Transactions Chasing Scalability and Instruction Locality on Multicores

THÈSE N^{O} 6411 (2014)

PRÉSENTÉE LE 7 NOVEMBRE 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÉES MASSIVES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Pınar TÖZÜN

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

I felt once more how simple and frugal a thing is happiness:
a glass of wine,
a roast chestnut,
a wretched little brazier,
the sound of the sea.
Nothing else.
— Nikos Kazantzakis, *Zorba the Greek*

To my fellow musketeers. . .

Acknowledgements

Below is the list of the people whom I shared my PhD journey with; without them this journey would not be possible.

When my advisor, *Natassa Ailamaki*, accepted me to her lab, I was already a second year PhD student that had already changed two labs. She basically gave me my second chance in PhD and supported me in every way so that I would use that chance well. She has been an excellent advisor and a true inspiration in my work. Natassa, thanks for appreciating my honesty at our meeting just before I joined the lab. That meeting was the reason why I still believed I can do a good PhD and it showed me that I can always rely on and be honest with you.

Even though *Andreas Moshovos* was not an official co-advisor, he has been practically my co-advisor for the past three years. I feel extremely lucky to be working with him during these years and am indebted to him for all the advice and feedback in my work.

Phil Bernstein has been one of my favorite people to interact with during conferences and his work was a huge goldmine when I first dived into the world of transactions. *Sam Madden* has always been a role model for the young academics with the diversity of the research topics he has been working on. As a person who primarily came to PhD to become a good systems researcher, I hope one day I would have as much breadth and depth in the field of computer systems as *Ed Bugnion* has. Thanks for accepting to be in my thesis committee and providing invaluable feedback to this thesis.

After being my candidacy exam president, *Willy Zwaenepoel* gave me great advice for my PhD and encouraged me the most to continue doing systems research. I was really glad that he accepted to be my thesis jury president as well.

I have two great academic brothers, *Ippokratis Pandis* and *Ryan Johnson*, who stood by me as two indestructible pillars throughout this journey. Everything I learned about the guts of database systems, I either learned from them or through their guidance. Thanks for being patient with me.

I was incredibly lucky to be part of the DIAS lab at EPFL. Ladies first. *Danica Porobic*, you were a great collaborator, office mate, and travel companion. Thanks for supporting my magnet collection and being patient with my frantic keyboard typing. *Erietta Liarou*, thanks for being my stunt double. *Renata Borovica*, I am going to miss our chats at random EPFL events with a wine-glass in our hands. *Eleni Zacharatou*, thank you for always looking lively and energetic. *Mirjana Pavlovic*, special thanks to you for joining me for a lunch at Tiffany's. Let's move on to the boys. Many thanks to *Manos Athanassoulis* for many random discussions about work and politics, *Radu Stoica* for the challenging questions, *Iraklis Psaroudakis* and *Utku Şirin* for being great junior students to advice, *Miguel Branco* for many feedback sessions, *Thomas Heinis* for translating my abstract to German, *Adrian Popescu* and *Farhan Tauheed* for all the good pictures, *Ioannis Alagiannis* for the frappes, *Manos Karpathiotakis* for the endless Game of Thrones discussions, and *Matt Olma* for being as enthusiastic as me in shooting videos. Finally, *Erika Raetz* and *Dimitra Tsaoussis*, thank you very much for being patient with my random bureaucratic questions and being my French-to-English translators.

In addition to the members of the DIAS lab, several people from the PARSA lab at EPFL and AENAO lab at University of Toronto have been influential in this thesis. It was a pleasure to work with *Islam Atta* who has been an essential collaborator during the last part of this thesis. *Cansu Kaynak* and *Djordje Jevdjic* were extremely resourceful while taming TPC-E. *Onur Koçberber* has been an encyclopedia for me while performing various workload characterization studies. Moreover, *Christoph Koch* and his students at the DATA lab at EPFL were all a great source of feedback.

I am very grateful to *Eric Sedlar* for giving me the opportunity to do an internship at Oracle Labs and for all the long chats during various conferences and EPFL visits. *Brian Gold* was a wonderful mentor at Oracle Labs and tremendously helpful for me to boost my knowledge on computer architecture.

I am also indebted to many people in our community who supported me over the years. Especially, I would like to thank *Shel Finkelstein* for inviting me to my first HPTS after my first conference talk. This gave me a great exposure early on that would not be possible otherwise.

There have been many friends who helped this journey to be as colorful as possible. First comes my *fantastic four*. *Duygu Ceylan*, you are the main reason I kept my sanity during these five years. You have become like my sister and my best friend here and were the only indicator of home in Lausanne for me. Thanks for listening to all my whinings, tolerating my weird habits, and watching Gilmore Girls for the Nth time with me. *Ippokratis Pandis*, in addition to being my academic brother, thank you for also being like a real brother to me; you complained about me not eating well like my mom, warned me about not dressing properly for cold weather like my grandma, and constantly gave me advice about life and work like my dad. *Esra Aslantürk*, thanks for being my kid and sharing my other world (the world of songs, books, films, and series) with me since college. *Helena Kotthaus*, thanks for turning Redwood Shores from a hell full of ducks to a wonderland for me.

I would also like to thank the *Turkish Gang* (*Cansu Kaynak*, *Onur Koçberber*, *Barış Kaşıkçı*, and *Kerem Kapucu*) for all the times you trashed my house and made me laugh as much as a South Park season, *Günseli Çakmakcı* for a never-ending friendship since high-school, *Ece Öztürk* for all the fun at Zurich, *Mihai Dobrescu* for the Starbucks chats, *Minh Dang* for the badminton Sundays, and *Tia Tsi* for the True Blood sessions.

Mom and dad, thanks for loving and supporting me no matter what I do. This gives me the biggest confidence and pulls me up in every step of the way. I love you.

Many thanks also to my extended family for all the home-time in Istanbul, Chicago, and Stuttgart.

Finally, I would like to thank *Rammstein* for being incredibly helpful during coding, *Green Day* for increasing my creativity while preparing slides, *Sigur Ros* for helping me concentrate on writing, and *Tori Amos* and *Lou Reed* for allowing me to relax.

This research has been supported by grants from Dotation (DIAS Lab), FN Sinergia, Naval Research Global (London), and Hasler Foundation.

Abstract

For several decades, online transaction processing (OLTP) has been one of the main server applications that drives innovations in the data management ecosystem, and in turn the database and computer architecture communities. Recent hardware trends oblige software to overcome two major challenges against systems scalability on modern multicore processors: (1) exploiting the abundant thread-level parallelism across cores and (2) taking advantage of the implicit parallelism within a core. The traditional design of the OLTP systems, however, faces inherent scalability problems due to its tightly coupled components. In addition, OLTP cannot exploit the full capability of the micro-architectural resources of modern processors because of the conventional scheduling decisions that ignore the cache locality for transactions. As a result, today's commonly used server hardware remains largely underutilized leading to a huge waste of hardware resources and energy.

In this thesis, we first identify the unbounded critical sections of traditional OLTP systems as the main enemy of thread-level parallelism. We design an alternative shared-everything system based on physiological partitioning (PLP) to eliminate the unbounded critical sections while providing an infrastructure for low-cost dynamic repartitioning and without introducing high-cost distributed transactions. Then, we demonstrate that L1 instruction cache stalls are the dominant factor leading to underutilization in the commodity servers. However, we also observe that independently of their high-level functionality, transactions running in parallel on a multicore system share significant amount of common instructions. By adaptively spreading the execution of a transaction over multiple cores through thread migration or multiplexing transactions on one core, we enable both an ample L1 instruction cache capacity for a transaction and reuse of common instructions across concurrent transactions.

As the hardware demands more from the software to exploit the complexity and parallelism it offers in the multicore era, this work would change the way we traditionally schedule transactions. Instead of viewing a transaction as a single big task, we split it into smaller parts that can exploit data and instruction locality through careful dynamic scheduling decisions. The methods this thesis presents are not only specific to OLTP systems, but they can also benefit other types of applications that have concurrent requests executing a series of actions from a predefined set and face similar scalability problems on emerging hardware.

Keywords: Database management systems, transaction processing, multicore and multi-socket hardware, micro-architectural behavior, instruction misses, transaction-aware scheduling, benchmarking.

Zusammenfassung

Online Transaction Processing (OLTP) ist bereits seit mehreren Jahrzehnten eine der wichtigsten Serveranwendungen welche Innovationen im Datenmanagement Ökosystem und dadurch in den Forschungsgebieten der Datenbank und der Computerarchitektur vorantreibt. Neueste Hardwaretrends zwingen Software zwei große Herausforderungen der Systemskalierbarkeit auf modernen Multicore-Prozessoren zu überwinden: (1) die Nutzung der reichlich vorhandenen Parallelismus über Cores auf Threadebene und (2) die Nutzung des impliziten Parallelismus innerhalb der Prozessorcores. Das traditionelle Design von OLTP-Systemen jedoch steht inhärenten Skalierbarkeitsprobleme aufgrund seiner eng gekoppelten Komponenten gegenüber. Darüber hinaus kann OLTP heute nicht die volle Leistungsfähigkeit der Mikroarchitekturressourcen moderner Prozessoren nutzen weil das Scheduling der Threads den Ort an dem Threaddaten gespeichert sind nicht berücksichtigt. Als Konsequenz bleibt die heute gängige Serverhardware weitgehend unausgelastet was zu einer großen Verschwendung von Hardware-Ressourcen und Energie führt.

In dieser Doktorarbeit identifizieren wir zunächst die unskalierbaren kritischen Abschnitte der traditionellen OLTP-Systeme als Hauptfeind des Parallelismus auf Threadebene. Wir entwerfen dann ein alternatives Shared-Everything System welches auf physiologischer Partitionierung (PLP) basiert um die unskalierbaren kritische Abschnitte zu beseitigen. PLP vermeidet die hohen Kosten von verteilten Transaktionen durch die Bereitstellung einer Infrastruktur für eine kostengünstige und dynamischen Repartitionierung. Weiter zeigen wir, dass Cache-Misses im L1-Instruktionscache der dominierende Faktor ist, der zur Unterauslastung von Servern führt. Allerdings beobachten wir auch, dass unabhängig von ihren übergeordneten Funktionen, parallel laufende Transaktionen auf einem Multicore-System einen erheblichen Anteil an gemeinsamen Anweisungen aufweisen. Indem wir eine Transaktion adaptiv auf mehreren Prozessorcores durch Threadmigration oder durch Multiplexing auf einem Core ausführen, ermöglichen wir einerseits ausreichend L1-Instruktionscachekapazität für eine Transaktion und ermöglichen andererseits die Wiederverwendung von gemeinsamen Instruktionen von nebenläufigen Transaktionen.

Während die Hardware immer mehr Anforderungen an die Software stellt um die Komplexität und den Parallelismus von heutigen Multicores zu nutzen, wird diese Arbeit die Art und Weise ändern wie wir Transaktionen planen. Anstatt eine Transaktion als eine einzige große Aufgabe zu verstehen, teilen wir sie in kleinere Teile, welche Daten- und Instruktionslokalität durch sorgfältige dynamische Schedulingentscheidungen nutzen können. Die Methoden welche wir in dieser Arbeit entwickeln sind nicht nur spezifisch auf OLTP-Systeme anwendbar und können daher auch für andere Arten von Anwendungen, in welchen die gleichzeitige Ausführung einer Reihe von Aktionen aus einer vorgegebenen Menge die Skalierbarkeit auf neuartiger Hardware limitiert, von großem Nutzen sein.

Stichwörter: Datenbanksysteme, Transaktionsverarbeitung, Multicore- und Multisocket-Hardware, Mikroarchitekturverhalten, Instruktionscache-Misses, Transaktionsbewusstes Scheduling, Benchmarking.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Data Management

We live in a data-driven world today [79, 80]. People have various opportunities to reach a wide range of information at any time and they themselves can contribute to the available information [49]. The past few years have even witnessed cases where people act as powerful media sources when the official news sources, such as broadcast networks and press, fail to do so [19, 165]. However, the maintenance and processing of the sheer amount of data to retrieve the essential information in an efficient and cost-effective way poses a tremendous challenge on traditional data management practices [90].

For several decades, database management systems have enabled many influential applications that transform data into useful information. These applications range from high-performance online services (social networks, online shopping, banks, financial markets, etc.) to big data analytics (scientific exploration, sensor networks, business intelligence, etc.). Online transaction processing (OLTP) [64] is one of the most important and challenging database applications and covers the online services applications above. OLTP applications were the primary reason why relational databases were invented back in the day [34, 35].

Some of the notable challenging characteristics of OLTP are that

- there are many concurrent read/write requests to the database,
- each request usually touches a few records in the whole database, and
- clients expect low and predictable response times while also interacting with fresh and consistent data.

Today OLTP is still among the most fundamental applications in the data management ecosystem and has a multi-billion dollar industry [63, 205]. The increased accessibility of the World Wide Web and big data volumes nowadays amplify the challenges of OLTP. Specifically,

- There are many more concurrent requests from various clients to the data. Therefore, OLTP has to exploit any parallelism opportunity from the underlying hardware in order to satisfy all the client requests.

- Touching a small portion of a database as the data volumes grow fast requires smart indexing and caching mechanisms to be able to maintain fast and predictable performance.

As a result, researchers and developers from the database and computer architecture communities lead many innovations targeting the usability and performance of OLTP systems on modern and emerging applications and hardware [86, 111, 142].

1.2 Evolution of Hardware

For the past five decades, processor technology has gone through major advancements mainly following Moore's law [132] that predicts the doubling of the number of transistors within a single chip every year or two. To exploit this increase in the transistor counts in a unit area, initially, computer architects focused on boosting the performance of a single thread while designing chips. More specifically, they kept clocking the processors at higher frequencies and designing complex micro-architectural features (e.g., aggressive pipelining, super-scalar execution, out-of-order execution, and branch prediction [78]) that enable instruction and data level parallelism implicitly (i.e., vertical parallelism). This led software developers to rely on this implicit/vertical parallelism within a chip to execute a single task as efficiently as possible.

Since the beginning of this decade, however, power draw and heat dissipation have prevented processor vendors from leaning on rising clock frequencies or more complex micro-architectural techniques for higher performance. Instead, they have started adding more processing cores or hardware contexts (i.e., horizontal parallelism) on a single processor to enable exponentially increasing parallelism and performance opportunities [138]. As a result, any software design today has to pay attention to both implicit/vertical and explicit/horizontal parallelism in order to get the best of the underlying hardware.

Unfortunately, the end of this trend is also upon us. While we will still be able to incorporate more cores on a single die, we will no longer be able to use them all at the same time. The main problem is again power-related. Even though Moore's law still holds today, Dennard scaling [43], which enables keeping the power density of the transistors constant, does not. The supply voltage required to power all the transistors up does not decrease at a proportional rate [50]. Putting more cores in a chip is not going to be able to overcome this problem any longer. This trend is referred to as *dark silicon* and fundamentally alters the focus of hardware designs [71]. In this new era, the focus needs to shift toward optimizing energy per instruction.

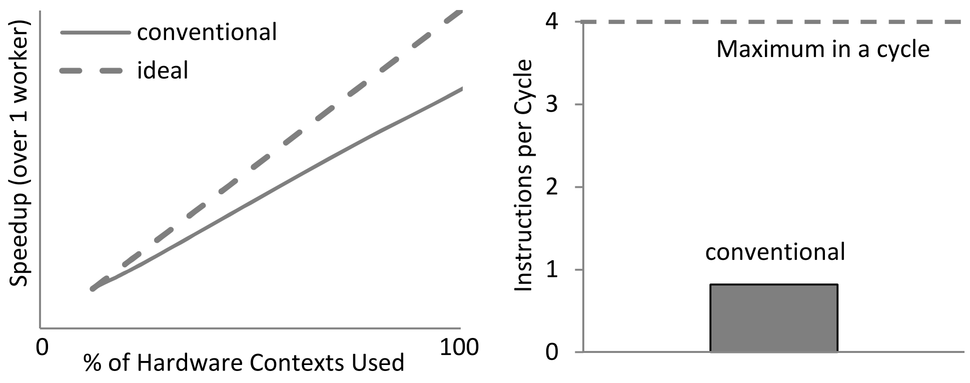


Figure 1.1: A conventional OLTP system on a commodity server; exploiting horizontal (left-hand side) and vertical parallelism (right-hand side) while running a simple short transaction that just reads a client's balance.

1.3 OLTP on Modern Hardware

As the previous section briefly mentions, the hardware technology has mainly evolved in the direction of providing further opportunities for parallelism in two dimensions, vertical and horizontal, to be able to continue exploiting the increasing number of transistors. However, the evolution of hardware does not automatically translate into proportional performance improvements for some complex software systems such as databases.

On the one hand, as shown by various early workload characterization studies, commercial workloads, especially transaction processing, exhibit diminishing returns from aggressive micro-architectural features [3, 69, 106, 177]. They cannot exploit the vertical parallelism in a core fully, especially due to poor instruction-level parallelism (ILP). On the other hand, exploiting the horizontal parallelism offered by multicores is limited by Amdahl's law [11], which states that the speedup of a program in parallel computing is bounded by the fraction of the program that can be parallelized. The tightly-coupled components in traditional data management systems lead to various scalability bottlenecks on multicores and hinder the parallel execution of even non-conflicting requests [8, 99, 116, 155, 166].

Figure 1.1 demonstrates how well traditional transaction processing systems utilize the resources of high-end server hardware (an Intel Sandy Bridge server [88]) in two dimensions:

- at the level of the whole machine (i.e., exploiting horizontal parallelism) and
- within a core (i.e., exploiting vertical parallelism).

The experiment uses the Shore-MT storage manager [96, 172] (and Section 2.6) executing a very simple read-only transaction that just looks up a client's key value in the database through an index search and reads the client's balance column from the record that belongs to this client.

The left-hand side of Figure 1.1 plots the relative throughput over the throughput achieved using a single worker thread as the number of worker threads executing transactions increases, i.e., as the number of hardware contexts used in the machine increases. The dashed line indicates the ideal case where each worker thread performs the same as the single worker thread in isolation. However, in practice, what we achieve is the solid line. Even though the overall throughput keeps increasing, the gap between the ideal line and the line of the conventional system also increases as we use more hardware contexts in the system. This highlights the poor scalability of the conventional system, which is only going to get worse with emerging hardware that offers more hardware contexts, and hence more horizontal parallelism.

On the other hand, the right-hand side of Figure 1.1 shows whether a worker thread in isolation can get the best of the micro-architectural features within a core. More specifically, the graph shows the instructions retired in a cycle (IPC) as one worker thread executes transactions based on the above setup. Even though the Intel server being used has four-way processors, i.e., each core has the ability to retire up to four instructions in a cycle, the conventional system barely retires one. Therefore, exploiting vertical parallelism is also a big challenge for OLTP.

Problem: *Traditional OLTP systems face two major challenges while trying to utilize modern hardware:*

- *Exploiting the abundant thread-level parallelism given by multicores.*
- *Taking advantage of the aggressive micro-architectural features within a core.*

The focus of this dissertation is to tackle the two challenges above in the context of *traditional* transaction processing systems running on a multicore server. We seek solutions that introduce minimal changes to existing software and hardware systems to maximize possible adoption of the proposed mechanisms.

1.4 Scaling Up on Multicores

Conventional shared-everything OLTP design is simple to configure, yet vulnerable to various scalability bottlenecks due to shared resources and tightly coupled internal components. The worker threads in the system can handle any client request. It is unpredictable which data each worker thread might access as they execute transactions [174]. As a result, the execution of a typical transaction is cluttered with critical sections to ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties over the shared data. These critical sections either lead to contention among multiple threads, which limits scalability [94, 99], or impose a significant penalty on single-thread performance even under no contention [76].

Various analysis studies demonstrate that as the available parallelism increases on multicore hardware, the time spent in different storage manager components change [95]. The lock manager becomes the component that contributes the most to the overall execution time under high concurrency [95, 145]. On the other hand, techniques that try to sidestep the problematic components often fail to be effective on the future generations of multicores [153, 154] and elimination of some scalability bottlenecks surfaces other unscalable components and critical sections [97, 147]. Therefore, one needs a methodological way to identify and minimize the scalability problems in order to scale up on emerging hardware.

Shared-nothing designs, on the other hand, choose to deploy independent database instances by physically partitioning the data [45, 178]. In this way, the contention on the shared data resources can be explicitly tuned by determining the number of processors assigned to each instance. Therefore, the shared-nothing design potentially leads to superior performance under perfectly partitionable workloads. However, its performance suffers when the workload triggers distributed transactions or when skew causes load imbalance [153].

Goal: *Providing an alternative shared-everything design and transaction execution model that allow more robust scalability on modern and future multicore architectures while preventing load imbalance across the worker threads in the system.*

1.5 Utilizing Resources within a Core

In the computer architecture community, it is more common to evaluate hardware advancements using the SPEC benchmarks [176] rather than the data management benchmarks [188]. The problem with this practice is that the data management applications tend to be a lot more complex in terms of memory access patterns (both in instruction and data accesses) compared to the compute-intensive benchmarks in the SPEC suite. Therefore, the aggressive micro-architectural properties that tend to boost the performance of the SPEC benchmarks usually do not benefit data management applications as much.

There has been a large body of workload characterization studies during the last two decades that investigate the micro-architectural behavior of OLTP workloads [16, 106, 161, 177]. They all conclude that OLTP cannot exploit aggressive micro-architectural features, thereby wasting most of its execution cycles in memory stalls and exhibiting low IPC.

After almost 15 years later than the initial detailed workload characterization studies for data-intensive applications, in a more recent study, Ferdman et al. [55] demonstrate that there is still a clear mismatch between what modern hardware offers and what data management systems can exploit from it. Large-scale data management workloads, including OLTP applications, still fail to take advantage of the full capability of today's commodity servers at the micro-architectural level due to poor instruction and data locality at different levels of the memory hierarchy. Such underutilization of micro-architectural features is a great waste of hardware resources.

Goal: *Identifying the dominant factors from both the hardware and software sides causing the underutilization of micro-architectural resources in a core while running transaction processing applications, and eliminating their effect through techniques that improve locality at the right level of the memory hierarchy.*

1.6 Thesis Statement and Contributions

This thesis contributes to the quest of bridging the gap between software and hardware in the context of transaction processing systems.

Thesis Statement

Typical database management systems process each transaction as an indivisible unit, thereby exploiting less than the abundant parallelism available in today's hardware platforms and underutilizing processor caches. To optimize the use of micro-architectural features and avoid wasting hardware resources, systems should break transactions dynamically into smaller parts, according to which data and instructions each part accesses, and schedule each part adaptively depending on the other transactions currently running in the system.

With the above statement, we depart from the traditional way of scheduling transactions, which considers them as a black-box. We advocate for a more fine-grained task scheduling that is aware of the actions a transaction executes. In this way, on the one hand, we can improve thread-to-data access locality and minimize contention on shared data, and on the other hand, maximize instruction cache locality to eliminate memory stalls that are hard to overlap.

A summary of the contributions of this thesis is below:

- We demonstrate that the scalability of a shared-everything transaction processing system depends on minimizing the *unbounded* communication points on the critical path of a transaction. Through physiological partitioning, we provide an infrastructure for eliminating *unbounded* critical sections during logical and physical data accesses, which in turn eliminates a majority of the *unbounded* communication in a transaction. The same infrastructure also minimizes the costs of multi-partition transactions and dynamic repartitioning.
- We perform thorough workload characterization of the Transaction Processing Performance Council's (TPC's) [188] transaction processing benchmarks. Our analysis
    - illustrates the evolution (e.g., complexity increase) with each generation of OLTP benchmark TPC standardizes,

  - highlights that first-level instruction cache misses are the dominant factor causing OLTP to exhibit low IPC and spend more than half of its execution cycles on memory stalls,
  - maps memory stalls to the storage manager components they stem from, and
  - reveals that transactions exhibit significant temporal code overlap as they run concurrently on multicore hardware.

- Based on the insights from our analysis, we design alternative ways of scheduling transactions at the hardware-level that aim to minimize L1 instruction misses through maximizing instruction reuse across concurrent transactions. Two of the techniques are programmer-transparent (pure hardware) techniques: one adaptively spreads the execution of transactions over multiple cores through thread migration and the other one time-multiplexes transactions on the same core. On the other hand, the last technique is more transaction-aware and gets software-side hints as it migrates transactions over cores in order to reduce the complexity required from the hardware side.

