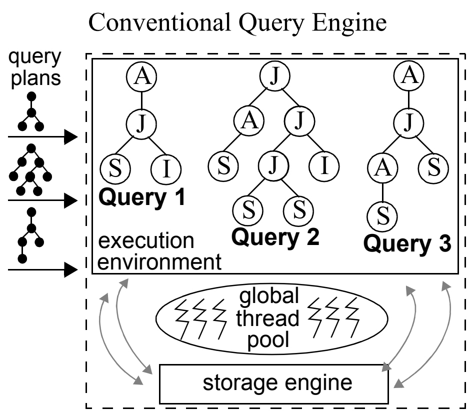


Figure 4-4. Conventional engines evaluate queries independently of each other. Disk requests are passed to the storage engine.

detect and exploit data and work sharing opportunities in any workload and can achieve up to 2x throughput speedup over traditional DBMS when running TPC-H queries. As Figure 4-2 shows, QPipe exploits all data sharing opportunities, while executing the same workload as the commercial DBMS.

In the remaining of this chapter, we describe the design and implementation of QPipe (Section 4.2), we present on-demand simultaneous pipelining (OSP) techniques for maximizing data and work sharing across queries (Section 4.3), we then describe how OSP is implemented in QPipe (Section 4.4), while in Section 4.5 we present our experimentation with the QPipe prototype. We address deadlocks in OSP in Section 4.6 and summarize the chapter in Section 4.7.

4.2. QPipe: design and implementation

In this section we first briefly describe the design philosophy behind conventional query engines (Section 4.2.1), we then introduce the QPipe engine design (Section 4.2.2) and provide details of the QPipe/BerkeleyDB prototype (Section 4.2.3).

4.2.1 Conventional engine design

Traditional relational query engine designs follow the "one-query, many-operators" model, and therefore are *query-centric*. Query plans generated by the optimizer drive the

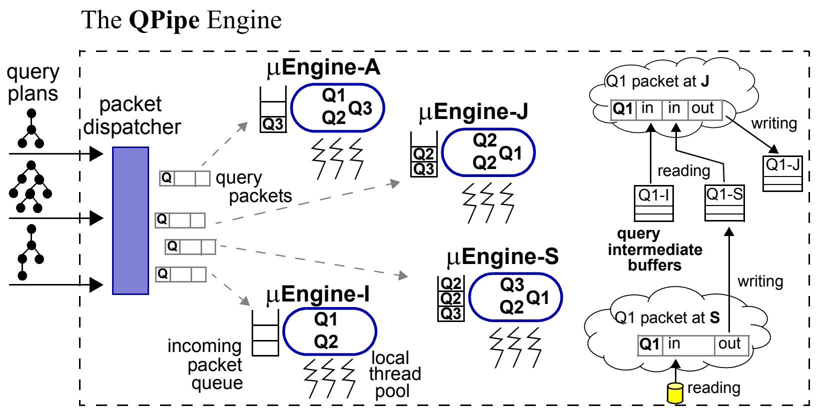


Figure 4-5. In QPipe every relational operator is a *micro*-engine. For simplicity, only four operators are shown (Scan, Index-scan, Join, Aggregation). Queries are broken into *packets* and queue up in the µEngines.

query evaluation process. A query plan is a tree with each node being a relational operator and each leaf an input point (either file scan or index scan) [Graefe 1996]. The execution engine evaluates queries independently of each other, by assigning one or more threads to each query. The high-level picture of the query engine consists of two components — the execution environment, where each query performs all of its intermediate computations, and the storage manager which handles all requests for disk pages (see also Figure 4-4). Queries dispatch requests to the disk subsystem (storage engine) and a notification mechanism informs the query when the data is placed in a pre-specified memory location. The storage engine optimizes resource management by deciding which pages will be cached or evicted. Since all actions are performed without having cumulative knowledge of the exact state of all current queries, conventional engines cannot fully exploit data and work sharing opportunities across queries.

4.2.2 The QPipe engine

QPipe implements a new, alternative execution model, based on the StagedDB design, that we call “one-operator, many-queries,” and therefore is an *operator-centric* architecture (Figure 4-5). In QPipe, each operator is promoted to an independent *micro*-engine (µEngine). µEngines accept requests (in the form of *packets*) and serve them from a queue. For example, the Sort µEngine only accepts requests for sorting a relation. The

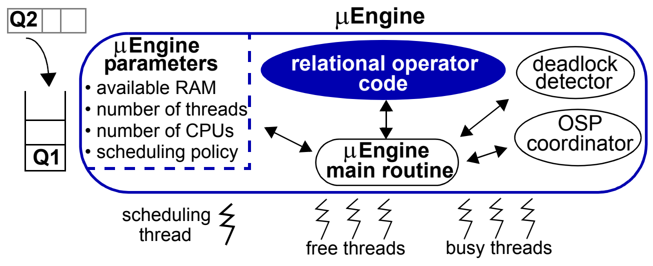


Figure 4-6. A μEngine in detail. The deadlock detector and OSP coordinator modules are explained in Section 4.4.

request itself must specify what needs to be sorted and which tuple buffer the result needs to be placed into. The way a query combines the independent work of all μEngines is by linking the output of one μEngine to the input of another, therefore establishing producer-consumer relationships between participating μEngines. In the current prototype, tuple buffers are implemented in shared-memory, however, this communication module can easily be replaced with a message passing mechanism, to deploy QPipe in distributed environments.

The input to QPipe is precompiled query plans (we use plans derived from a commercial system's optimizer). Query plans pass through the packet dispatcher which creates as many packets as the nodes in the query tree and dispatches them to the corresponding μEngines. Each μEngine has a queue of incoming requests. A worker thread that belongs to that μEngine removes the packet from the queue and processes it. Figure 4-6 shows the components of a μEngine. Packets mainly specify the input and output tuple buffers and the arguments for the relational operator (e.g., sorting attributes, predicates etc.). μEngines work in parallel to evaluate the query. The evaluation model resembles a *push-based* execution design [Graefe 1994], where each operator independently produces tuples until it fills the parent's input buffer. If the output is consumed by a slower operator, then the intermediate buffers regulate the data flow.

Since QPipe involves multiple local thread pools (one for each μEngine), efficient scheduling policies are important to ensure low query response times. We follow a two-level scheduling approach. At the higher level, the scheduler chooses which μEngine runs next and on which CPU(s). Within each μEngine, a local scheduler decides how the worker threads are scheduled. In our prototype we use a round-robin schedule for the μEngines, with a fixed number of CPUs per μEngine, and the default, preemptive processor-sharing (PS) that the OS provides for the worker threads. Since this simple policy guarantees that the system always makes progress, response times for all queries were held low.

QPipe improves performance (throughput and response time) when compared to tuple-by-tuple evaluation engines (iterator model) by saving extraneous procedure calls and by improving temporal locality. Recent work [Zhou and Ross 2004] introduces a buffer operator to increase the number of tuples processed at one time at each operator, thereby improving instruction temporal locality for long running queries. By processing a batch of tuples for each query at every μEngine, QPipe improves instruction temporal locality, as also shown in the previous chapter. QPipe also provides full intra-query parallelism, taking advantage of all available CPUs in a multi-processor server for evaluating a single query, regardless of the plan's complexity.

QPipe can achieve better resource utilization than conventional engines by grouping requests of the same nature together, and by having dedicated μEngines to process each group of similar requests. In the same way a disk drive performs better when it is presented with a large group of requests (because of better disk head scheduling), each μEngine can better optimize resource usage by processing a group of similar requests. Over the next sections of the chapter we focus on QPipe's ability to reuse data pages and similar computation between different queries at the same μEngine.

4.2.3 The QPipe/BerkeleyDB prototype

The QPipe prototype is a multi-threaded, parallel application that runs on shared-memory multiprocessor systems. Each μEngine is a different C++ class with separate classes for the thread-pool support, the shared-memory implementation of queues and buffers (including query packets), and the packet dispatcher. Calls to data access methods are wrappers for the underlying storage manager. The bare system is a runtime consisting of a number of idle threads, as many as the specified μEngines times the number of threads per μEngine. The OS schedules the threads on any of the available CPUs. Client processes can either submit packets directly to the μEngines or send a query plan to the packet dispatcher which creates and routes the packets accordingly. The basic functionality of each μEngine is to dequeue the packet, process it, optionally read input or write output to a buffer, and destroy the packet. The client process reads the final results from a shared-memory buffer.

We implement relational-engine functionality by inserting relational processing code to each μEngine, and providing the packet dispatcher code to transform precompiled query plans into packets. The database storage manager adds the necessary transactional support, a buffer-pool manager, and table access methods. In the current prototype we use the BerkeleyDB database storage manager and have implemented the following relational operators: table scan (indexed and non-indexed), nested-loop join, sort, merge-join, hybrid hash join, aggregate (both simple and hash-based). The implementation is about 7,000 lines of C++ code (BerkeleyDB itself is around 210,000 lines).

In addition to BerkeleyDB, we have successfully applied the QPipe runtime to two other open source DBMS, MySQL[1] and Predator [Seshadri et al. 1997]. Since there is typically a clear division between the storage manager and the rest of the DBMS, it was straightforward to transfer all function calls to the storage manager inside the μEngines. Taking the optimizer's output and redirecting it to the packet dispatcher was also straightforward. The time consuming part of the conversion is to isolate the code for each relational operator. Fortunately, each relational operator uses a limited set of global variables which makes it easy to turn the operator into a self-contained module with parameters being passed as an encoded structure.

4.3. Simultaneous pipelining

Despite a plethora of mechanisms to share data and work across queries (as those were analyzed in Chapter 2), the prevailing relational query execution paradigm is characterized by two key properties that preclude full exploitation of sharing opportunities. First, it deprives individual queries from knowing about the state of other, concurrent queries. In doing so, it prevents the system from taking action at run time, once an overlapping operation across different queries appears. Second, traditional query engines adhere to a static

1.http://www.mysql.com/

evaluation plan and to a page-level interface to the storage manager. Despite the fact that disk page access patterns are known in advance, sharing opportunities are limited since the system cannot adjust the query evaluation strategy at run time.

If two or more concurrent queries contain the same relational operator in their plans, and that operator outputs the same tuples on behalf of all queries (or a query can use these tuples with a simple projection), then we can potentially "share" the operator. The operator will execute once, and its output will be pipelined to all consuming nodes simultaneously. We refer to the ability of a single relational operator to pipeline its output to multiple queries concurrently as simultaneous pipelining. *On-demand simultaneous pipelining* (OSP) is the ability to dynamically exploit overlapping operations at run time. OSP is desirable when there exist opportunities for reusing data pages that the buffer pool manager has evicted early, or intermediate computations across queries that are not covered by pre-computed materialized views.

In this section we first characterize what a "missed opportunity" for data and work sharing is (Section 4.3.1). Then, we classify all relational operators with respect to their effective "window of opportunity," i.e., what percentage of the operation's lifetime is offered for reuse (Section 4.3.2). Lastly, we describe the challenges in exploiting overlap between relational operators (Section 4.3.3).

4.3.1 Data and work sharing misses

Whenever two or more concurrent queries read from the same table, or compute the same (or subset of the same) intermediate result, there is potentially an opportunity to exploit overlapping work and reduce I/O traffic, RAM usage, and processing time. A *sharing miss* in a workload is defined in terms of memory page faults and computation as follows:

A query Q begins execution at time T_s and completes at time T_c.

Definition 1*. At time T_r, Q requests page P, which was previously referenced at time T_p. If the request results in a page fault, and $T_s < T_p$, the page fault is a* ***data sharing miss****.*

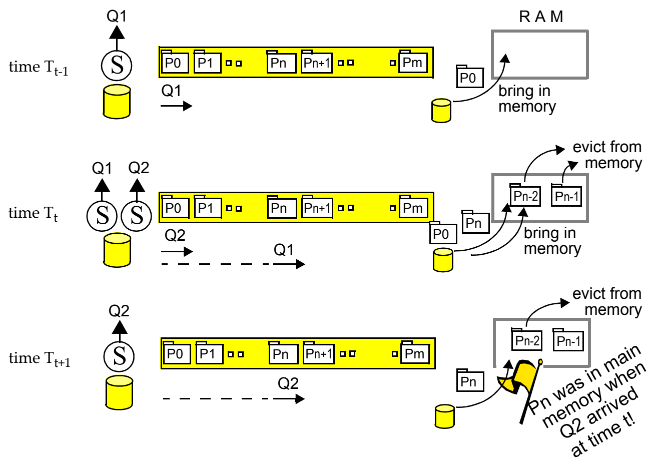


Figure 4-7. Two queries independently start a file scan on the same table. Query 2 is missing the opportunity to reuse all pages, after Pn, that Query 1 brings momentarily in RAM.

***Definition 2**. At time T_w, Q initiates new computation by running operator W. If W was also executed between T_s and T_w, then there is a **work sharing miss**.*

Sharing misses can be minimized by proactively sharing the overlapping operator across multiple queries. To clarify this procedure, consider the scenario illustrated in Figure 4-7 in which two queries use the same scan operators. For simplicity, we assume that the main memory holds only two disk pages while the file is $M \gg 2$ pages long. Query 1 starts a file scan at time T_{t-1}. As the scan progresses, pages are evicted to make room for the incoming data. At time T_t, Query 2 arrives and starts a scan on the same table. At this point, pages P_{n-2} and P_{n-1} are in main memory. These will be replaced by the new pages read by the two scans: P_n for Q1 and P_0 for Q2. At time T_{t+1}, Q1 has finished and Q2 is about to read page P_n. The main memory now contains P_{n-2} and P_{n-1} that Q2 just read. Page P_n, however, was in main memory when Q2 arrived in the system. This page (and all subsequent ones) represent *data sharing misses* by Q2.

With simultaneous pipelining in place, Query 2 can potentially avoid all data sharing misses in this scenario. Assuming that Q2 is not interested in which order disk pages are read — as long as the entire table is read — then, at time T_t, Q2 can “piggyback” on Q1’s scan operator. The scan operator will then pipeline all pages read simultaneously to both

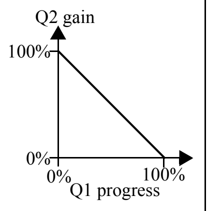
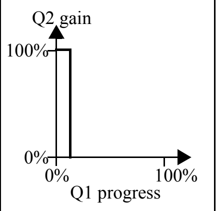
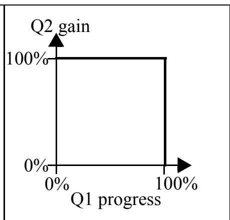
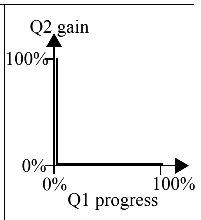
linear	**step**	**full**	**spike**
			
• *table scan (either as an operator or part of reading sorted files, hashed partitions etc.)*	• *index scan*	• *hash join (partitioning)*	• *ordered table scan*
	• *hash join (probe)*	• *sort*	
	• *group-by*	• *single aggregate*	
	• *nested-loop join*	• *non-clustered index scan (RID list creation)*	
	• *merge join*		

Figure 4-8. Windows of Opportunity for the four basic operator overlap types.

queries, and, on reaching the end of file, a new scan operator, just for Q2, will read the skipped pages. What happens, however, if Q2 expects all disk pages to be read in the order stored in file? To help understand the challenges involved in trying to minimize sharing misses, the next subsection classifies relational operators with respect to their sharing opportunities.

4.3.2 Window of Opportunity (WoP)

Given that query Q1 executes a relational operator and query Q2 arrives with a similar operator in its plan, we need to know whether we can apply simultaneous pipelining or not, and what are the expected cost savings for Q2 (i.e., how many sharing misses will be eliminated). We call the time from the invocation of an operator up until a newly submitted identical operator can take advantage of the one in progress, *window of opportunity* or *WoP.* Once the new operator starts taking advantage of the in-progress operator, the cost savings apply to the entire cumulative cost of all the children operators in the query's plan.

Figure 4-8 shows a classification of all basic operations in a relational engine with respect to the WoP and the associated cost savings for a simultaneously pipelined second query. We identify four different types of overlap between the various relational operations (shown on the top of the figure). ***Linear*** overlap characterizes operations that can always take advantage of the uncompleted part of an in-progress identical operation, with cost savings varying from 100% to 0%, depending how late in the process Q2 joins Q1. For example, *unordered* table scans (which do not care about the order in which the tuples

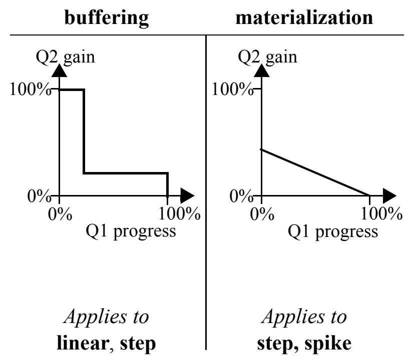


Figure 4-9. WoP enhancement functions.

are received) fall in this category. ***Step*** overlap applies to concurrent operations that can exploit each other completely (100% cost savings), as long as the first output tuple has not been produced yet. For example, in the probing phase of hash-join, it may take some time before the first match is found; during that time, Q2 can join Q1. ***Full*** overlap is the ideal case: 100% cost savings for the entire lifetime of the in-progress operation (for example, computing a single aggregate). The last category, ***spike*** overlap, is all operations that cannot be overlapped, unless they start at the exact same time; for example, a table scan that must output tuples in table order can only piggyback on any other scan if the first output page is still in memory. A *spike* overlap is the same as a *step* overlap when the latter produces its first output tuple instantaneously.

Figure 4-9 shows two "enhancement" functions that can apply to the aforementioned categories in order to increase both the WoP and the cost savings. The ***buffering*** function refers to the ability of an operator to buffer a number of output tuples. Since output is not discarded immediately after it is consumed, an incoming request has a wider window of opportunity for exploiting the precomputed output tuples. For example, an ordered table scan that buffers N tuples can be converted from *spike* to *step*. The ***materialization*** function stores the results of an operator to be used later on. For example, consider an ordered table scan. If a new, highly selective (few qualifying tuples) query needs to scan the same table in stored tuple order, then we can potentially exploit the scan in progress by storing the qualifying tuples for the new query. This way we trade reading part of the table with storing and then reading a potentially significantly smaller number of tuples. This function can convert *spike* to *linear*, albeit with a smaller effective slope for the cost savings.

Next, we break down each operator to its basic overlap types.

File scans. File scans have only one phase. If there is no restriction on the order the tuples are produced, or the parent operator needs the tuples in order but can output them in any order, then file scans have a *linear* overlap. If tuple ordering is strict then file scans have a *spike* overlap.

Index scans. Clustered index scans are similar to file scans and therefore exhibit either *linear* or *spike* overlap depending on the tuple ordering requirements. Unclustered index scans are implemented in two phases. The first phase probes the index for all matches and constructs a list with all the matching record IDs (RID). The list is then sorted on ascending page number to avoid multiple visits on the same page. This phase corresponds to a *full* overlap as a newly arrived operator can exploit work in progress at any point of time. The second phase is similar to file scan and so is either *linear* or *spike* overlap.

Sort. Sorting consists of multiple phases, though, in our context, we treat it as a two-phase operator. In the first phase the input is sorted on the sorting attribute (either in memory or disk, depending on the size of the file). During this phase any new arrival can share the ongoing operation, and therefore it is a *full* overlap. The second phase is pipelining the sorted tuples to the parent operator and it is similar to a file scan (either *linear* or *spike*).

Aggregates. All aggregate operators producing a single result (min, max, count, avg) exhibit a *full* overlap. Group-by belongs to *step* overlap, since it produces multiple results. *Buffering* can potentially provide a significant increase in the WoP, especially if the provided buffer size is comparable to the output size.

Joins. The most widely used join operators are hash-join, sort-merge join, and nested-loop join. Nested-loop join has a *step* overlap (it can be shared while the first match is not found yet). The sorting phase of sort-merge join is typically a separate sort operator. The merging phase is similar to nested-loop join (*step*). Hash-join first hashes and partitions the input relations. This phase is a *full* overlap. The joining phase is again *step* overlap. Both *buffering* and *materialization* can further increase the WoP.

Updates. By their nature, update statements cannot be shared since that would violate the transactional semantics.

4.3.3 Challenges in simultaneous pipelining

A prerequisite to simultaneous pipelining is the decoupling of operator invocation and query scope. Such a decoupling is necessary to allow an operator to copy its output tuples to multiple queries-consumers. In commercial DBMS this decoupling is visible only at the storage layer (as shown in Figure 4-4). Whenever a query needs tuples from the disk it waits for them to be placed at a specified buffer. From the query's point of view, it does not make a difference whether there is a single or multiple I/O processes delivering the same tuples to multiple queries. A similar decoupling should apply to all relational operators to implement simultaneous pipelining techniques. Following, we outline the remaining challenges.

Run-time detection of overlapping operations. To make the most out of simultaneous pipelining, the query engine must track the progress of all operators for all queries at all times. Whenever a query is submitted, the operators in its plan must be compared with all the operators from all active queries. The output of each comparison should specify whether there is an overlapping computation in-progress and whether the window of opportunity (WoP) has expired. This run-time detection should be as efficient as possible and scale well with the number of active queries.

Multiple-scan consumers. When new scan requests for the same table arrive repeatedly and dynamically share a single scan, a large number of partial scans will then be active on the same relation. Ideally, these partial scans should again synchronize the retrieval of common tuples, which requires additional bookkeeping. File scans with different selectivities and different parent consumption rates can make the synchronization difficult. If one file scan blocks trying to provide more tuples than its parent node can consume, it will need to detach from the rest of the scans. This might create a large number of partial scans covering different overlapping and disjoint regions of the relations, further complicating synchronization efforts.

