StagedDB: Designing Database Servers for Modern Hardware

Stavros Harizopoulos, Anastassia Ailamaki
Carnegie Mellon University
{stavros,natassa}@cs.cmu.edu

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 the CPU 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.

In this article we describe StagedDB, a new DBMS 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, thus effortlessly exploiting data and work commonality. We present two systems based on the StagedDB design. STEPS, a transaction coordinating mechanism demonstrated on top of Shore, minimizes instruction-cache misses without increasing the cache size, eliminating two thirds of all instruction misses when running on-line transaction processing applications. QPipe, a staged relational query engine built on top of BerkeleyDB, maximizes data and work sharing across concurrent queries, providing up to 2x throughput speedup in a decision-support workload.

1 Introduction

Research shows that the performance of Database Management Systems (DBMS) 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 [1, 2, 10]. 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.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

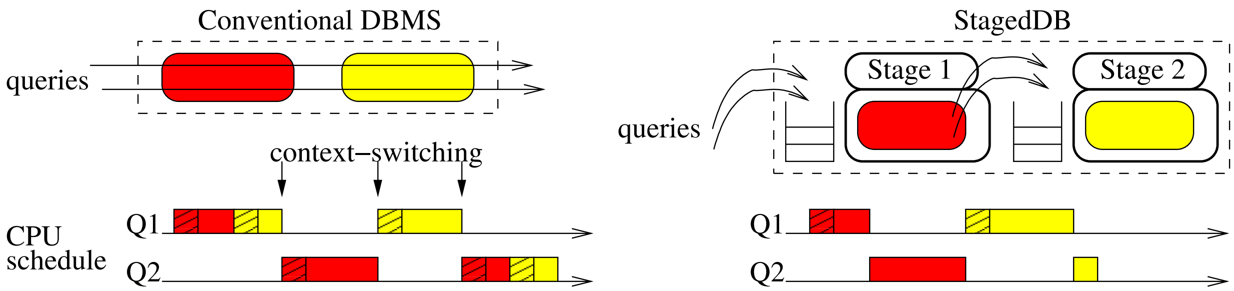

Figure 1: Example of conventional and Staged DBMS architecture with two system modules. Striped boxes in the query schedule (assuming single CPU) correspond to time spent accessing instructions and data common to both queries; these need to be reloaded when modules are swapped. StagedDB loads a module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

1.1 Techniques to improve memory hierarchy performance

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 relational processing algorithms to improve data and instruction temporal locality [16, 14], and also indexing structures and memory page layouts that improve spatial locality [15]. Buffer pool replacement policies seek to improve main memory utilization by increasing the reusability of data pages across requests [5, 12]. Computer architecture researchers propose to rearrange binary code layout [13] 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 [4], while at the microarchitecture level, out-of-order processors tolerate cache access latencies by executing neighboring instructions [9]. 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.

1.2 StagedDB design: exploiting concurrency to improve locality

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. 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.

StagedDB [6], is a new DBMS 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, thus effortlessly exploiting data and work commonality. The StagedDB 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 is accessed, when, and by which requests. Figure 1 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). A StagedDB system (right part of Figure

1) 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 benefits of the StagedDB design are extensively described elsewhere [6].

In this article we present an overview of two systems based on the StagedDB design. STEPS, a transaction coordinating mechanism, minimizes instruction misses at the first-level (L1-I) cache (Section 2), while QPipe, a staged relational query engine, maximizes data and work sharing across concurrent queries (Section 3).

2 *STEPS* for Cache-Resident Code

According to recent research, instruction-related delays in the memory subsystem account for 25% to 40% of the total execution time in Online Transaction Processing (OLTP) applications [2, 10]. 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 [11] 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.

STEPS (for Synchronized Transactions through Explicit Processor Scheduling) is a technique that minimizes instruction-cache misses in OLTP by multiplexing concurrent transactions and exploiting common code paths [7]. 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 (left part of Figure 2). 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 2).

2.1 Implementation of STEPS

We implement STEPS inside the Shore database storage manager [3] by wrapping each transactional operation to form groups of threads. To achieve overhead-free context-switching between threads in the same group, we execute only the *core* context-switch code, updating only the CPU state, and postponing updates to thread-specific data structures until those updates become necessary. In our implementation, the fast context-switch