To be able to have better software/hardware integration for transaction processing systems, this thesis aims to lead people to re-think scheduling decisions for transaction processing workloads to better utilize the micro-architectural resources of underlying modern hardware and also give guidance on how to specialize future hardware designs for OLTP. In addition, even though this thesis targets transaction processing applications, the insights and techniques presented here have potential to benefit any other application that executes concurrent requests formed of some predefined tasks, suffers from unbounded communication on the critical path, and exhibits micro-architectural inefficiencies due to memory stalls.

1.7 Roadmap

The three parts of this thesis cover the three contributions above. The details for each chapter are as follows:

- Chapter 2 gives background on improving hardware utilization while running OLTP on modern multicore hardware. It also introduces the OLTP benchmarks and the Shore-MT storage manager used throughout this thesis. Readers familiar with these concepts can skip this chapter.

- In Part I:

  - Chapter 3 first classifies the critical sections in a typical OLTP system into three: *fixed*, *unbounded*, and *cooperative*. It shows that not all critical sections cause scalability bottlenecks and the key to scale-up OLTP in a single node is to either remove unbounded critical sections or downgrade them into fixed or cooperative types. Based on this insight, the chapter presents *physiological partitioning (PLP)* to eliminate unscalable locking and latching, which form the majority of the unbounded critical sections in a shared-everything OLTP system.

    - Chapter 4 designs a lightweight yet effective dynamic load balancing mechanism for PLP after demonstrating with a cost model that PLP provides a very good infrastructure for dynamic repartitioning. The resulting design enables a shared-everything OLTP system to run free of most *unbounded* communication without introducing costly distributed transactions, and adapt to workload changes and skew with low-cost dynamic repartitioning.

- In Part II:
    - Chapter 5 investigates the evolution of the OLTP benchmarks introduced by TPC [188] focusing mainly on the latest benchmark, TPC-E [193]. The chapter demonstrates that TPC-E is significantly more complex in terms of its schema and data accesses compared to its predecessors and suffers due to logical lock contention. However, at the micro-architectural level all OLTP benchmarks behave the same: more than half of the execution cycles go to stalls and on machines that have the ability to execute four instructions per cycle, OLTP exhibits barely one instruction per cycle.
    - Chapter 6 shows that the large instruction footprint of transactions is the dominant factor causing the low utilization of the existing micro-architectural resources. OLTP workloads mainly suffer from L1 instruction cache misses coming from very common clear execution paths within the index, lock, and buffer manager components of a typical storage manager, mainly during an index probe. The next problematic component is the long-latency data misses from the last-level cache. On the other hand, the worker threads of an OLTP system usually execute similar transactions in parallel and each transaction is formed of a subset of the predefined database operations. As a result, even though the threads running on different cores do not execute exactly the same code, they do share a non-negligible amount of instructions.

- In Part III:
    - Chapter 7 proposes two programmer-transparent scheduling techniques that aim to improve instruction locality at L1 caches via exploiting the instruction overlap across concurrent transactions. By adaptively spreading the execution of a transaction over multiple cores through thread migration or multiplexing transactions on one core, these scheduling mechanisms create an ample L1 instruction cache capacity and enable instruction reuse.
    - Chapter 8 presents transaction-aware thread migration to dynamically allocate cores to each database operation based on their frequencies and instruction footprint. Through getting software-side hints, this chapter minimizes the functionality that needs to be added to hardware for the type of scheduling mechanisms presented in this part.

- Finally, Chapter 9 concludes this thesis focusing on the possible next steps in the quest of better hardware/software integration for data management applications.

2 Background

This chapter provides a brief overview of [1]

- some of the terminology regarding the database transactions (Section 2.1),
- the memory hierarchy of high-end server hardware and techniques that provide implicit parallelism within a core (Section 2.2),
- related work aiming to exploit implicit/explicit hardware parallelism in the context of transaction processing systems (Section 2.3), and
- industry standard online transaction processing benchmarks (Section 2.4 and Section 2.5) and the storage manager (Section 2.6) used throughout this thesis for the analysis studies and evaluation of the proposed ideas.

2.1 Transaction Processing

A transaction is a unit of work in a database system that satisfies the properties (abbreviated as *ACID* [64]) given below:

- **Atomicity:** When a transaction completes the execution of its actions, either all or none of their effects are visible to the other transactions.
- **Consistency:** The transactions start from a consistent state of the database and complete their execution through transforming the database to another consistent state.
- **Isolation:** Transactions should not interfere with each other's effects to the database. [2]

[1] This chapter uses material from [64, 78, 96, 137, 186, 188].

[2] Some isolation levels would allow a transaction to see the changes done by another incomplete transaction. However, after all the transactions commit, the changes should appear as if the transactions were run in isolation.

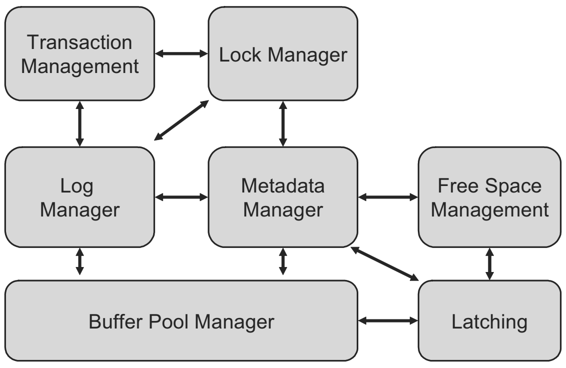


Figure 2.1: Main components of a storage manager (taken from [96]).

- **Durability:** The effects of complete transactions must be persistent in the database.

The maintenance of these properties in the face of many concurrent client requests is a big challenge for any transaction processing system and complicates the design of a storage manager. Multiple storage manager components are involved in satisfying each of the ACID properties and a storage manager component is usually involved in providing several properties. For example, both isolation and consistency properties rely on atomicity to rollback the changes of a failed transaction, where atomicity itself depends on logging since upon a transaction abort the log is used to undo the changes of the aborted transaction. Therefore, as Chapter 1 mentions, the components of a typical storage manager in transaction processing systems are very tightly coupled, which leads to various forms of underutilization on modern multicore hardware.

Figure 2.1 illustrates the main components of a conventional storage manager. A brief explanation for each of the components is given below.

- **Transaction Management:** The transaction manager does the bookkeeping about all active transactions (e.g., thread-to-transaction assignments, transaction order, etc.), coordinates the checkpointing, and orchestrates the recovery upon system crashes.
- **Lock Manager:** Conventional transaction processing systems maintain a centralized lock manager in order to dictate isolation among concurrent transactions. It keeps information about which worker threads (and hence active transactions) are holding or waiting for a particular record's/table's/volume's lock and in which lock mode (shared, exclusive, etc.). Each worker thread must consult the lock manager before acquiring the locks for the records they are going to access. The lock manager grants these requests if no other thread currently holds the lock or the lock modes of the thread holding the lock and the thread requesting the lock are compatible. If the lock cannot be granted, the thread is added to the list of threads waiting this lock.
- **Log Manager:** The log manager records all the modifications performed by transactions in the database. For each transaction, the in-memory log buffer keeps the changes it

performs. Before a transaction commits successfully, the log entries of that transaction must be flushed to disk. The database log allows the recovery of the database when the system crashes before the changes of some of the committed transactions were flushed to disk and rollback of the modifications done by the aborted transactions.

- **Metadata Manager:** The metadata manager keeps information about the objects a database stores: the files that keep the database records and table indexes. From the perspective of the storage manager, the metadata information is just another type of data to be stored that is not frequently updated.
- **Buffer Pool:** The buffer pool is the virtual memory of a database management system. It gives the illusion that all the data is in-memory and manages the retrieval/flushing of the database pages from/to disk.
- **Free Space Management:** Space manager tracks the free space on each database page and handles the allocation of new pages if needed.
- **Latching:** Even though the lock manager ensures consistent updates of database records at the logical level, it does not protect the contents of database pages against concurrent updates since the database pages keep multiple records or information about multiple records. This is the duty of page latching. Latches are at the granularity of a database page and usually held for short periods, only for the duration of a read/write operation on a particular page.

2.2 Micro-architecture of OLTP's Playground

As Chapter 1 also mentions, hardware vendors heavily innovated on implicit parallelism till 2005 through aggressive micro-architectural techniques. The main goal behind all these innovations was to prevent CPUs from stalling due to either memory accesses or core functionality while processing an instruction. The main sources of implicit parallelism are described below:

- **Pipelining:** The execution of an instruction is composed of a sequence of steps. For example, the classic RISC pipeline consists of five stages for an instruction:
  - *fetching* the instruction from the cache,
  - *decoding* the instruction to specify which operation it performs and inputs it needs,
  - *executing* the operation,
  - accessing the *memory* for inputs if needed, and
  - *writing back* the results into registers.

  Early processor designs processed instructions one after the other requiring several cycles per instruction. Instruction pipelining allows overlapping of the above stages, and hence, partially overlaps the processing of several instructions.

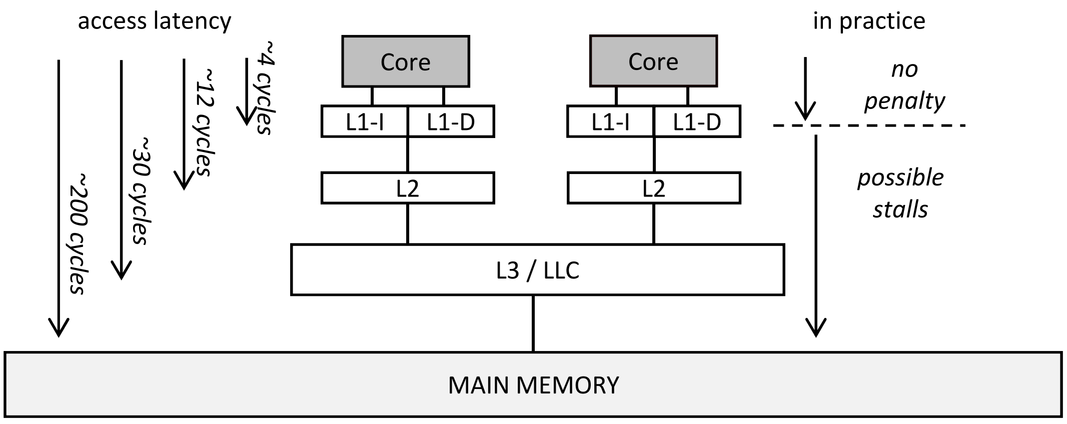


Figure 2.2: Memory hierarchy of commodity servers.

- **Superscalar:** A superscalar CPU is able to maintain multiple instruction pipelines, and therefore, can issue multiple instructions in a cycle.
- **Out-of-Order Execution (OoO):** Out-of-order execution mainly enhances the *execute* stage of the instruction pipeline and allows a processor to execute instructions based on the availability of input data rather than strictly following the instruction ordering of a program.
- **SIMD:** The three techniques presented above all focus on instruction-level parallelism. SIMD instructions, on the other hand, provide data-level parallelism. They apply the same instruction over multiple data items simultaneously. Hence, they prevent the cost of processing the same instruction over and over.

All these techniques, however, become ineffective either when an application exhibits high memory dependencies (a memory address to be accessed depending on the memory access that comes right before) or excessive memory stalls. Unfortunately, data management applications suffer from both aspects. For example, during an index probe operation the next index node to be accessed depends on the index node that is currently accessed and the key value that is searched. In addition, the instruction and data footprints for data management applications usually exceed the size of the typical L1 caches and lead to excessive memory stalls.

Since this thesis targets the problem of memory stalls, next we focus on why they happen. Figure 2.2 shows the memory hierarchy of a typical server processor today. There are usually three levels of caches. The first-level caches are split between instructions and data, whereas the lower levels of the memory hierarchy are shared by instructions and data. The L1 instruction and data caches as well as the L2 caches are private per core and the cores of a processor share the L3 or last-level cache (LLC). While going down in this hierarchy, the access latencies drastically increase at each level. However, in practice, a superscalar OoO core easily hides the latency of accessing the L1 caches. On the other hand, if a core cannot find a memory address in the L1 caches, then the lower levels of the memory should be searched. This might stall the core till it gets the instructions or data needed to continue the execution. Such memory stalls are the dominant factor in the underutilization of a core's resources and have to be minimized.

2.3 Exploiting Modern Hardware while Running OLTP

As Chapter 1 also mentioned, hardware trends oblige OLTP systems to overcome two major challenges against hardware utilization:

- exploiting explicit/horizontal parallelism provided by an increasing number of available hardware contexts on multicore architectures and
- taking advantage of implicit/vertical parallelism supported by the micro-architectural features within a core.

There is a large body of related work that tackles both of these challenges, which are highlighted in the following two subsections, respectively. This section aims to give an overview of the work in this field. In the following parts of the thesis, each chapter compares its contributions to the corresponding related work presented in this section.

2.3.1 Scaling Up OLTP

The traditional shared-everything design for transaction processing systems is simple to deploy since there is only a single database instance to maintain and transactions are assigned to worker threads randomly regardless of which data they are going to access. However, shared-everything systems were not designed with abundant horizontal parallelism in mind. Even though they can handle several concurrent requests very efficiently while satisfying the *ACID* properties on a few single-core processors, their performance suffers on multicore architectures due to the tightly-coupled and centralized internal components and unpredictable accesses to the shared data [96, 145].

Proposals that depart from traditional transaction processing in order to scale up on multicores can be grouped into two broad categories based on whether they apply some sort of data partitioning or not. The ones that are based on partitioning, in turn, fall into two categories: *physical* or *logical* partitioning. On the other hand, the systems that prefer avoiding any kind of data partitioning rely on lock-free algorithms and multiversion concurrency control.

Physical Partitioning

Shared-nothing systems [45, 178] deploy multiple instances of a database and physically partition the data across these instances. One can determine which part of the data belongs to which instance of the database and how many worker threads each instance should have. In a way, physically partitioned systems give explicit control to users in terms of managing data access patterns and contention on the data. For example, systems like VoltDB [199] (commercial version of H-Store [179]) and Hyper [107] apply the idea of physical partitioning to the extreme and deploy single-threaded database instances. Such a design eliminates all unbounded communication within the database engine and has the potential to achieve perfect scalability as long as the workloads are perfectly partitionable; i.e., all the transactions are handled by a single database instance and data accesses are balanced across the instances.

However, for workloads that are not amenable to partitioning, physically partitioned systems suffer from multi-partition/distributed transactions [39, 77, 149] or load imbalance [150]. Dynamic repartitioning to minimize distributed transactions or load balancing is highly costly due to the amount of data to be moved from one partition to another and reorganization of the index structures [169].

Logical Partitioning

In contrast with physical partitioning, there are shared-everything systems that apply logical partitioning of the data in a single database instance to regulate the data access patterns. In such designs, each logical partition is owned by a single worker thread [145] or a few worker threads [114, 143]. Through managing the number of worker threads per partition, logically-partitioned systems can also control contention over the data. By logically-partitioning the accesses to the database records, this design also partitions the accesses to the lock manager. Therefore, logical partitioning decentralizes the lock manager and ensures each partition correctly maintains its own lock manager.

However, the accesses to shared database pages are still unpredictable and would eventually lead to scalability bottlenecks with increased hardware parallelism. On the other hand, online repartitioning is very cheap under this design since no data movement is necessary [146]. Part I enhances logical partitioning and partitions physical data page accesses as well; this eliminates contention on both database records and pages while still remaining in a shared-everything setting.

Lock-free Techniques and Multiversion Concurrency Control

The OLTP system designs that choose to scale up without relying on partitioning use lock-free techniques while maintaining the consistency of their internal data structures, and optimistic and multiversion concurrency control to ensure isolation and atomicity of concurrent transactions. There are various commercial OLTP systems that follow this approach: Hekaton [46], MemSQL [127], TokuDB [182], SAP HANA [117], etc. In addition to typical OLTP workloads, this design especially benefits workloads that combine short-running transactions that might modify the database and long-running read-only transactions since it does not block transactions with pessimistic locking techniques.

However, most of the lock-free algorithms proposed by these systems do not eliminate the possible central communication points or unbounded critical sections (Section 3.2.1), which all the worker threads in the system have potential to execute. They rather minimize the time spent in such points through optimistic concurrency control, decentralized record locks, frequent use of atomic operations, or read-copy-update mechanisms [116, 119, 196]. Even though these operations perform and scale very well on architectures with a few (one to four) processor sockets, scaling them up on multisocket multicore hardware with more than four processors or emerging many-core processors is not straightforward [41, 154, 204]. In addition, these techniques suffer under update-heavy workloads with hotspots similar to the statically partitioned designs [135].

2.3.2 Minimizing Memory Stalls

Techniques aiming to minimize memory stalls target instruction and data cache locality and utilization for a wide variety of applications. This section groups such techniques that specifically target OLTP-like applications into two based on whether they are hardware or software approaches.

Hardware-Side Approaches

One of the well-studied techniques for improving instruction and data locality at various levels of the memory hierarchy is hardware prefetching. Even though the simpler prefetching techniques that exist on modern hardware (e.g., next-line, stream, stride prefetchers [78]) help in terms of reducing some of the instruction and data misses, they are not enough to minimize the memory stalls for memory-intensive applications like OLTP as we (Part II) and many other workload characterization studies highlight [3, 54, 106, 161, 177].

Sophisticated temporal stream predictors [52, 105, 175] are an especially good fit for OLTP-like applications where the execution follows a long but predictable path. For example, whenever one searches a record that maps to a key value in a traditional OLTP system, first the index lookup routine is invoked. Index lookup, then, starts the B-tree traversal from index root page to leaves. The traversal routine, in turn, invokes the binary search routine for each page touched during the tree traversal to find the key value. Even though this is a long execution path, it is executed over and over in an OLTP system. The goal of temporal streaming techniques is to exploit such recurring execution paths. However, they trade simplicity for accuracy; i.e., the more accurate they are the more complex their functionality becomes. As a result of the complexity of these techniques, hardware vendors still prefer using simpler prefetching techniques such as next-line prefetching and branch prediction for instructions; and next-line, stream, and stride prefetchers for data.

In addition to prefetching, recent work proposes computation spreading, which involves multiple cores in the execution of a task to separate the execution of the system code from application code in order to improve instruction cache locality [30]. In Part III, we take this idea further and consider tasks at a finer-granularity to maximize instruction locality.

Software-Side Approaches

There are two significant approaches to reduce memory stalls from the software-side:

- improving code or data layout to increase cache line utilization and
- departing from traditional execution models to change the way instructions and data are accessed.

In the case of instructions, the most straightforward approach would be to simplify the code of a system. For example, databases optimized for main-memory usually adopt more lightweight concurrency control mechanisms and omit the buffer pool component. Therefore, they execute fewer instructions than traditional disk-based database systems. On the other hand, for the systems where a simplified codebase is not enough to eliminate a majority of the instruction related memory stalls, smart static or dynamic compilation techniques can optimize the code layout [159]. A better code layout leads to smoother instruction streams and helps programs to exploit the next-line prefetcher in a more effective way. In the database community, such approaches are used during query compilation to compile a query (or transaction) with an optimized instruction and data layout for that query [46, 109, 112, 136].

Alternative data page layouts have been widely studied in the data management community. Even though the initial proposals target minimizing the overhead due to disk accesses for disk-based systems, the main insights from these work can also be applied today to provide more cache conscious data layouts and minimize accesses to memory. Row-stores are the original choice for data layouts, where records of a table are allocated as a whole one after the other in a database page. Column-stores [24, 37, 180], on the other hand, vertically partition each table to its columns and store the values that belong to each column from a table together. There have been proposals for hybrid data layouts [5, 9, 65] that combine these two techniques as well. While choosing the optimal layout for an application, the data access patterns must be observed to know which layout would maximize

- the benefits of the simple stream or next-line prefetchers via ensuring mostly sequential data accesses and
- data cache utilization through maximizing the usage of a cache line brought into the data cache.

For example, since OLTP workloads tend to access several columns from a few records, a row-store is the dominant choice for the data layout. On the other hand, for analytical workloads that go over a huge number of records by just checking a few columns, column-stores or hybrids are preferred. Where the alternative data layouts target the allocation of the database records, proposals for cache conscious index structures have the same goals given above for index pages [32, 125, 163] through changing the size of the index pages or the way index pages are allocated in memory.

Orthogonal to improving instruction and data layouts, one can also change the way a program accesses instructions and data to maximize cache locality. Vectorized execution [24] is one such technique. Unlike the Volcano-style iterator model [60] (which processes tuples/columns one at a time), vectorized execution processes a vector of items at each database operator to enable instruction and data reuse. Staged databases [75] or STEPS [72] are other alternative execution models for analytical and transaction processing applications, respectively, which also enable instruction and data reuse at the first-level caches. In Part III, we also propose an alternative execution/scheduling model for OLTP in order to maximize instruction cache locality through instruction reuse.

2.4 Evolution of TPC's OLTP benchmarks

Transaction processing benchmarks are the gold standard for DBMS performance evaluation and they are frequently used for marketing purposes. The Transaction Processing Performance Council (TPC) [188] is a non-profit IT organization founded to define database benchmarks and disseminate objective, verifiable performance data to the industry. This section describes the four important database transaction processing benchmarks that have been used under the trademark of TPC and highlights how they have evolved over the years with each new benchmark.

2.4.1 The obsolete TPC-A and TPC-B

The first widely accepted database benchmark was formalized in 1985 [12]. That specification included three workloads, of which the *DebitCredit* stressed the database engine. The DebitCredit benchmark was an instant success. Soon database and hardware vendors started reporting extraordinary results, often achieved by removing key constraints from the specification. Therefore, in 1988 a consortium of analysts and hardware, operating system, and database vendors formed the Transaction Processing Performance Council in order to enforce some order in database benchmarking. Its first benchmark specification, TPC-A [189], essentially formalized the DebitCredit benchmark.

TPC-A is straightforward. It models deposits to and withdrawals from random bank accounts, with the associated double-entry accounting on a database that contains x `Branches`, $10x$ `Tellers`, and $100{,}000x$ `Accounts`. It also captures the entire system, including terminals and network. Transactions usually originate from their *home* `Branch`, but can go anywhere. Conflicts are possible requiring the system to recover occasionally from failed transactions.

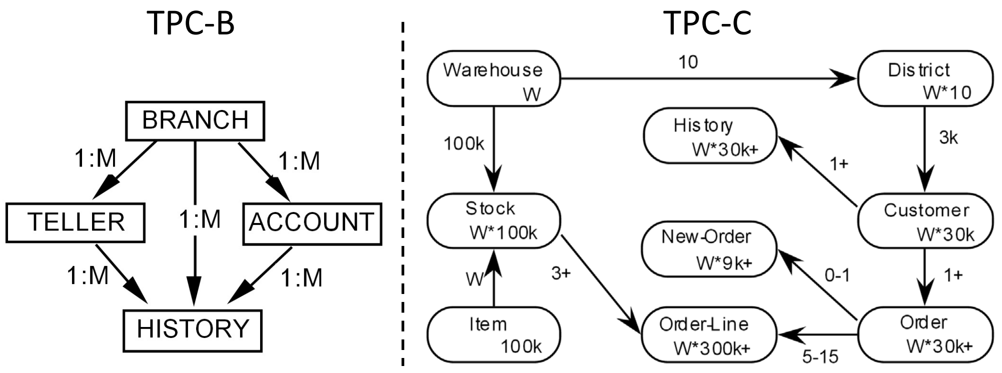


Figure 2.3: Schemas of the TPC-B and TPC-C benchmarks (taken from [188]).

An important aspect of this benchmark is its scaling rule: for a result to be valid, the database size must be proportional to the reported throughput.

Even though it is simple, the TPC-A benchmark highlighted the importance of quantifying the performance and correctness of different systems. Early benchmarking showed vast performance differences among different vendors (400x), as well as exposing serious bugs, which had lurked undiscovered for many years in mature products.

TPC's second benchmark, TPC-B [190], is very similar to TPC-A, but eliminates the network and terminal handling to create a database engine stress test. Like TPC-A, the TPC-B database contains four tables: `Branch`, `Teller`, `Account`, and `History`. These tables are accessed in a double-entry accounting style as customers make deposits on and withdrawals from various tellers. The benchmark consists of a single transaction, `AccountUpdate`, which simply updates one record in the `Branch`, `Teller`, and `Account` tables while appending a record to the `History` table. Therefore, it is a very update-heavy transaction that stresses the transaction processing engine; especially the logging and concurrency control modules.

2.4.2 The ubiquitous TPC-C

For its third benchmark specification, TPC-C [191], TPC moves away from banking to commerce. TPC-C models an online transaction processing database for a wholesale supplier. The transactions follow customer orders from initial creation to final delivery and payment.

A TPC-C database consists of nine tables in total where

- one of them has fixed size (*fixed*),
- four of them scale proportionally with the number of `Warehouses` (*scaling*), and
- four of them might change size, mostly grow, due to insert and delete operations (*growing*).