Order-sensitive operators. Query optimizers often create plans that exploit "interesting" table orders by assuming that the scanned tuples will be read in table order. For example, if a table is already sorted on a join attribute, the optimizer is likely to suggest a merge-join and avoid sorting the relation. Such scans have a *spike* WoP and therefore cannot take advantage of an ongoing scan. In case the ordered scan is highly selective (few qualifying tuples), a materialization function could help by storing the qualifying tuples, and reusing them later, in order. The challenge, however, is to exploit the scan in progress even if the new, order-sensitive scan does not perform any filtering.

Deadlocks in pipelining. The simultaneous evaluation of multiple query plans may lead to deadlocks. Consider for example two queries that share the results of two different scans (table A and B). If one query needs to advance scan A to be able to process the last value read from B, while the other query has the opposite need, advancing B to process A's last read value, then the two queries become deadlocked. The existence of a buffer can only delay the appearance of a deadlock in this case. The challenge is to efficiently detect potential deadlock situations and avoid them while still making the most out of overlapping computations.

In the next section we describe how QPipe implements OSP techniques and addresses the above-mentioned challenges. The treatment of deadlocks in simultaneous pipelining is separately covered in Section 4.6.

4.4. Support for simultaneous pipelining in QPipe

In QPipe, a query packet represents work a query needs to perform at a given μEngine. Every time a new packet queues up in a μEngine, we scan the queue with the existing packets to check for overlapping work. This is a quick check of the encoded argument list for each packet (that was produced when the query passed through the packet dispatcher). The outcome of the comparison is whether there is a match and which phase of the current operation can be reused (i.e., a sorted file, and/or reading the sorted file). Each μEngine employs a different sharing mechanism, depending on the encapsulated relational operation (sharing opportunities were described in the previous section).

There are two elements that are common to all μEngines: the **OSP Coordinator** and the **Deadlock Detector** (Figure 4-6, page 60). The OSP Coordinator lays the ground for the new packet (the "satellite" packet) to attach to the in-progress query's packet (the "host" packet), and have the operator's output simultaneously pipelined to all participating queries. The OSP Coordinator handles the additional requirements and necessary adjustments

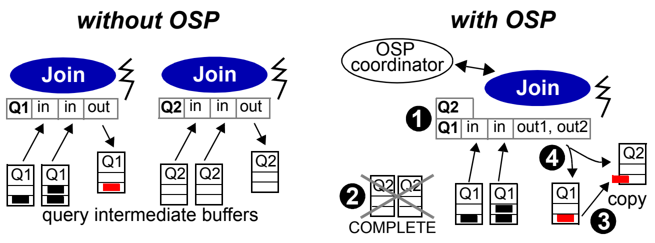


Figure 4-10. Simultaneous pipelining on two join operations.

to the evaluation strategy of the satellite's packet original query. For example, it may create an additional packet to complete the non-overlapping part of an operation (this scenario is described in Section 4.4.2). The Deadlock Detector ensures a deadlock-free execution of simultaneously pipelined schedules. The pipelining deadlock problem is treated in detail in Section 4.6.

Figure 4-10 illustrates the actions the OSP coordinator takes when two queries have an overlapping operation. In this scenario, we assume Query 1 has already initiated a join of *step* overlap (e.g., merge-join), and a few tuples have already been produced, but are still stored in Q1's output buffer. Without OSP (left part of Figure 4-10), when Q2 arrives, it will repeat the same join operation as Q1, receiving input and placing output to buffers dedicated to Q2. When the OSP Coordinator is active, it performs the following actions:

1. It attaches Q2's packet (satellite) to Q1 (host).

2. It notifies Q2's children operators to terminate (recursively, for the entire subtree underneath the join node).

3. It copies the output tuples of the join that are still in Q1's buffer, to Q2's output buffer.

4. While Q1 proceeds with the join operation, the output is copied simultaneously to both Q1's and the satellite's output.

The above steps are illustrated in Figure 4-10 (right part).

Once the OSP Coordinator attaches one or more satellite packets to a host packet, a "1-producer, N-consumers" relationship is formed between the participating queries. QPipe's intermediate buffers regulate the dataflow. If any of the consumers is slower than the producer, all queries will eventually adjust their consuming speed to the speed of the slowest consumer. Next, we describe (a) how QPipe deals with the burden of frequently arriving/departing satellite scans, (b) the actions the OSP Coordinator takes to exploit order-sensitive overlapping scans, and how QPipe handles lock requests and update statements.

4.4.1 Synchronizing multiple scan consumers

Scan sharing of base relations is a frequently anticipated operation in QPipe. A large number of different scan requests with different requirements can easily put pressure on any storage manager and make the bookkeeping in a design that shares disk pages difficult. Sharing of multiple scans to the same table was first described in the RedBrick Data Warehouse implementation [Fernandez 1994], and several other commercial systems such as Teradata and SQL Server mention a similar functionality in their implementation. Details of the mechanisms employed, such as what kind of bookkeeping the storage manager performs and how the technique scales to multiple concurrent scans with different arrival times, are not publicly disclosed. Moreover, the existing literature describes only scenarios where queries do not depend on the table scan order.

To simplify the management of multiple overlapping scans in QPipe, we maintain a dedicated scan thread that is responsible for scanning a particular relation. Once a new request for scanning a relation arrives, a *scanner thread* is initiated and reads the file (Figure 4-11). The scanner thread essentially plays the role of the host packet and the newly arrived packet becomes a satellite (time T_{t-1} in Figure 4-11). Since the satellite packet is the only one scanning the file, it also sets the termination point for the scanner thread at the end of the file. When later on, (time T_t), a new packet for scanning the same relation arrives, the packet immediately becomes a satellite one and sets the new termination point for the scanner thread at the current position of the file. When the scanner thread reaches the end-of-file for the first time, it will keep scanning the relation from the beginning, to serve the unread pages to Query 2.

This circular scan implementation simplifies the bookkeeping needed to track which queries are attached at any time. Moreover, it is the job of the OSP Coordinator to allow a packet to attach to the scanner thread or not, depending on the query requirements. For example, the query may need to start consuming pages only after another operator in the

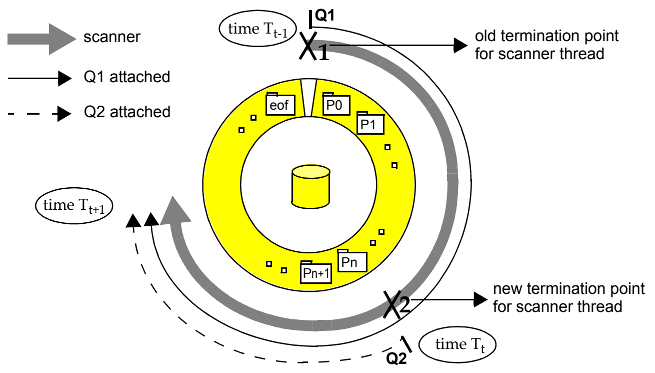


Figure 4-11. Circular scan operation. Q1 initiates the scanner thread (T_{t-1}). Q2 attaches immediately when it arrives (T_t) and sets the new termination point for the circular scan at page P_n.

plan has started producing output. In our implementation, the OSP coordinator applies a late activation policy, where no scan packet is initiated until its output buffer is flagged as ready to receive tuples. Late activation prevents queries from delaying each other.

4.4.2 Order-sensitive scans

Consider a join operation where the base relations are already sorted on a joining key. In this case, the query plans may use a merge operator directly on the sorted files. If a scan is already in progress and a second query arrives, it encounters a spike overlap, and thus, it will not be able to attach. There are two cases, however, that the scan in progress can still be exploited.

First, if the parent operator of the merge-join does not depend on the order in which its input tuples are received, then the OSP Coordinator creates two merge-join packets for the same query. The first packet joins the remaining portion of the shared relation with the non-shared relation, providing output tuples to the order-insensitive parent. Afterwards, the second packet processes the unread part of the shared relation and joins it again with the non-shared relation. To avoid increasing the total cost, the OSP Coordinator always assumes the worst case scenario of reading the non-shared relation twice in order to merge the two disjoint parts of the shared relation. If the total cost does not justify sharing the operation, the OSP Coordinator does not attach the packets.

Second, if the selectivity of the scan is high (few qualifying tuples) or the selectivity of the merge operation is high, then the OSP Coordinator may choose to use the *materializa-*

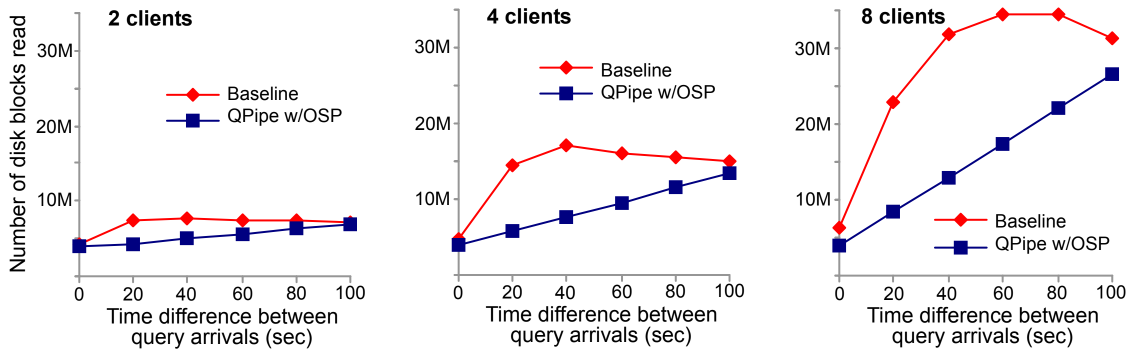
tion function to save out-of-order results that are cheap to produce. Once the scan reaches the beginning of the relation, the query resumes regular execution, passing the result tuples to the parent of the merge. Once the scan reaches the page it first attached to, the saved results are used to compute the rest of the merge.

4.4.3 Locks and updates

Data and work sharing techniques are best exploited in read-mostly environments, such as concurrent long-running queries in data warehouses, where there is high probability of performing overlapping work. Workloads with frequent concurrent updates to the database limit the percentage of time that scans can be performed (due to locking), and therefore restrict the overall impact of data sharing techniques. QPipe runs any type of workload, as it charges the underlying storage manager (BerkeleyDB in the current implementation) with lock and update management by routing update requests to a dedicated μEngine with no OSP functionality. As long as a sharing opportunity appears, even in the presence of concurrent updates, QPipe will take advantage of it. If a table is locked for writing, the scan packet will simply wait (and with it, all satellite ones), until the lock is released.

4.5. QPipe evaluation

This section presents our experimentation with the QPipe prototype. We experiment using two datasets. The first dataset is based on the Wisconsin Benchmark [DeWitt 1993] which specifies a simple schema with two large tables and a smaller one. We use 8 million 200-byte tuple tables for the big tables (BIG1 and BIG2 in the experiments) and 800,000 200-byte tuples for the small table (SMALL). The total size of the tables on disk is 4.5GB. The second dataset is a 4GB TPC-H database generated by the standard *dbgen* utility. The total size of the dataset on disk (including indices and storage engine overhead) is 5GB. All experiments are run on a 2.6 GHz P4 machine, with 2GB of RAM and four 10K RPM SCSI drives (organized as software RAID-0 array), running Linux 2.4.18. We discard all result tuples to avoid introducing additional client-server communication overhead. In all of the graphs, "Baseline" is the BerkeleyDB-based QPipe implementation with OSP dis-



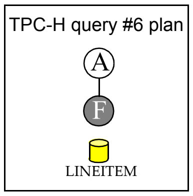


Figure 4-12. Total number of disk blocks read for three different configurations (2, 4, and 8 concurrent users sending TPC-H Query 6) with varying user interarrival times (0-100 sec). The number of blocks read remains flat for longer than 120 sec interarrival times.

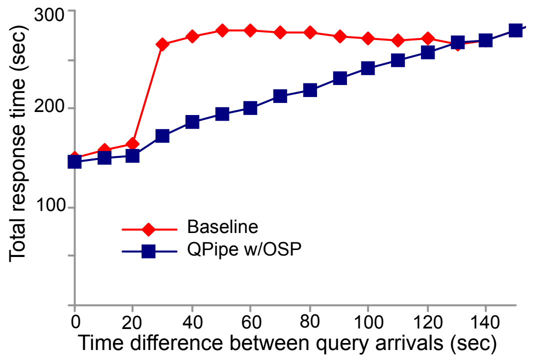
abled, “QPipe w/OSP” is the same system with OSP enabled, and “DBMS X” is a major commercial database system. When running QPipe with queries that present no sharing opportunities, we found that the overhead of the OSP coordinator is negligible (less than 1% of the query execution time).

4.5.1 Sharing data pages

4.5.1.1 Exploiting overlapping unordered scans

In this experiment we examine how well QPipe with OSP performs when exploiting the linear overlap of table scans. We evaluate three different workloads with 2, 4, and 8 concurrent clients running TPC-H Query 6. The 99% of execution time is spent performing an unordered table scan of the LINEITEM relation. We evaluate the performance of circular scans in QPipe as a function of different query interarrival times. We vary the interarrival time for a set of queries from 0 sec (highest overlap) to 100 sec (relatively little overlap). The goal of the experiment is to investigate the amount of redundant I/O that we can save by employing OSP.

The results are shown in Figure 4-12. The vertical axis is the total number of disk blocks read during the workload execution time. For workloads where queries arrive simultaneously, traditional disk page sharing through the buffer pool manager performs well.



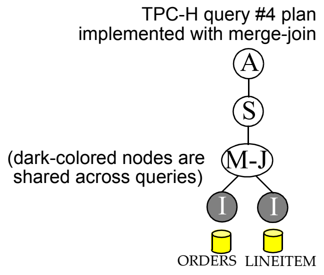

TPC-H query #4 plan implemented with merge-join
A
S
M-J
I
I
(dark-colored nodes are shared across queries)
ORDERS
LINEITEM

Figure 4-13. Sharing order-sensitive clustered index scans (I) on ORDERS and LINEITEM between two queries starting at different time intervals. Merge-join (M-J) expects tuples in key order. Since sort (S) does not assume a specific ordering, QPipe w/OSP performs 2 separate joins to share the in-progress scan.

However, as the query interarrival time grows to 20 sec, the data brought in by the running query are completely evicted from the buffer pool by the time the next query arrives. On workloads with a high degree of overlap (20 sec interarrival time) QPipe with OSP can save up to 63% of the total I/O cost. As the interarrival time grows and the overlap between queries shrinks the two curves approach each other (and remain flat at the same point for 120 sec or more when there is no overlap). Note that the curve for the Baseline system, for 4 or 8 clients, does not grow monotonically; the reason is that multiple concurrent scans in our implementation utilize large scan buffers, causing contention in the buffer pool and therefore the eviction of useful meta-data pages that need to be re-read. QPipe w/OSP always exploits all data sharing opportunities, making optimal use of the existing buffer pool space, whereas the baseline system depends on the timing of different arrivals to share data.

4.5.1.2 Exploiting overlapping clustered index-scans

In this experiment we evaluate our technique for exploiting overlaps between ordered scans, essentially converting a spike overlap to linear. We submit to QPipe two instances of TPC-H Query #4 which includes a merge-join at different time intervals. The full plan of Query #4 is shown in Figure 4-13 (right part). Even though the merge join relies on the input tuples being ordered, there is no need for the output of the join to be properly

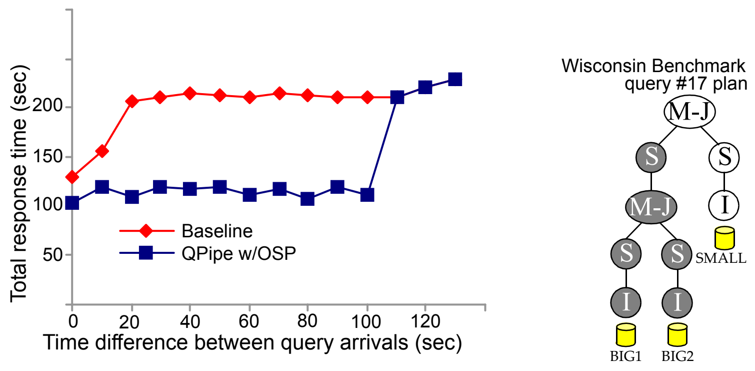


Figure 4-14. Sharing multiple operators, with sort (S) at the highest level. The two queries have the same predicates for scanning BIG1 and BIG2, but different ones for SMALL.

ordered. QPipe with OSP takes advantage of this property of the query plan, and allows ordered scans to attach to the existing one even though it is already in progress. Once the merge join consumes the input produced by the overlapping scans it initiates a new partial scan to retrieve the records it missed due to the late arrival. Figure 4-13 shows that QPipe with OSP significantly outperforms the baseline system with OSP disabled.

4.5.2 Reusing computation in aggregates / joins

4.5.2.1 Sort-merge join

The sort-merge join operator consists of a sort which is a full + linear overlap, followed by a merge which is a step overlap. In the next experiment, we use two similar 3-way join queries from the Wisconsin Benchmark. The graph in Figure 4-14 shows the total elapsed time from the moment the first query arrives until the system is idle again. We vary the interarrival time for the two queries from 0 sec up to the when there is no overlap between the queries. The graph shows that QPipe with OSP can exploit commonality for most of the query's lifetime (that's why the line for QPipe w/OSP remains flat most of the time) resulting in a 2x speedup. In this case, QPipe w/OSP is able to merge the packets from the two different queries during the merge phase of the sort-merge join. The baseline system performs better when the queries arrive close to each other (point zero on the horizontal axis), as it can share data pages in the buffer pool.

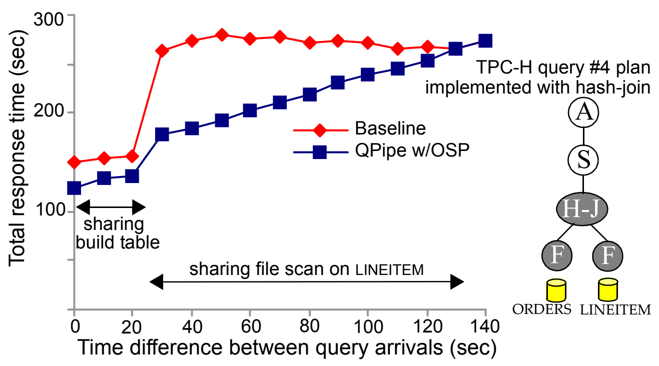


Figure 4-15. Changing the plan of TPC-H query #4 to use hash-join allows for a window of opportunity on sharing the build phase of the hash-join (first 20 secs). If the second query arrives later than that, it still can share the scan on LINEITEM.

4.5.2.2 Hash join

In this experiment we evaluate a full + step overlap operator, hash join. We submit to QPipe two instances of TPC-H query #4 which uses a hash join between the LINEITEM and ORDERS relations varying interarrival time. We expect that QPipe with OSP will be able to reuse the building phase of hash join. The graph axes are the same as in the previous figures. Figure 4-15 shows that QPipe with OSP can reuse the entire results of the build phase of the hash-join (20 seconds mark). After the hash join starts producing the first output and it is no longer possible to reuse the results of the build phase, QPipe still is able to significantly reduce the I/O costs by sharing the results of the scan in progress on LINEITEM.

4.5.3 Running full workloads

In the next experiment we compare the performance of QPipe with OSP against the baseline system and the commercial DBMS X, using a set of clients executing a random mix of queries from the TPC-H benchmark. The query mix is based on TPC-H queries #1, #4, #6, #8, #12, #13, #14, and #19. To make sure that multiple clients do not run identical queries at the same time, the selection predicates for base table scans were generated randomly using the standard *qgen* utility. Even though all the queries had different selection predi-

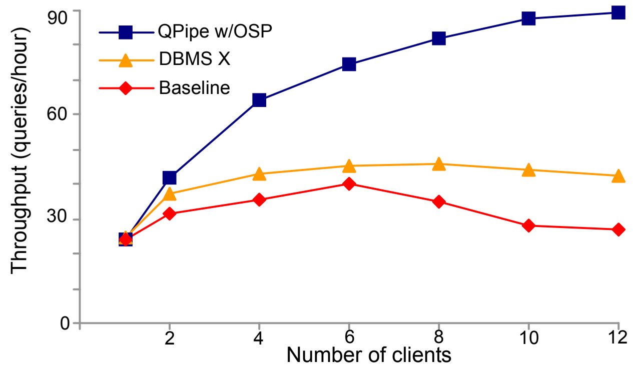


Figure 4-16. TPC-H throughput for the three systems increasing the number of concurrent users from 1 to 12, and keeping think time to zero.

cates for table scans, QPipe's circular scans are able to take advantage of the common accesses to LINEITEM, ORDERS and PART. We use hybrid hash joins exclusively for all the join parts of the query plans. Since hash joins do not rely on the ordering properties of the input streams, we are able to use unordered scans for all the access paths, which have large windows of opportunity. We vary the number of clients from 1 to 12 and measure the overall system throughput. Each client is given 128MB of memory to use for the sort heap and the hash tables. When running a single client we observe that the workload is disk-bound.

Figure 4-16 shows that QPipe w/OSP outperforms both the baseline system and X. For a single client, the throughput of QPipe and X is almost identical since the disk bandwidth is the limiting factor. As the number of clients increases beyond 6, DBMS X is not able to significantly increase the throughput. On other hand, QPipe with OSP takes full advantage of overlapping work and achieves a 2x speedup over DBMS X. The difference in the throughput between the baseline system and DBMS X shows that X's buffer pool manager achieves better sharing than the one BerkeleyDB employs. In Figure 4-17 we show the average response time for the same mix of TPC-H queries, for QPipe w/OSP and the baseline system, using 10 concurrent users and changing the think time of each user. As this experiment shows, QPipe w/OSP achieves high throughput without sacrificing query response times.