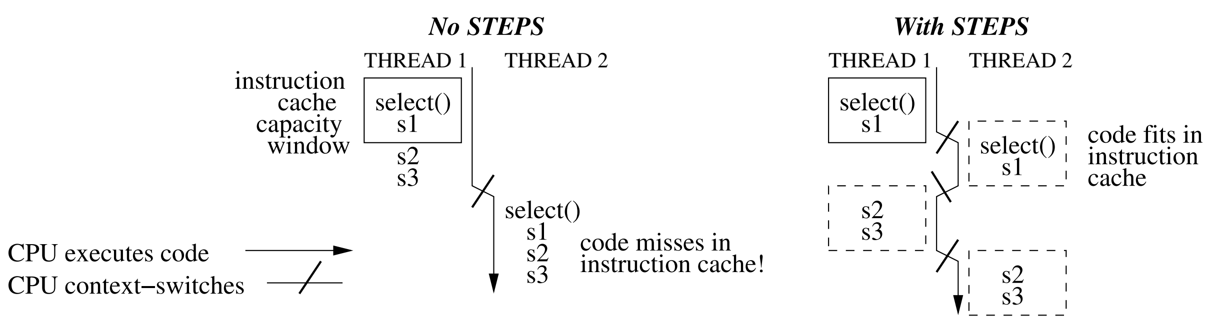

Figure 2: Illustration of instruction-cache aware context switching with STEPS.

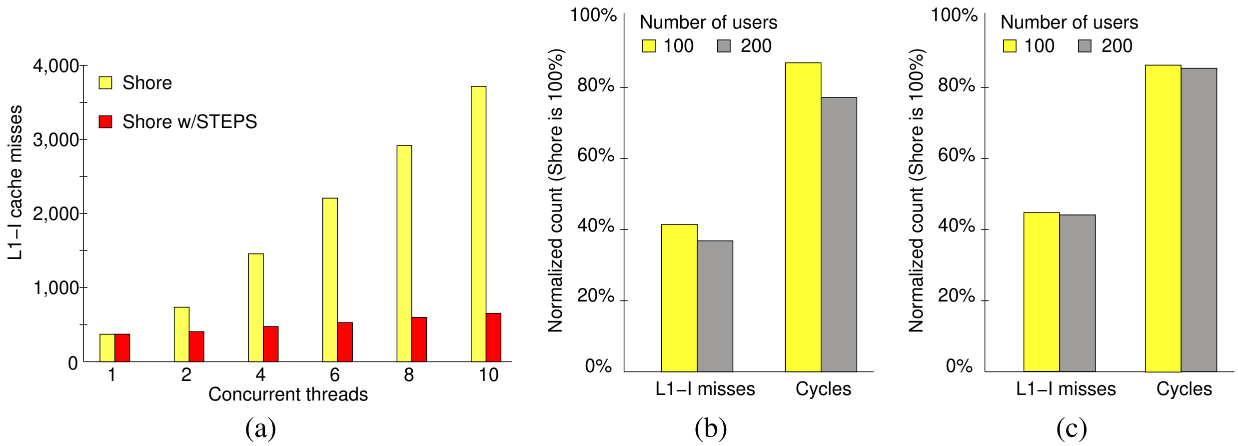

Figure 3: (a) L1-I cache misses for Shore without and with STEPS, when running index select. (b) L1-I miss and execution cycles reduction with STEPS running a single TPC-C transaction, and, (c) a mix of four transactions.

code is 76 bytes and therefore only occupies three 32-byte or two 64-byte blocks in the instruction cache. We locate points in the source code to insert context-switch statements by using a cache simulation tool and also by measuring actual misses in the code using hardware counters. The rest of the code changes are restricted to the thread package, to make sure that every time a thread yields control (either because it needs to wait on I/O, or failed to acquire a lock, or for any other reason, such as aborting a transaction), all thread-package structures that were not updated during fast context-switches are checked to ensure system consistency.

2.2 Evaluation

We conducted an extensive evaluation of STEPS, using both real hardware (two different CPU architectures) and full-system simulation, on both microbenchmarks and full workloads. The full results are presented elsewhere [7]. In this article we present microbenchmark and TPC-C results on a server with a single AMD AthlonXP processor, which has a large, 64KB instruction cache. Figure 3a shows that, when running a group of transactions performing an indexed selection with STEPS, L1-I cache misses are reduced by 96% for each additional concurrent thread. Figures 3b and 3c show that when running a single transaction (Payment) or a mix of transactions from TPC-C in a 10-20 Warehouse configuration (100-200 concurrent users), STEPS reduces instruction misses by up to 65% while at the same time produces up to 30% speedup.

3 QPipe: A Staged Relational Query Engine

Modern query execution engines 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. Queries can share work through materialized views which, however, have a maintenance overhead and also require prior workload knowledge.