Thereby, compared to TPC-B, TPC-C offers a more complex database schema; where the TPC-B schema can be represented as a tree with only four nodes, the TPC-C schema is a directed acyclic graph with nine nodes (see Figure 2.3).

Like the database schema, the TPC-C transactions are more complex than the `AccountUpdate` transaction of TPC-B. The benchmark combines the five transactions listed below in a transaction mix at frequencies given in parenthesis:

- `NewOrder` **(45%)** inserts a new sales order to the database. It is a medium-weight transaction with a 1% failure rate due to invalid inputs.
- `Payment` **(43%)** is a short transaction, very similar to the `AccountUpdate` transaction of TPC-B, which makes a payment on an existing order.
- `OrderStatus` **(4%)** is a read-only transaction that computes the shipping status and the line items of an order.
- `Delivery` **(4%)** is the largest and the most contentious update transaction. It selects the oldest undelivered orders for each warehouse and marks them as delivered.
- `StockLevel` **(4%)** is also a read-only transaction. It joins on average 200 order line items with their corresponding stock entries in order to produce a report.

The specification also lays out strict requirements about response time, consistency, and recovery in the system, and brings back the testing of an end-to-end system that includes network and terminal handling.

TPC-C stresses the entire stack (database system, operating system, and hardware) in several ways. First, it mixes short and long, read-only and update-intensive transactions, exercising a wider variety of features and situations than the TPC-B benchmark. In addition, the benchmark has major hotspots, partly due to the way transactions access the `Warehouse` table and partly due to the design of the `Delivery` transaction. The resulting contention and deadlocks stress the system's concurrency control mechanisms. Finally, the database grows throughout the benchmark run; not just because of the append-only `History` table as in TPC-B, but also because of the insert and delete operations on different tables, stressing code paths that the TPC-B benchmark did not reach.

TPC-C has been the most popular OLTP benchmark for over twenty years. Major database vendors have published results on TPC's website, and on several occasions they have used TPC-C for marketing purposes [86, 142].

2.4.3 The unexplored TPC-E

To represent real-life OLTP workloads more realistically, TPC presented TPC-E [193] as an alternative to the dominant TPC-C. This subsection gives an overview of TPC-E while pointing out its differences mainly from TPC-C.

Model

TPC-E models a brokerage house. The database tables keep information about customers, brokers, and a market. The transactions simulate a workload where either the customers initiate requests to the brokerage house (*customer initiated transactions*) or the market sends ticker feeds or trade results to the brokerage house (*market-triggered transactions*). The brokerage house responds to the customers, checks the orders to decide whether to submit them or not, submits the related brokerage requests (*brokerage initiated transactions*), and analyzes or updates the database. One could say that the TPC-E benchmark represents a more complicated business model compared to the TPC-C benchmark.

Database

TPC-E has more tables than TPC-C; thirty-three tables instead of nine:

- nine of TPC-E's tables are of *fixed* size,
- sixteen are *scaling* based on the number of `Customers`, and
- eight are *growing*.

However, the growth rate of the *growing* tables varies and, in general, it is greater than the growth rates of the *growing* tables in TPC-C. In addition, the TPC-E tables are populated with pseudo-real data and exhibit data skew. By contrast, TPC-C tables have randomly generated data that face a low degree of skew.

The scaling factor determines the number of `Branches` in TPC-B and the number of `Warehouses` in TPC-C. TPC-E has a scaling factor that controls the number of `Customers` in the database. However, unlike TPC-B and TPC-C, where a single scaling factor (via the number of `Branches` and `Warehouses`) is the only parameter that determines the initial size of the database, TPC-E has two additional parameters that affect the database size right after database population. In particular, the parameters called *working days* and *scaling factor* control the cardinality of the `Trade` table and in turn all the other *growing* tables in TPC-E. TPC-E also has a *growing* table, `Trade_Request`, which right after database population starts as an empty table and then grows. Neither TPC-B nor TPC-C has empty tables after the initial database population.

Table 2.1: TPC-E transactions.

Transaction	Weight	Access	Category	Frames Executed	% in Mix
BrokerVolume	Mid to Heavy	RO	BI	1 (out of 1)	4.9
CustomerPosition	Mid to Heavy	RO	CI	2/3 (out of 3)	13
MarketFeed	Medium	RW	MT	1 (out of 1)	1
MarketWatch	Medium	RO	CI	1 (out of 1)	18
SecurityDetail	Medium	RO	CI	1 (out of 1)	14
TradeLookup	Medium	RO	BI/CI	1 (out of 4)	8
TradeOrder	Heavy	RW	CI	2/5/6 (out of 6)	10.1
TradeResult	Heavy	RW	MT	5/6 (out of 6)	10
TradeStatus	Light	RO	CI	1 (out of 1)	19
TradeUpdate	Medium	RW	BI/CI	1 (out of 3)	2
BI: Brokerage Initiated, CI: Customer Initiated, MT: Market Triggered, RO: Read-Only, RW: Read-Write					

Transactions

TPC-E contains twelve transactions in total, which are shown in Table 2.1. Only ten of the transactions belong to the regular transaction mix. Two of them, `DataMaintenance` and `TradeCleanup`, get executed separately. `DataMaintenance` is executed periodically, every minute, alongside the transaction mix, whereas `TradeCleanup` needs to be executed before each run if one wants to clean up the submitted or pending trades from a previous run in order to restore the initial database state. In TPC-C, all the five transactions are included in the transaction mix.

The TPC-E transactions consist of *frames*, which are parts of a long transaction with a distinctive task. For some transactions only a subset of their frames are executed depending on the input values or whether they are initiated by a customer or brokerage; like in `TradeLookup` and `TradeUpdate`. TPC-C does not contain as complicated and long transactions. All transactions in TPC-C have only one frame.

Another significant distinction of TPC-E from its predecessors is that a majority of the transactions in the mix are Read-Only (*RO*). That is, in TPC-E around 75% of the transactions executed are read-only, whereas TPC-C has 92% Read-Write (*RW*) transactions in the mix.

TPC-E also enforces dependencies among some of its transactions. More specifically, the market-triggered transactions, `TradeResult` and `MarketFeed`, require the `TradeOrder` transactions to submit input for them. Therefore, they cannot be executed independently from the transaction mix. In TPC-C none of the transactions have such dependencies.

The TPC-E specification also introduces skew in transaction inputs, harness control measures within the transactions, and checks for referential integrity constraints, which do not exist in TPC-C. Moreover, for high performance, TPC-E needs to perform lookups and scans through non-primary indexes in almost all its transactions (ten out of twelve), whereas TPC-C uses secondary indexes in only two of its transactions.

Overall, TPC-E is a much more sophisticated OLTP benchmark compared to its predecessors and therefore, it offers a more interesting and mature environment for testing OLTP engines. On the other hand, it is also harder to adopt for people from both industry and academia, which have been optimizing their systems mainly based on TPC-C for the last twenty years.

Table 2.2: Evolution of TPC's OLTP benchmarks.

		TPC-A	TPC-B	TPC-C	TPC-E
First release		Nov 1989	Aug 1990	Aug 1992	Feb 2007
Last update		Jun 1994	Jun 1994	Feb 2010	Apr 2014
Business model		Banking	Banking	Wholesale supplier	Brokerage house
Tables	Fixed	0	0	1	9
	Scaling	3	3	4	16
	Growing	1	1	4	8
	Total	4	4	9	33
Transactions	RW	1	1	3	6
	RO	0	0	2	6
Transaction Mix %	RW	100%	100%	92%	23.1%
	RO	0%	0%	8%	76.9%
Transactions using secondary indexes		None	None	2	10
Data population		Random	Random	Random	Pseudo-real

2.4.4 The evolution summary

Table 2.2 summarizes the high-level comparison of the four OLTP benchmarks of TPC, which we detailed above. What we can conclude from this section and Table 2.2 is that with each benchmark TPC standardized, we see a significant complexity increase, which is driven by the facts listed below:

- A more sophisticated business model.
- A larger variety of transaction types.
- Longer-running and less deterministic transactions, causing longer and less predictable instruction streams.
- Increase in the number of read-only transactions that need to be run together with update-heavy ones.
- Increase in the number of scan operations and dependency on the secondary indexes, which makes physical database partitioning less effective.

- More fundamental stress within the storage manager and exploration of an increased number of code-paths.

The above items are going to be crucial while explaining the behavior of these workloads within a storage manager and micro-architecturally in Part II.

2.5 The TATP benchmark

In addition to the benchmarks TPC provides, the TATP benchmark [137] is another widely-used OLTP benchmark in the database community. The benchmark was originally designed by Nokia Networks and called TM1. The goal was to create a benchmark to test Nokia's own infrastructure. Therefore, the TATP benchmark simulates the actions of a telecommunication business; e.g., call forwarding, retrieving/updating a subscriber's information, etc.

The database has 4 tables. All of them are *scaling* tables; their cardinality is proportional to the number of `Subscribers`. For each `Subscriber`, there are ~2.5 `Access_Info`, ~2.5 `Specifical_Facility`, and ~3.75 `Call_Forwarding` records. Therefore, it is very straightforward to partition each table based on the `Subscriber` ids. On the other hand, only the `Call_Forwarding` table observes insert/delete operations during the execution of the workload mix. Since the inserts/deletes to this table are at the same rate, however, the table's (and hence the database's) size does not change visibly after the database population.

The TATP transaction mix consists of three read-only and four read-write transactions, which are 80% and 20% of the mix, respectively. These transactions are very short compared to the ones in TPC's OLTP benchmarks; at most four database records are accessed in one transaction. In addition, except for the two transactions that only touch the `Subscriber` table, the TATP transactions exhibit very high abort rates leading to an abort rate of 25% in the workload mix. The details of the TATP transactions and their frequencies in the mix are as follows:

- `GetSubscriberData` **(35%)** is one of the read-only transactions in the mix and never fails. It just probes for one `Subscriber` in the database and retrieves its information (e.g., number, location, etc.).

- `GetNewDestination` **(10%)** is another read-only transaction in the mix. Its goal is to fetch the information regarding a `Subscriber`'s call forwarding destination. However, it aborts ~76% of the time due to a `Subscriber` not having an active call forwarding request.

- `GetAccessData` **(35%)** is the last read-only transaction of TATP and returns the access validation information of a `Subscriber`. It can abort with a rate of 37.5%.

- `UpdateSubscriberData` **(2%)** just updates a `Subscriber`'s profile information. It also has an abort rate of 37.5%.

- `UpdateLocation` **(14%)** updates a `Subscriber`'s location information without any aborts.

- `InsertCallForwarding` **(2%)** adds call forwarding information for a `Subscriber` and has a ~30% failure rate.
- `DeleteCallForwarding` **(2%)** removes a call forwarding request from a `Subscriber` and also has a ~30% failure rate.

Due to the short nature, low application logic, and high failure rate of its transactions, the TATP workload mix spends majority of its execution time stressing the internals of a storage manager.

Currently, the IBM Corporation maintains the TATP benchmark.

2.6 Shore-MT and Shore-Kits: Benchmarks on Top of Shore-MT

This thesis uses the Shore-MT storage manager to prototype the proposed ideas and perform workload characterization and hardware simulation studies. Shore-MT [7, 172] is an enhanced version of the SHORE storage manager [28], whose micro-architectural behavior is very close to that of commercial disk-based database management systems [4, 6]. Shore-MT adds a multithreaded storage manager kernel to SHORE and is particularly developed to adapt SHORE to the multicore era, mainly by focusing on eliminating scalability bottlenecks when running on multicore hardware [96]. Today, Shore-MT is one the most scalable open-source shared-everything storage managers within a single database node [96, 102]. It has been used in various research projects as a test-bed both by the team who develops and maintains it [99, 145, 154, 155, 186] and by other well-known teams in the database and computer architecture communities [62, 102, 144, 152, 168].

In order to study the behavior and challenges the standardized OLTP benchmarks pose on modern storage managers, we implement them on top of Shore-MT and distribute them as an open-source suite of database benchmarks, called Shore-Kits. Since Shore-MT does not have an SQL front end, a query parser, and an optimizer, the benchmarks are implemented in C++ using direct calls to Shore-MT's storage manager API, which is linked as a static library to the executable. With some programming effort and code refactoring, one can port Shore-Kits to other storage managers by changing the API calls to match the target storage manager's API.

Both Shore-MT and Shore-Kits are available at [171]. The latest online version of Shore-MT incorporates the techniques proposed in [95, 97, 145, 147], whereas Shore-Kits provides the TPC-B [190], TPC-C [191], TPC-E [193], and TATP [137] benchmarks for transaction processing and TPC-H [195] and SSB [139] benchmarks for analytical applications.

Part I

Scalable and Dynamically Balanced Shared-Everything OLTP with Physiological Partitioning

3 Latch-free Shared-everything OLTP

As the previous chapters discuss, scaling the performance of shared-everything transaction processing systems to highly-parallel multicore hardware is a challenge for database system designers. In this chapter, we initially analyze the scalability of a conventional shared-everything transaction processing system through an analysis of its critical sections to determine the unscalable/unbounded communication points. This analysis identifies page latching as a significant source of unbounded communication in conventional transaction processing. Then, with the goal of minimizing the unbounded communication in mind, we propose physiological partitioning (PLP). PLP applies logical-only partitioning, maintaining the desired properties of shared-everything designs, and introduces a multi-rooted B+Tree (MRBTree) index structure. Logical partitioning and MRBTrees together enable the partitioning of the accesses to both database records and pages, which minimizes the unbounded communication due to locking and latching. Profiling a PLP prototype running on different multicore machines shows that PLP acquires 90% and 75% fewer critical sections than an optimized conventional design and a design based on logical-only partitioning, respectively. As a result, PLP also improves performance up to 50% over the existing designs. [1]

3.1 Introduction

Due to concerns over power draw and heat dissipation, processor vendors can no longer rely on rising clock frequencies or increasingly aggressive micro-architectural techniques to boost performance. Instead, they focus on parallelism by placing many independent processing cores in each chip. The resulting multicore designs require software to expose enough execution parallelism in order to exploit the abundant and rapidly growing hardware parallelism. However, this is a challenging task. Conventional systems tend to have application threads that exhibit high resource sharing with each other since they are not designed with increasing hardware parallelism in mind. The coordination of accesses to these shared resources prevents systems from exploiting the multicores.

[1] This chapter uses material from [147, 185].

Online transaction processing (OLTP) is a particularly complex data management application that needs to perform efficiently on modern hardware. Previous studies show that conventional shared-everything OLTP systems face major scalability problems while running on highly parallel hardware [96]. One significant source of scalability problems is the conventional transaction-oriented work assignment policy, which assigns each transaction as a whole to a single worker thread (mostly randomly) [145]. The transaction, along with the physical arrangement of the records within the data pages, determines which resources (e.g., records and pages) each thread accesses.

The random nature of transaction processing requests leads to unpredictable data accesses [145, 174] that complicate resource sharing. Such unpredictability favors pessimistic policies while protecting the consistency of the data and isolation among transactions, which clutter the execution path of a transaction with many lock and latch acquisitions. These critical sections often lead to *contention* that limits scalability [96] and in the best case imposes a significant penalty to single-thread performance [76]. In addition, the performance of shared-everything systems is vulnerable to the *false sharing* of database pages, where hot but unrelated records happen to reside on the same page. Careful tuning is often needed to detect and resolve such issues; e.g., padding problematic records to spread them out.

Following a different approach, shared-nothing systems deploy many independent database instances that collectively serve the workload [45, 178]. In shared-nothing designs, the contention for the shared data resources can be explicitly tuned; i.e. the database administrator (DBA) can determine the number of processors assigned to each database instance. Such designs potentially lead to superior performance as long as inter-instance communication is not excessive. Systems like H-Store [103, 179] (or it is commercial version VoltDB [199]) and Hyper [107] take this approach to the extreme, with single-threaded database instances that remove critical sections altogether when there is no inter-instance communication. However, shared-nothing systems physically partition the data and deliver poor performance when the workload triggers distributed transactions [39, 77, 149] or when skew causes load imbalance [39, 150, 153]. Repartitioning to minimize distributed transactions or balance load requires the system to physically move and reorganize all affected data. These weaknesses become especially problematic as partitions become smaller and more numerous in response to the increasing multicore parallelism.

3.1.1 Multi-rooted B+Trees

To alleviate the difficulties imposed by page latching and repartitioning, we propose a new physical access method, a type of multi-rooted B+Tree called *MRBTree*. The root of each subtree in this structure corresponds to a logical partition of the data, and the mapping of the key ranges to subtree roots forms the durable part of the index's metadata. Partition sizes are non-uniform, making the tree robust against skewed access patterns, and repartitioning is cheap because it involves very little data movement.

When deployed in a conventional shared-everything system, the MRBTree eliminates latch contention at the index root; i.e., fewer threads access the same index root concurrently. Furthermore, the MRBTree can also benefit systems that use shared-nothing parallelism in a shared-memory environment (e.g., H-Store [179]).

3.1.2 Physiological Partitioning

Recent work proposes logical-only partitioning [145] to address problems with conventional execution while avoiding the weaknesses of shared-nothing approaches. Logical-only partitioning assigns each partition to one worker thread to manage the data locally without the overhead of centralized locking. However, logical partitioning alone neither prevents the conflicts due to false sharing of database pages nor addresses the overhead and complexity of the page latching protocols.

Ideally, we would like to have a system with the best properties of both shared-everything and shared-nothing designs: a centralized data store that sidesteps the challenges of moving data during (re)partitioning and a partitioning scheme that eliminates contention and the need for page latches.

This chapter presents *physiological partitioning (PLP)*, a transaction processing design that logically partitions the physical data accesses to alleviate the difficulties imposed by page latching. While achieving its goal, PLP uses the MRBTree indexes to enhance logical partitioning and enable partitioned physical data accesses in a shared-everything infrastructure. A partition manager assigns threads to subtrees of the MRBTrees and ensures that requests distributed to each thread reference only the corresponding subtree. As a result, threads can bypass the partition mapping and their accesses to the subtrees are entirely latch-free. In addition, PLP can easily extend the partitioning down to the heap pages where non-clustered records are stored, eliminating another class of page latching (similar to shared-nothing systems).

3.1.3 Contributions and Organization

The structure and contributions of the remaining of this chapter is as follows:

- Section 3.2 categorizes the communication patterns in traditional transaction processing. This categorization highlights the unbounded critical sections that create latent scalability bottlenecks, which might surface with any new generation of multicore hardware since their effect is proportional to the available hardware parallelism. We also identify page latching as one such bottleneck in OLTP.

- Section 3.3 shows that deploying a design based on physiological partitioning can eliminate the unbounded critical sections due to both locking and page latching during transaction execution within a shared-everything OLTP system.

- Section 3.4 evaluates a prototype implementation of PLP. PLP acquires 90% and 75% fewer contentious critical sections per transaction, respectively, than an optimized conventional design and logical partitioning. As a result, PLP improves scalability and yields up to ~50% higher performance on multicores.

Finally, while PLP advances the state-of-the-art design options for OLTP systems as discussed in Section 3.5, it has some limitations as well, which we detail in Section 3.6. Nevertheless, we conclude by promoting PLP as a very promising OLTP system design in the light of the upcoming hardware trends in Section 3.7.

3.2 Communication Patterns

Traditional transaction processing systems excel at providing high concurrency, or the ability to interleave multiple concurrent requests or transactions over limited hardware resources. However, as core counts increase exponentially, performance increasingly depends on execution parallelism, i.e., the ability for multiple requests to make forward progress simultaneously in different execution contexts. Even the smallest of serializations on the software side therefore impact scalability and performance [81]. Unfortunately, recent studies show that high concurrency in transaction processing systems does not necessarily translate to sufficient execution parallelism [96, 97] due to the high degree of irregular and fine-grained communication they exhibit.

As Section 2.3.1 mentions, proposals to tackle overhead and scalability bottlenecks fall into two general categories:

- reducing the degree of communication and contention within shared-everything systems, relying on efficient communication via shared caches to keep synchronization overhead low; and
- taking a shared-nothing approach [178], relying on the low-latency of multicore hardware to keep overhead manageable in spite of the challenges that accompany distributed transactions and load balancing.

In this section we first categorize the types of communication that can occur in an OLTP system, and from this point of view we analyze the execution of a modern shared-everything system. Then, we revisit the debate between the shared-everything and shared-nothing approaches.

3.2.1 Types of Communication

OLTP systems employ several types of communication and synchronization. *Database locking* operates at the logical (application) level to enforce isolation and atomicity between transac-

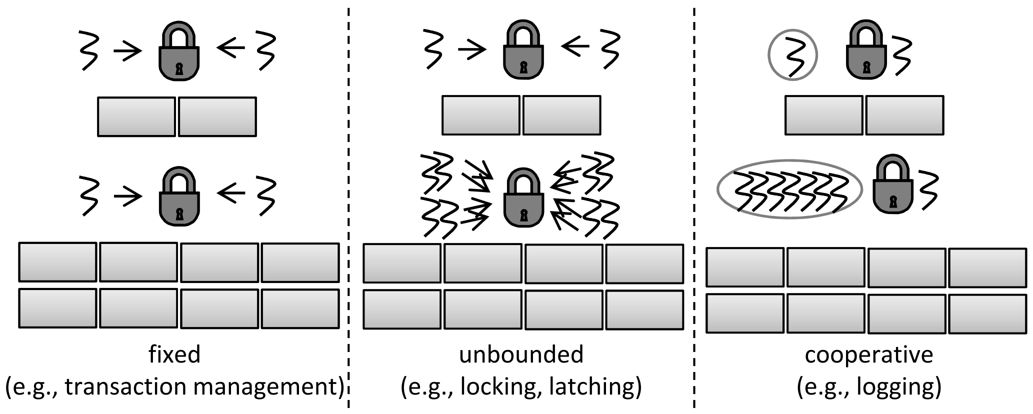


Figure 3.1: Categorization of the critical sections of OLTP based on the type of contention they create as the parallelism increases.

tions. *Page latching* operates at the physical (database page) level to enforce the consistency of the physical data stored on disk in the face of concurrent updates from multiple transactions. Finally, at the lowest levels, *critical sections* protect various code paths that must execute serially to protect the consistency of the system's internal state. Critical sections are traditionally protected by mutex locks, atomic instructions, etc. We note that locks and latches, which form a crucial part of the systems' internal state, are themselves protected by critical sections. Therefore, analyzing the behavior of critical sections captures nearly all forms of communication in the DBMS.

Critical sections, in turn, fall into three categories depending on the nature of the contention they tend to trigger in the system. Figure 3.1 illustrates these categories. For example, pairs of threads that form producer-consumer pairs protect their communication with a critical section but cannot generate significant contention. We refer to these as *fixed* critical sections (leftmost part of Figure 3.1) because contention is independent of the underlying hardware and depends only on the (fixed) number of threads that communicate. At the other extreme, *unbounded* critical sections (middle part of Figure 3.1) have the highly undesirable tendency to affect most threads in the system and lead to unbounded contention. As hardware parallelism increases the degree of contention also increases and inevitably grows into a bottleneck. Making these critical sections shorter or less frequent provides a little slack but does not fundamentally improve scalability. Finally, Moir et al. [130] introduce the notion of *cooperative* critical sections (rightmost part of Figure 3.1), those having the property that multiple threads can aggregate their operations. Cooperative critical sections are highly resistant to contention because threads take advantage of queuing delays to combine their requests and drop out of the queue. The critical section is thus self-regulating: adding more threads to the system gives more opportunity for threads to combine work rather than competing directly for the critical section.

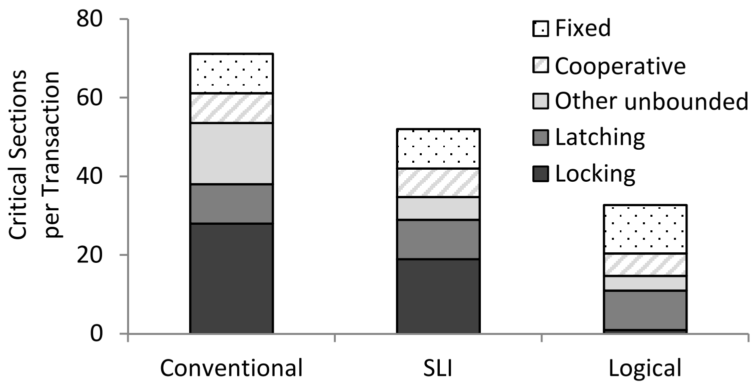


Figure 3.2: Breakdown of the critical sections for conventional, optimized conventional (SLI), and logically-partitioned shared-everything OLTP designs when running the `UpdateLocation` transaction of TATP.