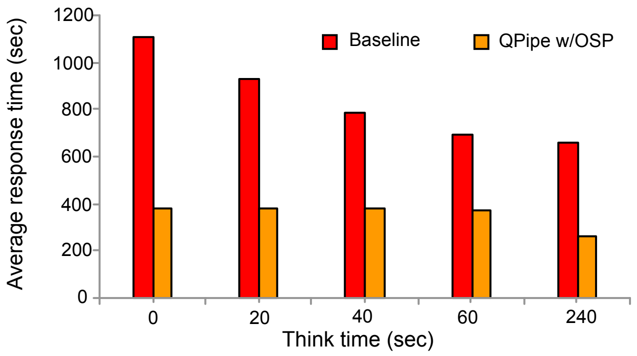


Figure 4-17. Average response time for QPipe w/OSP and the baseline system for a mix of TPC-H queries, varying the think time, for 10 concurrent users (low think times correspond to high system load).

4.6. Deadlocks in simultaneous pipelining

In this last section of the chapter, we study the deadlock problem in simultaneous pipelines: whenever the execution engine simultaneously pipelines tuples produced by a query node to multiple consumers, it introduces the possibility of deadlock. Since nodes can only produce tuples as fast as the slowest consumer allows them to, loops in the combined query plans can lead to deadlocks. One such scenario was described in Section 4.3.3. This problem is not specific to QPipe; it has been also identified and studied in the context of multi-query optimization [Dalvi et al. 2001], where materialization of intermediate results is used as a deadlock prevention mechanism. It also appears in the context of parallel sorting [Graefe 1993], where multiple producers sort partitions of a relation and pipeline their results to multiple consumers who perform a merge.

In all of the above-mentioned contexts, pipelining is applied to query evaluation strategies that are based on *query plan graphs* (more specifically, *Directed Acyclic Graphs*—DAGs) instead of trees. The difference is that a single node can have multiple parents, which can either belong to the same or different queries. While pipelined query graphs can speed up execution when compared to pipelined tree plans, by eliminating redundant computation and data accesses, they may also lead to run-time execution deadlocks. This section introduces a dynamic mechanism for detecting and resolving deadlocks in pipelined query graphs. Instead of statically determining what nodes to materialize (as proposed in

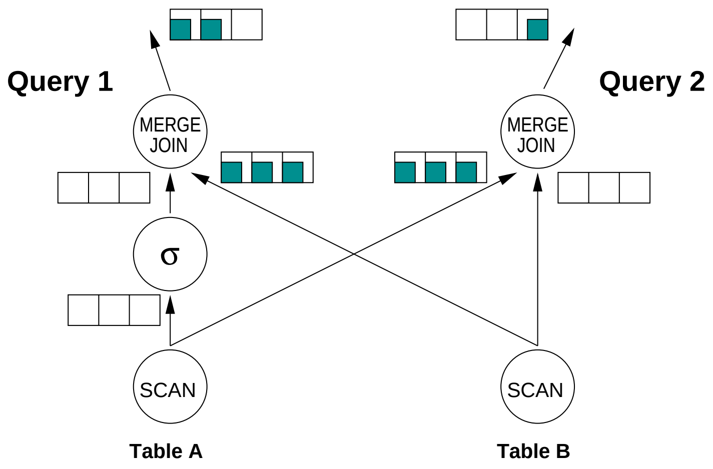


Figure 4-18. Example of a deadlocked query graph.

[Dalvi et al. 2001]), we pipeline every pipelinable operator in the query plan, resolving deadlock when it arises by materialization.

Over the next paragraphs of this section we describe the problem in more detail and discuss related work (4.6.1), we then propose mechanisms to detect and resolve deadlocks dynamically (4.6.2), and, lastly, we evaluate the proposed techniques (4.6.3).

4.6.1 Problem description and related work

To illustrate the problem consider the plan-DAG shown in Figure 4-18. Queries 1 and 2 each perform a merge-join on tables A and B, with Query 1 applying a selection predicate on table A. Since both queries scan the same tables, the results from each scan are pipelined to both queries. Each operator in a query plan has a finite buffer in which incoming tuples are temporarily stored, before they are processed; the produced tuples are sent to the buffer of the parent operator. In Figure 4-18, next to each edge, we show a snapshot of the state of those buffers. In this specific scenario, where two of the buffers are full, the evaluation of the plan-DAG cannot proceed. In order for the merge-join of Query 2 to proceed (by consuming a tuple from its left full buffer), it needs to receive a tuple from scan B. However, scan B cannot provide a tuple since it would overflow the right buffer of Query 1's merge join. Similarly, Query 1 cannot consume any tuples from its right full buffer, causing a deadlock in the evaluation of the plan-DAG.

Given a DAG $G = (V, E)$, we define the directionless graph of G to be the undirected graph $G_d = (V, \{\{u, v\} \mid ((u, v) \in E) \vee ((v, u) \in E)\})$. G_d is also known as the *shadow* of G. Deadlocks may occur in a pipelined query plan graph every time there exists a cycle in the directionless graph, due to different tuple consuming/producing rates across nodes and the finite buffers between producers and consumers. In a pipelined plan, both the consuming and producing rate of a node may depend on the producing rate of the children nodes and the consuming rate of the parent. In general, two nodes may indirectly affect each other's consuming/producing rate as long as there is some graph connection between them (a path between the two nodes in the directionless graph). Intuitively, a cycle in the directionless graph means that a node can be in a situation where a requirement for a tuple production/consumption places an additional requirement to that node. If those two requirements are conflicting, a deadlock may occur. For instance, a node can set the requirement "to produce a tuple, a tuple needs to be consumed," which may trigger a requirement on the same node of the form "to consume a tuple, a tuple needs to be produced."

A proactive way to remove the possibility of deadlock in query plan graphs is to always materialize the results produced by shared query nodes. This is the default strategy in Multi-Query Optimization [Sellis 1988]. Ideally, we would like to use pipelining as much as possible and rely on selective materialization only when it is absolutely necessary. The authors in [Dalvi et al. 2001] propose to solve the deadlock problem in pipelined MQO plans by deciding at query optimization which shared nodes will be pipelined and which will be materialized, forming this way a *valid schedule* (deadlock-free evaluation of the plan-DAG). They observe that whenever a cycle exists in the directionless graph (called a C-cycle) and certain conditions apply, then a schedule is not realizable (may deadlock at run time). Once the deadlocking C-cycles are identified, they are broken by selective materialization of a subset of shared nodes. The proposed algorithm makes conservative decisions since it relies on static analysis of the query plans. Due to pessimistic assumptions, a large number of safe query graphs will be flagged as potentially deadlocking and will lead to unnecessary materialization. Since the analysis of the query graph is done at query optimization time, the algorithm suffers from a lack of information about operator selectivities and real materialization costs. As a result, even if the graph is going to deadlock at run time, the algorithm can make suboptimal choices.

4.6.2 Detecting and resolving deadlocks

Each node in a plan-DAG corresponds to either a base relation, the final result of some query, or an intermediate result. Edges represent producer-consumer relationships: edge (u, v) means that tuples of u are used by v. Each edge of a plan-DAG has a finite buffer associated with it. Over time, some buffers may become full and some may become empty. We label full-buffer edges as “heavy” and empty-buffer edges as “light.” Heavy and light edges can slow the execution of the overall plan, and if enough edges are heavy or light, the execution will introduce deadlocking structures within the graph. A worst-case deadlock prevention approach, such as the one in [Dalvi et al. 2001], attempts to materialize as many nodes as needed to avoid heavy and light edges altogether.

We propose an optimistic approach, fixing deadlock when and if it arises. In our model, we attempt to pipeline everything, only materializing nodes in the event of a deadlock. Deadlock detection and resolution (materialization choices) are done by monitoring the status of buffers in the query graph. To capture this, we define a *buffered* plan-DAG, a graph which represents the current state of buffers at some point during the query execution with a *buffer state function* $\boldsymbol{s}: E \rightarrow \{\mathbf{F}, \mathbf{E}, \mathbf{N}\}$. Each edge of a buffered plan-DAG is labeled either **F** (for a full buffer), **E** (for empty), or **N** (for neither). We can now formulate a notion of “waiting” which captures producer and consumer interaction. Given any final consumer v (that is, a node with outdegree zero), at some point in time, v may be waiting on a tuple from another node u; in that case, the buffer state function would give $\boldsymbol{s}(u, v) = \mathbf{E}$. Similarly, for every initial producer v (that is, a node with indegree zero), v may be waiting to put something on the input buffer of one or more u (in that case, $\boldsymbol{s}(u, v) = \mathbf{F}$). To model the notion of waiting for a buffered plan-DAG $G' = ((V, E), \boldsymbol{s})$, we define a corresponding *waits-for* graph $\mathsf{Waits}(G')$ as the directed graph (V, E') such that

$$E' = \{(v, u) \mid [(u, v) \in E \wedge \boldsymbol{s}(u, v) = \mathbf{E}] \vee [(v, u) \in E \wedge \boldsymbol{s}(v, u) = \mathbf{F}]\}.$$

Given this definition, a deadlock is equivalent to the existence of a cycle in the above waits-for graph. That is, in a deadlocked situation, all final consumers are waiting on one of its input buffers to contain something, and all initial producers are waiting on one of its output buffers to pull a tuple.

As an example, consider the query plan-DAG shown in Figure 4-18. There are two paths (modulo edge directions) from one SCAN node to the other. The intuitive argument in [Dalvi et al. 2001] for why some of the nodes here need to be materialized (prior to execution) is simply that the rates of consumers and producers along the two paths could possibly vary greatly, even if one side has a materialized node and the other one does not, leading to buffer overflow on one side. If this were to occur in our model, then the two paths between the SCAN nodes would have all **F** and **E** edges with the proper edge directions. This results in a cycle in the waits-for graph. Finding a *waits-for* cycle is equivalent to detecting the precise moment at which a particular buffer configuration will eventually stall computation. That is, we can both detect and optimally resolve the problem immediately and efficiently as it arises, when the last buffer in the cycle empties or becomes full.

Detection. To detect a cycle efficiently, we use data structures for *dynamic connectivity*, a problem that has been studied extensively in the algorithms literature. The details, which are out of the scope of this dissertation, are in [Shkapenyuk et al. 2005], where we show that $\mathsf{Waits}(G')$ can be checked for cycles in linear time using dynamic connectivity. Note that this is a theoretical result; in practice, checking for cycles can typically be done even faster (dynamically) if the graph is sparse.

Resolution. Once a deadlock is detected, we must resolve it by materializing some problematic nodes. This materialization problem is much easier than the static case [Dalvi et al. 2001], in that it can be optimally solved in polynomial (quadratic) time (the proof is in [Shkapenyuk et al. 2005]). At a high level, deadlock resolution can be done as follows. When deadlock arises due to a bad buffer configuration, every cycle in the *waits-for* graph contains a specific vertex v (the last node "responsible" for deadlock). Therefore, the *waits-for* graph has a particular structure that we can exploit. We build an undirected graph G'' that represents the possible cycles through v. G'' has weights on nodes, corresponding to materialization costs. G'' also has two distinguished nodes v_{in} and v_{out}; nodes connected to v_{in} (respectively, v_{out}) correspond to nodes with paths to v (respectively, paths from v) in the *waits-for* graph. We prove (in [Shkapenyuk et al. 2005]) that finding a minimum vertex cut in G'' that disconnects v_{in} from v_{out} is equivalent to choosing a minimum cost set of nodes to materialize that breaks the existing deadlock.

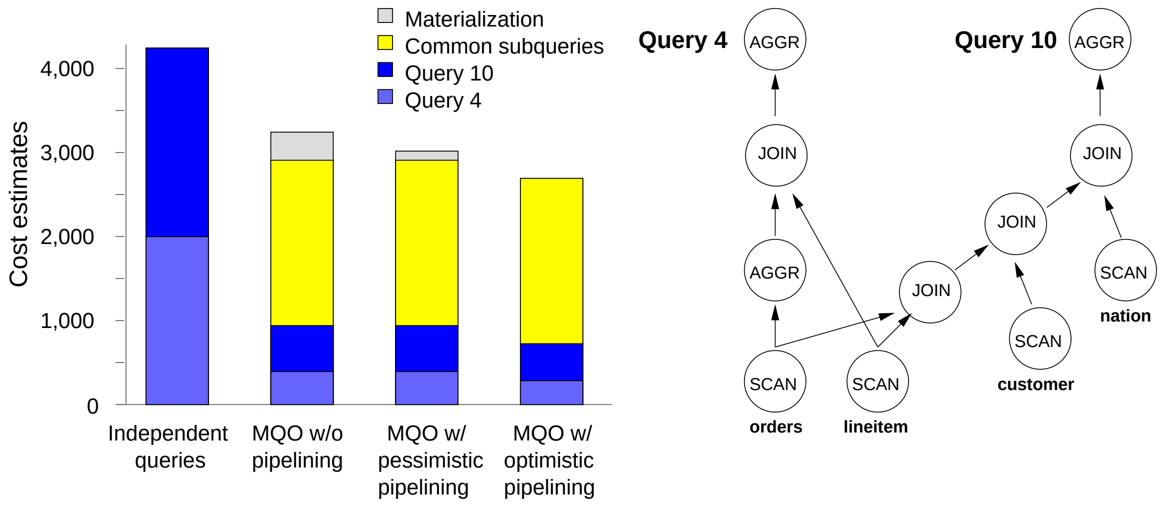


Figure 4-19. Cost estimate comparison for four different approaches in evaluating two TPC-H queries.

4.6.3 Experimental evaluation

We evaluate the potential benefits of our dynamic materialization strategy on a TPC-H workload that can take advantage of query subplan sharing. We configure TPC-H with scale factor 10 (database size is 10 GB) and use queries Q4 and Q10 with shared scans on the LINEITEM table and the smaller ORDERS table. To better compare the static materialization policy and our own dynamic one, we use Microsoft SQL Server's EXPLAIN facility for run-time estimates of the submitted queries. The SQL Server optimizer uses sophisticated cost models which are more accurate for estimating relative query costs than the simplistic cost models largely used the multi-query optimization literature. Since SQL Server does not support multi-query optimization, we mimic it by manually materializing common subexpressions and appropriately adjusting the query costs to reflect materialization or multiple reads of shared results. The experiments are conducted on a Dual Pentium III 900MHz with 1GB of RAM running SQL Server 2000 on Windows Server 2003.

The right part of Figure 4-19 shows the merged query plan graph for the two TPC-H queries. Since the plan-DAG contains a cycle, a static deadlock analysis (*pessimistic* pipelining) will need to materialize all edges coming out of one of the relations. Since the scan on LINEITEM is significantly more expensive than the scan on ORDERS, a greedy static algorithm chooses to pipeline the results of the scan on LINEITEM and materialize the scan on ORDERS. However, since the consumption rates for the ORDERS and LINEITEM relations are not significantly different for different consumers, an optimistic approach that pipelines everything will never reach deadlock. This is reflected on the cost estimates for running the batch of two queries shown in the left part of Figure 4-19. The y-axis indicates units of time used by SQL Server. The leftmost bar corresponds to the cost evaluating the two queries independently. The next bar is the cost of performing multi-query optimization with full materialization, and provides a speedup of 1.32. Applying the pessimistic static pipelining algorithm provides a speedup of 1.67 by avoiding materialization and incurring read costs on the ORDERS relation. Our optimistic approach further removes any materialization costs and shared read costs by using a fully pipelined plan-DAG, and provides a speedup of 1.89.

4.7. Chapter summary

This chapter described QPipe, a staged relational query engine that enhances locality across concurrent queries. It also introduced a set of techniques and policies to exploit overlapping work between concurrent queries at run time and showed how these techniques are incorporated inside QPipe.

QPipe offers several advantages over traditional query engine designs. It provides full intra-query parallelism, taking advantage of all available CPUs in multiprocessor servers for evaluating a single query, regardless of the plan's complexity. By processing a batch of tuples for each query at every μEngine, QPipe improves instruction temporal locality and avoids extraneous procedure calls in the DBMS code, when compared to tuple-by-tuple query engines. Furthermore, by applying on-demand simultaneous pipelining of common intermediate results across queries, QPipe avoids costly materializations and can efficiently evaluate plans produced by a multi-query optimizer.

Multiple concurrent queries often operate on the same set of tables, using the same set of basic operators in their query plans. Modern DBMS can therefore execute concurrent queries faster by aggressively exploiting commonalities in the data or computation involved in the query plans. Current query engines execute queries independently and rely on the buffer pool to exploit common data, which may miss sharing opportunities due to unfortunate timing. Previous efforts to explore overlapping work include multiple query optimization, materialized views, and exploitation of previous results; all these approaches, however, involve caveats and overhead that makes them impractical in several cases. In this chapter we changed the query engine philosophy from query-centric (one-query, many-operators) to operator-centric (one-operator, many-queries) and showed how we can proactively detect and exploit common data and computation at execution time with no additional effort or overhead.

We also proposed a new model that characterizes deadlocks in the generalized context of *query plan graphs*. Our dynamic model uses the well-understood concept of *waits-for* graphs to define deadlocks in pipelined execution engines. We use information about the state of the intermediate query buffers (full, empty, or non-empty) without making any assumptions about operator consumer/producer rates. This allows us to pipeline the results of every query node, only materializing the tuples in the event of a real deadlock. Based on our model, we proposed an efficient algorithm to detect deadlocks at run time and choose an optimal set of nodes to materialize that minimizes the total cost of executing all concurrent queries.

Locality and predictability of different tasks running in a system has long been the key property that computer and storage architects, along with software designers have exploited to build high-performance computing systems. In this chapter we showed that the key to optimal performance is exposing all potential locality both in data and in computation, by grouping similar tasks together. QPipe, our proposed query execution engine architecture, leverages the fact that a query plan offers an exact description of all task items needed by a query, and employs techniques to expose and exploit locality in both data and computation across different queries. Most importantly, by successfully applying the QPipe architecture to two open source DBMS and two storage managers, we demonstrated that the QPipe design can apply with relatively few changes to any DBMS.

Chapter 5

Improving instruction cache performance

Instruction-cache misses account for 25-40% of execution time in Online Transaction Processing (OLTP) database workloads. In contrast to data cache misses, instruction misses cannot be overlapped with out-of-order execution. Current chip design practices do not allow increases in the size or associativity of instruction caches that would help reduce misses. On the contrary, the effective instruction cache size is expected to further decrease with the adoption of Chip Multithreading designs (multiple on-chip processor cores and multiple simultaneous threads per core). Different concurrent database threads, however, execute similar instruction sequences over their lifetime, too long to be captured and exploited in hardware. The challenge, from a software designer's point of view, is to identify and exploit common code paths across threads executing arbitrary operations, thereby eliminating extraneous instruction misses.

This chapter describes *Synchronized Threads through Explicit Processor Scheduling* (STEPS), a mechanism based on the principles of StagedDB to increase instruction locality in database servers executing transaction processing workloads. STEPS works at two levels to increase reusability of instructions brought in the cache. At a higher level, STEPS applies a staged execution scheme by using synchronization barriers to form teams of threads executing the same system component. Since typically transactional system components can overwhelm the instruction cache, STEPS schedules special fast context-switches within a team, at very fine granularity, to reuse sets of instructions across team members. To find points in the code where context-switches should occur, we develop *autoSTEPS*, a code processing tool that runs directly on the DBMS software binary.

We demonstrate the effectiveness of STEPS on *Shore*, a research prototype database system shown to be governed by similar bottlenecks as commercial systems. Using microbenchmarks on real and simulated processors, we observe that STEPS eliminates up to 96% of instruction-cache misses for each additional team thread, and at the same time eliminates up to 64% of mispredicted branches by providing a repetitive execution pattern to the processor. When performing a full-system evaluation, on real hardware, with TPC-C, the industry-standard transactional benchmark, STEPS eliminates two thirds of instruction-cache misses and provides up to 1.4 overall speedup. This chapter shows that STEPS can minimize both capacity and conflict instruction cache misses for arbitrarily long code paths.

5.1. Introduction

Good instruction-cache performance is crucial in modern Database Management System (DBMS) installations running Online Transaction Processing (OLTP) applications. OLTP applications include banking, e-commerce, reservation systems, inventory management; a common requirement is the ability of the DBMS software to execute fast multiple concurrent transactions, which are typically short in duration. In large scale installations, server software is leveraging increasingly higher-capacity main memories and highly parallel storage subsystems to efficiently hide I/O latencies. As a result, DBMS software performance is largely determined by the ability of the processors to continuously execute instructions without *stalling*. Recent studies show that between 22% and 45% of the execution time in the prevailing OLTP benchmark (TPC-C), is attributed to instruction-cache misses [Shao and Ailamaki 2004; Keeton et al. 1998; Barroso et al. 1998].

Over the past few years, a wide body of research has proposed techniques to identify and reduce CPU performance bottlenecks in database workloads [Shatdal et al. 1994; Graefe and Larson 2001; Ailamaki et al. 2001]. Since memory access times improve much slower than processor speed, performance is bound by instruction and data cache misses that cause expensive main-memory accesses [Ranganathan et al. 1998; Ailamaki et al. 1999]. Related research efforts propose hardware and compiler techniques to address instruction-cache performance [Ranganathan et al. 1998; Ramirez et al. 2001]. The focus, however, is on single-thread execution (single transaction). While the proposed techniques can enhance instruction-cache performance by increasing *spatial locality* (utilization of instructions contained in a cache block), they cannot increase *temporal locality* (instruction reusability) since that is a direct function of the application nature. A study on Oracle reports a 556KB OLTP code footprint [Lo et al. 1998]. With modern CPUs having 16-64KB instruction cache size (L1-I cache), OLTP code paths are too long to achieve cache-residency.

Although instruction reusability (temporal locality) cannot be increased within a single execution thread by hardware or compiler techniques, DBMS software is inherently multi-threaded. It executes multiple concurrent threads (each carrying out a transaction execution), and therefore exists the opportunity to alter software design to reuse instructions in the cache *across* different concurrent threads. The payoff in such a software approach can be large, and complementary to any hardware/compiler technique. For example, consider a software mechanism that can identify ten concurrent threads, executing the same function call (the code path of which is far larger than the instruction cache). Each thread execution will yield a default number of instruction misses. If, however, we perfectly reuse the instructions one thread brings gradually in the cache, across all ten threads, we can achieve a 90% reduction in instruction cache misses. That is, in a group of threads, we could potentially turn one thread's instruction cache misses to cache hits for all the other threads. Applying this idea, however, to a DBMS consisting of millions lines of code, executing different types of transactions, each with different requirements, and unpredictable execution sequences (due to lock requests, frequent critical sections, I/O requests) is a challenging task. To the best of our knowledge, this line of work is the first to address instruction cache misses in transaction processing from within the DBMS software.