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

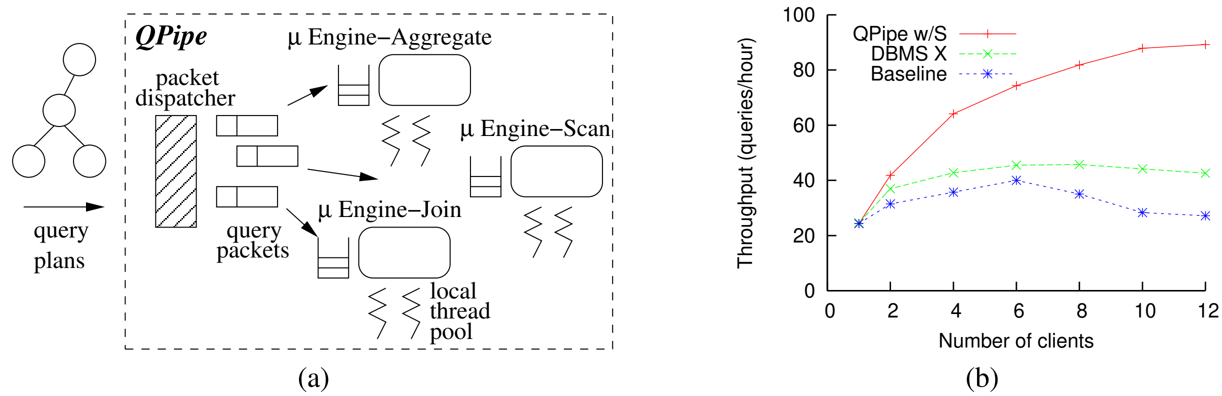

Figure 4: (a) QPipe architecture: queries with the same operator queue up at the same micro-engine (for simplicity, only three micro-engines are shown). (b) Throughput results for a set of TPC-H queries.

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 efficiently detect and exploit overlaps across queries, we introduce *QPipe*, a new query engine architecture which follows a "*one-operator, many-queries*" design philosophy [8]. Each relational operator is promoted to an independent micro-engine which manages a set of threads and serves queries from a queue (Figure 4a). 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. 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.

3.1 QPipe prototype

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. Each micro-engine is a different C++ class with separate classes for the thread-pool support, the shared-memory implementation of queues and buffers, the packet dispatcher, and the modules for detecting and exploiting overlapping work. The basic functionality of each micro-engine is to dequeue the packet, process it, optionally read input or write output to a buffer, and destroy the packet. Calls to data access methods are wrappers for the underlying database storage manager (BerkeleyDB) which adds the necessary transactional support, a buffer-pool manager, and table access methods. Client processes can either submit packets directly to the micro-engines or send a query plan to the packet dispatcher which creates and routes packets accordingly.

3.2 Evaluation

We experiment with a 4GB TPC-H database 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 use a pool of eight representative TPC-H queries, randomly generating selection predicates every time a query is picked. Figure 4b shows the throughput achieved, when varying the number of clients from 1 to 12, for three systems. "Baseline" is the BerkeleyDB-based QPipe implementation with sharing disabled, "QPipe w/S" is the same system with sharing enabled, and "DBMS X" is a major commercial database system. Although different TPC-H queries do not exhibit overlapping computation by design, all queries operate on the same nine tables, and therefore there often exist data page sharing opportunities. For a single client, all systems perform almost the same since the disk bandwidth is the limiting factor. As clients increase beyond six, X is not able to significantly increase the throughput while QPipe with sharing enabled takes full advantage of overlapping work and achieves a 2x speedup over X. A more extensive evaluation of QPipe is presented elsewhere [8].

4 Conclusions

As DBMS performance relies increasingly on good memory hierarchy utilization and future multi-core processors will only put more pressure on the memory subsystem, it becomes increasingly important to design software that matches the architecture of modern hardware. This article shows that the key to optimal performance is to expose all potential locality in instructions, data, and high-level computation, by grouping similar concurrent tasks together. We presented two systems based on the StagedDB design. STEPS minimizes instruction cache misses of OLTP with arbitrary long code paths, without increasing the size of the instruction cache. QPipe, a novel 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. Moreover, by providing fine-grained software parallelism, the StagedDB design is a unique match to future highly-parallel infrastructures.

References

[1] A. Ailamaki, D. J. DeWitt, et al. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA*, 1998.

[3] M. Carey et al. Shoring Up Persistent Applications. In *SIGMOD*, 1994.

[4] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance through Prefetching. In *SIGMOD*, 2001.

[5] H.T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *SIGMOD*, 1985.

[6] S. Harizopoulos and A. Ailamaki. A Case for Staged Database Systems. In *CIDR*, 2003.

[7] S. Harizopoulos and A. Ailamaki. STEPS Towards Cache-Resident Transaction Processing. In *VLDB*, 2004.

[8] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[9] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan-Kaufmann, 1996.

[10] K. Keeton, D. A. Patterson, et al. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*, 1998.

[11] J. Lo, L. A. Barroso, S. Eggers, et al. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *ISCA*, 1998.

[12] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, 2003.

[13] A. Ramirez, L. A. Barroso, et al. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA*, 2001.

[14] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, 2001.

[15] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling Memory Page Layout from Storage Organization. In *VLDB*, 2004.

[16] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.