3.2.2 Communication Patterns in OLTP

As the previous section hints, the real key to scalability lies in converting all unbounded communication to either the fixed or cooperative type, thus removing the potential for bottlenecks to arise. The three bars of Figure 3.2 compare the number and types of critical sections executed by a conventional OLTP system (labeled as *Conventional*) and two others designed to reduce contention due to locking: speculative lock inheritance [95] and data-oriented execution with logical-partitioning [145] (labeled as *SLI* and *Logical*, respectively). Each bar shows the average number of critical sections entered as the system runs 10000 instances of the `UpdateLocation` transaction of the TATP benchmark [137]. The critical sections are categorized based on the classification in Figure 3.1 by looking at which storage manager component triggers them (details about the storage manager are in Section 3.4.1).

Locking and latching form a significant fraction of the total communication for the baseline system. SLI exploits the observation that almost all transactions request high-level locks (e.g., table-level locks) in compatible modes and allows the worker threads to inherit such locks from one transaction to another without releasing them. Therefore, SLI reduces trips/requests to the lock manager for the common case and achieves a performance boost by sidestepping the most problematic critical sections associated with the lock manager. However, it fails to address the remaining (still-unbounded) communication in that category. Logical partitioning, in contrast, eliminates nearly all types of locking, replacing both contention and overhead of centralized communication with efficient, fixed communication via message passing. With locking removed, latching remains by far the largest source of critical sections. There is no predefined limit to the number of threads that might attempt to access a given page simultaneously, so page latching represents an unbounded form of communication, which should be eliminated or converted to a scalable type. The remaining categories represent either fixed communication (e.g., transaction management), cooperative operations (e.g., logging [97]), or a minor fraction of the total unbounded component (e.g., buffer pool).

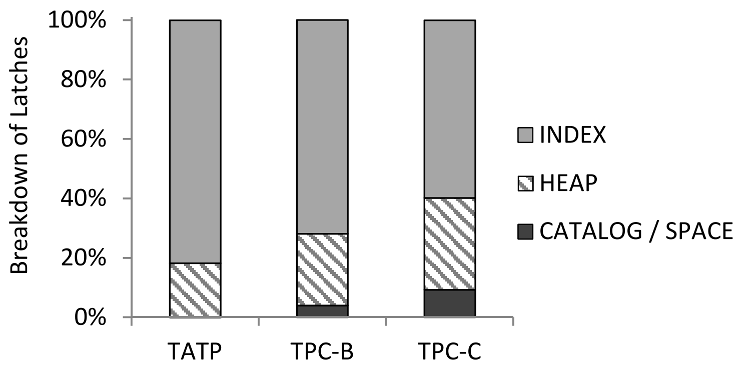


Figure 3.3: Breakdown of the page latches based on the page types they latch using TATP, TPC-B, and TPC-C benchmarks.

Examining page latching more closely, Figure 3.3 decomposes the page latches acquired by three popular OLTP benchmarks (TATP [137], TPC-B [190], and TPC-C [191]) into the different types of database pages: pages that keep the index on database records (*index*), pages that keep the database records (*heap*), pages that keep the metadata information (*catalog*). The figure demonstrates that the majority of the page latches (60%-80%) reside in index structures whereas the heap page latches are the next non-negligible component, accounting for nearly all the remaining page latches.

3.2.3 Physical vs. Logical Partitioning

With the preceding characterization of communication patterns in mind, we now return to the question of logical partitioning (shared-everything) vs. physical partitioning (shared-nothing). As its name suggests, logical partitioning eliminates unbounded communication at the logical level, namely database locking. However, it has little impact on the remaining communication, which arises in the physical layers and cannot be managed cleanly from the application level. Even when requests do not communicate at the application level, threads must acquire page latches and potentially perform other unbounded communication.

Shared-nothing systems [45, 178], on the other hand, are an appealing design, giving the designer explicit control over the number of threads per instance. Therefore, the contention on each component of the system can be controlled or even eliminated. However, such designs give up too much by eliminating all communication within the engine. Even the cooperative and fixed types of critical sections, which do not threaten scalability, become problematic. For example, logging is not amenable to distribution [98], and physically-partitioned systems either use a shared log [123] or eliminate it completely [179].

In addition, one of the biggest challenges for shared-nothing systems arises due to distributed transactions when requests access data from multiple physically distributed database instances [153]. The scalable execution of distributed transactions has been an active field of

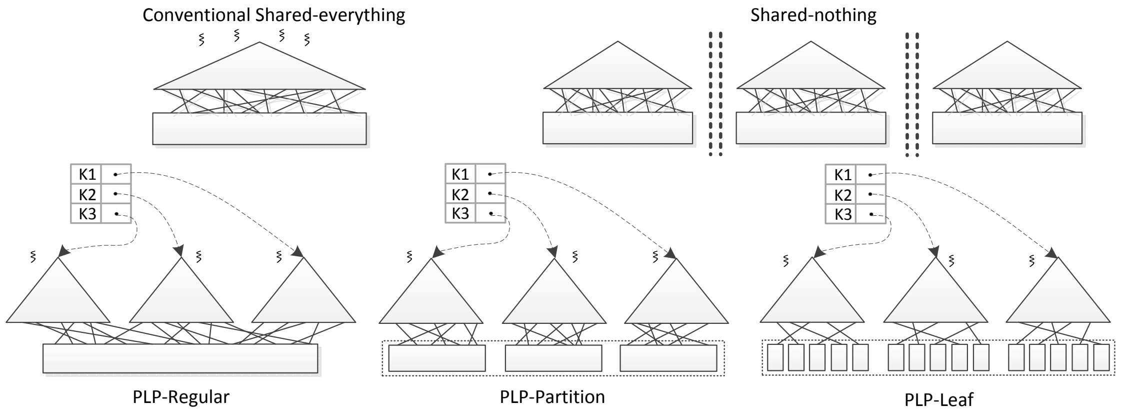


Figure 3.4: The conventional shared-everything and shared-nothing designs and the variations of physiological partitioning.

research for the past three decades, with researchers from both academia and industry persuasively arguing that they are fundamentally not scalable [27, 77]. Furthermore, the performance of shared-nothing systems is very sensitive to imbalances in load arising from data or access skew across different physical instances [39, 150] while non-partition aligned operations (such as non-clustered secondary indexes) may pose significant barriers to physical partitioning.

3.3 Physiological Partitioning

We have seen how both logically- and physically-partitioned designs offer desirable properties, but also suffer from weaknesses that threaten their scalability. In this work we therefore propose physiological partitioning (or PLP), a hybrid of the two approaches that combines the best properties of both. Like a physically-partitioned system a majority of physical data accesses occurs in a single-threaded environment, which obviate the need for page latching; like the logically-partitioned system, locking is distributed without resorting to distributed transactions and load balancing requires almost no data movement.

3.3.1 Design Overview

Each transaction in a typical OLTP workload accesses a very small subset of records via indexes (sequential scans are prohibitively expensive). PLP therefore centers around the indexing structures of the database. Figure 3.4 gives a high-level overview of a physiologically-partitioned system. We adapt the traditional B+Tree [18] (top left of Figure 3.4) for PLP by splitting it into multiple subtrees, each covering a contiguous subset of the key space (bottom part of Figure 3.4). A *partitioning table* becomes the new *root* and maintains the partitioning as well as pointers to the corresponding subtrees. We call the resulting structure a *multi-rooted B+Tree (MRBTree)*. The MRBTree partitions the data but unlike a horizontally-partitioned workload (top right of Figure 3.4), all subtrees belong to the same database file and can

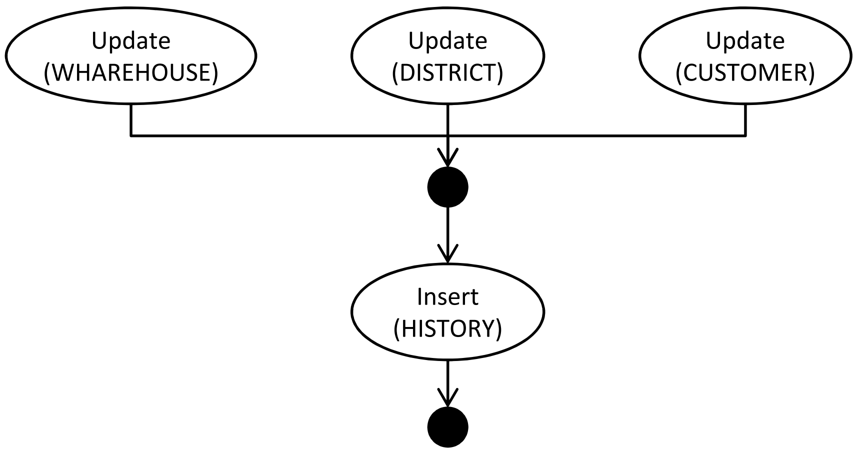


Figure 3.5: Transaction flow graph of TPC-C's `Payment` under PLP. Each node represents an action executed by the transaction and the synchronization points (dark circles) coordinate these actions based on the data dependencies among them.

exchange pages easily; the partitioning, though durable, is dynamic and malleable rather than static.

With the MRBTree in place, the system assigns each subtree to a single thread, guaranteeing exclusive access for latch-free execution. A *partition manager* layer controls all partition tables and makes assignments to threads. The threads in PLP do not reference partition tables during normal processing, which might otherwise become a bottleneck. Instead, the partition manager ensures that all work given to a thread involves only the data it owns.

The partition manager breaks transactions into directed graphs, passing each node to the appropriate thread and assembling the results into complete transactions. Figure 3.5 illustrates the transaction flow graph of TPC-C's `Payment` transaction under PLP. Each node represents an action executed by the transaction. For example, the node labeled *Update(WAREHOUSE)* indicates the action of updating a record in TPC-C's `Warehouse` table after performing an index lookup for that record. Each action reports to a synchronization point once it is completed. The synchronization points maintain the data dependencies across different actions. The final synchronization point informs all the participating worker threads (or partitions) after committing a transaction so that these threads can release the partition-local locks they acquired for each database record they accessed as part of the committed transaction. These worker threads, however, can continue working on other non-conflicting transactions once they are finished with the action they are responsible from in the current transaction. They do not have to wait for the current transaction to commit to perform other work since the partition-local lock managers ensure isolation among transactions [145].

PLP assigns different set of worker threads to each table. Therefore, if two actions are data independent, they can run in parallel (e.g., the top three nodes of Figure 3.5). Since actions on different tables are handled by different set of worker threads, whenever a transaction touches more than one table it becomes a multisite transaction under PLP. However, multisite transactions are not expensive as in a shared-nothing system since the state information for the participating sites is much less in a shared-everything environment; the synchronization points keep only pointers for the participating actions and the data that needs to be passed from one action to another.

All indexes in the system (primary, secondary, clustered, non-clustered) can be implemented as MRBTrees; data are stored directly in clustered indexes, or in tightly integrated heap file pages referenced by record ID. When the system can infer partitions from secondary (non-clustered) index columns, the partition's thread manages them directly (e.g., when the columns used for partitioning form a prefix of the secondary index columns). The remaining (non-partition aligned) secondary indexes are accessed as in the conventional system. However, the leaf pages of the secondary index also keep the columns used for partitioning for each data entry. Therefore, the result of each secondary index probe can be passed to the thread that owns the partition of the probed data for further processing.

3.3.2 Multi-rooted B+Tree

The *root* of an MRBTree is a partition table that identifies the disjoint subsets of the key range assigned to each subtree as well as a pointer to the root of each tree. Because the routing information is cached in memory as a ranges map by the partition manager, its on-disk layout favors simplicity rather than optimal access performance. We therefore employ a standard slotted page format to store key-root pairs. If the partitioning information cannot fit on a single page (for example, if the number of partitions is high or the keys are very long) the routing page is extended as a linked list of routing pages. In our experiments we have never encountered the need to extend the routing page as several dozen mappings fit easily in 8KB, even assuming rather large keys.

Record insertion (deletion) takes place as in a regular B+Tree. When the key to insert (delete) is given, the ranges map routes it to the subtree that corresponds to the key range the key belongs to and the insert (delete) operation is performed as in a regular B+Tree in that subtree. The other subtrees, ranges map, and the routing page are not affected by the insert (delete) operation at all.

When deployed in a conventional shared-everything system, the MRBTree eliminates latch contention at the index root; fewer threads attempt to grab the latch for the same index root at a time. Partitioning also reduces the expected tree level by at least one, which reduces the index probe time. Moreover, the MRBTree can also potentially benefit systems that use shared-nothing parallelism in a shared-memory environment (e.g., H-Store [179]).

3.3.3 Heap Page Accesses

In PLP a heap file scan is distributed to the partition-owning threads and performed in parallel. Large heap file scans reduce the concurrency of OLTP applications and PLP has little to offer. Still, heap page management opens up additional design options, since we can extend the partitioning of the accesses to the heap pages. That is, when records reside in a heap file rather than in the MRBTree leaf pages, PLP can ensure that accesses to pages are partitioned in the same way as index pages.

We propose three options on how to place and access records in the heap pages, depicted in the bottom part of Figure 3.4:

- *PLP-Regular* keeps the existing heap page design,
- In *PLP-Partition*, each heap page keeps records of only one logical partition, and
- In *PLP-Leaf*, only one leaf page of the primary MRBTree points to a particular heap page.

PLP-Regular simply keeps the existing heap page operations. Without any modification, the heap pages still need to be latched because they can be accessed by different threads in parallel. This may be acceptable because heap page accesses are not the biggest fraction of the total page accesses in OLTP (as low as 30% according to Figure 3.3). Thus, there is room for significant improvement even if we ignore them. However, allowing heap pages to span partitions prevents the system from responding automatically to false sharing or other sources of heap page contention.

In PLP-Partition and PLP-Leaf, the MRBTree and heap operations are modified so that heap page accesses are partitioned as well. The difference between the two is that in PLP-Partition a heap page can be referenced by many MRBTree leaf pages as long as all the pages belong to the same partition, while in PLP-Leaf a heap page is referenced by only one MRBTree leaf page.

Both variations provide latch-free heap page accesses, but they suffer from some disadvantages. Forcing a heap page to contain records that belong to a specific partition causes fragmentation. In the worst case, each leaf has room for one more entry than fits in the heap page, almost doubling the total space requirement (Section 3.4.8 measures this cost). Furthermore, in PLP-Leaf every leaf page split must also move the records that are referenced by the new leaf page to a new heap page, increasing the overhead of record insertion (deletions are simple because a leaf page may point to many heap pages). On the other hand, PLP-Partition, by allowing multiple leaf pages from a partition to share a heap page, forces the system to reorganize potentially significant numbers of heap pages with every repartitioning. Significant reorganization costs go against the philosophy of physiological partitioning, so we favor PLP-Leaf.

The two extensions impose one additional piece of complexity: During record insertion, the system must identify the correct MRBTree entry before selecting a heap page for the record. Because the storage management layer is completely unaware of the partitioning strategy (by design), it must make callbacks into the upper layers of the system to identify an appropriate heap page for each insertion.

Similarly, a partition split may split heap pages as well, invalidating the record IDs of migrated records. The storage manager, therefore, exposes another callback so the metadata management layer can update indexes and other structures that reference the stale record IDs. We note that when PLP-Leaf splits leaf pages during record insertion, the same kinds of record relocations arise and the same callbacks are used.

3.3.4 Page Cleaning

In conventional systems, there is a set of background threads that periodically traverse the whole buffer pool to write the dirty pages back to stable storage. This process is called page cleaning. Those threads may access arbitrary pages in the buffer pool, which breaks the invariant of PLP where a single thread can access a page at each point of time.

To handle the problem of page cleaning in PLP, each thread does the page cleaning for its logical partition. Each logical partition has an additional input queue for system requests and the page cleaning requests go to that queue. The system queue has higher priority than the queue of completed actions. Their execution is not delayed by more than the execution time of one action (typically very short since an action is part of a transaction). In addition, since page cleaning is a read-only operation, the thread can continue to work (and even re-dirty pages) during the write-back I/O.

3.3.5 Benefits of Physiological Partitioning

Under physiological partitioning, each partition is permanently locked for exclusive physical access by a single thread, which then handles all the requests for that partition. This allows the system to avoid several sources of overhead as described below.

Latching contention and overhead

Though page latching is inexpensive compared with acquiring a database lock, the sheer number of page latches acquired imposes some overhead and can serialize B+Tree operations as transactions crab down the tree during a probe. The problem becomes more acute when the lower levels of the tree do not fit in memory, because a thread that fetches a tree node from disk holds a latch on the node's parent until the I/O completes, which might be preventing access to 80-100 mostly memory-resident siblings. Section 3.4.3 evaluates a case where latching becomes expensive for B+Tree operations and how PLP eliminates this problem by allowing latch-free accesses on index pages.

False sharing of heap pages

One significant source of latch contention arises when multiple threads access unrelated records that reside on the same physical database page. In a conventional system false sharing requires padding to force problematic database records to different pages. PLP variations that allow latch-free heap page accesses achieve the same effect automatically (without the need of expensive tuning) as they split hot pages across multiple partitions. Section 3.4.3 evaluates this case as well.

Serialization of structural modification operations

The traditional ARIES/KVL indexes [128] allow only one structural modification operation (SMO), such as a leaf split/merge, to occur at a time, serializing all other accesses until the SMO completes. Partitioning the tree physically with MRBTrees eases the problem by distributing SMOs across subtrees (whose roots are fixed) without having to apply more complicated protocols, as such those described in [92, 129]. The benefits of parallel SMOs are apparent in the case of insert-heavy workloads, which we evaluate in Section 3.4.5.

Repartitioning

In PLP, repartitioning can occur at a higher level in the partition manager and therefore can be latch-free as well; the partition manager can simply halt affected threads until the process completes. Moreover, it can be performed very efficiently as it requires very few pointer updates and data movement as the next chapter (Chapter 4) demonstrates.

Code complexity

Finally, with all latching eliminated, we can also eliminate the code paths that handle contention and failure cases as well, simplifying the code significantly. In the end, the index can be substituted with a much simpler implementation. For example, a huge source of complexity in traditional B+Trees arises due to the sophisticated protocols that maintain consistency during an SMO in spite of concurrent probes from other threads. The simpler code is not only more efficient but also easier to maintain. In this chapter, we do not attempt to perform the code refactoring needed to exploit these opportunities and the performance results we report are therefore conservative. However, we note that B+Tree probes are the most expensive remaining component of PLP. Therefore, we expect significant performance improvements if, for example, we substitute the B+Tree implementation of our prototype with a cache-conscious [162, 163] and/or prefetching-based [32] B+Tree.

3.4 Evaluation

The evaluation consists of three parts.

- The first part measures how useful PLP can be. In particular, Section 3.4.2 quantifies how different designs impact page latching and critical section frequency, Section 3.4.3 examines how effectively PLP reduces latch contention on index and heap page latches, and Section 3.4.4 shows the performance impact of those changes.
- The second part (Section 3.4.5), quantifies how useful MRBTrees can be also for conventional and logically-partitioned systems.
- The third part analyzes the possible overhead of PLP. To do that we demonstrate PLP's behavior under challenging workloads that seem not to fit well with physiological partitioning, such as transactions that require joins (Section 3.4.6) and secondary index accesses (Section 3.4.7). In addition, Section 3.4.8 inspects the fragmentation overhead of the three PLP variations.

Finally, Section 3.4.9 highlights the key conclusions of the whole evaluation.

3.4.1 Experimental Setup

To ensure reasonable comparisons, all the prototypes are built on top of the same version of the Shore-MT storage manager [96, 172], incorporate the logging optimizations of [97], and share the same driver code.

We consider five different designs:

- An optimized version of a conventional, non-partitioned system, labeled as *Conventional* or *Conv.*. This system employs speculative lock inheritance [95] to reduce the contention due to locking.
- *Logical* is a data-oriented transaction processing prototype [145] that applies logical-only partitioning.
- *PLP or PLP-Regular* prototypes the basic PLP variation. This variation accesses the MRBTree index pages without latching.
- *PLP-Partition* extends PLP-Regular, so that one logical partition *owns* each heap page, allowing latch-free index and heap page accesses.
- *PLP-Leaf* assigns heap pages to leaves of the primary MRBTree index, also allowing latch-free index and heap page accesses.

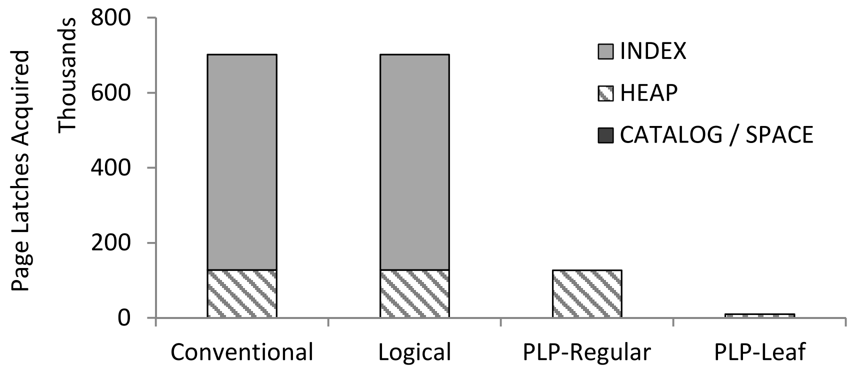


Figure 3.6: Average number of page latches acquired per transaction by different designs when running the TATP workload mix.

All experiments are performed on two machines: an x64 box, with four sockets of quad-core AMD Opteron 8356 processors, clocked at 2.4GHz and running Red Hat Linux 5; and a Sun UltraSPARC T5220 server with a 64-core Sun Niagara II chip clocked at 1.4GHz and running Solaris 10. Due to unavailability of a suitably fast I/O sub-system, all experiments are with memory-resident databases. But the relative behavior of the systems will be similar with larger databases.

To get accurate time breakdowns within the storage manager, we profile our system using the DTrace [48] framework on the SPARC machine. The profiler takes 7777 samples within a microsecond and we report the breakdown based on these sample counts. Therefore, we do not have a conventional time unit on the y-axes of the time breakdown graphs. However, please note that, the relative time is sufficient to analyze these graphs.

Finally, the number of partitions for each table in the evaluated benchmarks is equal to the number of hardware contexts available on the machine used for a particular experiment and the load is balanced across partitions (Chapter 4 targets the problem of load imbalance).

3.4.2 Page Latches and Critical Sections

First we measure how PLP reduces the number of page latch acquisitions in the system. Figure 3.6 shows the number and type of page latches acquired per transaction by the conventional, logically-partitioned, and two PLP design variations: PLP-Regular and PLP-Leaf. Each system executes the same number of transactions from the transaction mix of the TATP benchmark.

Since logical-partitioning does not target page latches, it acquires the same number of page latches as the conventional design. On the other hand, PLP-Regular reduces the amount of page latching per transaction by more than 80%, whereas PLP-Leaf eliminates almost all the page latching required in the conventional system. The remaining latches are associated with metadata and free space management.

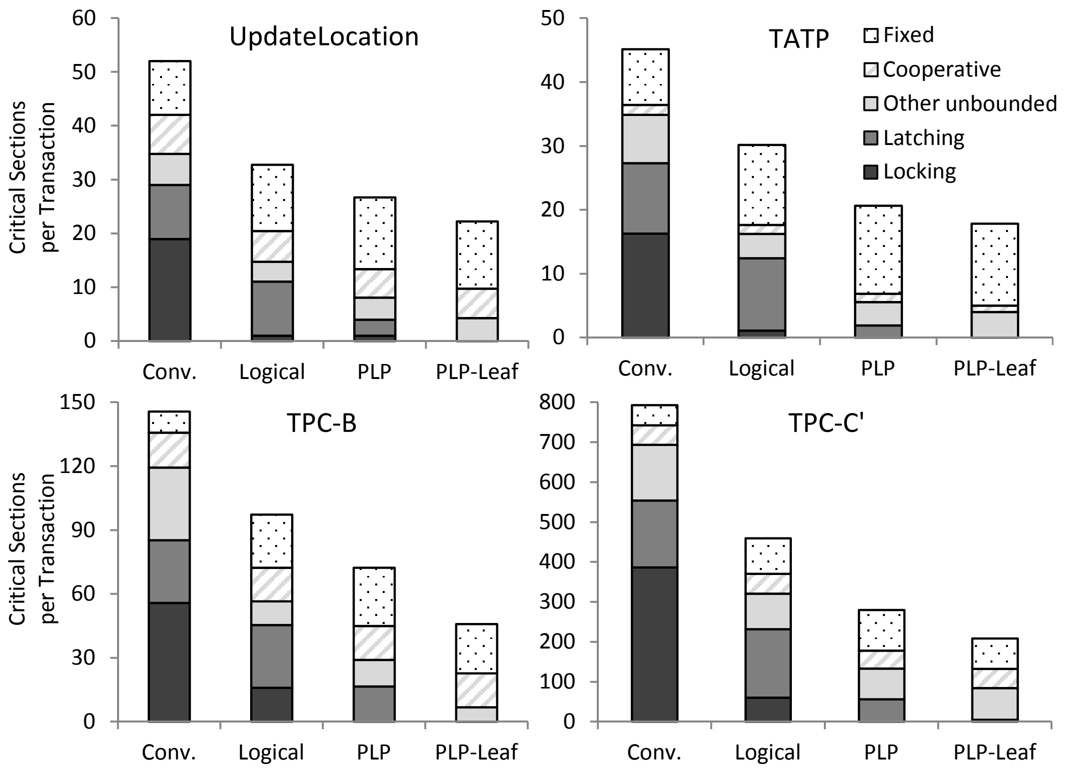


Figure 3.7: Breakdown of the critical sections for different shared-everything OLTP system designs when running the `UpdateLocation` transaction of TATP, TATP workload mix, TPC-B workload mix, and a 50%-50% mix of the `NewOrder` and `Payment` transactions of TPC-C.