This thesis chapter presents STEPS (*Synchronized Threads through Explicit Processor Scheduling)*, a methodology and its application for increasing instruction locality in database servers executing transaction processing workloads, and we also present autoSTEPS, a code profiling tool that automates the application of STEPS to any database system. STEPS works at two levels to increase reusability of instructions brought in the cache. At a higher level, synchronization barriers form *teams* of threads that execute the same system component. Within a team, STEPS schedules special fast context-switches at very fine granularity, to reuse sets of instructions across team members. Both team formation and fine-grain context-switching are low-overhead software mechanisms, designed to co-exist with all DBMS internal mechanisms (locking, logging, deadlock detection, buffer pool manager, I/O subsystem), and flexible enough to work with any arbitrary OLTP workload. STEPS requires only few code changes, targeted at the thread package. To find points in the DBMS core code where context-switches should occur, we develop autoSTEPS, a code analysis tool that runs directly on the DBMS software binary, thus eliminating the need to examine and modify DBMS source code. STEPS can minimize both *capacity* and *conflict* instruction cache misses for arbitrarily long code paths.

We demonstrate the effectiveness of STEPS on *Shore*, a research prototype database system shown to be governed by similar bottlenecks as commercial systems. First, using microbenchmarks on real and simulated processors, we show that STEPS eliminates up to 96% of instruction-cache misses for each additional team thread, and at the same time eliminates up to 64% of mispredicted branches by providing a repetitive execution pattern to the processor. When performing a full-system evaluation, on real hardware, with TPC-C, the industry-standard transactional benchmark, STEPS eliminates two thirds of instruction-cache misses and provides up to 1.4 overall speedup. To the best of our knowledge, this is the first software approach to provide explicit thread scheduling for improving instruction cache performance. The contributions of the chapter are:

- A novel technique that enables thread scheduling at very fine granularity to reuse instructions in the cache across concurrent threads.
- A tool to automatically find the boundaries in the code that the instruction cache fills up.
- The implementation and evaluation of the presented techniques inside a full-blown research prototype database system running a multi-user transactional benchmark on real hardware.

The rest of the chapter is organized as follows. Section 5.2 presents background information and discusses related work. Section 5.3 introduces our methodology and evaluates the basic implementation of STEPS. Section 5.4 describes and evaluates the application of STEPS to full OLTP workloads. Section 5.5 presents autoSTEPS and discusses applicability to commercial DBMS. The last section contains the chapter summary.

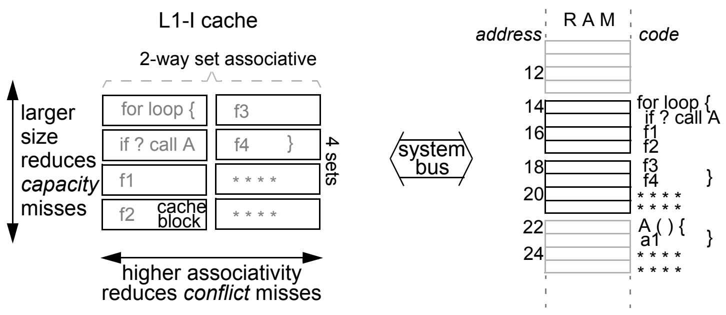


Figure 5-1. Example of a 2-way set associative, 4-set (8 cache blocks) L1-I cache. Code stored in RAM maps to one set of cache blocks and is stored to any of the two blocks in that set. For simplicity we omit L2/L3 caches. In this example, the *for-loop* code fits in the cache only if procedure A is never called. In that case, repeated executions of the code will always hit in the L1-I cache. Larger code (more than eight blocks) would result in *capacity* misses. On the other hand, frequent calls to A would result to *conflict* misses because A's code would replace code lines f3 and f4 needed in the next iteration. A code layout optimization technique [Romer et al. 1997; Ramirez et al. 2001] would place procedure A's code on address 20, so that it does not conflict with the for-loop code.

5.2. Background

To bridge the CPU/memory performance gap, today's processors employ a hierarchy of caches that maintain recently referenced instructions and data close to the processor. Figure 5-1 shows an example of an instruction cache organization and explains the difference between capacity and conflict cache misses. Recent processors — e.g., IBM's Power5 — have up to three cache levels. At each hierarchy level, the corresponding cache trades off lookup speed for size. For example, level-one (L1) caches at the highest level are small (e.g., 16KB-64KB), but operate at processor speed. In contrast, lookup in level-two (L2) caches typically incurs up to an order of magnitude longer time because they are several times larger than the L1 caches (e.g., 512K-8MB). L2 lookup, however, is still several orders of magnitude faster than memory accesses (typically 300-400 cycles). Therefore, the effectiveness of cache hierarchy is extremely important for performance.

5.2.1 The instruction cache problem

In contrast to data cache accesses, instruction cache accesses are serialized and cannot be overlapped. Instruction cache misses prevent the flow of instructions through the proces-

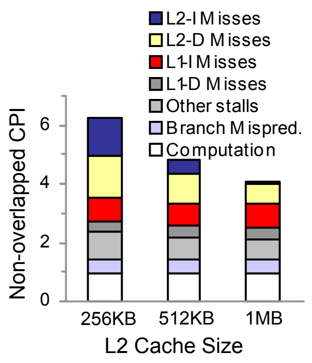


Figure 5-2. (a) TPC-C CPU stall breakdown on PentiumPro. With larger L2 cache size L1-I misses become the dominant stall factor (3rd box from top) [Keeton et al. 1998]

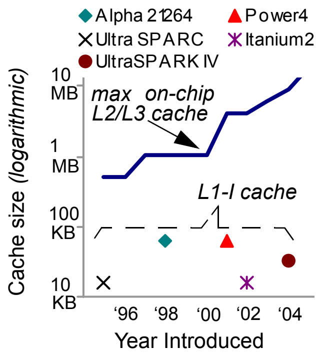


Figure 5-2. (b) A decade-spanning trend shows that L1-I caches do not grow, while secondary, on-chip caches become increasingly larger.

sor and directly affect performance. To maximize first-level instruction cache utilization and minimize stalls, application code should have few branches (exhibiting high *spatial locality*), a repeating pattern when deciding whether to follow a branch (yielding low *branch misprediction* rate), and most importantly, the "working set" code footprint should fit in the L1-I cache. Unfortunately, OLTP workloads exhibit the exact opposite behavior [Keeton et al. 1998]. A study on Oracle reports a 556KB OLTP code footprint [Lo et al. 1998]. With modern CPUs having 16-64KB L1-I cache sizes, OLTP code paths are too long to achieve cache-residency. Moreover, the importance of L1-I cache stalls increases with larger L2 caches, as shown in Figure 5-2a (stalls shown as non-overlapping components; I-cache stalls are actually 41% of the total execution time [Keeton et al. 1998]). As a large (or highly associative) L1-I cache may adversely impact the CPU's clock frequency, chip designers cannot increase L1-I sizes (associativity) despite the growth in secondary caches, as shown in Figure 5-2b.

Current chip design trends towards improving process performance are leading to thread-parallel architectures, where multiple threads or processes can run simultaneously on a single chip via multiple on-chip processor cores (chip multiprocessors — CMP) and/or multiple simultaneous threads per processor core (simultaneous multithreading — SMT)[1]. To be able to fit more cores on a single chip without overheating, and also save time in hardware verification, chip designers are expected to use simpler, "lean" cores as building blocks. The instruction cache size of these cores is not expected to grow. SMT chips already operate on a reduced effective instruction cache size, since the instruction cache is shared among all simultaneous threads. In future processors, the combined effect of larger L2 cache sizes and small (or shared) L1-I caches will make instruction cache stalls the key performance bottleneck.

5.3. STEPS: Introducing cache-resident code

All OLTP transactions, regardless of the specific actions they perform, execute common database mechanisms (e.g., index traversing, buffer pool manager, lock manager, logging). In addition, OLTP typically processes hundreds of requests concurrently (the top performing system in the TPC-C benchmark suite supports over one million users and handles hundreds of concurrent client connections[1]). High-performance disk subsystems and high-concurrency locking protocols ensure that, at any time, there are multiple threads in the CPU ready-to-run queue.

We propose to exploit the characteristics of OLTP code by reusing instructions in the cache across a group of transactions, effectively turning an arbitrarily large OLTP code footprint into nearly cache-resident code. We synchronize transaction groups executing common code fragments, improving performance by exploiting the high degree of OLTP concurrency. The rest of this section describes the basic implementation of STEPS, and details its behavior using transactional microbenchmarks.

5.3.1 Basic implementation of STEPS

Transactions typically invoke a basic set of operations: begin, commit, index fetch, scan, update, insert, and delete. Each of those operations involves several DBMS functions and can easily overwhelm the L1-I cache of modern processors. Experimenting with the Shore database storage manager [Carey et al. 1994] on a CPU with 64KB L1-I cache, we find

1. As of the time this dissertation is written, all major chip manufacturers have announced or made available CMP and SMT designs (Intel's Pentium4 currently implements a 2-way SMT design which is marketed as *Hyperthreading* design).
1. Transaction Processing Performance Council. `http://www.tpc.org`

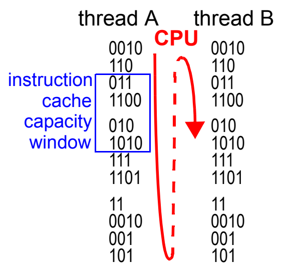


Figure 5-3. (a) As the instruction cache cannot fit the entire code, when the CPU switches (dotted line) to thread B, this incurs the same number of misses.

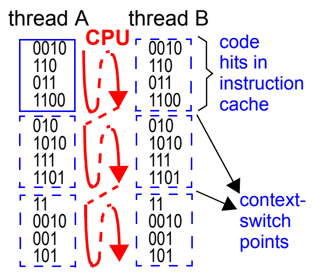


Figure 5-3. (b) By "breaking" the code into three pieces that fit in the cache, and switching back and forth between the two threads, thread B finds all instructions in the cache.

that even repeated execution of a single operation always incurs additional L1-I misses. Suppose that N transactions, each being carried out by a thread, perform an index fetch (traverse a B-tree, lock a record, and read it). For now, we assume that transactions execute uninterrupted (all pages are in main memory and locks are granted immediately). A DBMS would execute one index fetch after another, incurring more L1-I cache misses with each transaction execution. We propose to reuse the instructions one transaction brings in the cache, thereby eliminating misses for the remaining N-1 transactions.

As the code path is almost the same for all N transactions (except for minor, key-value processing), STEPS follows the code execution for one transaction and finds the point at which the L1-I cache starts evicting previously-fetched instructions. At that point STEPS context-switches the CPU to another thread. Once that thread reaches the same point in the code as the first, we switch to the next. The Nth thread switches back to the first one, which fills the cache with new instructions. Since the last N-1 threads execute the same instructions as the first, they incur significantly fewer L1-I misses (conflict misses, since each code fragment's footprint is smaller than the L1-I cache).

Figures 5-3a and 5-3b illustrate the scenario mentioned above for two threads. Using STEPS, one transaction paves the L1-I cache, incurring all compulsory misses. A second, similar transaction follows closely, finding all the instructions it needs in the cache. Next, we describe (a) how to minimize the context-switch code size, and, (b) where to insert the context-switch calls in the DBMS source code.

5.3.2 Fast, efficient context-switching

Switching execution from one thread (or process) to another involves updating OS and DBMS software structures, as well as updating CPU registers. Thread switching is less costly than process switching (depending on the implementation). Most commercial DBMS involve a light-weight mechanism to pass on CPU control (Shore uses user-level threads). Typical context-switching mechanisms, however, occupy a significant portion of the L1-I cache and take hundreds of processor cycles to run. Shore's context-switch, for instance, occupies half of Pentium III's 16KB L1-I cache.

To minimize the overhead of context-switch we apply a universal design guideline: make the common case fast. The common case here is switching between transactions executing the same operation. STEPS executes only the core context-switch code and updates only CPU state, ignoring thread-specific software structures such as the ready queue, until they must be updated. The minimum code needed to perform a context-switch on a IA-32 architecture — save/restore CPU registers and switch the base and stack pointers — is 48 bytes (76 in our implementation). Therefore, it only takes three 32-byte (or two 64-byte) cache blocks to store the context-switch code. One optimization that several commercial thread packages (e.g., Linux threads) make is to skip updating the floating point registers until they are actually used. For a subset of the microbenchmarks we apply a similar optimization using a flag in the core context-switch code.

5.3.3 Finding context-switching points in Shore

Given a basic set of transactional operations, we find appropriate places in the code to insert a call to `CTX(next)` (the context-switch function), where next is a pointer to the next thread to run. STEPS tests candidate points in the code by executing the DBMS operations (on simple, synthetic tables) and by inserting CTX (next) calls before or after major function calls. Using hardware counters (available on almost all processors[1]), we measure the L1-I cache misses for executing various code fragments. Starting from the beginning of a DBMS operation and gradually moving towards its end, STEPS compares the number

1. Intel Corporation. "IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide." (Order Number 253668).

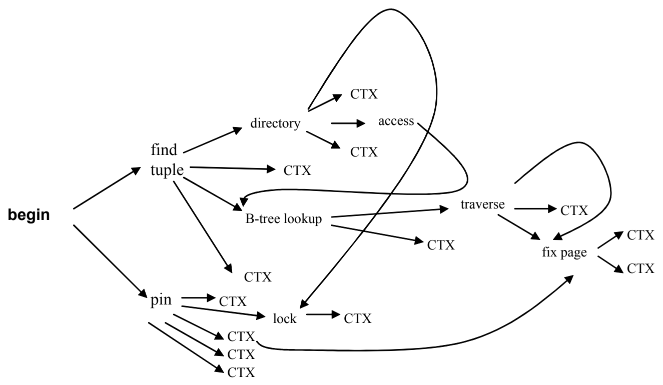


Figure 5-4. Context-switch (CTX) calls placed in Shore's source code for index_fetch. Arrows represent code paths, to be followed from left to right and from top to bottom until a CTX is encountered. Note that the actual number of context-switches can be larger than the number of CTX, since a CTX can be placed inside a for loop (i.e., for each level of a B-tree).

of L1-I misses the execution of a code fragment incurs alone with the total number of misses when executing the same fragment twice (using the fast CTX call). A CTX point is inserted as soon as STEPS detects a knee in the curve of the number of L1-I cache misses. STEPS continues this search until it covers the entire high-level code path of a DBMS operation, for all operations.

The method of placing CTX calls described above does not depend on any assumptions about the code behavior or the cache architecture. Rather, it dynamically inspects code paths and chooses every code fragment to reside in the L1-I cache as long as possible across a group of interested transactions. If a code path is self-conflicting (given the associativity of the cache), then our method will place CTX calls around a code fragment that may have a significantly smaller footprint than the cache size, but will have fewer conflict misses when repeatedly executed. Likewise, this method also explicitly includes the context-switching code itself when deciding switching points. Figure 5-4 shows the context-switch calls (CTX) placed in index_fetch in Shore, using a machine with 64KB instruction cache and 2-way set associativity.

The rest of this section evaluates STEPS using microbenchmarks, whereas the complete implementation for OLTP workloads is described in Section 5.4. In all experiments we refer as "Shore" to the original unmodified system and as "STEPS" to our system built on top of Shore.

CPU	Cache characteristics	
AMD AthlonXP	L1 I + D cache size associativity / block size	64KB + 64KB 2-way / 64 bytes
	L2 cache size	256KB
Pentium III	L1 I + D cache size associativity / block size	16KB + 16KB 4-way / 32 bytes
	L2 cache size	256KB
Simulated IA-32 (SIMFLEX)	L1 I + D cache size associativity	[16, 32, 64KB] [direct, 2, 4, 8, full]

Table 5-1. Processors used in microbenchmarks.

5.3.4 STEPS in practice: microbenchmarks

We conduct experiments on the processors shown in Table 5-1. Most experiments run on the AthlonXP, which features a large, 64KB L1-I cache. High-end installations typically run OLTP workloads on server processors (such as the ones shown in Figure 5-2b). In our work, however, we are primarily interested in the number of L1-cache misses. From the hardware perspective, this metric depends on the L1-I cache characteristics: size, associativity, and block size (and not on clock frequency, or the L2 cache). Moreover, L1-I cache misses are measured accurately using processor counters, whereas time-related metrics (cycles, time spent on a miss) can only be estimated and depend on the entire system configuration. Instruction misses, however, translate directly to stall time since they cannot be overlapped with out-of-order execution.

Shore runs under Linux 2.4.20. We use PAPI [Browne et al. 1999] and the `perfctr` library to access the AthlonXP and PIII counters. The results are based on running *index fetch* on various tables consisting of 25 `int` attributes and 100,000 rows each. The code footprint of *index fetch* without searching for the index itself (which is already loaded) is 45KB, as measured by a cache simulator (described in Section 3.2.4). Repeatedly running *index fetch* would incur no additional misses in a 45K *fully-associative* cache, but may incur conflict misses in lower-associativity caches, as explained in Figure 5-1. We report results averaged over 10 threads, each running *index fetch* 100 times.

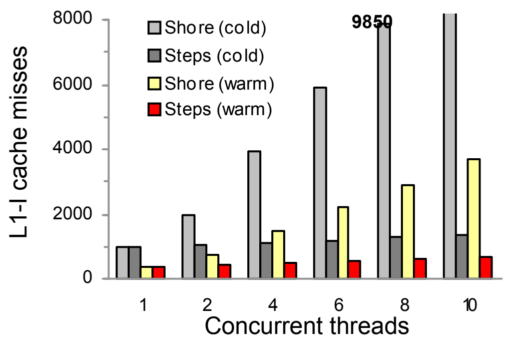


Figure 5-5. STEPS reduces significantly instruction-cache misses as the number of concurrent threads increases, both with cold and with warm caches.

5.3.4.1 Instruction misses and thread group size

We measure L1-I cache misses for *index fetch*, for various thread group sizes. Both STEPS and Shore execute the fast CTX call, but STEPS multiplexes thread execution, while Shore executes the threads serially. We first start with a cold cache and flush it between successive *index fetch* calls, and then repeat the experiment starting with a warm cache. Figure 5-5 shows the results on the AthlonXP.

STEPS only incurs 33 misses for every additional thread, with both a cold and a warm cache. Under Shore, each additional thread adds to the total exactly the same number of misses: 985 for a cold cache (capacity misses) and 373 for a warm cache (all conflict misses since the working set of *index fetch* is 45KB). The numbers show that Shore could potentially benefit from immediately repeating the execution of the same operation across different threads. In practice, this does not happen because: (a) DBMS threads suspend and resume execution at different places of the code (performing different operations), and, (b) even if somehow two threads did synchronize, the regular context-switch code would itself conflict with the DBMS code. If the same thread, however, executes the same operation immediately, it will enjoy a warm cache. For the rest of the experiments we always warm up Shore with the same operation, and use the fast CTX call, therefore reporting worst-case lower bounds.

The following brief analysis derives a formula for the L1-I cache miss reduction bounds as a function of the thread group size (for similarly structured operations with no exceptional events). Suppose executing an operation P once, with cold cache, yields m_P misses.

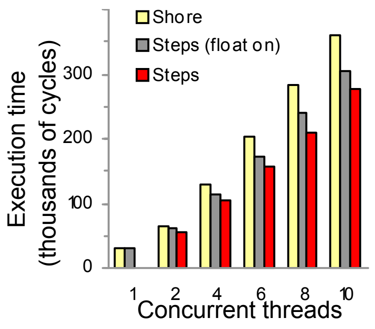


Figure 5-6. (a) Execution time for one to ten concurrent threads. STEPS with float on always updates floating point registers.

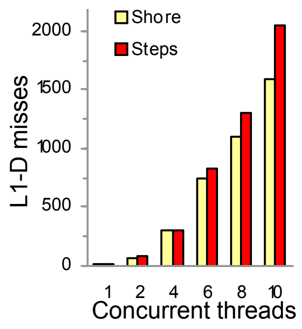


Figure 5-6. (b) L1-D cache misses for one to ten concurrent threads.

Executing *P*, *N* times, flushing the cache in-between, yields $N \cdot m_P$ misses. A warm cache yields $N \cdot a \cdot m_P$, $0 < a \le 1$ misses because of fewer capacity misses. In STEPS, all threads except the first incur $N \cdot \beta \cdot m_P$ misses, where $0 < \beta < 1$. For a group size of *N*, the total number of misses is $m_P + (N-1) \cdot \beta \cdot m_P$. For an already warmed-up cache this is: $a \cdot m_P + (N-1) \cdot \beta \cdot m_P$. When comparing STEPS to Shore, we express the miss reduction percentage as: $(1 - \text{\#misses after} / \text{\#misses before}) \cdot 100\%$. Therefore, the bounds for computing the L1-I cache miss reduction are:

$$\frac{(N-1)}{N} \cdot (1-\beta) \cdot 100\% \qquad\qquad \frac{N-1}{N} \cdot \left(1 - \frac{\beta}{a}\right) \cdot 100\%$$

for cold cache for warm cache

N: group size

For *index fetch*, we measure $a = 0.373$, $\beta = 0.033$, giving a range of 82% - 87% of overall reduction in L1-I cache misses for 10 threads, and 90% - 96% for 100 threads. For the *tuple update* code in Shore, the corresponding parameters are: $a = 0.35$ and $\beta = 0.044$.

The next microbenchmarks examine how the savings in L1-I cache misses translate into execution time and how STEPS affects other performance metrics.