Figure 3.7 compares the number and types of critical sections entered per transaction under *Conventional*, *Logical*, *PLP-Regular*, and *PLP-Leaf* as we run 10000 transactions from the TATP, TPC-B, and TPC-C workload mixes. The TPC-C mix consists of only the `NewOrder` and `Payment` transactions. Therefore, it is marked as TPC-C' (Section 3.6 explains the reasoning behind this setup). Figure 3.7 also includes a graph with the results for TATP's `UpdateLocation` transaction, which is an extended version of Figure 3.2.

The two PLP variants eliminate the vast majority of locking- and latching-related critical sections. PLP-Regular eliminates all the latching on index pages whereas PLP-Leaf eliminates the remaining latching related critical sections. The largest remaining component of the critical sections comes from the transaction manager. This component mostly employs fixed-contention communication to serialize threads that attempt to modify the transaction object's state. Similarly, the buffer pool-related critical sections are mostly due to the communication between cleaner threads, which again do not impact scalability. Overall, on average, PLP-Leaf acquires 90% and 75% fewer contentious critical sections than the conventional and logically-partitioned systems, respectively.

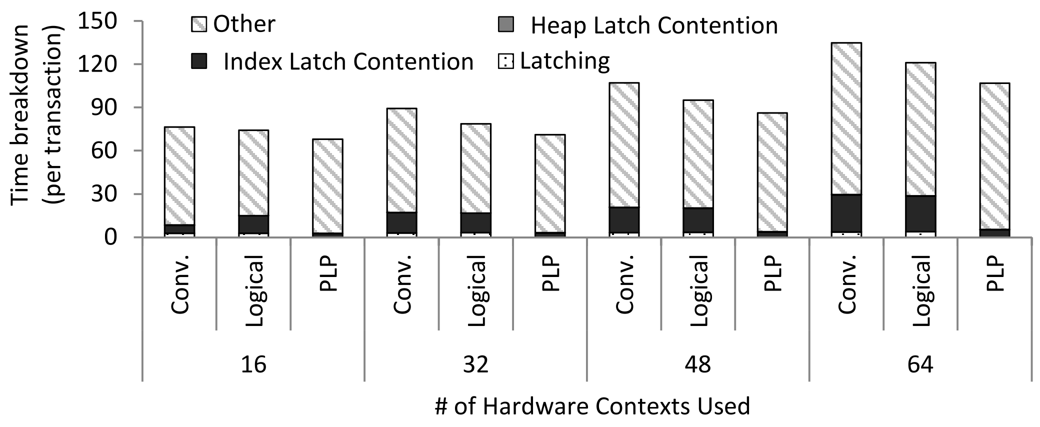


Figure 3.8: Time breakdown per transaction in an insert/delete-heavy micro-benchmark.

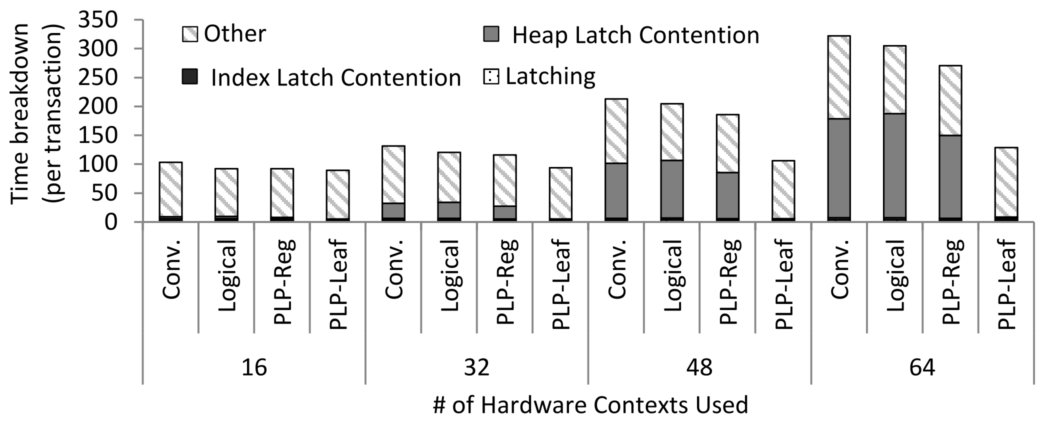


Figure 3.9: Time breakdown per transaction while running the `AccountUpdate` transaction of TPC-B, which suffers from false sharing on heap pages under conventional design.

3.4.3 Reducing Index and Heap Page Latch Contention

Having established that PLP effectively reduces the number of page latch acquisitions and critical sections, we measure the impact of this change in the time breakdown of a transaction.

Figure 3.8 shows the impact on the transaction execution time as PLP eliminates the contention on index page latches. The graph gives the time breakdown per transaction for the different designs as the number of threads that run an insert/delete-heavy workload on the TATP database increases. In this micro-benchmark, each transaction makes an insertion or a deletion request to the `Call_Forwarding` table, causing page splits and contention for the index pages that map to the records being inserted/deleted. As Figure 3.8 shows, the conventional and the logically-partitioned systems experience contention on the index page latches. They both spend 15-20% of their time waiting to acquire a latch, while PLP eliminates this contention on the index pages achieving proportional performance improvements.

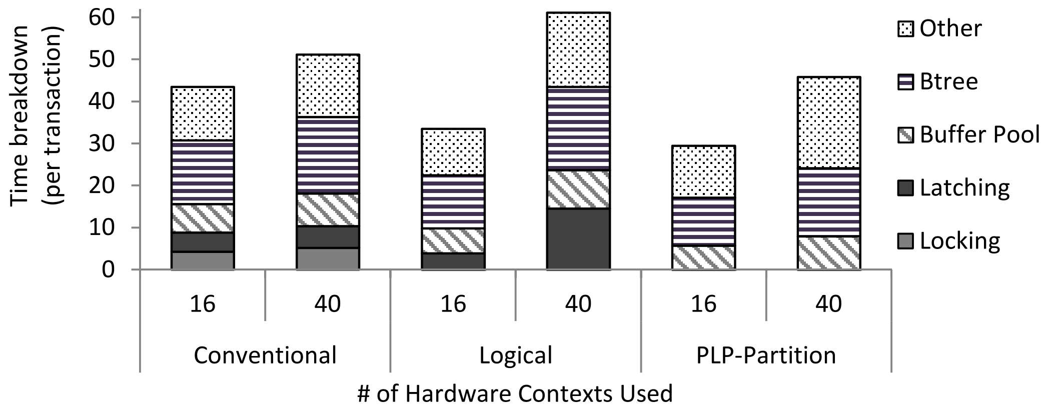


Figure 3.10: Time breakdown per transaction while running the `StockLevel` transaction of TPC-C joining 2000 tuples.

On the other hand, Figure 3.9 gives the time breakdown per transaction when we run the `AccountUpdate` transaction of the TPC-B benchmark. In this experiment we do not pad records to force them onto different pages. Transactions often wait for others because the record(s) they update happen to reside on latched heap pages. The conventional, logically-partitioned, and PLP-Regular designs all suffer from false sharing of the heap pages. At high utilization this contention wastes more than half of the total execution time. On the other hand, PLP-Leaf is immune, reducing the response time by 13-60% and achieving proportional performance improvement. In a way, PLP-Leaf provides automatic and more robust padding for the workloads that suffer from false sharing and require manual padding under the conventional system to reduce contention on the heap pages.

Finally, Figure 3.10 has the time breakdown per transaction when 16 and 40 hardware contexts are utilized by the conventional, logically-partitioned, and PLP-Partition systems when they run a slightly modified version of the `StockLevel` transaction of the TPC-C benchmark. `StockLevel` contains a join operation that joins 200 tuples. In this version, we join 2000 tuples instead of 200. We see that the conventional system wastes 20-25% of its time in contention in the lock manager and for page latching. Interestingly, even though logical-partitioning eliminates the contention due to locking, this elimination is not translated into performance improvement for the case with 40 hardware contexts. Instead the contention shifts to page latching. However, PLP eliminates the contention both inside the lock manager and for page latches achieving higher performance than all the other designs.

3.4.4 Impact on Scalability and Performance

Since PLP effectively reduces the contention (and the time wasted) to acquire and release index and heap page latches, we next measure its impact on performance and overall system scalability. We initially investigate how PLP behaves for workloads with no contention. Then,

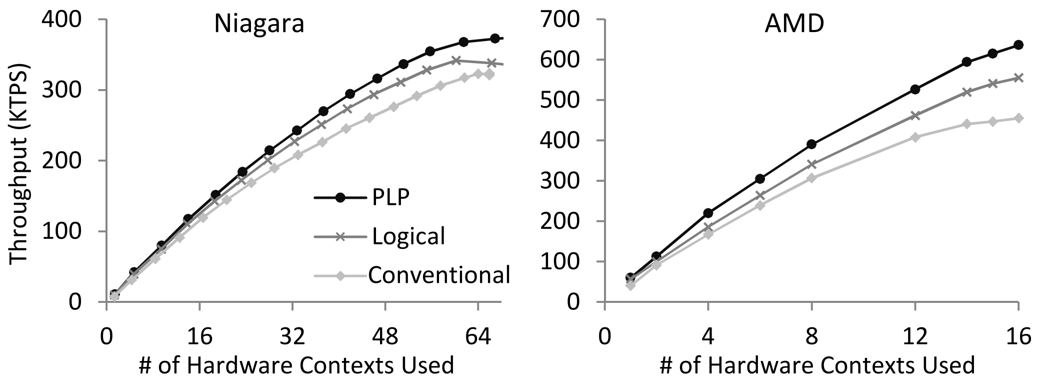


Figure 3.11: Throughput while running the `GetSubscriberData` transaction of TATP on two multicore machines.

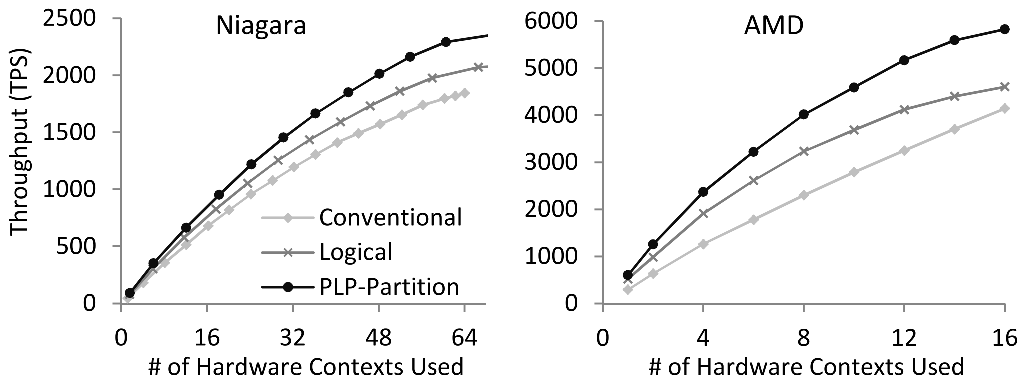


Figure 3.12: Throughput while running the `StockLevel` transaction of TPC-C on two multicore machines.

we measure its benefits for the more complex workload mixes that combine read-only and update-heavy transactions.

Figure 3.11 and Figure 3.12 show the throughput of the three main designs under comparison as we increase hardware utilization on the two multicore machines. The workloads consist of clients that repeatedly submit the TATP-`GetSubscriberData` and TPC-C-`StockLevel` transactions, respectively, which are read-only and ideally should impose no contention whatsoever. As expected, PLP shows superior scalability, evidenced by the widening performance gap with the other two systems as utilization increases. For example, from Figure 3.12 we see that for `StockLevel` the logically-partitioned system delivers an 11% speedup over the conventional system on the 4-socket Quad x64 machine. PLP delivers an additional 26% over logical partitioning or nearly 50% over the conventional. The corresponding improvements in the Sun machine's slower but more numerous cores are 13% and 34%, respectively. Note that eight cores of the x64 machine match the fully-loaded Sun machine, so the latter does not expose bottlenecks as strongly despite its higher parallelism.

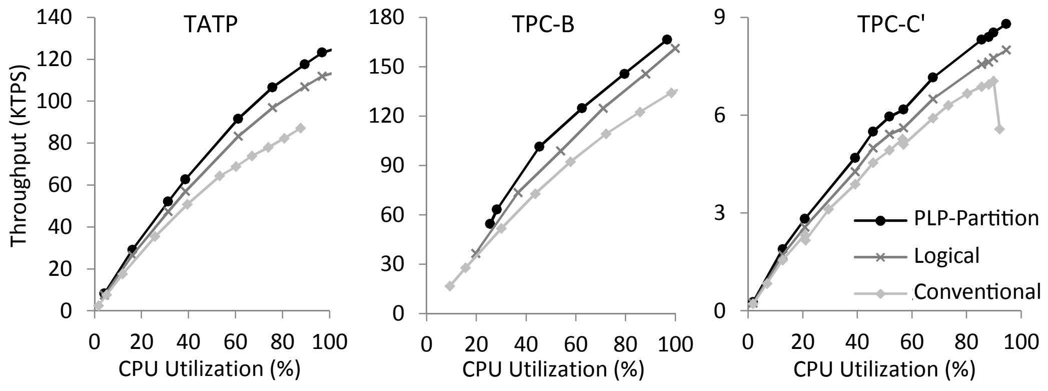


Figure 3.13: Throughput while running the workload mix of TATP, TPC-B, and TPC-C on the Sun Niagara server.

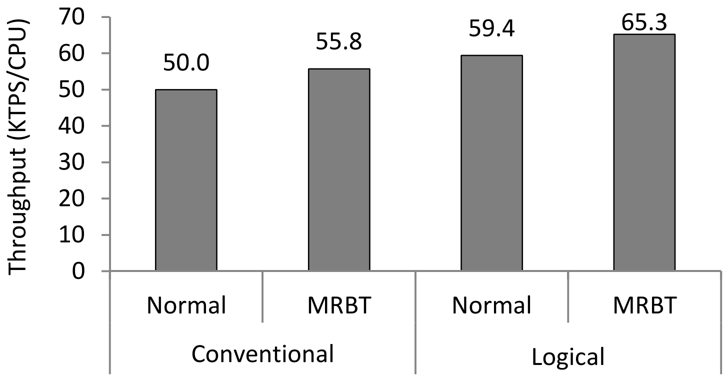


Figure 3.14: Performance of the conventional and the logically-partitioned system with and without MRBTrees while running the TATP mix.

Figure 3.13, on the other hand, shows the performance of the three design options when running the workload mix of the TATP, TPC-B, and TPC-C benchmarks as the load on the Sun Niagara machine increases. The TPC-C mix again consists of the `NewOrder` and `Payment` transactions as in Figure 3.7. We observe that PLP is still superior to both conventional and logically-partitioned systems under these complex benchmarks.

3.4.5 MRBTrees in Non-PLP Systems

The MRBTree can improve performance even in the case of conventional systems in three ways. First, since it effectively reduces the height of the index by one level, each index probe traverses one fewer node and hence is faster. Second, any possible delay due to contention on the root index page is also reduced roughly proportionally with the number of subtrees. Third, MRBTrees allow each subtree to have a structure modification operation (SMO) in flight at any time, increasing the number of concurrent SMOs the system can perform. We see the effect of the first two cases in Figure 3.14, whereas Figure 3.15 demonstrates the third case.

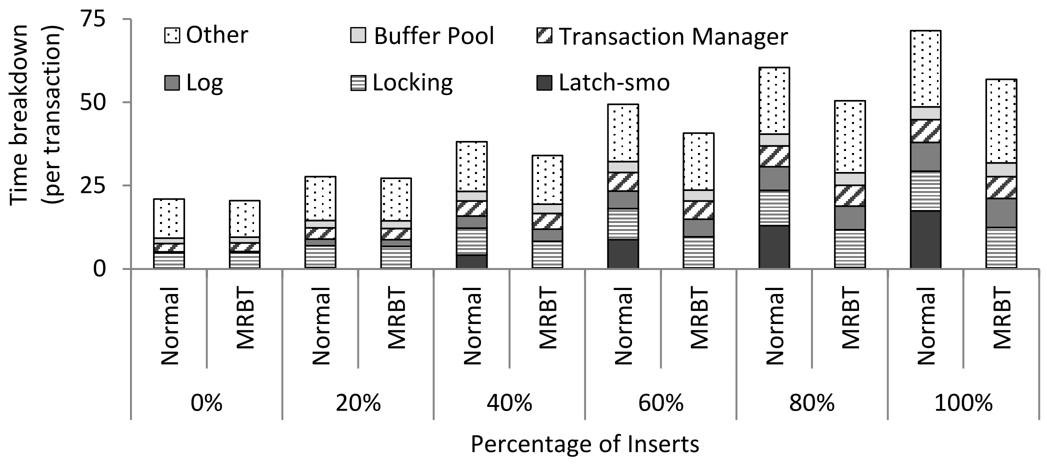


Figure 3.15: Time-breakdown of a transaction under conventional system with and without MRBTrees.

Figure 3.14 highlights the difference in the peak performance of the conventional and the logically-partitioned system when they run with and without MRBTrees. Both of the systems run the workload mix of the TATP benchmark. In both cases the improvement in performance is on the order of 10% when MRBTrees are used.

In workloads with high record insertion (deletion) rates, the MRBTree improves performance by parallelizing the SMOs. Figure 3.15 shows the time breakdown of the conventional system with and without MRBTrees as we run a micro-benchmark that consists of either a record probe or insert as we increase the percentage of inserts in the mix. Without MRBTrees, the system spends an increasing amount of the total execution time blocked waiting for SMOs to complete as the insertion rate increases. When MRBTrees are used, there is no time wasted waiting for SMOs and performance improves by up to 25%. Overall, there are compelling reasons for systems other than PLP to adopt MRBTrees.

3.4.6 Transactions with Joins in PLP

Next we turn our attention to workloads that seem not to fit well with physiological partitioning. Initially, we inspect how PLP behaves with transactions that have join operations, an operation that heavily involves work from multiple partitions in a transaction.

To evaluate the performance of PLP on transactions with joins, we slightly modified the `StockLevel` transaction from the TPC-C benchmark to determine the number of tuples joined. In its un-modified version, `StockLevel` joins 200 tuples between two tables. We created different versions of the transaction where 20, 200, 2000, 20000, and 200000 tuples are joined. For each number of tuples joined, Figure 3.16 plots the maximum throughput the conventional, the logically-partitioned, and the PLP-Partition systems achieve normalized to

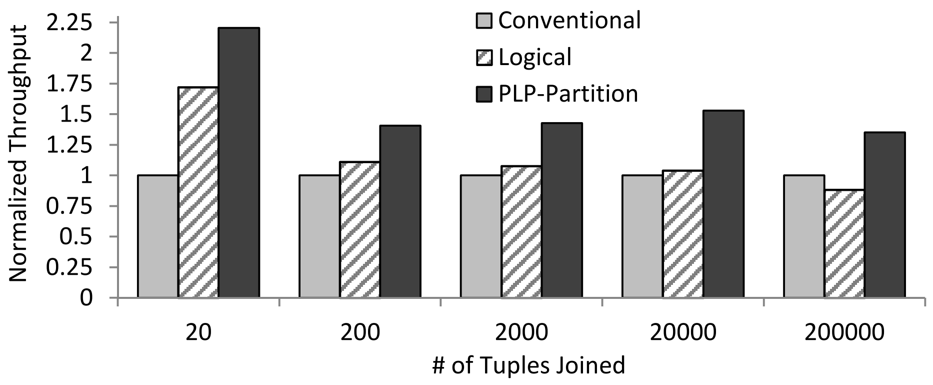


Figure 3.16: Throughput normalized to *Conventional* when running `StockLevel` on fully-utilized 4-socket Quad x86_64.

the maximum throughput of the conventional. The three systems achieve their maximum throughput when the 4-socket Quad x64 machine is 100% utilized.

Figure 3.16 shows that PLP achieves higher performance than the conventional system regardless of the number of tuples joined. When only 20 tuples are joined PLP achieves 2.1x higher performance than conventional, while when 200K tuples are joined PLP achieves 33% higher performance. The main reason for this drop in PLP's performance benefits when joining more than 20 tuples is that Shore-MT escalates to higher-level locking from row-level locking when a single transaction accesses more than a threshold of records (the default value is 25 in Shore-MT) under the conventional system. Lock-escalation reduces the lock requests to the centralized lock manager drastically for the conventional system, minimizing the bottlenecks due to locking for this particular read-only transaction. PLP achieves higher performance mainly because it eliminates the contention for page latches, as Figure 3.10 illustrates. That is in contrast with the logically-partitioned system, which for large number of tuples (200K) joined performs lower than conventional due to increased stress on page latches.

3.4.7 Secondary Index Accesses

Non-clustered secondary indexes are pervasive in transaction processing, since they are the only means to speed up transactions that access records using non-primary key columns. Nevertheless, secondary index accesses pose several challenges to PLP, which we explore in Figure 3.17. We break this analysis into two cases: (1) when the secondary index is aligned with the partitioning scheme and (2) when it is not.

If the routing columns are a prefix of the secondary index columns, then the secondary index is aligned with the partitioning scheme. For example, `subscriber_id` can be the routing column for a table where <`subscriber_id`, `subscriber_number`> is the primary key and <`subscriber_id`, `subscriber_location`> form the secondary index columns. In this case, a secondary index scan may return a large number of matched RIDs (record ids of entries

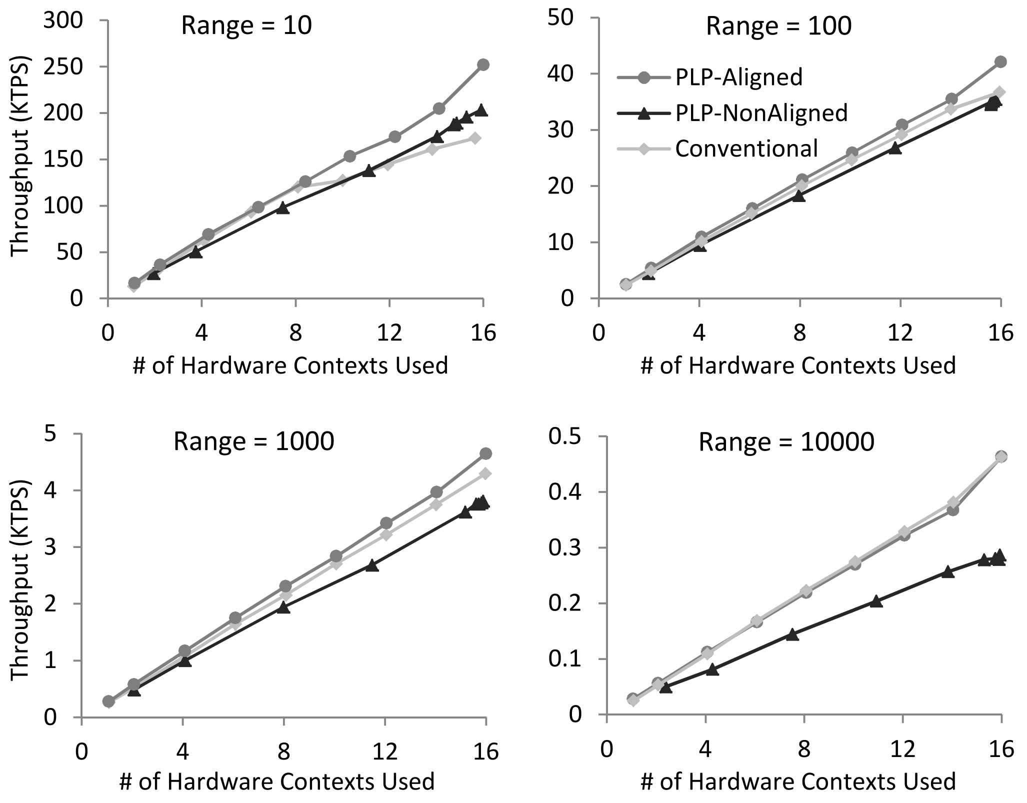


Figure 3.17: Effect of aligned and non-aligned secondary index scans on the performance of PLP as the scan range increases.

that match the selection criteria) from several partitions. All the executors need to send a pointer for the scanned data to a synchronization point (dark circles in Figure 3.5) where an aggregation of the partial results takes place. As the range of the index scans become larger (or the selectivity drops), this causes a bottleneck due to increased number of partitions participating in data coordination.

On the other hand, if the secondary index is built on the `subscriber_location` column for the table mentioned in the above example, then the secondary index is not aligned with the partitioning scheme. In this case, on top of the above mentioned bottleneck, there is also another important overhead. This overhead is because each record probe becomes a two-step process, where the secondary index probe is done by one thread conventionally and then this thread requests from the worker threads that own the partition of the scanned data to retrieve the selected records.

To quantify the overhead of using secondary indexes with PLP, we conduct an experiment where we modify TATP's `GetSubscriberData` transaction to perform a range scan on the secondary index: one that is built on <`subscriber_id`, `subscriber_number`> columns and another one that is built on `subscriber_number` only. The original transaction probes for only one `Subscriber` and the partitioning/routing column is `subscriber_id`. In the modified

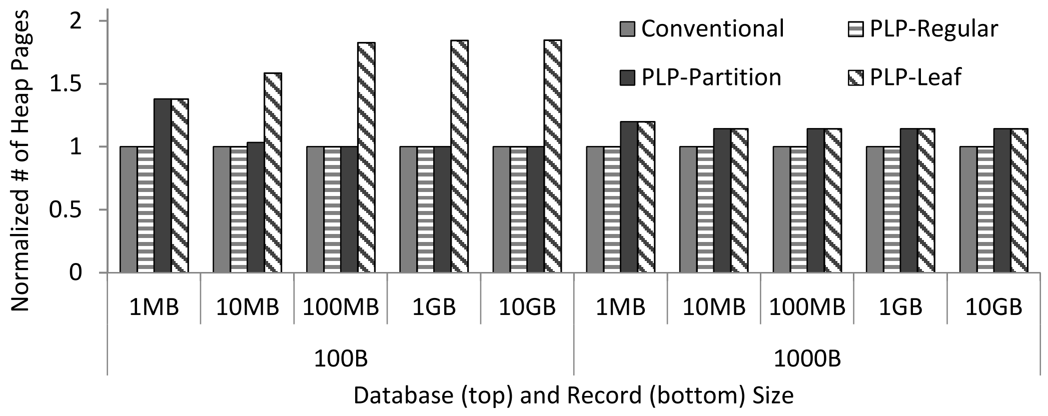


Figure 3.18: Space overhead of the three PLP variations.

version, we probe for 10, 100, 1000, and 10000 `Subscribers`, even though index scans for thousands of records are not typical in high-throughput transactional workloads.

Figure 3.17 compares the performance of the *Conventional* system with *PLP-Part-Aligned*, which performs partitioning aligned secondary index accesses, and *PLP-Part-NonAligned*, which performs non-partitioning aligned secondary index accesses, as more hardware contexts are utilized in the system. *PLP-Part-Aligned* improves performance over *Conventional* by 46%, 14%, 8%, and 1%, respectively, for ranges 10, 100, 1000, 10000. On the other hand, even though *PLP-Part-NonAligned* improves performance by 11% when 10 records are scanned, for larger ranges it hinders performance. *PLP-Part-NonAligned* is 3%, 11%, and 38% slower than *Conventional* for ranges 100, 1000, and 10000, respectively.

As expected, the performance improvement for *PLP-Part-Aligned* gets smaller as the range of the index scan increases, mainly because of the lock-escalation under the *Conventional* system as also discussed in Section 3.4.6. However, as long as the index scans of partitioning-aligned secondary indexes are selective and touch a relatively small number of records, PLP provides decent performance improvement. For *PLP-Part-NonAligned*, however, secondary index scans can be very unfriendly due to the additional overhead explained above, though unless the scan range is over 1000 records the result is not disastrous.

3.4.8 Fragmentation Overhead

PLP-Partition and PLP-Leaf create some fragmentation on the heap file since they change the regular heap file structure (see Section 3.3.3). Given the increased number of data pages due to fragmentation, we expect the heap file scan times to increase proportionally.

Figure 3.18 shows the ratio between the number of pages used in the three PLP variations and the conventional system as we increase the database size. The x-axis shows the total size of the database when each record is 100B (left-hand side of the graph) and 1000B (right-hand side of the graph). The y-axis is the ratio between the number of pages used in each

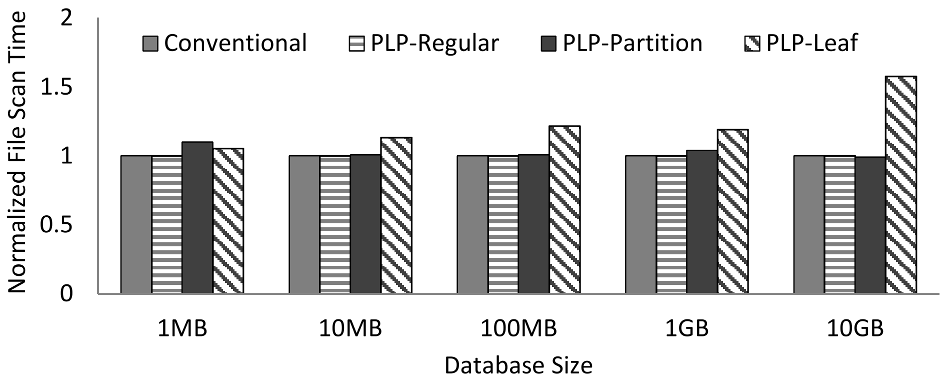


Figure 3.19: Overhead of PLP variations during file scan.

design and the conventional system. The conventional system has one partition, where the PLP variations have 100 and 10 partitions for the cases where record size is 100B and 1000B, respectively. The heap page size is 8KB. As expected, PLP-Regular does not create any fragmentation since it maintains the regular heap file format. For PLP-Partition, the amount of fragmentation becomes negligible as the database size increases for small records. However, PLP-Leaf uses up to 80% more heap pages than a conventional system for the same case creating a visible fragmentation on the heap file. On the other hand, as we increase the record size, the fragmentation decreases since each heap page is able to keep fewer records, and thus the amount of empty space on each heap page is reduced.

Figure 3.19 shows the time to scan the heap file for each PLP variation compared to the conventional system as we increase the size of the database. The setup is the same as in Figure 3.18 when the record size is 100B. The size of the buffer pool is 4GB for each measurement. From Figure 3.19, the fragmentation cost of PLP-Leaf does not significantly increase the file scan time when there are no disk accesses performed (from 1MB to 1GB) because the total number of records that are scanned is the same. However, for the larger database (10GB), PLP-Leaf increases the heap file scan time by 60% since there are more page requests from disk.

Overall, among the PLP variations, only PLP-Leaf may introduce some significant fragmentation when a heap page can keep many database records. As the number of records a heap page can keep decreases, this cost becomes less significant. We also note that PLP is a design optimized for high-performance transactional applications, where entire heap file scans are rare.

3.4.9 Summary

As the experimental results show, PLP successfully manages to eliminate two major sources of unbounded critical sections in conventional shared-everything systems; locking and latching. It is important to note that each PLP variation has its drawbacks. For example, PLP-Regular does not eliminate unbounded communication due to heap page latching (Section 3.4.2), PLP-Leaf comes with some fragmentation (Section 3.4.8), and PLP-Partition cannot repartition efficiently (as Section 3.3.3 mentions and Chapter 4 experimentally shows). We favor PLP-Leaf for workloads that need dynamic load balancing. If the workload does not heavily suffer from heap page latching, but only from index page latching, then PLP-Regular is definitely a great design choice as well.

3.5 Related Work

The related work can be categorized in two: analyzing and reducing the critical sections in DBMSs and partitioned B+Trees and concurrency control mechanisms.

3.5.1 Critical Sections

The complexity and overhead of database management systems are well-known. For example, [76] shows that, even in a single-threaded OLTP system, logging, locking, latching, and buffer-pool accesses contribute roughly equal overhead and together account for the majority of the machine instructions executed during a transaction. Other work shows that these sources of overhead become scalability burdens on multicore hardware [96]. PLP eliminates the bottlenecks due to locking and latching in a shared-everything setting.

In the shared-everything arena, recent proposals for speculative lock inheritance (SLI) [95], lightweight intent locks (LIL) [108], and data-oriented transaction execution [145] minimize the need for interaction with a centralized lock manager. Where speculative lock inheritance allows the system to spread lock operations across multiple transactions to reduce contention, data-oriented systems replace the central lock manager with thread-local lock management. Reducing lock contention with data-oriented execution is also studied for data-stream operators [40], making threads delegate the work on some data to the thread that already holds the lock for that data and move to the next operation in their queues.

Other proposals tackle the weakness posed by the centralized log manager, [97, 98] presenting a scalable log buffer and [31] exploiting flash technology to reduce logging latencies. These proposals show even seemingly-pervasive forms of communication can be reduced or sidestepped to great effect. However, none of them addresses physical data accesses involving page latching and buffer pool, the other two major sources of overhead in the system, which PLP eliminates.

Oracle RAC [143], with Cache-Fusion [114], allows database instances in the shared-disk cluster to share their buffer pools and avoid accesses to the shared-disk. It can also partition the data to reduce both logical and physical contention on a particular portion of the data. However, it does not enforce each partition to be accessed only by a single thread. Therefore, it does not eliminate physical latch contention while accessing pages from the shared-cache as much as PLP does.

As discussed previously, shared-nothing systems [45, 107, 178, 179] have an appealing design that eliminates critical sections altogether. However, they struggle both pro-actively to reduce the need to execute distributed transactions through efficient partitioning [39] as well as re-actively to reduce overhead when distributed transactions cannot be avoided [100]. On the other hand, PLP, in addition to eliminating a big portion of the unbounded critical sections, offers a less costly way of load balancing and communication for distributed (multi-site) transactions since partitions share the same memory address space.

3.5.2 B+Trees and Alternative Concurrency Control

There are alternatives to traditional B+Tree concurrency control to allow multiple SMOs at the same time [92, 129]. The MRBTree index structure provides an alternative to such techniques, allowing concurrent SMOs with less code complexity. However, these techniques can be implemented alongside MRBTrees to achieve concurrency within a partition, should that be desirable for a conventional system. As an addition to these techniques MRBTrees also allow multiple root split operations in parallel. Several earlier works propose B+Trees having multiple roots to reduce contention due to locking [61, 134]. However, none of these proposals targets physical latch contention in the system.

In addition, some latch-free B+Tree implementations use alternative synchronization methods. CO B-Tree [21] uses load-linked/store-conditional (LL/SC), whereas Masstree [125] relies on read-copy-update mechanisms instead of latching to synchronize operations on a B+Tree. Bw-tree [119] is another recent latch-free index structure proposal, which does not perform in place updates on the B-Tree pages and relies on atomic updates using compare-and-swap (CAS) instructions. These designs minimize the time spent in unbounded critical sections executed during index operations. However, they do not completely remove such critical sections. In addition, even though atomic operations scale and perform well on single/two-socket machines, they are shown to be problematic on multisocket machines with four or more processors [41, 154]. PALM [170], on the other hand, eliminates both page latching and contention on the B+Trees by using Bulk Synchronous Parallel model. However, it has to perform B+Tree operations in batches in order to exploit this technique, which might increase the average latency for individual operations and be harder to integrate within a database management system. Overall, the latch-free B+Tree proposals can be combined with MRBTrees, especially for the non-partitionable workloads. Instead of using fine-grained partitions (e.g., partition per hardware-context) like in PLP, one can build coarser-grained MRBTrees (e.g., partition per socket or close-by cores). Then, the worker threads assigned to an MRBTree can traverse it using latch-free techniques. This way, one can bound the contention on the unbounded critical sections that still exist in latch-free techniques to fixed number of worker threads that belong to a partition (i.e., downgrading unbounded contention to fixed contention), and also minimize the drawbacks related to partitioning.

Finally, optimistic and multiversion concurrency control schemes [22, 113, 116, 196] may improve concurrency by resolving conflicts lazily at commit time instead of eagerly blocking them at the moment of a potential conflict. In other words, they minimize the time spent in unbounded critical sections due to locking. When conflicts are rare this allows the system to avoid the overhead of enforcing database locks. On the other hand, if the conflicts occur frequently the performance of the system drops rapidly, since the transaction abort rate is high. Prior work compares the concurrency control schemes in database systems [2, 204], while the book of Bernstein et al. [23] and Thomasian's survey [181] are good starting points for the interested reader. Even though the focus of PLP is on the contention for latches rather than the concurrency control scheme, PLP can also be integrated with multiversion concurrency control schemes similar to the possible integration of MRBTrees with latch-free index structures as described above.

We also note that there is a large body of work on cache-conscious index implementations (e.g., [29, 32, 162, 163]). PLP eliminates the need for latching and concurrency control at the index level. Therefore, we expect to get a significant performance boost if we substitute the index implementation with a cache-friendlier B+Tree alternative, since the B+Tree probes are the most expensive remaining component of PLP.

3.6 Limitations of PLP

Applications that have less pressure on the storage manager

First of all, PLP is designed for high performance transaction processing that imposes great pressure on the internals of the database storage layer. Therefore, certain classes of applications may not benefit from it, or even get penalized. For example, the business intelligence applications with large file scans or joins do not stress the parts of the storage manager PLP improves; since these workloads have mostly read-only requests, the lock manager and latching can even be disabled. In such workloads PLP may penalize performance since it may require coordinating large volumes of data among participating threads from different partitions. It is common practice, however, to employ dedicated database engines (usually column-stores [24, 180]) for such workloads.

Non-partition aligned index accesses

PLP partitions each table using range-based partitioning to the keys of a specific subset of the columns of the table. The DBA, however, may decide to build indexes (usually non-clustered secondary indexes) that do not contain the columns that PLP uses for its partitioning. We refer to such indexes as *non-partitioning aligned indexes* and they may become performance bottlenecks. Data-oriented execution and PLP handles such accesses by appending each index leaf entry with the fields of the record that are needed for identifying the partition-owning thread. The non-partitioning aligned index is accessed as a conventional index, without avoiding any locking or latching, in order to retrieve the id of the record to be accessed in the heap file and then the access is passed to the appropriate thread.

As Figure 3.17 shows, such accesses can be burdensome for PLP. However, as a proactive measure, we implemented tools that help the application developer and the DBA to avoid having workloads with very frequent such index accesses [146].

Breaking transactions

As mentioned previously (Section 3.3.1), the transactions need to be divided into smaller actions based on the data accessed in different parts of the transaction. These actions are represented as a directed graph to understand the transaction flow and dependencies among the actions. This representation also helps us to exploit intra-transaction parallelism for the independent actions. However, it introduces the initial cost of identifying these actions. We implemented a tool that automatically forms such a transaction flow graph given the SQL statement for the transaction to ease this initial cost [146].

Scheduling challenge

As mentioned in the above paragraph, PLP breaks transactions into smaller actions and routes each action to the worker thread responsible from the data the action wants to access. Therefore, whenever a transaction involves multiple actions, it becomes a multi-partition transaction. Looking at the results in Section 3.4.4, we see that PLP performs well regardless of such transactions in a variety of cases. However, such transactions also create a scheduling challenge for PLP. In the case of TATP and TPC-B benchmarks, this is not as apparent since the transactions are either too short (TATP) or there is not much variety in the workload mix (TPC-B). On the other hand, for the TPC-C benchmark running the whole transaction mix either overloads the machine, since it requires too many worker threads to be active at the same time, or causes the critical sections of the fixed type to become problematic, which are extensively used while coordinating the different actions of a transaction. Therefore, Section 3.4.2 and Section 3.4.4 use a simpler version of the TPC-C mix, which run only the most frequent two transactions of TPC-C (`NewOrder` and `Payment`).

The scheduling challenge of PLP especially becomes problematic on multisocket multicore architectures with non-uniform memory access (NUMA) latencies. We have recently dealt with this problem through designing the ATraPos infrastructure [154], which makes PLP aware of the underlying hardware topology and dynamically controls the scheduling of the worker threads based on the workload characteristics. More specifically, ATraPos adjusts the number of partitions for each table at run-time and schedules the worker threads of the partitions that are frequently accessed together in a transaction on the same processor socket.

3.7 PLP on Future Hardware and Conclusions

Unlike conventional systems, which either embrace fully shared-everything or shared-nothing philosophies, physiological partitioning takes the best features of both to produce a hybrid system that operates nearly latch- and lock-free, while still retaining the convenience of a common underlying storage pool and log. We achieve this result with a new multi-rooted B+Tree structure and careful assignment of threads to data.

As multicore hardware trends evolve, PLP becomes increasingly attractive for several reasons. Conventional OLTP is ill-suited to modern and upcoming hardware since;

- the code of an OLTP system is full of unbounded critical sections [96, 99],
- the access patterns are so unpredictable [174] that even the most advanced prefetchers fail to detect data access patterns for a transaction [175],
- the majority of the accesses are shared read-write; hence, they under-perform on caches with non-uniform access latency [20, 69, 70].

As we have seen, PLP, combined with previous advances in logging, succeeds in all three problems. The majority of the unbounded critical sections are completely eliminated, access patterns are regularized by the thread assignments, and threads no longer share data to communicate, eliminating the shared R/W problem. This regularity is going to become increasingly important as hardware continues to make more and more demands of the software. Unfortunately, OLTP will only be able to utilize these new architectures effectively if it can eliminate the majority of accesses that are shared among multiple processors. In short, by eliminating a large class of unbounded/unscalable communication, PLP leaves OLTP engines much better-poised to take advantage of the upcoming hardware, whatever form it may take.

4 Dynamic Load Balancing for PLP

Partitioning is an increasingly popular solution for scaling up the performance of database management systems even within a single (multicore or multisocket) machine. However, it is not a panacea since there are many challenges associated with it. This chapter focuses on one of the most troublesome challenges for partitioning-based transaction processing systems, which is their behavior in skewed and dynamically changing workloads. Such workloads are the norm rather than the exception and highly problematic for statically partitioned systems.

We demonstrate the non-optimal performance of single-node partitioning-based transaction processing systems and analyze the costs and challenges toward robust and efficient dynamic load balancing mechanisms for such systems. This analysis highlights that physiologically-partitioned (PLP) shared-everything online transaction processing systems offer a good infrastructure for lightweight repartitioning. Based on this observation, we propose a dynamic load balancing mechanism (called DLB) specialized for the PLP design. Evaluation on different multicore machines shows that the overhead of DLB is low in normal operation (in the worst case at most 8%), while it enhances the system with robust behavior achieving very low response times in both detecting and handling load imbalances. [1]

4.1 Introduction

Database management systems need to provide enough execution parallelism to exploit modern multicore and multisocket hardware. Unfortunately, exhibiting high execution parallelism is not easy, even for transaction processing workloads, which are characterized by high *concurrency* at the request level. In particular, conventional transaction processing results in complicated and unpredictable access patterns [145]. In order for the system to maintain the consistency of the data shared by the parallel processes, it needs to employ synchronization points, which form *critical sections* that serialize transaction execution. Critical sections not only hurt single-thread performance, especially in transaction processing workloads [76], but they also quickly become scalability bottlenecks [96].

[1] This chapter uses material from [147, 185].

The common solution for improved scalability is to either remove critical sections completely or reduce the contention on them. A very popular technique to achieve that is *partitioning*. The database is broken into multiple partitions and the data that belong to one partition are operated on by just one worker thread. As a result, the number of threads that share some part of the data is reduced along with the contention on the critical sections that protect that data. If only a single thread accesses each partition, then the need for critical sections is eliminated [107, 145, 179].

If configured correctly, a partitioned database system (shared-nothing [45, 179] or shared-everything [145], Chapter 3) can perform better than corresponding non-partitioned systems. Achieving high performance, however, is not a simple task when running realistic, dynamically changing workloads. Depending on the access patterns, the load of each partition might be different. Skewed access patterns can lead to *load imbalance* and reduce or eliminate any benefits due to partitioning. Therefore, system designers need to be careful in order to benefit from and not to be hindered by partitioning.

There are two orthogonal ways to attack the problem of skewed access in partitioning-based transaction processing systems:

- *proactively* by configuring the system with an appropriate initial partitioning scheme and
- *reactively* by using a dynamic balancing mechanism.

Starting with the appropriate partitioning configuration is key. If the workload characteristics are known a priori, previously proposed techniques [39, 164] can be used to create effective initial configurations. If the workload characteristics are not known, then simpler approaches like round-robin, hash-based, and range-based partitioning [45] would work. As time progresses, however, skewed access patterns gradually lead to load imbalance during execution. The initial configuration eventually becomes useless no matter how carefully it is chosen. Thus, it is far more important and challenging to *dynamically* balance the load through repartitioning based on the observed, and ever changing, access patterns. A robust dynamic load balancing mechanism should eliminate any bad choices that might be made during initial assignments.

In this chapter, we focus on dynamic load balancing and online repartitioning in the context of partitioned database management systems within a single node. After a thorough comparison of different partitioning mechanisms in terms of their repartitioning costs, we design a lightweight yet effective dynamic load balancing and repartitioning mechanism, called *DLB*, for physiologically-partitioned (PLP) OLTP systems. To collect information about the current access patterns and load in a workload, DLB uses the existing request queues of the partitions and employs a new data structure, called an *aging two-level histogram*. These structures help in observing recent load and data access patterns across and within partitions. DLB also exploits the *multi-rooted B+Tree (MRBTree)* index structure that is at the core of PLP (Chapter 3) for efficient reorganization of partitions.

The contributions and the organization of this chapter are as follows:

- Section 4.2 demonstrates that access skew can severely hurt performance in statically partitioned databases, rendering partitioning useless in many realistic workloads and underlining the need for dynamic repartitioning.
- Section 4.3 devises a cost model for repartitioning and shows that PLP provides a very good infrastructure for dynamic repartitioning, mainly due to its key component MRBTrees.
- Section 4.4 designs a lightweight yet effective dynamic load balancing and repartitioning mechanism, DLB, for PLP.
- Section 4.5 integrates the DLB mechanism in a prototype transaction processing system that employs PLP. The evaluation quantifies the overhead of DLB under static workloads, which is in the worst case at most 8%, and its effectiveness during dynamic workloads, where DLB achieves low response times in both detecting and fixing imbalances.

Finally, Section 4.6 contrasts DLB to the related work on dynamic load balancing and Section 4.7 concludes the chapter.

4.2 Need for Dynamic Repartitioning

In general one of the disadvantages of partitioning-based transaction processing designs is that they are vulnerable to skewed and dynamically changing workloads; in contrast with shared-everything systems that do not employ any form of partitioning and tend to suffer less when the workload is not stable. Unfortunately, skewed and dynamically changing workloads are the rule rather than an exception in transaction processing. Therefore, it is imperative for partitioning-based designs to alleviate the problem of skewed and dynamically changing accesses.

To exhibit how vulnerable partitioning-based systems are to skew, Figure 4.1 plots the throughput of a non-partitioned (shared-everything) system and a statically partitioned system that deploys physiological-partitioning when all the clients in a TATP [137] database submit the `GetSubscriberData` read-only transaction. Initially the distribution of requests is uniform over the entire database. However, at time point 10 (sec) the distribution of the load changes: 50% of the requests are sent to 30% of the database (see Section 4.5.1 for experimental setup).