5.3.4.2 *Speedup and level-one data cache misses*

Keeping the same setup as before and providing Shore with a warmed-up cache we measure the execution time in CPU cycles and the number of level-one data (L1-D) cache

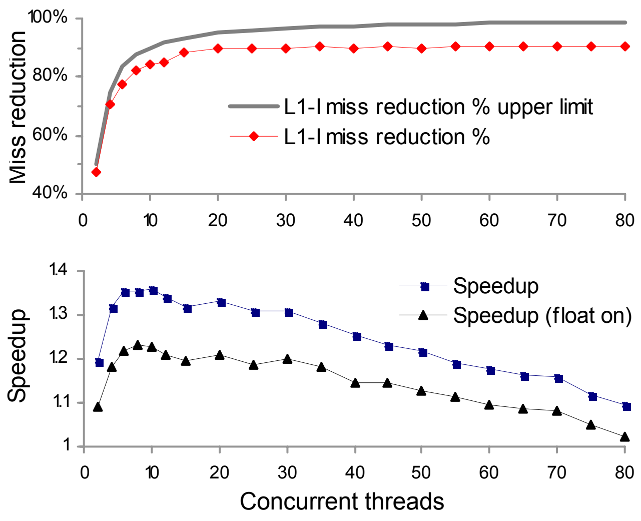


Figure 5-7. Lower bounds for speedup using a warm cache for Shore (bottom graph) and percentage of reduction in L1-I cache misses (top graph) of STEPS over Shore, for 2-80 concurrent threads. The top line shows the maximum possible reduction.

misses on the AthlonXP. Figure 5-6a shows that STEPS speedup increases with the number of concurrent threads. We plot both STEPS performance with a CTX function that always updates floating point registers (float on) and with a function that skips updates. The speedup for 10 threads is 31% while for a cold cache it is 40.7% (not shown).

While a larger group promotes instruction reuse it also increases the *collective* data working set. Each thread operates on a set of private variables, buffer pool pages, and metadata which form the thread's data working set. Multiplexing thread execution at the granularity STEPS does, results in a larger collective working set which can overwhelm the L1-D cache (when compared to Shore). Figure 5-6b shows that STEPS incurs increasingly more L1-D cache misses as the thread group size increases. For up to four threads, however, the collective working set has comparable performance to single-thread execution. Fortunately, L1-D cache misses have minimal effect on execution time (as also seen by the STEPS speedup). The reason is that L1-D cache misses that hit in the L2 cache (i.e., are serviced within five to ten cycles) can be easily overlapped by out-of-order execution [Ailamaki et al. 1999]. Moreover, in the context of Simultaneous Multithreading (SMT), it has been shown that for eight threads executing simultaneously an OLTP workload and sharing the CPU caches, additional L1-D misses can be eliminated [Lo et al. 1998].

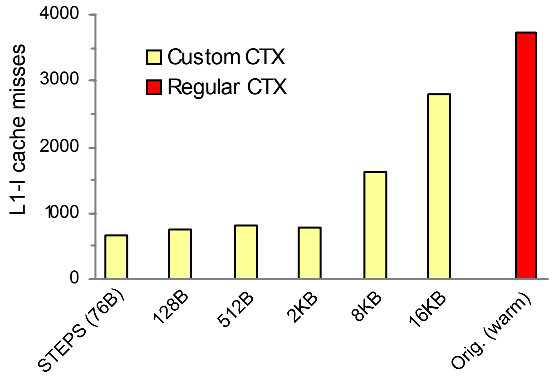


Figure 5-8. Number of L1-I cache misses for ten concurrent threads executing index_fetch, varying the footprint of the context-switch code. The leftmost bar corresponds to STEPS CTX code, the intermediate bars correspond to STEPS CTX padded with consecutive *nops*, and the rightmost bar provides a comparison with the default number of L1-I cache misses with no STEPS, when the cache is warm.

On the other hand, there is no real incentive in increasing the group size beyond 10-20 threads, as the upper limit in the reduction of L1-I cache misses is already 90-95%. Figure 5-7 plots the STEPS speedup (both with float on/off) and the percentage of L1-I cache misses reduction for 2-80 concurrent threads. The reason that the speedup deteriorates for groups larger than 10 threads is because of the AMD's small, 256KB unified L2 cache. In contrast to L1-D cache misses, L2-D misses cannot be overlapped by out-of-order execution. STEPS always splits large groups (discussed in Section 4) to avoid the speedup degradation.

5.3.4.3 Increasing the size of context-switching code

Next, we examine the effect of the context-switch code size on L1-I cache misses for STEPS, when keeping the breaking points in the source code the same. In our implementation, the CTX code size is 76 bytes. For this experiment, we pad the CTX function with a varying number of *nops*[1] to achieve CTX code sizes of up to 16KB. Figure 5-8 shows the total number of L1-I cache misses for ten threads executing index_fetch, for various sizes of the CTX code (128B to 16KB). The leftmost bar corresponds to our original STEPS implementation, while the rightmost bar corresponds to the default Shore configuration with no STEPS, under a warmed-up cache. This experiment shows that the CTX

1. A `nop` is a special assembly instruction used for padding cache blocks; it provides no other functionality.

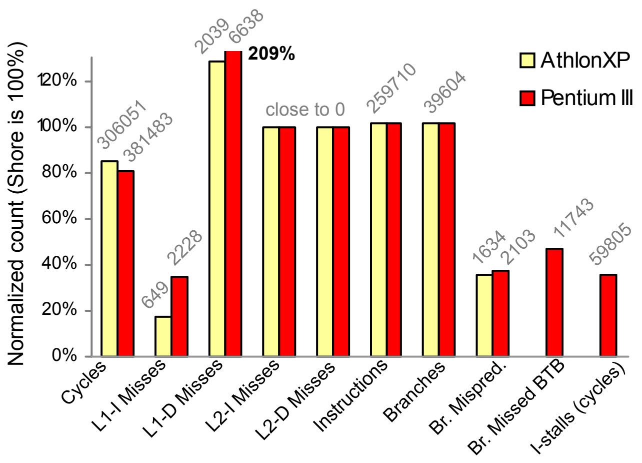


Figure 5-9. Relative performance of STEPS compared to Shore, for index fetch with 10 concurrent threads, on both the AthlonXP and the Pentium III. The two last bars are events exclusively available on the Pentium.

code could possibly include more functionality than the bare-minimum and still provide a significant reduction in the number of instruction cache misses. In our setup, a CTX function that is 25 times larger than the one we use, would result in almost the the same miss reduction (albeit with a lower overall speedup improvement). Having this flexibility in the footprint of the CTX code is important for portability of the STEPS implementation.

5.3.4.4 Detailed behavior on two different processors

The next experiment examines a wide range of changes in hardware behavior between STEPS and Shore for *index fetch* with 10 threads. We experiment with both the Athlon XP and the Pentium III, using the same code and a CTX function that updates all registers (float optimization is off). The Pentium III features a smaller, 16KB L1-I and L1-D cache (see also Table 5-1 for processor characteristics). Since the CTX points in Shore were chosen when running on the AthlonXP (64KB L1-I cache), we expect that this version of STEPS on the Pentium III will not be as effective in reducing L1-I cache misses as on the AthlonXP. The results are in Figure 5-9. Our observations for each event counted, in the order they appear in the graph, follow.

Execution time and L1-I cache misses. STEPS is also effective on the Pentium III despite its small cache, reducing L1-I cache misses to a third (66% out of a maximum possible 90% reduction). Moreover, the speedup on the Pentium is higher than the AthlonXP, mainly because the absolute number of misses saved is higher (absolute numbers for STEPS are on top of each bar in Figure 5-9). The last bar in Figure 5-9 shows the reduction in the cycles the processor is stalled due to lack of instructions in the cache (event only available on the Pentium III). The reduction percentage matches the L1-I cache miss reduction.

Level-one data cache. STEPS incurs significantly more L1-D cache misses on the Pentium's small L1-D cache (109% more misses). However, the CPU can cope well by overlapping misses and perform 24% faster.

Level-two cache. L2 cache performance does not have an effect on the specific microbenchmark since almost all data and instructions can be found there. We report L2 cache performance in the next section, when running a full OLTP workload.

Instructions and branches retired. As expected, STEPS executes slightly more instructions (1.7%) and branches (1.3%) due to the extra context-switch code.

Mispredicted branches. STEPS reduces mispredicted branches to almost a third on both CPUs (it eliminates 64% of Shore's mispredicted branches). This is an important result coming from STEPS' ability to provide the CPU with frequently repeating execution patterns. We verify this observation via an event available to Pentium III (second to last bar in Figure 5-9), that shows a reduction in the number of branches missing the Branch Target Buffer (BTB), a small cache for recently executed branches.

5.3.4.5 Varying L1-I cache characteristics

The last microbenchmark varies L1-I cache characteristics using SIMFLEX [Hardavellas et al. 2004], a Simics-based [Magnusson et al. 2002], full-system simulation framework developed at the Computer Architecture Lab of Carnegie Mellon. We use Simics/SIMFLEX to emulate a x86 processor (Pentium III) and associated peripheral devices (using the same setup as in the real Pentium). Simics boots and runs the exact same binary code of Linux and the Shore/STEPS microbenchmark, as in the real machines. Using SIMFLEX's cache component we modify the L1-I cache characteristics (size, associativity, block size) and run the 10-thread *index fetch* benchmark. The reported L1-I cache misses are exactly the same as in a real machine with the same cache characteristics. Metrics in simulation involving timing are subject to assumptions made by programmers and cannot possibly

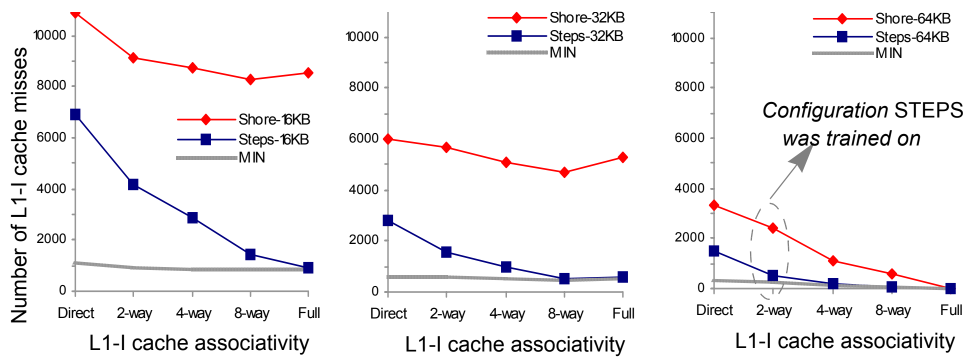


Figure 5-10. (a), (b), (c) Simulation results for index fetch with 10 threads. We use a L1-I cache with a 64-byte cache block, varying associativity (direct, 2-, 4-, 8-way, full) and size (16KB, 32KB, 64KB). STEPS eliminates all capacity misses and achieves up to 89% overall reduction (out of 90% max possible) in L1-I misses (max performance is for the 8-way 32KB and 64KB caches).

match real execution times. Figures 5-10 (a), (b), and (c) show the results for a fixed 64-byte cache block size, varying associativity for a 16KB, 32KB, and 64KB L1-I cache.

As expected, increasing the associativity reduces instruction conflict misses (except for a slight increase for fully-associative 16KB and 32KB caches, due to the LRU replacement policy resulting in more capacity misses). The conflict miss reduction for STEPS is more dramatic in a small cache (16KB). The reason is that with a 45KB working set for *index fetch* even a few CTX calls can eliminate all capacity misses for the small caches. Since STEPS is trained on a 2-way 64KB cache, smaller caches with the same associativity incur more conflict misses. As the associativity increases those additional L1-I misses disappear. Despite a fixed training on a large cache, STEPS performs very well across a wide range of cache architectures, achieving a 89% overall reduction in L1-I misses — out of 90% max possible — for the 8-way 32KB and 64KB caches. Experiments with different cache block sizes (not shown here) find that larger blocks further reduce L1-I misses, in agreement with the results in [Ranganathan et al. 1998].

5.4. Applying STEPS to OLTP workloads

So far we saw how to efficiently multiplex the execution of concurrent threads running the same transactional DBMS operation when (a) those threads run uninterrupted, and (b) the DBMS does not schedule any other threads. This section removes all previous assumptions and describes how STEPS works in full-system operation. The design goal is to take advantage of the fast CTX calls and maintain high concurrency for similarly structured operations in the presence of locking, latching (which provides exclusive access to DBMS structures), disk I/O, aborts and roll-backs, and other concurrent system operations (e.g., deadlock detection, buffer pool page flushing).

The rest of this section describes the full STEPS implementation (Section 5.4.1), presents the experimentation setup (5.4.2) and the TPC-C results (5.4.3).

5.4.1 Full STEPS implementation

STEPS employs a two-level transaction synchronization mechanism. At the higher level, all transactions about to perform a single DBMS operation form execution teams. We call S-threads all threads participating in an execution team (excluding system-specific threads or processes and threads which are blocked for any reason). Once all S-threads belong to a team, the CPU proceeds with the lower-level transaction synchronization scheme within a single team, following a similar execution schedule as in the previous section. Next, we detail synchronization mechanisms (Section 5.4.1.1), different code paths (Section 5.4.1.2), and threads leaving their teams (Section 5.4.1.3). Section 5.4.1.4 summarizes the changes to Shore code.

5.4.1.1 Forming and scheduling execution teams

To facilitate a flexible assignment of threads to execution teams and construct an efficient CPU schedule during the per-team synchronization phase, each DBMS operation is associated with a double-linked list (Figure 5-11). S-threads are part of such a list (depending on which operation they are currently executing), while all other threads have the `prev` and `next` pointers set to zero. The list for each execution team guides the CPU scheduling decisions. At each CTX point the CPU simply switches to the next thread in the list. S-threads may leave a team (disconnect) for several reasons. Transactions give up (yield) the CPU when they (a) block trying to acquire an exclusive lock (or access an exclusive resource), or on an I/O request, and, (b) when they voluntarily yield control as part of the code logic. We call *stray* the threads that leave a team.

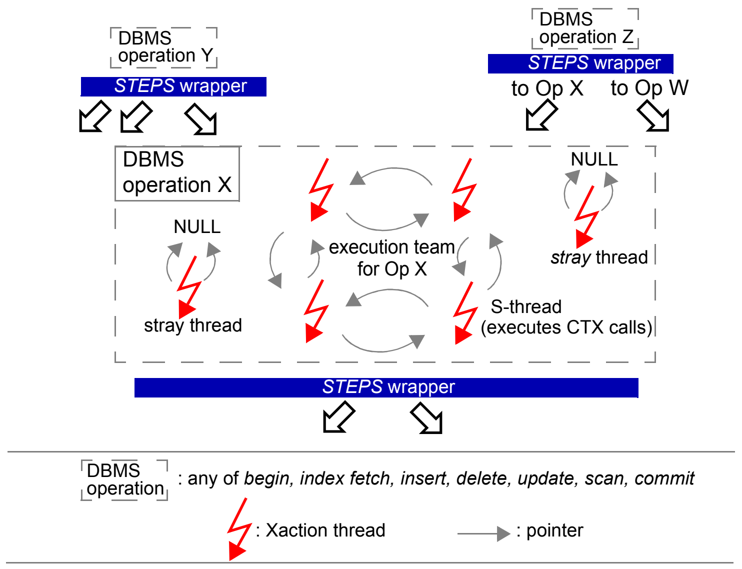


Figure 5-11. Additions to the DBMS code: Threads are associated with list nodes and form per-operation lists, during the STEPS setup code at the end of each DBMS operation.

The code responsible for team formation is a thin wrapper that runs every time a transaction finishes a single DBMS operation ("STEPS wrapper" in Figure 5-11). It disconnects the S-thread from the current list (if not stray) and connects it to the next list, according to the transaction code logic. If a list reaches the maximum number of threads allowed for a team (a user-defined variable), then the transaction will join a new team after the current team finishes execution. Before choosing the next team to run, all stray threads are given a chance to join their respective teams (next DBMS operation on their associated transaction's code logic). Finally, the STEPS wrapper updates internal statistics, checks with the system scheduler if other tasks need to run, and picks the next team to run[1].

Within each execution team STEPS works in a "best-effort" mode. Every time a transaction (or any thread) encounters a CTX point in the code, it first checks if it is an S-thread and then passes the CPU to the next thread in the list. All S-threads in the list eventually complete the current DBMS operation, executing in a round-robin fashion, the same way as in Section 3. This approach does not explicitly provide any guarantees that all threads will remain synchronized for the duration of the DBMS operation. It provides, however, a

1. Different per-team scheduling policies may apply at this point. In our experiments, picking the next operation that the last member of a list (or the last stray thread) is interested, worked well in practice since the system scheduler makes sure that every thread makes progress.

DBMS operation	cross-transaction code overlap		
	always	same tables	same tables + split Op
begin / commit	✓		
fetch		✓	
insert			✓
delete			✓
update	✓		
scan	✓		

Table 5-2. Operation classification for overlapped code

very fast context-switching mechanism during full-system operation (the same list-based mechanism was used in all microbenchmarks). If all threads execute the same code path without blocking, then STEPS will achieve the same L1-I cache miss reduction as in the previous section. Significantly different code paths across transactions executing the same operation or exceptional events that cause threads to become stray may lead to reduced benefits in the L1-I cache performance. Fortunately, we can reduce the effect of different code paths (Section 5.4.1.2) and exceptional events (5.4.1.3).

5.4.1.2 Maximizing code overlap across transactions

If an S-thread follows a significantly different code path than other threads in its team (e.g., traverse a B-tree with fewer levels), the assumed synchronization breaks down. That thread will keep evicting useful instructions with code that no one else needs. If a thread, however, exits the current operation prematurely (e.g., a key was not found), the only effect will be a reduced team size, since the thread will wait to join another team. To minimize the effect of different code paths we follow the next two guidelines: 1. Have a separate list for each operation that manipulates a different index (i.e., *index fetch (table1), index fetch (table2),* and so on). 2. If the workload does not yield high concurrency for similarly structured operations, we consider defining finer-grain operations. For example, instead of an *insert* operation, we maintain a different list for creating a record and a different one for updating an index.

Table 5-2 shows all transactional operations along with their degree of cross-transaction overlapped code. *Begin, commit, scan,* and *update* are independent of the database structure and use a single list each. *Index fetch* code follows different branches depending on the B-tree depth, therefore a separate list per index maximizes code overlap. Lastly, *insert* and *delete* code paths may differ across transactions even for same indices, therefore it may be necessary to define finer-grain operations. While experimenting with TPC-C we find that following only the first guideline (declaring lists per index) is sufficient. Small variations in the code path are unavoidable (e.g., utilizing a different attribute set or manipulating different strings) but the main function calls to the DBMS engine are generally the same across different transactions. For workloads with an excessive number of indices, we can use statistics collected by STEPS on the average execution team size per index, and consolidate teams from different indices. This way STEPS trades code overlap for an increased team size.

5.4.1.3 Dealing with stray transactions

S-threads turn into stray when they block or voluntarily yield the CPU. In *preemptive* thread packages the CPU scheduler may also preempt a thread after its time quantum has elapsed. The latter is a rare event for STEPS since it performs switches at orders of magnitude faster times than the quantum length. In our implementation on Shore we modify the thread package and intercept the entrance of `block` and `yield` to perform the following actions:

1. Disconnect the S-thread from the current list.

2. Turn the thread into stray, by setting pointers `prev` and `next` to zero. Stray threads bypass subsequent CTX calls and fall under the authority of the regular scheduler. They remain stray until they join the next list.

3. Update all thread package structures that were not updated during the fast CTX calls. In Shore these are the current running thread, and the ready queue status.

4. Pass a hint to the regular scheduler that the next thread to run should be the next in the current list (unless a system or a higher priority thread needs to run first).

5. Give up the CPU using regular context-switching.

Except for I/O requests and non-granted locks, transactions may go astray because of mutually exclusive code paths. Frequently, a database programmer protects accesses or modifications to a shared data structure by using a mutex (or a latch). If an S-thread calls CTX while still holding the mutex, all other threads in the same team will go astray as they will not be able to access the protected data. If the current operation's remaining code (after the mutex release) can still be shared, it may be preferable to skip the badly placed CTX call. This way STEPS only suffers momentarily the extra misses associated with executing a small, self-evicting piece of code.

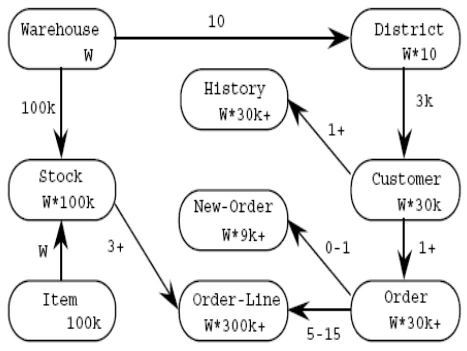
Erasing CTX calls is not a good idea since the specific CTX call may also be accessed from different code paths (for example, through other operations) which do not necessarily go through acquiring a mutex. *STEPS* associates with every thread a counter that increases every time the thread acquires a mutex and decreases when releasing it. Each CTX call tests if the counter is non-zero in which case it lets the current thread continue running without giving up the CPU. In Shore, there were only two places in the code that the counter would be non-zero.

5.4.1.4 Summary of changes to the DBMS code

The list of additions and modifications to the Shore code base is the following. We added the wrapper code to synchronize threads between calls to DBMS operations (STEPS wrapper, 150 lines of C++), the code to perform fast context-switching (20 lines of inline assembly), and we also added global variables for the list pointers representing each DBMS operation. We modified the thread package code to update the list nodes properly and thread status whenever blocking, yielding, or changing thread priorities (added/changed 140 lines of code). Finally, we inserted calls to our custom CTX function into the source code (as those were found during the microbenchmarking phase). Next, we describe the experimentation testbed.

5.4.2 Experimentation setup

We experiment with TPC-C, the most widely accepted transactional benchmark, which models a wholesale parts supplier operating out of a number of warehouses and their asso-



New Order	Payment
begin	begin
fetch (D) *lock*	fetch (D) *lock*
fetch (W)	fetch (W) *lock*
fetch (C)	scan (C) *60% prob.*
update (D)	fetch (C) *lock*
for (avg 10)	fetch (C) *lock, 10% prob.*
.....fetch (I)	update (C)
.....fetch (S) *lock*	update (D)
.....update (S)	update (W)
.....insert (O-L)	insert (H)
insert (O)	commit
insert (N-O)	
commit	

Figure 5-12. Database schema for TPC-C benchmark and code outline (in terms of system calls to the DBMS) of the two most frequently executed transactions, New Order and Payment (capital letters correspond to TPC-C tables, “fetch” and “scan” are implemented through SQL SELECT statements that retrieve a single or multiple records correspondingly.

ciated sales districts[1]. The benchmark is designed to represent any industry that must manage, sell, or distribute a product or service. It is designed to scale just as the supplier expands and new warehouses are created. The scaling requirement is that each warehouse must supply ten sales districts, and each district serves three thousand customers. The database schema along with the scaling requirements (as a function of the number of warehouses W) is shown in Figure 5-12 (left part).

The database size for one warehouse is 100MB (10 warehouses correspond to 1GB and so on). TPC-C involves a mix of five concurrent transactions of different types and complexity. These transactions include entering orders (the New Order transaction), recording payments (Payment), delivering orders, checking the status of orders, and monitoring the level of stock at the warehouses. The first two transactions are the most frequently executed (88% of any newly submitted transaction), and their code outline (in terms of calls to the DBMS in our implementation of TPC-C on top of Shore) is shown in Figure 5-12.

The TPC-C toolkit for Shore is written at CMU. Table 5-3 shows the basic configuration characteristics of our system. To ensure high concurrency and reduce the I/O bottleneck in

1. Transaction Processing Performance Council. `http://www.tpc.org`

CPU	AthlonXP, 2GB RAM, Linux 2.4.20
Storage	one 120GB main disk, one 30GB log disk
Buffer pool size	Up to 2GB
Page size	8192 Bytes
Shore locking hierarchy	Record, page, table, entire database
Shore locking protocol	Two phase locking

Table 5-3. System configuration

our two-disk system we cache the database in the buffer pool and allow transactions to commit without waiting for the log to be flushed on disk (the log is flushed asynchronously; this technique is also known as *lazy commit*). A reduced buffer pool size would cause I/O contention allowing only very few threads to be runnable at any time. High-end installations can hide the I/O latency by parallelizing requests on multiple disks. To mimic a high-end system's CPU utilization, we set user thinking time to zero and keep the standard TPC-C scaling factor (10 users per Warehouse), essentially having as many concurrent threads as the number of users. We found that, when comparing STEPS with Shore running New Order, STEPS was more efficient in inserting multiple subsequent records on behalf of a transaction (because of a slot allocation mechanism that was avoiding overheads when inserts were spread across many transactions). We modified slightly New Order by removing one insert from inside a for-loop (but kept the remaining inserts).

For all experiments we warm up the buffer pool and measure CPU events in full-system operation, including background I/O processes that are not optimized using STEPS. Measurement periods range from 10sec - 1min depending on the time needed to complete a pre-specified number of transactions. All reported numbers are consistent across different runs, since the aggregation period is large in terms of CPU time. Our primary metric is the number of L1-I cache misses as it is not affected by the AthlonXP's small L2 cache (when compared to server processors shown in Figure 5-2b).

STEPS setup: We keep the same CTX calls used in the microbenchmarks but without using floating point optimizations, and without re-training STEPS on TPC-C indexes or tables. Furthermore, we refrain from using STEPS on the TPC-C application code. Our

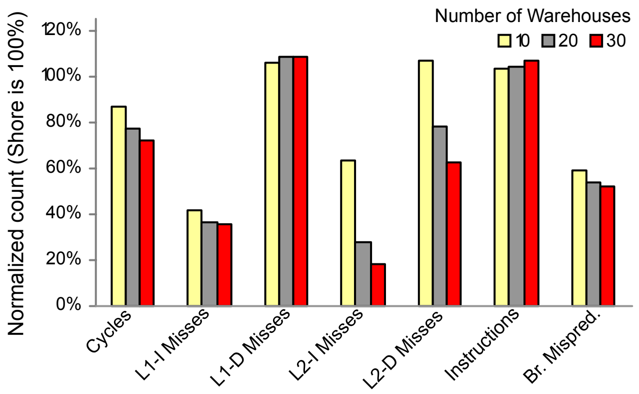


Figure 5-13. Transaction mix includes only the Payment transaction, for 10-30 Warehouses (100-300 threads).

goal is to show that STEPS is workload-independent and report lower bounds for performance metrics by not using optimized CTX calls. We assign a separate thread list to each *index fetch, insert,* and *delete* operating on different tables while keeping one list for each of the rest operations. Restricting execution team sizes has no effect since in our configuration the number of runnable threads is low. For larger setups, STEPS can be configured to restrict team sizes, essentially creating multiple independent teams per operation.

5.4.3 TPC-C results

Initially we run all TPC-C transaction types by themselves varying the database size (and number of users). Figure 5-13 shows the relative performance of STEPS over Shore when running the Payment transaction with standard TPC-C scaling for 10, 20, and 30 warehouses. The measured events are: execution time in CPU cycles, cache misses for both L1 and L2 caches, the number of instructions executed, and the number of mispredicted branches. Results for other transaction types were similar. STEPS outperforms Shore, achieving a 60-65% reduction in L1-I cache misses, a 41-45% reduction in mispredicted branches, and a 16-39% speedup (with no floating point optimizations). The benefits increase as the database size (and number of users) scale up. The increase in L1-D cache misses is marginal. STEPS speedup is also fueled by fewer L2-I and L2-D misses as the database size increases. STEPS makes better utilization of AMD's small L2 cache as fewer L1-I cache misses also translate into more usable space in L2 for data.

Warehouses →	10		20		30	
Operation (table)↓	in	out	in	out	in	out
index fetch (C)	8.6	8.6	16	16	25.2	24.7
index fetch (D)	8.9	1.7	16.2	2.6	31.7	5.3
index fetch (W)	8.9	0.5	16.6	1	30	1.9
scan (C)	9.4	8.2	16	14.3	26.2	23.7
insert (H)	7.9	7.8	14.9	14.6	24	23.2
update (C, D, W)	7.5	7.2	14	12.3	21.6	19
average team size	8.6	6.9	15.9	12.3	26.4	20.4
# of ready threads	15		28		48.4	

Table 5-4. Team sizes per DBMS operation in Payment

Table 5-4 shows for each configuration (10, 20, and 30 warehouses running Payment) how many threads on average enter an execution team for a DBMS operation and exit without being strays, along with how many threads are ready to run at any time and the average team size. The single capital letters in every operation correspond to the TPC-C tables/indices used (Customer, District, Warehouse, and History). STEPS is able to group on average half of the available threads. Most of the operations yield a low rate for producing strays, except for *index fetch* on District and Warehouse. In small TPC-C configurations, exclusive locks on those tables restrict concurrency.

Next, we run the standard TPC-C mix, excluding the non-interactive Delivery transaction (TPC-C specifies up to 80sec queueing delay before executing Delivery). Figure 5-14 shows that the four-transaction mix follows the general behavior of the Payment mix, with the reduction in instruction cache misses (both L1 and L2) being slightly worse. Statistics for the team sizes reveal that this configuration forces a smaller average team size due to the increased number of unique operations. For the 10-warehouse configuration, there are 14 ready threads, and on average, 4.3 threads exit from a list without being stray. Still, this means a theoretical bound of a 77% reduction in L1-I cache misses, and STEPS achieves a 56% reduction while handling a full TPC-C workload and without being optimized for it specifically. Results for different mixes of TPC-C transactions were similar.

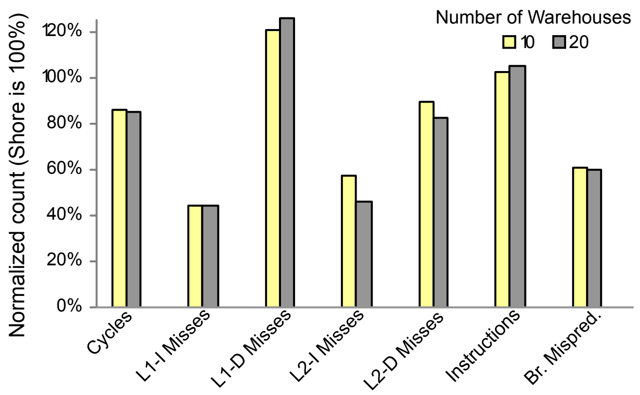


Figure 5-14. Transaction mix includes all transactions except the non-interactive Delivery transaction, for 10-20 Warehouses.

5.5. Applicability to commercial DBMS

STEPS has the following two attractive features that simplify integration in a commercial DBMS: (a) its application is incremental as it can target specific calls to the DBMS and also co-exist with other workloads which do not require a STEPS runtime, (e.g., decision support applications simply bypass all CTX calls), and (b) the required code modifications are restricted to a very specific small subset of the code, the thread package. Most commercial thread packages implement preemptive threads. As a result, DBMS code is *thread safe*: programmers develop DBMS code anticipating random context-switches that can occur at any time. Thread safe code ensures that any placement of CTX calls throughout the code will not break any assumptions.

To apply STEPS to a thread-based DBMS the programming team first needs to augment the thread package to support fast context-switching. Database software using processes instead of threads may require changes to a larger subset of the underlying OS code. For the rest of this section, we target DBMS software that assigns transactions to threads (and not processes). In general, a STEPS fast CTX call needs to bypass the operating system's scheduler and update only the absolute minimum state needed by a different thread to resume execution. Whenever a thread gives up CPU control through a mechanism different than fast CTX (e.g., disk I/O, unsuccessful lock requests, failure to enter a critical section, or expired time quantum), all state needed before invoking a regular context-switch needs to be updated accordingly. This is the state that the OS scheduler needs to make scheduling decisions. The next phase is to add a STEP wrapper in each major DBMS transactional operation. This thin wrapper will provide the high-level, per-operation transaction synchronization mechanism used in STEPS. The only workload-specific tuning required is the creation of per-index execution teams, which can be done once the database schema is known. The previous two sections described our approach into implementing the above mentioned guidelines in Shore running on Linux. The implementation does not depend on the core DBMS source code, since it only affects the thread package and the entry (or exit) points of a few high-level functions.

The final phase in applying STEPS is to decide how many fast CTX calls to insert in the DBMS code and where exactly to place them. So far, we identified candidate insertion points by “test-and-try.” We performed a manual search by executing DBMS operations on the target hardware, and using the CPU performance counters to count instruction cache misses. In our implementation we could afford the time to perform a manual search since Shore’s code is relatively small (around 60,000 lines of code). Commercial systems, however, may contain tens of millions of code lines. To aid STEPS deployment in large-scale DBMS software we need a tool that can automatically decide on where to insert CTX calls throughout the source code. Moreover, such a tool should be versatile enough to produce different outputs for different targeted cache architectures. This way, a DBMS binary with STEPS could ship optimized for a specific CPU.

In the remaining of this section we describe `autoSTEPS`, our approach towards automating the deployment of STEPS in commercial DBMS. Section 5.5.1 presents the functionality and usage of the tool, Section 5.5.2 describes the implementation, while Section 5.5.3 evaluates the accuracy of `autoSTEPS`.

5.5.1 `AutoSTEPS`: A tool to generate CTX insertion points

We leverage existing open-source software to obtain a trace of all instruction references during the execution of a transactional operation. `Valgrind`[1] / `cachegrind`[2] is a cache profiling tool which tracks all memory references (both data and instruction) from a

1. http://valgrind.org/
2. http://valgrind.org/docs/manual/cg_main.html

binary executable and passes them through a cache simulator to report statistics. We modify `cachegrind` to output all memory addresses of instructions executed between two "magic" instructions, placed anywhere in the DBMS source code (or user application code). `AutoSTEPS` is a cache-simulator script, written in Python, that takes as input the memory address trace and outputs the source code lines where fast CTX calls should be placed for STEPS-like execution. The tool usage is the following:

- First, the user specifies the DBMS operation for processing (such as index_fetch, insert, update, or any desired function call), by creating a sample client application which contains the operation to be profiled.
- If the targeted platform is the same as the one used to run the tool, no further action is needed (the tool automatically extracts cache parameters and outputs them for verification). Otherwise, the user needs to specify the cache characteristics (L1-I cache size, associativity, and block size) using a switch.
- The user first executes the modified version of `valgrind` to obtain the trace (i.e., "`>steps_valgrind sample_transaction.exe > trace`"), and then runs `autoSTEPS` on the collected trace.
- The tool outputs the memory addresses of the code that CTX calls need to be inserted, along with various cache statistics, both for plain and STEPS execution. Note that, depending on the underlying cache architecture, the tool may output different addresses for the CTX calls.

A specific DBMS operation can be profiled for STEPS by compiling a client application that issues sample transactions to a set of tables in the database. If the targeted operation is a function call exported by the DBMS to the client application, then the user inserts a "magic" instruction right before and right after the function call to the DBMS, and recompiles the client application. If the function call to be profiled is an internal one, then the magic instruction should be inserted at the DBMS source code. The magic instruction is a unique sequence of native assembly instructions with no effect, that differs on each platform and comes with the tool documentation; its purpose is to instruct the tool to start and stop processing the binary.

To translate the memory addresses to source code line numbers, `autoSTEPS` invokes the system debugger in batch mode and outputs source file names and line numbers. Once the lines in the source code are known, the CTX calls are inserted in place (the code of the CTX call is always the same). Then, the user re-compiles and runs the transactional application. Note that this procedure does not guarantee a deadlock-free execution since it may have introduced potential races. In our experiments with Shore, this was not the case, but it may occur in other systems. The `autoSTEPS` tool is an aiding tool in the process of instrumenting code, not a complete binary solution. We envision that a commercial application would also include two more tools. The first is a binary modification tool, similar to [Srivastava and Eustace 1994], to insert the CTX calls directly to the DBMS binary with no need for recompiling. The second tool is a race-detection binary tool, similar to [Savage et al. 1997], to pinpoint badly placed CTX calls which may cause races or force S-threads to go astray. Since similar tools already exist, it is out of the scope of this work to re-implement such functionality. Next, we describe the implementation of `autoSTEPS`.

5.5.2 `AutoSTEPS` algorithm

To profile code for STEPS we only need to examine L1-I cache traffic. We re-implement the cache simulator of `cachegrind`, to compare with STEPS-like execution. To find points in the code to place fast CTX calls, we need to consider cache misses only for *non-leading*[1] threads in an execution team. A regular cache simulator will count all misses for the single executing thread. Under STEPS, these are the default, compulsory misses that will always be caused by the leading thread. Since STEPS performance is governed by the misses caused by non-leading threads, we need to track which cache blocks are *evicted* during execution of a code segment by the leading thread. Those evicted cache blocks will need to be reloaded when non-leading threads execute the same code segment (immediately after the next CTX call). This way, we can decide on where to place CTX calls and also compute overall misses for all threads in the execution team, by processing a trace of *only one* thread executing the entire transactional operation.

1. A *leading* thread in an execution team is the first thread to execute (and therefore load in the cache) a new code segment, enclosed by two consecutive CTX calls. *Non-leading* threads are the rest n-1 threads in the execution team that will execute the same code segment and will incur significantly fewer misses.

Our algorithm works on top of the regular L1-I cache simulation. Starting from the beginning of the trace, the autoSTEPS simulator marks each cache block accessed with the number of the current code segment. These are the cache blocks loaded by the leading thread and, initially, the segment number is 0. The algorithm does not take into consideration regular misses. However, whenever a cache block with the same segment number as the currently executed segment is evicted, we count it as a *STEPS miss*. A STEPS miss corresponds to a miss that would have been caused by a non-leading thread. When the number of STEPS misses reaches a threshold, we mark the current memory address as the place where a CTX call will be inserted, increase the code segment number, reset the number of STEPS misses, and continue processing the trace.

To better match CTX call selection with code behavior, instead of using a fixed threshold for STEPS misses, we place CTX calls close to the beginning of a "knee" in the order that misses occur. Such a knee appears when (a) STEPS misses occur close to each other in terms of executed instructions (the number of in-between instructions is the width of the knee), and, (b) when the number of consecutive misses that are spaced out by fewer instructions than the knee width, reaches a predefined threshold (the "height" of a knee). Whenever two consecutive STEPS misses are separated by more than the knee width, we reset the knee height counter, up to a maximum of twice the knee height for STEPS misses between two consecutive CTX calls. The input to autoSTEPS is the width of the knee (as a number of instructions) and the height of the knee (as a number of STEPS misses). Upon detection of a knee, the CTX call is placed before the instruction that marks the beginning of the knee. For the code segment following the newly placed CTX call, we can afford more evictions before one results into a STEPS miss (since it is a new segment for all team threads), effectively "absorbing" the knee in the misses altogether.

To test autoSTEPS we collect a trace of instructions executed during index_fetch, using the same setup as in Section 5.3. We run autoSTEPS with a knee of width 200 instructions and height 4 misses and compare the number of misses for a single thread (non-leading thread when in STEPS mode) for the original code and STEPS execution. Figure 5-15 shows the cumulative number of misses as those add up for every executed instruction (the number of executed instructions at any time is in on the x-axis). The top line corresponds to the L1-I cache misses of the original code, when the cache is warmed up with

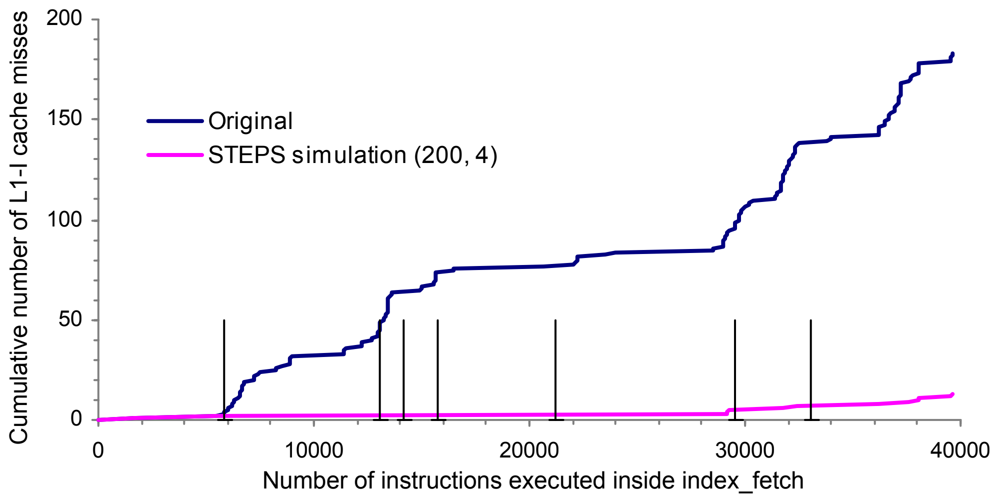


Figure 5-15. Cumulative number of instruction cache misses for single-thread execution of index_fetch for all executed instructions. “Original” corresponds to warm-cache behavior for any thread. The line for “STEPS simulation (200,4)” shows misses for any non-leading STEPS thread. The vertical lines show the points where a fast CTX takes place, when autoSTEPS is configured for a “knee” of 200 instructions width and height 4.

the exact same operation, as computed by the cache simulator of cachegrind. The bottom line corresponds to STEPS misses (misses caused by non-leading threads), while the vertical lines show where a CTX call takes place. In this run, autoSTEPS outputs 3 insertion points for CTX code which results into a total of 7 CTX calls. As Figure 5-15 shows, the resulting configuration keeps the overall number of misses very low with only 3 CTX insertion points in the source code. Note that during manual tuning, a total of 16 CTX insertion points was used.

To examine the effect of varying the knee input parameters to autoSTEPS, Figure 5-16 plots the number of L1-I cache misses (y-axis) against the number of CTX calls executed (x-axis) for various inputs. The observed trend is an expected one: as the height of the knee is reduced, autoSTEPS results into more recommendations for CTX calls which bring the overall number of misses lower. For the same height, a wider knee essentially allows more flexibility in defining a knee, absorbing more misses. Note that these are only trends and not rules that always hold, since autoSTEPS does not try to minimize the actual number of CTX calls executed. A CTX insertion point is picked according to the miss behavior up to that point, and not according to how many total calls it will generate. Fig-

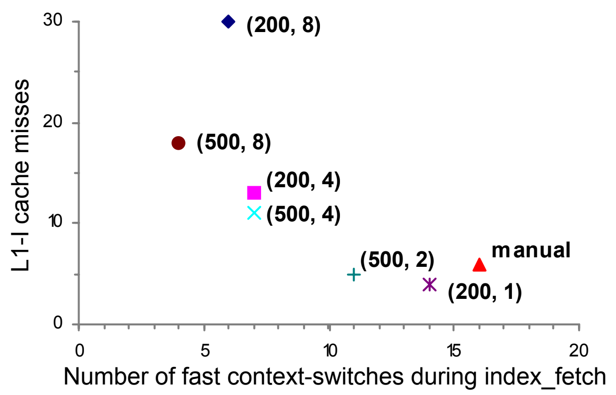


Figure 5-16. Number of L1-I cache misses and total number of context-switches executed for non-leading STEPS threads, for different runs of autoSTEPS using different input parameters. The "manual" point is produced by using the same CTX calls as in the manually tuned system.

ure 5-16 also shows the performance of the manually configured STEPS runtime used in the previous sections. The number of CTX calls in the manually tuned system exceed those resulting from autoSTEPS recommendations. Note that the cache simulation results into fewer misses for both the original and STEPS execution when compared to the simulation results of Section 5.3.4.5, because of the simplicity of the simulator (e.g., it does not simulate prefetching and assumes a perfect LRU replacement policy) and because of the fact we are assuming a perfectly warmed-up cache with no pollution from context-switching code.