As we can see from the graph in Figure 4.1, initially and as long as the distribution of requests is uniform, the performance of the non-partitioned system is around 15% lower than the partitioned one. After the load change the performance of the non-partitioned system remains pretty much the same, while the performance of the partitioned system drops sharply by around 35%. The drop in the performance is severe even though the skew is not that extreme; easily a higher fraction of the requests could go to a smaller portion of the database, for

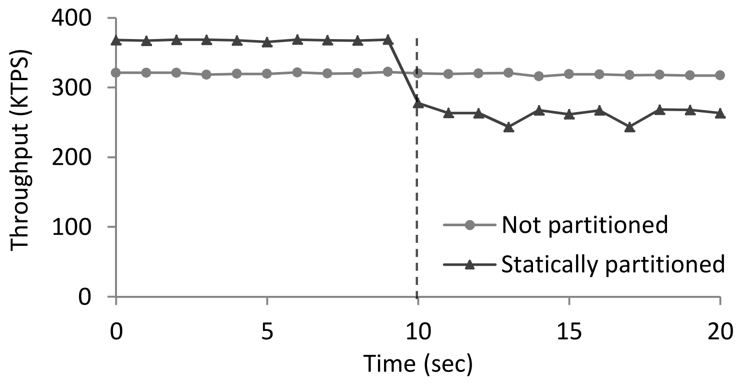


Figure 4.1: Throughput of a statically partitioned system when load changes at runtime; at time t=10 50% of the requests are sent to 30% of the database.

example following the 80-20 rule of thumb where the 80% of the accesses go to only 20% of the database. Figure 4.1 clearly underlines the need for dynamic repartitioning for partitioning-based designs.

4.3 Repartitioning Cost

A dynamic load balancing mechanism would be useless if the cost of repartitioning in a partitioning-based transaction processing system is very high. The lower the cost of repartitioning, the more frequently the system can trigger load balancing procedures and the faster it can react to load changes. This subsection models the cost of repartitioning for a shared-nothing (physically-partitioned) system and the three PLP variations (Section 3.3.3) to highlight the clear advantage of PLP-Regular and PLP-Leaf. It also describes the way to perform repartitioning with PLP.

The basic case of repartitioning is when a partition needs to split into two. Therefore, for all the PLP variations and the shared-nothing design our repartitioning cost model calculates

- the number of records and index entries that have to be moved,
- the number of update/insert/delete operations on the indexes,
- the number of pointer updates on the index pages and the routing page that keeps the information on the key-ranges for each partition, and
- the remaining number of read operations that have to be performed

when a partition is split into two. We also discuss merging two partitions but do not give a detailed cost model.

Algorithm 1 Splitting an MRBTree subtree.

```
1: {binary-search routine used below performs binary search to find the $key$ on the $page$. If
   an exact match for the $key$ is found, $found$ is returned as true and the function returns
   the slot for the $key$ on the $page$. Otherwise, $found$ is false and the function returns at
   which slot on the $page$ the $key$ should reside.}
2: $page = root$
3: $found = false$
4: while $page\,!= NULL\ \&\ !found$ do
5:     $slot$ = binary-search($page, key, found$)
6:     $slots.push(slot)$
7:     $pages.push(page)$
8:     $page = page[slot].child$
9: while $pages.size > 0$ do
10:    $slot = slots.pop()$
11:    $page = pages.pop()$
12:    Create $page_{new}$
13:    Move entries starting from $slot$ at $page$ to $page_{new}$
```

Let's assume that there is a heap file (table) with an index on it, which in the case of PLP is an MRBTree. When a partition needs to split into two, a subtree in the index needs to split into two as well. In that case we define:

- h as the height of the tree,
- n as the number of entries in a non-leaf B+Tree page,
- m_i as the number of entries to be moved from the B+Tree at level i, and
- M as the number of records in the heap file that have to be moved.

The number of read operations during a key value search in the B+Tree is omitted since it is the same for all the systems (a binary search at each level from root to leaf).

4.3.1 Splitting Non-clustered Indexes

The first case we consider is when the heap file that needs to be repartitioned has a unique non-clustered primary and a secondary index and the data are partitioned based on the primary index key values.

PLP-Regular

The cost of repartitioning in PLP-Regular is very low. Only a few index entries need to move from one subtree of the MRBTree index(es) to another newly created subtree. Algorithm 1 shows the procedure that needs to be executed to split an MRBTree subtree. First, we need

Table 4.1: Repartitioning costs for splitting a partition into two.

System	#Records Moved (M)	Primary Index					Secondary Index Changes
		#Entries Moved	#Reads	#Pages Read	#Pointer Updates	Changes	
PLP-Regular	-	$\sum_{k=1}^{h} m_k$	-	-	$2 \times h + 1$	-	-
PLP-Leaf	m_1	$\sum_{k=1}^{h} m_k$	M	1	$2 \times h + 1$	M *updates*	M *updates*
PLP-Partition	$m_1 + \sum_{l=0}^{h-2} (n^{h-l-1} \times (m_{h-l} - 1))$	$\sum_{k=1}^{h} m_k$	M	$1 + \frac{M-m_1}{n}$	$2 \times h + 1$	M *updates*	M *updates*
Shared-Nothing	$m_1 + \sum_{l=0}^{h-2} (n^{h-l-1} \times (m_{h-l} - 1))$	-	M	$1 + \frac{M-m_1}{n}$	-	M *inserts* M *deletes*	M *inserts* M *deletes*
PLP (Clustered)	m_1	$\sum_{k=2}^{h} m_k$	-	-	$2 \times h + 1$	-	M *updates*
Shared-Nothing (Clustered)	$m_1 + \sum_{l=0}^{h-2} (n^{h-l-1} \times (m_{h-l} - 1))$	-	-	-	-	M *inserts* M *deletes*	M *inserts* M *deletes*

to find the leaf page where the starting key of the new partition should reside (lines 4–8 in Algorithm 1). Let's assume that there are m_1 entries that are greater than or equal to the starting key on the leaf page where the slot for this key is found. All that needs to be done is to move these m_1 entries on that leaf page to a newly created (MRBTree) index page. This procedure has to repeat as the tree is traversed from this leaf page to the root (lines 9–13 in Algorithm 1). It is not necessary to move any entry from the pages that keep the key values greater than the ones in the leaf page containing the starting key. Setting the previous/next pointers of the pages at the boundaries of the old and new partitions is sufficient. Finally, a new entry to the routing page should be added for the new partition.

The overall cost is given in the first row of Table 4.1. The cost model in Table 4.1 describes the worst case scenario for PLP-Regular. If the starting key of the new partition is in one of the non-leaf index pages, there is no need to move any entries from the pages that are below this page because the moved entries from the non-leaf page already have pointers to their corresponding child pages, resulting in fewer reads, updates, and moved entries.

PLP-Leaf

Figure 4.2 shows the three-step process for splitting a partition into two in PLP-Leaf. The height of the subtree is two and the dark slot in Figure 4.2(a) indicates the slot that contains the leaf entry with the starting key of the new partition. Figure 4.2(b) shows that a new subtree is created as a result of the split. Those two steps are the same as the repartitioning process in PLP-Regular and hence have the same cost.

However, as mentioned in Section 3.3.3, in addition to modifying the index structure, we also have to move records from the heap file to new heap pages when repartitioning in PLP-

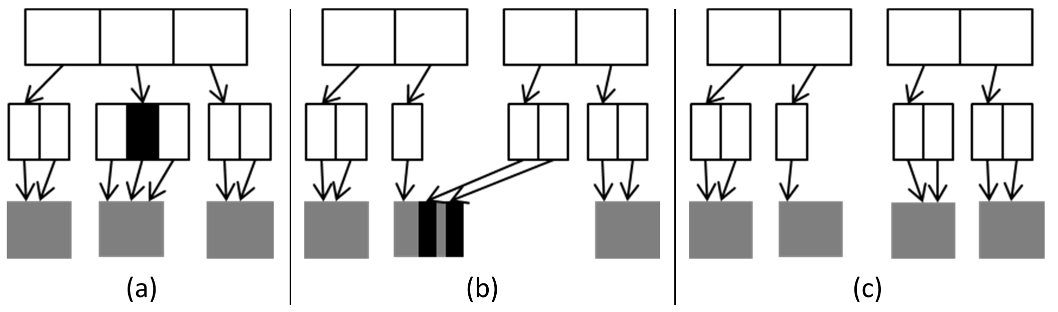


Figure 4.2: Example of splitting a partition in PLP-Leaf, which is a three-step process.

Algorithm 2 Splitting heap pages in PLP-Leaf and PLP-Partition.

```
1: $leaf$ = leftmost leaf page
2: Create $page_{new}$
3: while $leaf != NULL$ {Omit for PLP-Leaf} do
4:   for all $t$ referenced by $leaf_{current}$ do
5:     if $page_{new}$ does not have space then
6:       Create $page_{new}$
7:     Move $t$ to $page_{new}$
8:     Update pointers at all the secondary indexes
9:   $leaf = leaf.next$ {Omit for PLP-Leaf}
```

Leaf. Algorithm 2 shows the pseudocode for updating the heap pages upon a partition split in PLP-Leaf (and also PLP-Partition). The dark records on the heap pages in Figure 4.2(b) indicate those records that belong to the new partition (subtree) and need to move. Those records are referenced by the m_1 leaf page entries that moved to the newly created subtree. Therefore, in the worst case m_1 records have to move (lines 4-7 in Algorithm 2). Since the index is non-clustered, we have to scan these m_1 entries in order to get the record ids (RIDs) of the records to be moved and spot their heap pages. The result of the split after the records are moved is shown in Figure 4.2(c). Whenever a record moves, its RID changes. Thus, once all the records are moved, all the indexes (primary and secondary) need to update their entries (line 8 in Algorithm 2) with the new RID values.

The cost for repartitioning in PLP-Leaf is given in the second row of Table 4.1. This cost, again, illustrates the worst case scenario. If the starting key of the new partition is found in one of the non-leaf pages, then no record movement has to be done since there will be no leaf page splits and the constraint of having all heap pages referenced by only one leaf page is already preserved. Moreover, even if the key is found on the leaf page, we might not have to move all the records that are specified by the model above. If all the records on a heap page are referenced only by leaf entries of the new partition, then these records can stay on that heap page.

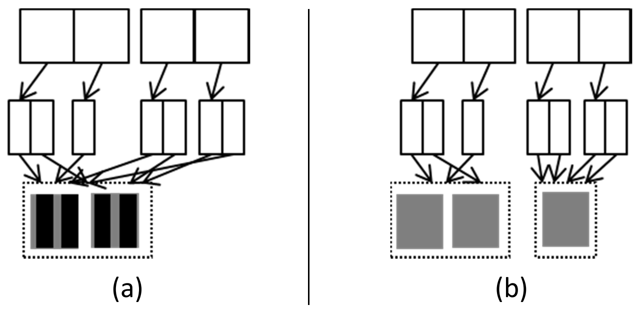


Figure 4.3: Splitting a partition when PLP-Partition is used.

PLP-Partition

In PLP-Partition, the process for splitting the index structure is the same as in PLP-Regular and PLP-Leaf. Therefore, it is omitted from Figure 4.3, which shows the rest of the process for splitting a partition into two in PLP-Partition.

In the worst case, in PLP-Partition we may have to move records from all the heap pages that belong to the old partition. Those records are indicated by the dark rectangles in the heap pages of Figure 4.3(a). The number of records to be moved is equal to the number of entries that are on the leaf pages of the new subtree. As in PLP-Leaf, the RIDs of the records are retrieved with an index scan on the newly created subtree, the records are moved to new heap pages and they get new RIDs, and all the indexes are updated with the new RIDs after the record movement is completed (shown in lines 3-9 in Algorithm 2). The result of the partitioning is shown in Figure 4.3(b), while the cost model for PLP-Partition is given in the third row of Table 4.1.

Shared-Nothing

In a shared-nothing system, the cost for the record movement is equal to the worst case of PLP-Partition since the entire old partition needs to be scanned for records that belong to the new partition. In addition, the cost of index maintenance may be prohibitively expensive.

In a shared-nothing system, each record move across partitions results to a deletion of an index entry (or entries if there are multiple indexes) from the old partition and an insertion of an index entry to the new partition. This is in contrast with the PLP variant where every record move is a result of a few MRBTree updates. The cost of index maintenance when repartitioning shared-nothing systems sometimes can be prohibitive. In order to avoid the index maintenance, a common technique is to drop and bulk-load the index from scratch upon every repartition. The repartitioning cost for a shared-nothing system is given in the fourth row of Table 4.1. Given how expensive repartitioning can be, shared-nothing systems are reluctant to frequently triggering repartitioning.

4.3.2 Splitting Clustered Indexes

Let's consider the case where we have a unique clustered primary index and a secondary index, and the data partitioning is done using the primary index key columns. In this setup, no heap file exists, since the primary index contains the actual data records rather than RIDs. Also, the three PLP variations are equivalent, because their differences lie in how they treat the records in the heap pages.

When the actual records are part of the clustered primary index, the cost of record movement for PLP is equal to the number of leaf page entries that need to move, while the cost of primary index maintenance is equal to the entry movements in the non-leaf pages of the MRBTree index. The cost model is given in the fifth row of Table 4.1.

On the other hand, the repartitioning cost for the shared-nothing system is similar to its non-clustered case. The only difference is there is no need to scan the leaf pages to get the RIDs of the records to be moved since the leaf pages have the actual records. The repartitioning cost model is given in the last row of Table 4.1.

4.3.3 Moving Fewer Records

With some additional information we can move less data during repartitioning while increasing the number of reads. For example, in PLP-Partition instead of directly moving all the records that belong to a new partition, we can scan all the index leaf pages to be split and collect information for all the records. With this information, we can determine whether a heap page has more records that belong to the old partition or the new partition and act accordingly. More specifically, if a heap page has more records that belong to the new partition, we can move the records that belong to the old partition instead. The number of reads while scanning the leaf pages during this process can easily become a bottleneck in disk-resident databases, due to the number of I/O operations that have to be performed. On the other hand, in in-memory databases or systems that use flash storage devices, such I/O bottlenecks can be prevented [31] and the above mentioned technique can reduce the amount of data movement during repartitioning. This technique, unfortunately, cannot be used in a shared-nothing system because the pages of the two partitions do not share the same storage space.

4.3.4 Example of Repartitioning Cost

Table 4.2 gives an example of the repartitioning cost for the different systems under consideration based on the cost model given in Table 4.1. In this example, a partition, which contains 433MB of 100-byte data records in a heap file, is split into two. We assume that there is a primary index of height 3 with 170 32-byte entries on each page. The first four rows of the table assume there are a unique non-clustered primary index and a secondary index in the system, whereas for the last two rows there are a unique clustered primary index and a secondary

Table 4.2: Repartitioning costs when splitting a partition with 466 MB data into two (U: Updates, D: Deletes, I: Inserts, M: Million).

System	Records Moved	Primary Index				Secondary Index Changes
		Entries Moved	#Pages Read	#Pointer Updates	Changes	
PLP-Regular	-	8KB	-	7	-	-
PLP-Leaf	8.3KB	8KB	1	7	85 U	85 U
PLP-Partition	233MB	8KB	14365	7	2.44M U	2.44M U
Shared-Nothing	233MB	-	14365	-	2.44M I + 2.44M D	2.44M I + 2.44M D
PLP (Clustered)	8.3KB	5.3KB	-	7	-	85 U
Shared-Nothing (Clustered)	233MB	-	-	-	2.44M I + 2.44M D	2.44M I + 2.44M D

index. For the three PLP variations the number of moved records represents the worst case scenario.

As Table 4.2 shows, the PLP variations, except for PLP-Partition, move very few records compared to the shared-nothing one. In the worst case, PLP-Partition moves the same number of records as the shared-nothing system. For the clustered index case, PLP is cheaper to repartition than the shared-nothing system in terms of both record movement and index maintenance. When we calculate the corresponding costs for a larger heap file with an index of height 4, the repartitioning cost for the shared-nothing system and PLP-Partition becomes prohibitive.

4.3.5 Cost of Merging Two Partitions

For any PLP variation, a merge operation only requires index reorganization and no data movement. During the index reorganization, there are three cases to consider;

- when two subtrees have the same level,
- when the subtree with lower key values (T_l) has a higher level than the other subtree, and
- when the subtree with higher key values (T_h) has a higher level than the other subtree.

When the two subtrees to be merged have the same level, the entries of T_h's root are appended to the entries of T_l's root. Since the entries of the root page have information about the pointers to the lower levels of the tree, copying the entries of the root page is sufficient for this merge operation. In this case the cost of the merge operation only depends on the number of entries in the root page of T_h. If the number of entries destined to the new root exceeds the page capacity, a new root page is created (through a structure modification operation), the same way a page split happens after a record insert.

When T_l is taller than T_h, T_l is traversed down to one level higher than the level of T_h. Then an entry is inserted at the rightmost page of this level that points to T_h and has the key value equal to the starting key of the key range of T_h. Therefore, the cost of the merge operation is only a tree traversal, which depends on the level difference between the two trees, and an insert operation in this case.

When T_h is taller, the merge operation is very similar to the second case and the cost is the same. T_h is traversed down to one level higher than the level of T_l and instead of the rightmost page, the leftmost page gets the entry that points to T_l and has the key value equal to the starting key of the key range of T_l.

After the delete operation, the partition table is updated according to the new key range and its corresponding subtree root page id.

In a shared-nothing system, however, we have to move all the records from one partition to the other and insert the corresponding index entries at the resulting partition. Therefore, the cost of the merge operation is proportional to the number of records in a partition and its way higher than the merge cost for any PLP variation.

We conclude that, in contrast with shared-nothing systems, the PLP-Regular and PLP-Leaf designs provide low repartitioning costs that allow frequent repartitioning attempts and facilitate the implementation of responsive and lightweight dynamic load balancing mechanisms. We present one such mechanism in the next section.

4.4 A Dynamic Load Balancing Mechanism for PLP

At a high level, any dynamic load balancing mechanism has the same functionality. During normal execution it has to observe the access patterns and detect any skew that causes load imbalance among the partitions. Once the mechanism detects the troublesome imbalance, it triggers a repartition procedure. It is very important for the detection mechanism to incur minimal overhead during normal operation and not to trigger repartitioning when it is not really needed. After the mechanism decides to repartition, it should determine a new partitioning configuration, so that the load is again uniformly distributed. This decision depends on various parameters, such as the recent load of each partition and the available hardware parallelism. Finally, after the new configuration has been determined, the system has to perform the actual repartitioning. The repartitioning should be done in a way that minimizes the drop in performance and the duration of this process.

Thus, any dynamic load balancing mechanism that we build on top of PLP (or any partitioning-based system in general) should;

- perform lightweight monitoring,
- make robust decisions on the new partition configuration, and
- repartition efficiently when such decision is made.

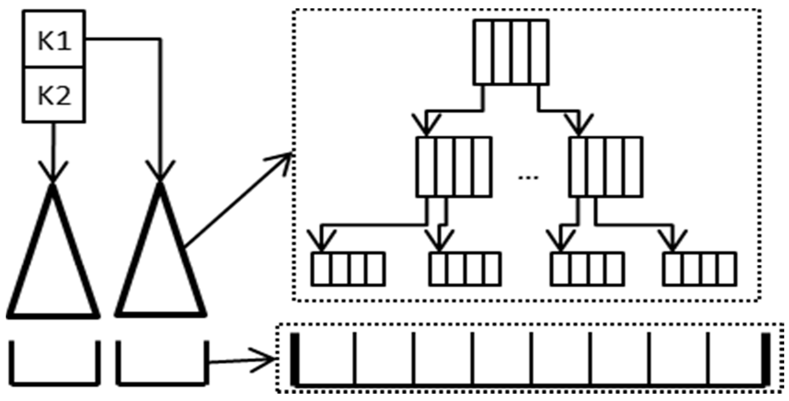


Figure 4.4: A two-level histogram for MRBTrees. The buckets that track the load across partitions (left-hand side) and the sub-buckets that track the load distribution within each partition (right-hand side).

Section 4.3 has already shown that PLP provides the infrastructure for efficient repartitioning. In this section, we present techniques for lightweight monitoring (Section 4.4.1) and deciding on the new partitions (Section 4.4.2). We call the overall mechanism *DLB*.

4.4.1 Monitoring

DLB collects information about the system's load across partitions and stability dynamically through monitoring some indicators of such system behavior. This monitoring is crucial to decide:

- when to trigger a repartition operation and
- what the new partitioning configuration should be.

Candidate indicators to monitor are

- the overall *throughput* of the system,
- the amount of work each partition performs (the length of the *request queues* of each partition's worker thread), and
- the number of data accesses in each partition (the statistics kept by the *two-level histogram* structure described below).

Throughput is one of the indicators of a system's stability. However, it is not enough to be able to trigger repartitioning whenever it is needed. For example, let's consider that DLB monitors only the overall throughput of the system and raises flags when changes in throughput are larger than a threshold value. If the initial partitioning configuration of the system is not

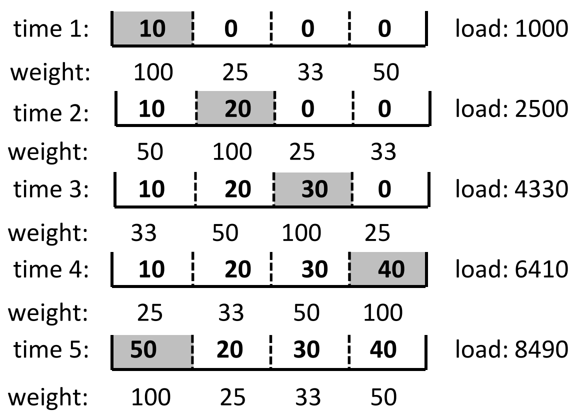


Figure 4.5: The aging algorithm example. Each row shows the state of a histogram sub-bucket that corresponds to a sub-range of a partition during the indicated time period at the left-hand side. This sub-bucket has four age-buckets; i.e., tracks four time periods. The gray age-bucket indicates the access count to this sub-range during the current time period, which is given higher weight while calculating the access load for this particular sub-bucket/sub-range.

optimal, then the throughput monitoring might fail in capturing the effect of load imbalance (e.g., having load imbalance, but stable throughput from the start). Furthermore, there might be uniform drops or increases in the incoming request traffic for each partition, which would trigger unnecessary repartitioning. Finally, the information about the throughput is not useful for the DLB component that decides on the new configuration (presented in Section 4.4.2). Therefore, DLB also maintains information about the load of each partition.

To determine the load of individual partitions, DLB monitors the request queue length of each partition (*request loads*) and a two-level histogram structure that employs *aging* (*access loads*). The request queues help in detecting the load imbalance across partitions, whereas the aging two-level histogram maintains load distribution within each partition. Figure 4.4 sketches the histogram structure. In addition to the one high-level bucket that keeps the access count for the whole partition (left-hand side of Figure 4.4), the histogram has sub-buckets for counting the accesses to the sub-ranges in a partition's key range (right-hand side of Figure 4.4). The number of sub-buckets within each partition is tunable and determines the monitoring granularity.

Each sub-bucket in the histogram is implemented as an array of *age-buckets*, shown in Figure 4.5 with an example. There is one active age-bucket at a time. When a record is accessed, the active age-bucket of the sub-bucket that this record's partition range corresponds to is incremented by one. At regular time intervals the age of the histogram increases. Whenever the age of the histogram increases, the next age-bucket is reset and starts to count the accesses.

When calculating the access load of a sub-bucket in the histogram, the recent age-buckets are given more weight than the older ones. More specifically, if a sub-bucket consists of A age-buckets, the access count value in the ith age-bucket is l_i, and the current age-bucket is the cth bucket, then we calculate the access load L for the sub-bucket as follows:

$$L = \sum_{i=c}^{A+c-1} \frac{100 \times l_{i mod(A)}}{(i-c+1)}.$$

DLB frequently monitors the throughput and the length of the request queues to detect load imbalance and trigger repartitioning. On the other hand, it analyzes the histograms only after repartitioning is triggered. The overall monitoring mechanism incurs very low overhead (as Section 4.5.2 evaluates) and it also provides adequate information for DLB to decide on the new partitions.

4.4.2 Deciding New Partitioning

DLB's algorithm for reconfiguring the partition key-ranges depends on the monitoring of the worker threads' request queues and aging two-level histogram structure discussed previously. This section first describes the algorithm that determines the new partition ranges for a single table and then it adopts this algorithm for multiple tables.