5.5.3 Evaluation

To evaluate the effectiveness of the recommendations produced by autoSTEPS, we pick the same input parameters as in Figure 5-15 (knee width 200 and height 4). AutoSTEPS outputs the line number and source file of three places in Shore's code to insert fast CTX calls. After inserting the CTX calls and recompiling the code, we run the index_fetch microbenchmark with 10 threads on AthlonXP. We compare three systems: the original Shore, Shore with STEPS using the manual search for CTX insertion points (which resulted in a total of 16 insertions), and Shore with STEPS using the three insertion points recommended by autoSTEPS. We expect that the specific autoSTEPS configuration will not outperform the manual one in reducing the number of misses, as shown in the simulation results of Figure 5-16, but the reduced number of CTX calls should give a relative

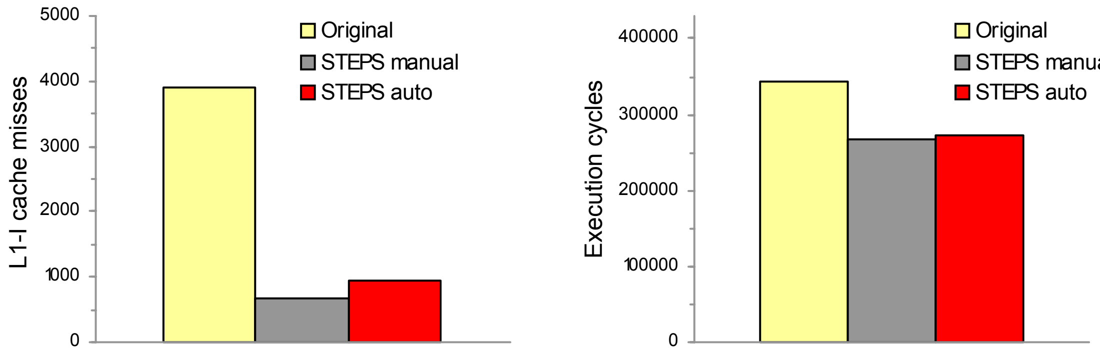


Figure 5-17. Performance comparison of Original, STEPS with manually placed CTX calls, and STEPS with an automatically produced configuration. We use autoSTEPS(200,4) to derive only 3 CTX calls (the manual configuration has 16 CTX calls).

speedup benefit to the autoSTEPS configuration. The results are in Figure 5-17. The left part of the figure shows the total number of L1-I cache misses for all 10 threads executing index_fetch, for the three systems. As expected, the autoSTEPS configuration significantly outperforms the original system and is slightly worse than the manual configuration. The right part of Figure 5-17 shows the total execution cycles. The autoSTEPS configuration performs almost as well as the manual configuration since it executes fewer context-switches.

5.6. Chapter summary

This chapter described STEPS, a transaction coordinating mechanism based on the StagedDB design, that addresses the instruction cache bottleneck in OLTP workloads. As recent studies have shown, instruction cache misses in transaction processing account for up to 40% of the execution time. Although compiling techniques and recently proposed architectural features can partially alleviate the problem, the database software design itself holds the key for eliminating cache misses by targeting directly the root of the problem. While database researchers have demonstrated the effectiveness of cache-conscious algorithms and data structures on data cache misses, instruction cache performance in transaction processing has yet to be addressed from within the software. The size of the code involved in transaction processing and the unpredictable nature of transaction execution make a software approach to eliminate instruction cache misses a challenging one.

STEPS is a mechanism that can apply with few code changes to any database software and shape the thread execution sequence to improve temporal locality in the instruction cache. Based on the grouped execution scheme of the StagedDB design, STEPS first forms teams of threads executing the same system component. It then multiplexes thread execution within a team, at such fine granularity that it enables reuse of instructions in the cache across threads. In this chapter we described the implementation of STEPS and showed how to automate the application of STEPS to any commercial DBMS.

To the best of our knowledge, STEPS is the first software approach to provide explicit thread scheduling for improving instruction cache performance. STEPS is orthogonal to compiler techniques and its benefits are always additional to any binary-optimized configuration. This chapter showed that STEPS minimizes both capacity and conflict instruction cache misses of OLTP with arbitrary long code paths, without increasing the size or the associativity of the instruction cache. The contributions of the chapter are:

- A novel technique that enables thread scheduling at very fine granularity to reuse instructions in the cache across concurrent threads.
- A tool to automatically find the boundaries in the code that the instruction cache fills up.
- The implementation and evaluation of the presented techniques inside a full-blown research prototype database system running a multi-user transactional benchmark on real hardware.

Chapter 6

New research opportunities and conclusions

The last chapter in this dissertation describes new research opportunities that become possible with the StagedDB design and presents the overall conclusions. Future research discussion is organized into two sections. First, we overview key performance characteristics of Chip Multiprocessors (CMP), which have already started replacing conventional (single core) processors in server computing. As of today, dual-core CMP systems are available and it is widely expected that over the next few years (2006 and onwards) chip designers will keep increasing the number of processor cores per chip. In Section 6.1.1, we argue that CMP systems underutilize their performance potential when running conventional DBMS software and propose mechanisms to adapt a Staged Database System for CMP environments and optimize performance. The second part of future research focuses on new opportunities for scheduling system resources in StagedDB. Section 6.1.2 describes how a Staged Database System can further improve performance by implementing new scheduling policies on stages involving disk requests and database lock requests.

6.1. Future work

6.1.1 High-performance database computing for CMPs

To exploit the increasingly large number of available transistors per chip, chip designers have introduced processors consisting of multiple cores on the same chip (Chip Multiprocessors, CMPs). Each core typically includes a private L1 cache and shares the on-chip L2

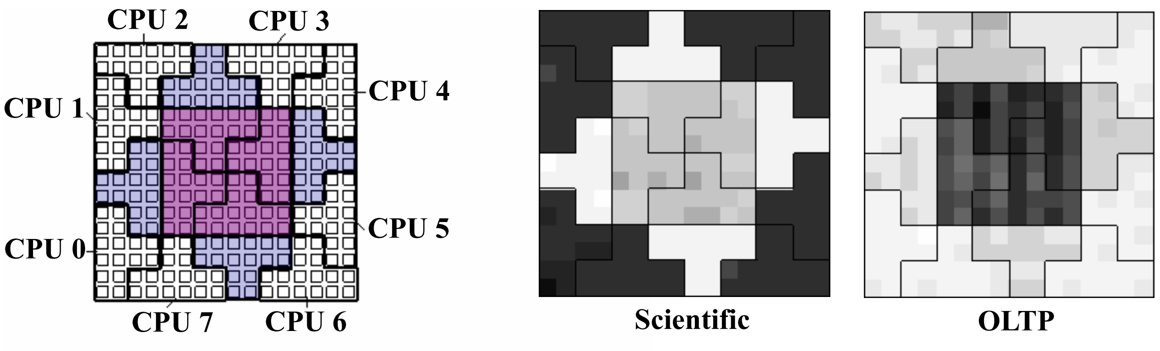


Figure 6-1. Non-uniform access shared L2 cache for 8-way CMP (left) and cache hit intensity for scientific and OLTP workloads (right).

cache with the rest of the cores. As of today, dual-core CMP systems are available and it is widely expected that over the next few years (2006 and onwards) chip designers will keep increasing the number of processor cores per chip. At the same time, computer architects expect that growing on-chip wire delays will force a re-design of on-chip large caches. Today's cache hierarchies consist of few discrete levels, each with a fixed access latency. With higher levels of chip integration, maintaining fixed access latency to the on-chip L2 cache will no longer be desirable, as areas that are physically closer to a core could be accessed in much shorter time. For example, for a 16MB on-chip L2 cache, built in a 50-nanometer process technology, it takes only 4 cycles to access the location closest to a core and 47 cycles to access the location that is the furthest to the core [Kim et al. 2002].

As a result, ongoing research in computer architecture is proposing cache architectures with non-uniform access times [Kim et al. 2002; Beckmann and Wood 2004]. An example of such a cache architecture is shown on the left side of Figure 6-1 [Beckmann and Wood 2004]. Eight CPU cores are placed around a large, shared L2 cache; cache blocks within the same colored-area have the same access latency while the overall access latency that a core sees, is a multiple of the colored areas that need to be crossed to retrieve a cache block. To minimize access times in shared CMP caches, computer architects propose cache block replication and migration mechanisms. Beckmann and Wood simulated a block migration policy using scientific and OLTP workloads on the cache architecture of Figure 6-1. The two rightmost pictures in Figure 6-1 show the overall cache utilization with darker areas corresponding to more heavily accessed areas. In the scientific workload, each core heavily accesses its private data set, and as a result, these blocks are gathered around each core, while shared data, which are accessed less frequently, are gathered in the center of the cache. The behavior of OLTP, however, is dramatically different. The majority of accesses is to shared data, and as a result of each core "pulling" these cache blocks, the center area of the cache becomes the most heavily used area.

With conventional DBMS designs there seems to be no way to provide low latency access to shared data since each core needs to access the center of the shared L2 cache. StagedDB, however, holds the potential of dramatically changing memory access patterns, essentially creating a picture similar to the one under the scientific workload in Figure 6-1. A suitable staging scheme can assign stages that access the same shared data (e.g., indexes, lock tables) to the same core and have transactions migrating through stages, depending on which indexes each transaction needs to access. This way, data shared across transactions become private to each core, while private transaction data will follow a migratory pattern. Since the majority of data cache accesses is to shared data, a staged transaction processing system can potentially behave similarly to the scientific workload shown in Figure 6-1.

In the above mentioned design, additional hardware mechanisms can be employed to facilitate the migration of transaction private data across cores. Since the application programmer can easily identify which data structures are private to each transaction, a hardware data streaming mechanism (similar to the one in [Wenisch et al. 2005]) can be used to hide access latency. Such a mechanism will automatically move data belonging to a pre-specified "stream" to the core that needs to access the stream, after the first access to the data stream occurs. The application programmer can provide hints to the hardware with respect to which data belong to the private environmental data of each transaction and therefore consist a stream.

6.1.2 Resource scheduling in StagedDB

The departure from a time-sharing thread-based execution model to stage-based database software introduces new scheduling opportunities. The CPU is no longer assigned directly to queries, rather it processes stage queues which include different queries at different phases of their execution. Depending on the nature of each stage there is a number of local optimizations to perform, utilizing information from the queries waiting in the queue. We focus on two classes of stages: (a) those responsible for issuing I/O requests, and, (b) stages that govern the access to lock tables. By giving the database programmer control over scheduling those two resources (disk and lock requests), it becomes easier to implement priority-based scheduling, where certain queries are given higher priority.

I/O stage queue management. Traditional database systems issue I/O requests at random points of time from the disks' point of view. That is, whenever the current execution thread incurs an I/O, the database system switches to a different thread without having any knowledge if and when the new thread will generate a disk request. In StagedDB, database tables are accessed through specialized stages. Each stage can examine its queue and determine the disk requests that will be generated. By presenting all disk requests together, the storage manager and the underlying disks can perform deeper and more effective scheduling. A future research direction is to leverage this effect and also further expose the I/O stages to the underlying storage architecture.

Lock stage queue management. Requests for locks share several similarities to disk requests. The current execution thread may issue a lock request at any time; if the request is not granted, the database system switches to a different thread. Typically there are no means to interrupt the current lock holder (same as when the disk head is about to serve a request). This fact deprives traditional database systems of the ability to control the execution sequence and optimize the system resource utilization. Consider, for example, two concurrent transactions that need to acquire the same exclusive lock. Assume that one transaction is short and about to commit, while the other one contains several more operations before committing. If the second transaction gets the lock first, the response time of the short transaction will be high. On the other hand, if both transactions were queued up in the same stage, we could potentially schedule the lock request form the short transaction first, thus reducing the average response time. A future research direction is to develop stages to control accesses to locks and implement stage queue scheduling policies to maximize resource utilization and minimize response times.

6.2. Dissertation conclusions

Recent research has shown that the performance of database management systems on modern hardware is tightly coupled to how efficiently the entire memory hierarchy, from disks to on-chip caches, is utilized. Unfortunately, according to recent studies, 50% to 80% of the execution time in database workloads is spent waiting for instructions or data. Although related work has identified memory-related bottlenecks and proposed techniques to boost performance, current DBMS designs do not have the means to exploit commonality across all levels of the memory hierarchy. Most of the research to date for improving locality examines data accessed and instructions executed by a single query (or transaction) at a time. Database systems, however, typically handle multiple concurrent users. Request concurrency adds a new dimension for addressing the locality optimization problem. Unfortunately, the operating system's context switching decisions are oblivious to the state of the request/thread and therefore cause severe context thrashing in the memory hierarchy.

This dissertation introduced a novel way to re-engineer database systems to improve utilization of all memory hierarchy levels. By properly synchronizing and multiplexing the concurrent execution of multiple requests there is a potential of increasing both data and instruction reusability at all levels of the memory hierarchy. Existing DBMS designs, however, pose difficulties in applying such execution optimizations as they abide by the execution primitives provided by the underlying operating system. Rather than rewriting the entire code of the database system, this dissertation provides the support to organize system components into self-contained stages and change request execution sequence to perform group-processing at each stage, thus effortlessly exploiting commonality across queries. The proposed design, StagedDB, requires only a small number of changes to the existing DBMS code base and provides a new set of execution primitives that allow software to gain increased control over what data is accessed, when, and by which requests.

This dissertation first introduced techniques to define and build individual stages out of a prototype DBMS along with query scheduling policies for staged architectures (Chapter 3). To prove feasibility of staged execution for database servers, we studied the performance tradeoff of delaying requests to execute them in groups, and found scheduling disciplines that outperform traditional architectures. We then evaluated the initial implementation of a staged database system on top of Predator, a research prototype DBMS, and described both performance and software engineering benefits of the case study.

In Chapter 4, we investigated how a staged execution engine can improve cross-query locality. We built and evaluated QPipe, a staged relational query engine that exploits overlapping work between concurrent queries at run time and offers several advantages over traditional query engine designs. It provides full intra-query parallelism, taking advantage of all available CPUs in multiprocessor servers for evaluating a single query, regardless of the plan's complexity. By processing a batch of tuples for each query at every staged relational operator, QPipe improves instruction temporal locality and avoids extraneous procedure calls in the DBMS code, when compared to tuple-by-tuple query engines. Furthermore, by applying on-demand simultaneous pipelining of common intermediate results across queries, QPipe avoids costly materializations and can efficiently evaluate plans produced by a multi-query optimizer.

Then, in Chapter 5, this dissertation investigated how the StagedDB design can improve instruction cache performance for large, commercial-grade systems processing multiple concurrent transactions. According to recent studies, instruction cache misses in transaction processing account for 25-40% of the execution time. While database researchers have demonstrated the effectiveness of cache-conscious algorithms and data structures on data cache misses, instruction cache performance in transaction processing has yet to be addressed from within the software. This dissertation introduced STEPS, a novel mechanism that can apply with few code changes to any database software and can shape the thread execution sequence to improve temporal locality in the instruction cache. To the best of our knowledge, STEPS is the first software approach to provide explicit thread scheduling for improving instruction cache performance. In chapter 5 we showed that STEPS minimizes both capacity and conflict instruction cache misses of OLTP with arbitrary long code paths, without increasing the size or the associativity of the instruction cache. The implementation and evaluation of the proposed techniques was done inside a full-blown research prototype database system running a multi-user transactional benchmark on real hardware.

The key results of this dissertation are summarized as follows:

1. *Database management systems can improve performance and software engineering characteristics by organizing system components into self-contained stages.*

2. *Relational query engines can improve performance (both throughput and single-query response time) by converting their design philosophy from query-centric (one-query, many-operators) to operator-centric (one-operator, many-queries).*

3. *The instruction cache bottleneck in transaction processing can be removed by forcing concurrent transactions to maximize reuse of instructions already present in the cache.*

Epilogue

Locality and predictability of different tasks running in a system has long been the key property that computer and storage architects, along with software designers have exploited to build high-performance computing systems. Different implementation iterations of caching and prefetching techniques both in hardware and software already span more than three decades. At this point of time, two trends in server computing have become (and are expected to remain) the norm: high-performance servers process multiple concurrent requests (varying from tens to thousands) and the utilization of the underlying hardware in its entirety (from CPUs to memory hierarchy and to disks) is a critical factor in deciding performance. This Ph.D. dissertation showed how to re-engineer server software to better match the underlying modern hardware, and how to exploit request concurrency to improve locality and predictability in the system, thereby improving performance. The focus has been on Database Management Systems, arguably the most challenging sever computing platform, but the lessons learned are general enough to apply to other server software as well.

Bibliography

AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. 2000. Automated selection of materialized views and indexes in sql databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496-505.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. 2001. Weaving relations for cache performance. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases.* Morgan Kaufmann Publisher. Inc., San Francisco, CA, USA, 169-180.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. Dbmss on a modern processor: Where does time go. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases.* Morgan Kaufmann Publisher. Inc., San Francisco, CA, USA, 266- 277.

ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles.* ACM Press, New York, NY, USA, 95-109.

ANNAVARAM, M., PATEL, J. M., AND DAVIDSON, E. S. 2003. Call graph prefetching for database applications. *ACM Trans. Comput. Syst. 21,* 4, 412-444.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: continuously adaptive query processing. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 261-272.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACTSIGART symposium on Principles of database systems.* ACM Press, New York, NY, USA, 1-16.

BANGA, G. AND MOGUL, J. C. 1998. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference.* New Orleans, LA.

BANSAL, N. AND HARCHOL-BALTER, M. 2001. Analysis of srpt scheduling: investigating unfairness. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems.* ACM Press, New York, NY, USA, 279- 290.

BARROSO, L. A., GHARACHORLOO, K., AND BUGNION, E. 1998. Memory system characterization of commercial workloads. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture.* IEEE Computer Society, Washington, DC, USA, 3-14.

BECKMANN, B. M. AND WOOD, D. A. 2004. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual international symposium on Microarchitecture.* IEEE Computer Society, Washington, DC, USA, 319-330.

BERNSTEIN, P., BRODIE, M., CERI, S., DEWITT, D., FRANKLIN, M., GARCIA-MOLINA, H., GRAY, J., HELD, J., HELLERSTEIN, J., JAGADISH, H. V., LESK, M., MAIER, D., NAUGHTON, J., PIRAHESH, H., STONEBRAKER, M., AND ULLMAN, J. 1998. The asilomar report on database research. *SIGMOD Rec. 27,* 4, 74-80.

BLAKELEY, J. A., LARSON, P.-A., AND TOMPA, F. W. 1986. Efficiently updating materialized views. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 61-71.

BROWNE, S., DEANE, C., HO, G., AND MUCCI, P. 1999. Papi: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference.*

CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. 1994. Shoring up persistent applications. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 383-394.

CAREY, M. J. AND HAAS, L. M. 1990. Extensible database management systems. *SIGMOD Record 19,* 4, 54-60.

CHANDRASEKARAN, S. AND FRANKLIN, M. J. 2003. Psoup: a system for streaming queries over streaming data. *The VLDB Journal 12,* 2, 140-156.

CHAUDHURI, S. AND WEIKUM, G. 2000. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1-10.

CHEKURI, C., HASAN, W., AND MOTWANI, R. 1995. Scheduling problems in parallel query optimization. In *PODS '95: Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM Press, New York, NY, USA, 255-265.

CHEN, I.-C. K., LEE, C.-C., AND MUDGE, T. N. 1997. Instruction prefetching using branch prediction information. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97).* IEEE Computer Society, Washington, DC, USA, 593.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 379-390.

CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. 2001. Improving index performance through prefetching. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 235-246.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 2000. Making pointer-based data structures cache conscious. *Computer 33,* 12, 67-74.

CHOU, H.-T. AND DEWITT, D. J. 1985. An evaluation of buffer management strategies for relational database systems. In *VLDB.* 127-141.

CROVELLA, M. E., HARCHOL-BALTER, M., AND MURTA, C. D. 1998. Task assignment in a distributed system (extended abstract): improving performance by unbalancing load. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems.* ACM Press, New York, NY, USA, 268-269.

DALVI, N. N., SANGHAI, S. K., ROY, P., AND SUDARSHAN, S. 2001. Pipelining in multi-query optimization. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* ACM Press, New York, NY, USA, 59-70.

DAR, S, FRANKLIN, M. J., JONSSON, B. T., SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 330-341.

DEWITT, D. AND GRAY, J. 1992. Parallel database systems: the future of high performance database systems. *Commun. ACM 35,* 6, 85-98.

DEWITT, D. J. 1993. The wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition),* J. Gray, Ed. Morgan Kaufmann.

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D. A., BRICKER, A., HSIAO, H. I., AND RASMUSSEN, R. 1990. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2,* 1, 44-62.

FERNANDEZ, P. M. 1994. Red brick warehouse: a read-mostly rdbms for open smp platforms. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 492.

FINKELSTEIN, S. 1982. Common expression analysis in database applications. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 235-245.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys, 25(2),* 73–170.

GRAEFE, G. 1994. Volcano: an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering 6,* 1, 120-135.

GRAEFE, G. 1996. Iterators, schedulers, and distributed-memory parallelism. *Softw. Pract. Exper. 26,* 4, 427-452.

GRAEFE, G. AND LARSON, P. 2001. B-tree indexes and cpu caches. In *Proceedings of the 17th International Conference on Data Engineering.* IEEE Computer Society, Washington, DC, USA, 349-358.

GRAY, J. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems.* Morgan Kaufmann Publisher. Inc., San Francisco, CA, USA.

GRAY, J. 2004. The next database revolution. In *SIGMOD Conference.* 1-4.

HARDAVELLAS, N., SOMOGYI, S., WENISCH, T. F., WUNDERLICH, R. E., CHEN, S., KIM, J., FALSAFI, B., HOE, J. C., AND NOWATZYK, A. G. 2004. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev. 31,* 4, 31-34.

HARIZOPOULOS, S. AND AILAMAKI, A. 2002. Affinity scheduling in staged server architectures. *Carnegie Mellon University Technical Report* CMU-CS-02-113.

HARIZOPOULOS, S. AND AILAMAKI, A. 2003. A case for staged database systems. In *CIDR '03: Proceedings of the First International Conference on Innovative Data Systems Research.*

HARIZOPOULOS, S. AND AILAMAKI, A. 2004. Steps towards cache-resident transaction processing. In *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases.* Morgan Kaufmann, 660-671.

HARIZOPOULOS, S. AND AILAMAKI, A. 2005. Stageddb: designing database servers for modern hardware. In *IEEE Data Eng. Bull. 28,* 2, 11-16.

HARIZOPOULOS, S., SHKAPENYUK, V., AND AILAMAKI, A. 2005. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD '05: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data.* ACM Press, New York, NY, USA, 383-394.

HENNESSY, J. L. AND PATTERSON, D. A. 1996 . *Computer architecture (2nd ed.): a quantitative approach.* Morgan Kaufmann Publisher. Inc., San Francisco, CA, USA.

JAYASIMHA, J. AND KUMAR, A. 1999. Thread-based cache analysis of a modified tpc-c workload. In *Second Workshop on Computer Architecture Evaluation Using Commercial Workloads.*

JOHNSON, T. AND SHASHA, D. 1994. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439-450.

KABRA, N. AND DEWITT, D. J. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 106-117.

KEETON, K., PATTERSON, D. A., HE, Y. Q., RAPHAEL, R. C., AND BAKER, W. E. 1998. Performance characterization of a quad pentium pro smp using oltp workloads. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture.* IEEE Computer Society, Washington, DC, USA, 15-26.

KIM, C., BURGER, D., AND KECKLER, S. W. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems.* ACM Press, New York, NY, USA, 211-222.

LARUS, J. R. AND PARKES, M. 2002. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference.* USENIX Association, Berkeley, CA, USA, 103-114.

LO, J. L., BARROSO, L. A., EGGERS, S. J., GHARACHORLOO, K., LEVY, H. M., AND PAREKH, S. S. 1998. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture.* IEEE Computer Society, Washington, DC, USA, 39-50.

MADDEN, S., SHAH, M., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 49-60.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HLLBERG, G., HGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Computer 35,* 2, 50-58.

MAYNARD, A. M. G., DONNELLY, C. M., AND OLSZEWSKI, B. R. 1994. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems.* ACM Press, New York, NY, USA, 145-156.

MEGIDDO, N. AND MODHA, D. S. 2003. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies.*

O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 297-306.

PADMANABHAN, S., MALKEMUS, T., AGARWAL, R. C., AND JHINGRAN, A. 2001. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering.* IEEE Computer Society, 567-574.

PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference.*

RAMIREZ, A., BARROSO, L. A., GHARACHORLOO, K., COHN, R., LARRIBA-PEY, J., LOWNEY, P. G., AND VALERO, M. 2001. Code layout optimizations for transaction processing workloads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture.* ACM Press, New York, NY, USA, 155-164.

RANGANATHAN, P., GHARACHORLOO, K., ADVE, S. V., AND BARROSO, L. A. 1998. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems.* ACM Press, New York, NY, USA, 307-318.

ROMER, T., VOELKER, G., LEE, D., WOLMAN, A., WONG, W., LEVY, H., BERSHAD, B., AND CHEN, B. 1997. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop.*

ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. The impact of architectural trends on operating system performance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles.* ACM Press, New York, NY, USA, 285-298.

ROUSSOPOULOS, N. 1982. View indexing in relational databases. *ACM Trans. Database Syst. 7,* 2, 258-290.

ROY, P., SESHADRI, S., SUDARSHAN, S., AND BHOBE, S. 2000. Efficient and extensible algorithms for multi query optimization. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 249-260.

SACCO, G. M. AND SCHKOLNICK, M. 1986. Buffer management in relational database systems. *ACM Trans. Database Syst. 11,* 4, 473-498.

SARDA, P. AND HARITSA, J. R. 2004. Green query optimization: Taming query optimization overheads through plan recycling. In *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases.* Morgan Kaufmann, 1333-1336.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15,* 4, 391-411.

SELLIS, T. K. 1988. Multiple-query optimization. *ACM Trans. Database Syst. 13,* 1, 23-52.

SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. 1997. The case for enhanced abstract data types. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 66-75.

SHAO, M. AND AILAMAKI, A. 2004. Dbmbench: Fast and accurate database workload representation on modern microarchitecture. *Carnegie Mellon University Technical Report* CMU-CS03-161.

SHAO, M., SCHINDLER, J., SCHLOSSER, S. W., AILAMAKI, A., AND GANGER, G. R. 2004. Clotho: decoupling memory page layout from storage organization. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 696-707.

SHATDAL, A., KANT, C., AND NAUGHTON, J. F. 1994. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 510-521.

SHIM, J., SCHEUERMANN, P., AND VINGRALEK, R. 1999. Dynamic caching of query results for decision support systems. In *SSDBM '99: Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management.* IEEE Computer Society, Washington, DC, USA, 254.

SHKAPENYUK, V., WILLIAMS, R., HARIZOPOULOS, S., AND AILAMAKI, A. 2005. Deadlock resolution in pipelined query graphs. *Carnegie Mellon University Technical Report* CMU-CS-05-122.

SILBERSCHATZ, A. AND ZDONIK, S. 1996. Strategic directions in database systems: breaking out of the box. *ACM Comput. Surv. 28,* 4, 764-778.

SQUIILLANTE, M. S. AND LAZOWSKA, E. D. 1993. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst. 4,* 2, 131-143.

SRIVASTAVA, A. AND EUSTACE, A. 1994. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation.* ACM Press, New York, NY, USA, 196-205.

STETS, R., GHARACHORLOO, K., AND BARROSO, L. A. 2002. A detailed comparison of two transaction processing workloads. In *WWC-5: IEEE 5th Annual Workshop on Workload Characterization.*

STONEBRAKER, M., HELD, G., WONG, E., AND KREPS, P. 1976. The design and implementation of ingres. *ACM Trans. Database Syst. 1,* 3, 189-222.

SUBRAMANIAM, S. AND EAGER, D. L. 1994. Affinity scheduling of unbalanced workloads. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing.* ACM Press, New York, NY, USA, 214-226.

TORRELLAS, J., XIA, C., AND DAIGLE, R. L. 1998. Optimizing the instruction cache performance of the operating system. *IEEE Trans. Comput. 47,* 12, 1363-1381.

URHAN, T. AND FRANKLIN, M. J. 2001. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 501-510.

WELSH, M., CULLER, D., AND BREWER, E. 2001. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles.* ACM Press, New York, NY, USA, 230-243.

WENISCH, T. F., SOMOGYI, S., HARDAVELLAS, N., KIM, J., AILAMAKI, A., AND FALSAFI, B. 2005 Temporal streaming of shared memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer architecture.* ACM Press, New York, NY, USA, 222-233.

WIEDERHOLD, G. 1992. Mediators in the architecture of future information systems. *Computer 25,* 3, 38-49.

ZHOU, J. AND ROSS, K. A. 2004. Buffering database operations for enhanced instruction cache performance. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* ACM Press, New York, NY, USA, 191-202.

APPENDIX

Analysis of scheduling policies

8.1. Analysis of PS and FCFS

Under both FCFS and PS each query sees all modules as if they were one M/G/1 server, with mean service time the mean module service time plus the module load time. That is, each query has to serve for *l* time units plus a variable amount of time drawn from an exponential distribution with mean *m*. For PS, the mean response time (expected time in system) in a M/G/1 server, is:

$$E[T_s] = \frac{1}{\frac{1}{m+l} - \lambda} \quad \textit{(PS)}$$

For FCFS, the Pollaczek - Khinchin formula applies to the expected time in system for a M/G/1 server:

$$E[T_Q] = \frac{\rho}{1-\rho} \cdot \frac{E[S^2]}{2E[S]} \quad \textit{(P-K)}$$

This formula needs the first 2 moments of the general query size distribution (Exp(*m*) + *l*):

$$E[S] = \int_l^{\infty} xf(x)dx = l + m \quad \textit{(1)}, \quad E[S^2] = \int_l^{\infty} x^2 f(x)dx = l^2 + 2lm + 2m^2 \quad \textit{(1)}$$

So, the expected response time for a query for the given problem definition under FCFS is:

$$E[T_s] = \frac{\lambda(l^2 + 2lm + 2m^2)}{2(1-\rho)} + l + m \quad \textit{(FCFS)}$$

8.2. Analysis of M/M/1 with a locality-aware policy

This section considers a special case of the M/M/1 queue where the notion of data locality is incorporated into the job service times. While in CPU, we assume that each job undergoes a series of execution steps (modules). At each of those modules the service time is affected by the cache hit ratio. Whenever a job moves to the next execution module, new data structures and new instructions need to be loaded in the cache. However, if a second job is preemptively scheduled to execute the very same module that the previous job was working on, then, the cache hit ratio increases, and thus, the service time for the second job is reduced.

Although the analysis described here is in the context of the staged server paradigm, it can also apply to a wider class of servers that follow a “production-line” model of operation. For instance, we can imagine a single robotic arm (could be part of a chain) performing several tasks on incoming items. Suppose some or all of those tasks require each time a special, time-consuming preparation on behalf of the robotic arm (such as switching position or functionality). Then, it could be more efficient if the robotic arm treated incoming items as a batch and performed each task on the whole batch (thus paying the penalty of preparation only once per incoming batch). In that system, the notion of the cache and common data structures is replaced by the preparation of the robotic arm that is common to each task.

In order to model this queue, we assume that all jobs pass through the same execution modules. The breakdown of the service requirement into modules is such that the common data associated with each module fit entirely in the cache (or in the higher levels of the memory hierarchy, in general). The total service time for a job that always suffers the default cache miss ratio (as in the FCFS case) is drawn from an exponential distribution with mean $1/\mu$. A job that always executes a module just after another job has brought the common data of that module in the cache, requires a total service time drawn from an exponential distribution with mean c/μ, with $0 < c \leq 1$.

All existing scheduling policies (preemptive and non-preemptive) for the M/M/1 queue are oblivious to cache performance. For exponential service times all non size-based policies result in the same expected time in system, namely:

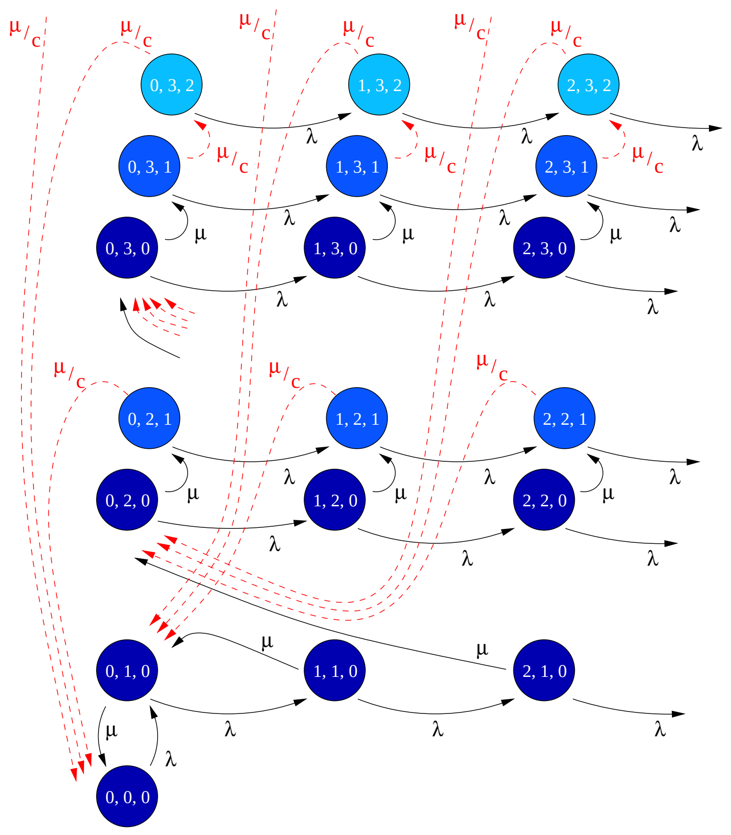


Figure 8-1. Markov chain with states: (jobs in queue, size of admitted batch, jobs completed within batch).

$$E[T_s] = \frac{1}{\mu - \lambda},$$

where λ is the job arrival rate. The rest of this Section analyses d-gated for the M/M/1 queue just described.

An equivalent way to think of the proposed cache-conscious execution is as a modified FCFS policy. Under this policy the first job in the queue (and in a group) gets uninterrupted service with rate μ and then waits until all jobs in its group finish. Each of those jobs in turn, executes in a FCFS fashion, but with an "accelerated" rate of μ/c, and then waits for the rest (note that ***c*** was defined to be between zero and one). When the last job of the group finishes execution, all jobs leave the system at the same time and the CPU looks at the queue: the jobs that wait there will consist the next group (if there are no jobs in the queue, then the first job to arrive in the system will consist a group by itself, in which case the cache-conscious execution reduces to a true FCFS policy). This modified

FCFS scheme behaves the same as the proposed cache-conscious execution in terms of the number of jobs in the system at any time.

Since both the interarrival and service times are exponential, a Markov chain can be written for this modified FCFS scheme (illustrated in Figure 8-1). A state (*m, n, z*) in this chain is defined as: *m* is the number of jobs waiting in the queue (not admitted), *n* is the number of jobs consisting the current batch, and, *z* is the number of jobs, out of the current batch, that would have finished execution under normal FCFS but now have to wait for all members of the batch to complete execution. Every batch comes to completion whenever there is a departure with rate μ (for batch size = 1) or rate μ/c (for batch size > 1), from a state in the form of: (*m, n, n-1)*; this departure leads to state (*0, m, 0*), since all *m* jobs waiting in the queue will consist the next batch.

Solution of Markov chain. We can write a balance equation for the state (*m, n, z*): the rate we leave that state equals the rate at which we enter into that state.

$$\left(\frac{\mu}{c}+\lambda\right)P_{m,n,z} = \frac{\mu}{c}P_{m,n,z-1}+\lambda P_{m-1,n,z}\ , for\ 1<z<n \quad (1)$$

For $z = 0$, the balance equation is:

$$(\mu+\lambda)P_{m,n,0} = \lambda P_{m-1,n,0} \rightarrow P_{m,n,0} = \left(\frac{\lambda}{\mu+\lambda}\right)^{m} P_{0,n,0}\ , \quad (2)$$

Convention: from now on we will write all probabilities of the form $P_{0,n,0}$*, as* P_n*..The goal is to express all state probabilities as functions of* P_n*.*

For $z = 1$, the balance equation is:

$$\left(\frac{\mu}{c}+\lambda\right)P_{m,n,1} = \mu P_{m,n,0}+\lambda P_{m-1,n,1}\ , \quad (3)$$

By substituting (2) into (3) and solving the recurrence, we derive:

$$P_{m,n,1} = \frac{\mu}{\mu/c+\lambda}\cdot\left(\sum_{i=0}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i}\right)\cdot P_n\ , \quad (4)$$

For *z = 2*, (1) and (4) give, after solving for the recurrence:

$$P_{m,n,2} = \frac{\mu/c}{\mu/c+\lambda}\cdot\frac{\mu}{\mu/c+\lambda}\cdot\left(\sum_{i=0}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i}+\ldots+\sum_{i=m}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i}\right)\cdot P_n\ , \quad (5)$$

Similarly, for *z = 3*, we have:

$$P_{m,n,3} = \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{2}\cdot\frac{\mu}{\mu/c+\lambda}\cdot\left(\sum_{i_3=0}^{m}\sum_{i_2=i_3}^{m}\sum_{i_1=i_2}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1}\right)\cdot P_n\ , \quad (6)$$

From (2), (4), (5), and (6) we can conclude what the formula for $P_{m,n,z}$ as a function of P_n is.

$$P_{m,n,z} = \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{z-1}\cdot\frac{\mu}{\mu/c+\lambda}\cdot\left(\sum_{i_z=0}^{m}\sum_{i_{z-1}=i_z}^{m}\cdots\sum_{i_1=i_2}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1}\right)\cdot P_n \qquad 0\le m,\ 0<z<n$$

$$P_{m,n,0} = \left(\frac{\lambda}{\mu+\lambda}\right)^{m}P_n \qquad 0\le m,\ 0<n \qquad , \quad (7)$$

Convention: from now on, a[m, z] will be the following sum

$$a[m,z] = \sum_{i_z=0}^{m}\sum_{i_{z-1}=i_z}^{m}\cdots\sum_{i_1=i_2}^{m}\left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1}\ , \quad (8)$$

a[m, z] can be thought as a sequence for *z > 0* with *m* being a non negative integer, and the following recursive definition can be written:

$$a[m,z] = \sum_{k=0}^{m}\left[\left(\frac{\lambda}{\mu/c+\lambda}\right)^{k}\cdot a[m-k,z-1]\right] \qquad 0\le m,\ 1<z$$

$$a[m,1] = \sum_{k=0}^{m}\left[\left(\frac{\lambda}{\mu/c+\lambda}\right)^{k}\cdot\left(\frac{\lambda}{\mu+\lambda}\right)^{m-k}\right] \qquad 0\le m \qquad , \quad (9)$$

$$a[m,0] = \left(\frac{\lambda}{\mu+\lambda}\right)^{m} \qquad 0\le m$$

This will be useful when we want to evaluate those probabilities. We now need to compute probabilities P_n. For *n > 1*, we have the following balance equation:

$$(\mu+\lambda)\cdot P_n = \mu P_{n,1,0} + \sum_{i=2}^{\infty} \frac{\mu}{c} P_{n,i,i-1} \ ,$$

and by using (7) and (9), we can rewrite that as:

$$P_n = \frac{\mu\cdot\lambda^n}{(\mu+\lambda)^{n+1}}\cdot P_1 + \frac{\mu}{\mu+\lambda}\cdot\sum_{i=2}^{\infty}\left[a[n,i-1]\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{i-1}\cdot P_i\right] \qquad 2\le n \qquad ,\ \textit{(10)}$$

For *n = 1*, we can write the following balance equation:

$$(\mu+\lambda)\cdot P_1 = \lambda P_0 + \mu P_{1,1,0} + \sum_{i=2}^{\infty}\frac{\mu}{c}P_{1,i,i-1} \ ,$$

which can be written as:

$$P_1 = \frac{\lambda}{\mu+\lambda}\cdot P_0 + \frac{\mu\cdot\lambda}{(\mu+\lambda)^2}\cdot P_1 + \frac{\mu}{\mu+\lambda}\cdot\sum_{i=2}^{\infty}\left[a[1,i-1]\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{i-1}\cdot P_i\right] \ ,\ \textit{(11)}$$

The probability of being in state 0 (system being idle) equals 1 minus the sum of the probabilities of being in every other possible state, namely:

$$P_0 = 1 - \sum_{m=0}^{\infty}\sum_{n=1}^{\infty}\sum_{z=0}^{n-1} P_{m,n,z} \ ,$$

or:

$$P_0 = 1 - \sum_{n=1}^{\infty}\sum_{m=0}^{\infty}\sum_{z=0}^{n-1}\left[\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{z-1}\cdot\frac{\mu}{\mu/c+\lambda}\cdot a[m,z]\cdot P_n\right] \ ,\ \textit{(12)}$$

From equations (10), (11), and (12) we can see that the probabilities P_1 through P_∞ are part of a set of an infinite number of linear equations, and their value could had been obtained if we were able to solve this linear system of infinite equations. Such a solution would have the following general representation:

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ \dots \\ P_n \\ \dots \end{bmatrix} = \begin{bmatrix} \frac{\mu \cdot \lambda}{(\mu+\lambda)^2} - 1 - S(1) & \frac{\mu}{\mu+\lambda} \cdot \frac{\mu/c}{\mu/c+\lambda} \cdot a[1,1] - S(2) & \dots & \frac{\mu}{\mu+\lambda} \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1} \cdot a[1,n-1] - S(n) & \dots \\ \frac{\mu \cdot \lambda^2}{(\mu+\lambda)^3} & \frac{\mu}{\mu+\lambda} \cdot \frac{\mu/c}{\mu/c+\lambda} \cdot a[2,1] - 1 & \dots & \frac{\mu}{\mu+\lambda} \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1} \cdot a[2,n-1] & \dots \\ \frac{\mu \cdot \lambda^3}{(\mu+\lambda)^4} & \frac{\mu}{\mu+\lambda} \cdot \frac{\mu/c}{\mu/c+\lambda} \cdot a[3,1] & \dots & \frac{\mu}{\mu+\lambda} \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1} \cdot a[3,n-1] & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \frac{\mu \cdot \lambda^n}{(\mu+\lambda)^{n+1}} & \frac{\mu}{\mu+\lambda} \cdot \frac{\mu/c}{\mu/c+\lambda} \cdot a[n,1] & \dots & \frac{\mu}{\mu+\lambda} \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1} \cdot a[n,n-1] - 1 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}^{-1} \cdot \begin{bmatrix} -\frac{\lambda}{\mu+\lambda} \\ 0 \\ 0 \\ \dots \\ 0 \\ \dots \end{bmatrix}$$

where the quantity *S(n)* is defined as:

$$S(n) = \sum_{m=0}^{\infty} \sum_{z=0}^{n-1} \left[\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{z-1} \cdot \frac{\mu}{\mu/c+\lambda} \cdot a[m,z] \right]$$

Evaluation. The above solution can be approximated by solving for a finite *n*. Since the last job on every batch sees the system as if it was a simple FCFS server, the system is stable and the sum of the state probabilities goes to 1 as *n* goes to infinity. We used dynamic programming to efficiently evaluate *a[m,z]*, and matlab to solve the linear equations, for *n* in the range of 100-500. The results matched exactly the simulation scripts, for all combinations of c, μ, λ we tried.