Deciding on the new partitioning within a single table

To describe the algorithm, let N be the total number of partitions and Q_i be the number of requests at the request queue of the ith partition. Then, the ideal number of requests for each partition's queue is:

$$Q_{ideal} = \sum_{i=1}^{N} \frac{Q_i}{N}.$$

Knowing Q_{ideal}, DLB has to decide on the ideal data access load for each partition. Let L_i be the access load of the ith partition, which can be calculated as the sum of the access loads of its sub-buckets. If the access load of the jth sub-bucket in ith partition is $L_{i(j)}$ (calculated as shown in Section 4.4.1) and there are M sub-buckets in each partition, then L_i is:

$$L_i = \sum_{j=1}^{M} L_{i(j)}.$$

DLB calculates the ideal data access load for partition i (LI_i) using the ideal request load (Q_{ideal}), the request load of partition i (Q_i), and the access load of partition i (L_i). Therefore, LI_i is:

$$LI_i = \frac{Q_{ideal} \times L_i}{Q_i}.$$

Algorithm 3 Calculating ideal loads.
Total number of partitions is N.
Total number of sub-buckets is M.
The access load for partition i is L_i.
The access load for sub-bucket j in partition i is $L_{i(j)}$.
Ideal access load for partition i is LI_i.
The ideal access load range for partition i is $[LI_i - t, LI_i + t]$.

```
1:  for $i = 1 \rightarrow N-1$ do
2:    while $L_i < LI_i - t$ do
3:      {Move leftmost (0th) sub-bucket range from $i+1$ to $i$}
4:      if $L_i + L_{i+1(0)} > LI_i + t$ then
5:        Split sub-bucket range for $L_{i+1(0)}$ into two
6:      else
7:        $L_i = L_i + L_{i+1(0)}$
8:    while $L_i > LI_i + t$ do
9:      {Move rightmost ($M$th) sub-bucket range from $i$ to $i+1$}
10:     if $L_i - L_{i(M)} < LI_i - t$ then
11:       Split sub-bucket range for $L_{i(M)}$ into two
12:     else
13:       $L_i = L_i - L_{i(M)}$
```

The reason for DLB to involve both the request loads from the request queues of the worker threads (Q_i) and access loads from the aging two-level histogram (L_i) while calculating the LI_i values is that not all data accesses create the same amount (or duration) of work. Q_i is a better indicator of a partition's load in terms of the amount of work the partition has to perform, whereas L_i correlates this amount to the data access within that partition.

Since the granularity of the access load information depends on the number of sub-buckets in the histogram, it is difficult for DLB to achieve precise ideal loads after repartitioning. Therefore, DLB only tries to approximate the ideal value. Algorithm 3 sketches how the new key-ranges are assigned. DLB iterates over all partitions except for the last one. While the estimated access load L_i at a partition is less than $LI_i - t$ for some t value, it moves the key range of the leftmost sub-bucket from the $(i+1)$th partition to ith. Similarly, while the load at a partition is larger than $LI_i + t$, it moves the key range of rightmost sub-bucket from the ith partition to $(i+1)$th. If the moved sub-bucket causes a significant change in the calculated load (more than $2 \times t$), then this sub-bucket's range is divided into two and the sub-bucket is substituted by two new sub-buckets. In this case, the number of sub-buckets in that partition increases by one and the access load of the old sub-bucket is distributed equally between the new sub-buckets.

Figure 4.6 has an example of how Algorithm 3 is applied. In the example, there are three partitions on a table and Figure 4.6 shows the two-level histogram for each partition. The first-level of the histogram tracks down the number of accesses to a partition's range, which is 40 units in this example. The second-level of the histogram, the 4 sub-buckets, keeps the

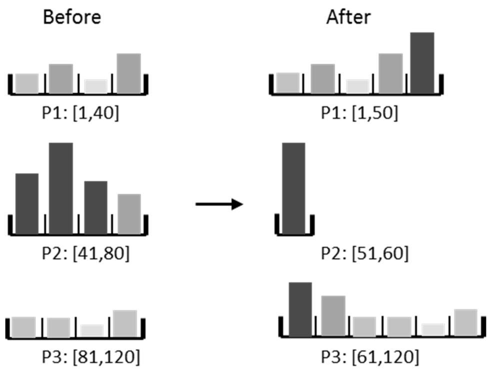


Figure 4.6: Example of how to decide on the new partition ranges.

number of accesses to sub-ranges in a partition, which is 10 units in this example. The higher bar in a sub-bucket indicates that the sub-range that corresponds to that sub-bucket has more load. Initially, each partition has equal key-ranges, shown in the left-hand side of Figure 4.6. If we assume that each partition has to perform an equal amount of work per request, the loads in this configuration are not balanced among the partitions. Therefore, the repartition manager triggers repartitioning. Based on Algorithm 3 the new partitions are decided by moving around the sub-buckets to create almost-equal loads among the partitions. The result is shown on the right-hand side of Figure 4.6; the most loaded regions end up in partitions with smaller ranges, like the second partition in Figure 4.6, and the lightly loaded regions are merged together.

Deciding the number of partitions of each table

The algorithm presented above is just for one table and assumes that the number of partitions before and after the repartitioning operation does not change. To determine how many partitions a table should have is another issue and requires knowledge of all the tables in the database. [2]

In our setting, initially, the number of partitions for a table is determined automatically to be equal to the number of hardware contexts supported by the underlying machine. To find what the number of partitions for a table should be dynamically, based on the workload trends; let

- T be the number of tables,
- N_{total} be the upper limit on the total number of partitions for the whole database,
- Q_i be the total number of requests for table i,
- N_i be the number of partitions for table i,

[2] In [154], we propose a more advanced version of this process.

- QT_{avg} be the average number of requests for all the tables,
- N_{avg} be the average number of partitions for a table, and
- $\#CTX$ be the total number of available hardware contexts supported by the machine being used to run the database system.

Based on the initial total number of partitions, we define N_{total} as:

$$N_{total} = T \times \#CTX.$$

As a result, N_{avg} will be:

$$N_{avg} = \frac{N_{total}}{T} = \#CTX.$$

The QT_i values are known from the request queues, and therefore, QT_{avg} can be calculated as:

$$QT_{avg} = \frac{\sum_{i=1}^{T} QT_i}{T}.$$

The goal is to find the N_i values, which can be derived from the following formula:

$$\frac{QT_{avg}}{N_{avg}} = \frac{QT_i}{N_i}.$$

4.4.3 Using Control Theory for Load Balancing

In our prototype, the system immediately tries to adjust to a new configuration, once a target load value is determined for each partition. Thus there is always the danger of over-fitting, especially for the workloads that observe data access skew with frequently changing hot-spots. Since repartitioning is not expensive for PLP (except for PLP-Partition), it can repartition again very quickly to alter the bad effects of a previous bad partitioning choice. Rather than directly aiming to reach the target load, a more robust technique would be to employ control theory while converging to the target load [120]. Control theory can increase the robustness of our algorithm, prevent the system from repartitioning unnecessarily and/or resulting with wrong partitions, and reduce the downtime faced by PLP-Partition during repartitioning. Nevertheless, it is orthogonal to the remaining infrastructure and it could be easily integrated in the current design. The prototype implementation does not employ control theory techniques. But the evaluation, presented next, shows that DLB allows PLP to balance the load effectively.

4.5 Evaluation

The goal of this section is to first quantify the overhead of the dynamic load balancing mechanism under normal operation, and then, measure how quickly and effectively it reacts against skew and load imbalances. More specifically, the evaluation measures the following:

- the overhead of DLB during regular transaction execution in Section 4.5.2,
- the effectiveness of DLB's dynamic load balancing and repartitioning across PLP variants in Section 4.5.2,
- DLB's impact while accessing the hot spots of the database in Section 4.5.2, and
- the effect of the secondary indexes on DLB's performance in Section 4.5.3.

4.5.1 Experimental setup

We evaluate three different design options:

- a non-partitioned system, labeled *Conventional*.
- a statically partitioned shared-everything system that employs the three PLP variations (Section 3.3.3), labeled *PLP-Regular, PLP-Partition, PLP-Leaf*.
- our prototype system that integrates the DLB mechanism on top of PLP using the PLP prototype from Section 3.4, labeled *PLP-Reg-DLB, PLP-Part-DLB, and PLP-Leaf-DLB*.

To allow fair comparisons the systems are built on top of the same storage manager, Shore-MT [96, 172], and use the same driver code. All the experiments use the `GetSubscriberData` transaction from the TATP benchmark. We picked this transaction since it probes a record from the `Subscribers` table, which provides 10000 tuples per scaling factor (and per partition in our experiments). Therefore, the number of records that have to change partitions after repartitioning is good enough to understand the record movement cost among different PLP variations. In addition, the short nature of the transaction also stresses the update of the data access histogram DLB uses.

Finally, the experiments are performed on the same two multicore machines that are used to evaluate PLP (Section 3.4.1): an x64 box, with four sockets of quad-core AMD Opteron 8356 processors, clocked at 2.4GHz and running Red Hat Linux 5; and a Sun UltraSPARC T5220 server with a 64-core Sun Niagara II chip clocked at 1.4GHz and running Solaris 10. We keep the buffer pool sizes big enough to maintain the entire database in memory.

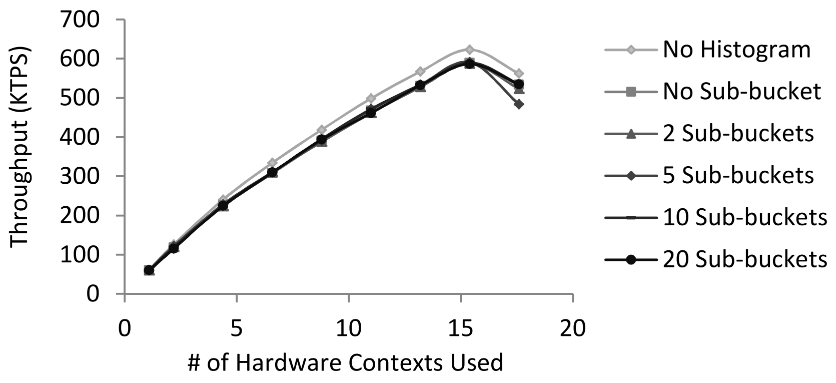


Figure 4.7: Overhead of updating histogram for DLB under normal operation.

4.5.2 Overhead in Normal Operation

Under normal operation, DLB should impose minimal overhead. DLB's monitoring component performs three operations:

- maintaining the histograms with access information,
- continuously monitoring the throughput, and
- periodically analyzing the request queues of the worker threads

for detecting load imbalances.

Since the monitoring of the throughput and request queues is performed by a separate thread, it should not affect the throughput of the system at all unless all the CPUs in the system are utilized by the threads executing transactions. Therefore, the main source of overhead for DLB is updating the histogram.

Figure 4.7 shows the overhead caused by updating the aging histogram for each data access. Since the number of threads that try to update the histogram increases as we utilize more CPUs, the overhead of updating the histogram increases as well. On the other hand, increasing the number of sub-buckets does not have much effect. We note that the histogram is not a source of contention since each partition has their own sub-buckets that they update. Therefore, the overhead in updating the histogram purely comes from the extra work that a partition's worker thread has to perform while updating the histogram.

Overall, the monitoring component of DLB is fairly lightweight. On average histogram updates cause a 6% drop in throughput compared to the system running without a histogram and the maximum drop is 7-8%. Considering that the transaction we execute in our system is a short read-only transaction, we actually evaluate the worst case behavior here. For a transaction with updates, the number of transactions executed per second, and hence, the number of data

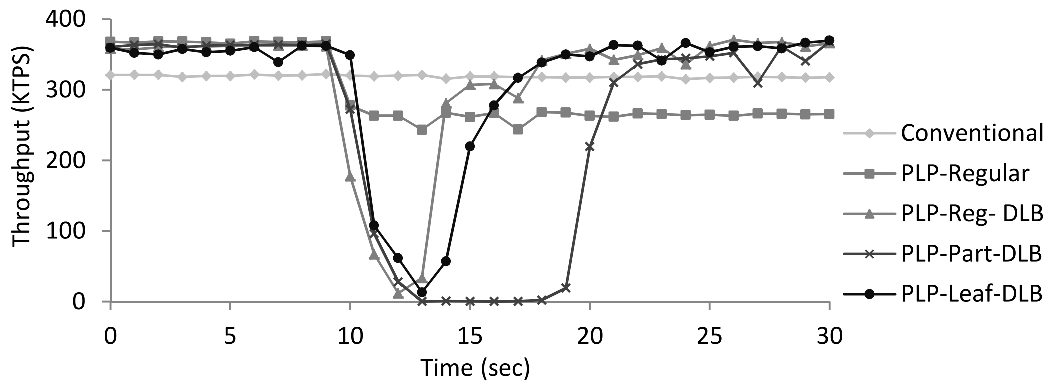


Figure 4.8: Dynamic load balancing when at time t=10 50% of the requests are sent to 30% of the database.

accesses would be lower. Fewer data accesses would cause fewer updates in the histogram, and therefore, less overhead.

Reacting to load imbalances

In order to evaluate how effectively DLB handles load imbalances, we perform the same experiment as the one in Figure 4.1. The PLP variations (*PLP-Regular*, *PLP-Reg-DLB*, *PLP-Part-DLB*, and *PLP-Leaf-DLB*) use 64 partitions, apply aging every second, and the load difference threshold value *t* is 10% of the ideal access load value for each partition. Initially the requests are distributed uniformly and at time point 10 (sec), 30% of the database starts to receive 50% of the requests.

As Figure 4.8 shows, the change in the access pattern causes a 30% drop in the throughput of *PLP-Regular*, making its performance worse than the performance of the non-partitioned *Conventional* system. On the other hand, the DLB-integrated PLP variations quickly detect the skew and bring the performance back to the pre-skew levels in less than 10 seconds. In particular, 2 seconds after the change in the access pattern, DLB has already decided on the new partitioning configuration, and around 8 seconds later it has performed ~126 repartition operations (63 splits and 63 merges). The throughput has some spikes for a short time after repartitioning, but in the end settles down.

In *PLP-Reg-DLB*, very few index entries are updated, leading to a shorter dip in throughput during repartitioning. *PLP-Leaf-DLB* experiences an almost equally short dip. However, *PLP-Part-DLB* suffers a much longer dip. For the statically partitioned PLP, Figure 4.8 has only the results for the statically partitioned PLP-Regular since the drop in throughput is almost the same for the other two statically partitioned PLP variations (PLP-Partition and PLP-Leaf).

DLB triggers a global repartitioning process, which affects all the partitions in the system. PLP-Regular and PLP-Leaf can handle this process very well. However, such global repartitioning is

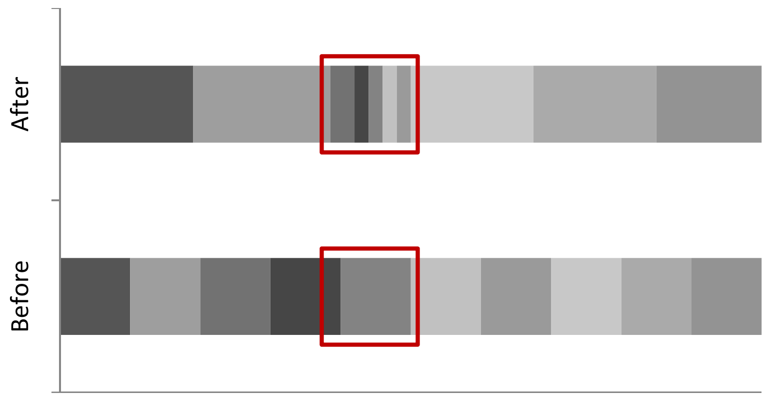


Figure 4.9: Partitions before and after the repartitioning.

Table 4.3: Average index probe times (in microseconds) for a hot record, as skew increases.

Skewed region (%)	Before Skew	After Skew	After Repartitioning
50	69	67	65
20	67	66	63
10	69	66	62
5	68	64	61
2	68	64	60

not suitable for PLP-Partition. PLP-Partition is the closest to a shared-nothing system in terms of repartitioning cost since it reorganizes a large number of heap pages (see Section 4.3 and Table 4.1). Therefore, its non-optimal behavior with DLB is as expected.

Speeding Up Accesses to Hot Spots

When DLB is effective, the *hot* regions end up in narrow partitions. The indexes for these partitions are shallower and provide shorter access times for the *hot* records. In addition, *hot* records that previously belong to the same partition, due to their key proximity, end up in different partitions. Figure 4.9 illustrates graphically the impact of DLB on the ranges of 10 partitions before and after repartitioning. The area within the rectangular region highlights the *hot* range; it is 10% of the total area that receives the 50% of the total load. Initially, labeled *Before*, the system has equal-length range partitions. After DLB kicks in and repartitioning completes, labeled *After*, the *hot* region has shorter-length range partitions while the not-so-loaded regions have larger-length partitions.

Table 4.3 shows the average index probe time (in microseconds) for a hot record as we increase the skew. For this experiment we use a single table with 640000 records. There is an index on this table, with 8KB pages and the primary key is an integer (4B). When there are 10 equal-range partitions, the height of each partition's subtree is 3. Each row in the table shows the average access time of a randomly picked record from a *hot* region which gets 50% of all the

Table 4.4: Average record probes per sec for a hot record, as skew increases.

Skewed region (%)	After Skew	After Repartitioning
50	13	13
20	7	29
10	7	73
5	32	108
2	63	155

requests, as the range of the *hot* region decreases – and the skew increases. The first column (*Before Skew*) shows the average access time when the requests are uniformly distributed. The second column (*After Skew*) shows the average access time when DLB is disabled and the request distribution is skewed. The third column has the average access time after DLB kicked in and completed a repartitioning.

As Table 4.3 shows, the access times for the randomly picked record are lower after we set the skew. This is probably due to some caching effect since the record is accessed more frequently when there is skew in data accesses. However, the access time after repartitioning is the lowest since the height of the subtree in the new *hot* partition is 2 whereas in the old partition it was 3 (the height of the subtrees for the other partitions remains as 3).

Table 4.4 shows the number of finished requests for the *hot* record after the skew and after DLB's repartitioning. Before repartitioning fewer requests are satisfied for the picked record because its partition is highly loaded with requests for other records in the same *hot* partition range. DLB distributes the *hot* range among multiple shorter-range partitions. Therefore, a single partition can serve more requests for the *hot* record. This results in a small throughput increase after repartitioning in Figure 4.8.

4.5.3 Overhead of Updating Secondary Indexes for DLB

In PLP-Leaf and PLP-Partition, whenever a record moves, every non-clustered index of the table needs to be updated with the record's new RID (see Section 3.3.3). This section measures the overhead of updating secondary indexes during repartitioning.

Figure 4.10 shows the effect of increasing the number of secondary indexes of a table on repartitioning under PLP-Leaf and PLP-Partition. Initially, there are 2 partitions of 320000 records each that receive uniform requests. After 5 seconds, 50% of the requests are sent to only 10% of the table and DLB triggers a repartitioning. We measure the throughput of the system as we increase the number of secondary indexes on the `Subscribers` table in the TATP database, from none up to 4 secondary indexes (indicated by the different lines on the graph).

As Figure 4.10 shows, the overhead for PLP-Leaf to update the secondary indexes is relatively low, because very few or none of the records needs to be moved. On the other hand, the

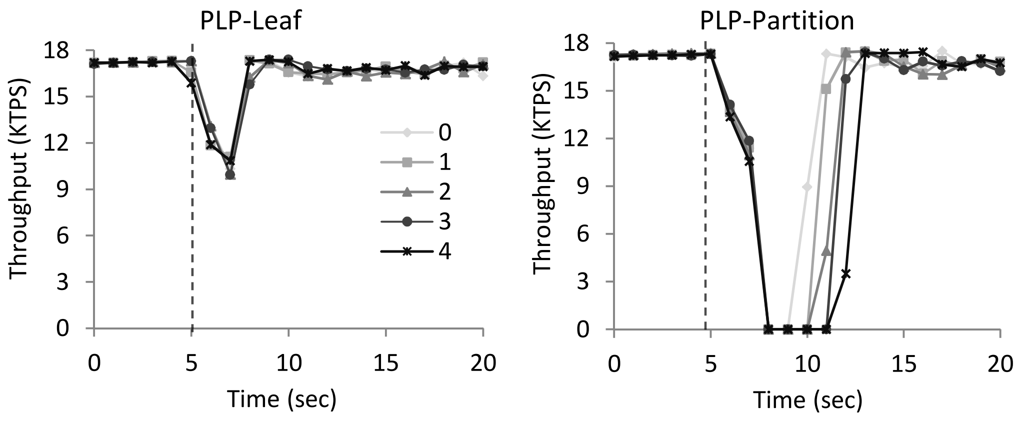


Figure 4.10: Overhead of updating secondary indexes during repartitioning for PLP-Leaf and PLP-Partition. At time t=5 50% of the requests are sent to only 10% of the database, which triggers repartitioning. Each line marked with number X indicates that the table has X secondary indexes.

overhead for PLP-Partition is much higher. PLP-Partition has to move a lot more records and hence update more entries in the secondary indexes. Therefore, repartitioning in PLP-Partition takes longer as we increase the number of secondary indexes for a table.

4.6 Related Work

Most of the related work on dynamic load balancing and repartitioning targets clustered (shared-nothing) environments. For example, Achyutuni et al. [1] analyze and compare different approaches for index reorganization during repartitioning in shared-nothing deployments. Lee et al. [118] propose an index structure similar to MRBTree, which eases the index reorganization during repartitioning in a shared-nothing system and Mondal et al. [131] extend this design by keeping statistics for each branch referenced by the root page of a partition's subtree. While the structure of [131] enables the observation of access patterns at a fine granularity, it gives all the accesses the same weight no matter how recent or old they are. Our aging two-level histogram assigns higher weight to the recent accesses. This allows us to have a more accurate view of the skewed access patterns and detect load imbalances quickly.

Our work is orthogonal to techniques that determine initial partitioning configuration. For example, in [164] the query optimizer is used to provide suggestions for the initial partitions, while Schism [39] creates initial static partitions in a way to minimize the number of distributed transactions by first representing the workload as a graph, and then, using a graph partitioning algorithm. Such tools only create the initial configuration. If the workload characteristics change over time, however, the system has to re-calculate the partitioning configuration and perform repartitioning.

An extension of Schism, Sword [157], proposes a graph algorithm that allows incremental data movement among different partitioning solutions improving the partitions over time. Horticulture [150] is another mechanism for automatic partitioning, which uses the database schema, stored procedures for the target workload, and a workload trace from a prior run while determining the data partitions. Both of these techniques enable dynamic repartitioning. However, they give much higher weight to find the best possible partitions rather than handling the repartitioning in a fast and lightweight manner at run-time.

Shinobi [201] uses a cost model to decide whether the benefits of a new partitioning configuration are worth the cost of repartitioning. Shinobi focuses on insert-heavy workloads, where data is rarely queried and when queried the queries focus on a small region of the most recently inserted records. Its benefits primarily come from avoiding indexing the large infrequently accessed parts of the database. We consider mainstream transactional workloads, where the entire database is accessed and we cannot drop any indexes.

Finally, the histogram-based technique we use is influenced by previous work on maintaining dynamic histograms on data distributions for accurately estimating the selectivity of query predicates [47, 59]. In our case, we are interested in the frequency of accesses to a particular data region rather than the data distribution.

4.7 Conclusions

Evolving and skewed data distributions and access patterns are one of the most important problems of partitioned database management systems, which become increasingly important due to their natural potential of exploiting the benefits of modern hardware. Such partitioned systems need mechanisms that enable them to quickly and effectively react to changes in the load. In this chapter, we discussed challenges of robust dynamic load balancing and described one such solution, called DLB, for physiologically-partitioned transaction processing systems since they provide a good infrastructure for repartitioning. Evaluation of the proposed technique shows that it is lightweight, yet manages to detect and react effectively to load imbalances.

Characterizing OLTP Benchmarks Part II

5 From A to E: Analyzing TPC's OLTP Benchmarks

Introduced in 2007, TPC-E is the most recently standardized OLTP benchmark by TPC. Even though TPC-E has already been around for seven years, it has not gained the popularity of its predecessor TPC-C: all the published results for TPC-E use a single database vendor's product. TPC-E is significantly different than its predecessors. Some of its distinguishing characteristics are the non-uniform input creation, longer-running and more complicated transactions, more difficult partitioning, etc. These factors slow down the adoption of TPC-E. In turn, there is little knowledge in the community about how TPC-E behaves micro-architecturally and within the database engine.

To shed light on TPC-E, we implement it on top of a scalable open-source database engine, Shore-MT, and perform a workload characterization study, comparing it with the previous, much better known OLTP benchmarks of TPC: TPC-B and TPC-C[1]*. In parallel, we study the evolution of the OLTP benchmarks throughout the decades. Our results demonstrate that TPC-E exhibits similar micro-architectural behavior to TPC-B and TPC-C, even though it incurs less stall time and higher instructions per cycle. On the other hand, within the database engine it suffers more from logical lock contention. Therefore, we argue that, on the hardware side, TPC-E needs less aggressive processors; whereas on the software side it can benefit from designs based on intra-transaction parallelism, logical partitioning, and optimistic concurrency control to minimize the effects of lock contention without introducing distributed transactions.* [2]

5.1 Introduction

For the past decades, the data management ecosystem and in turn the database and hardware markets have evolved primarily around two applications: online transaction processing (OLTP) and online analytical processing (OLAP). Transaction processing benchmarks are the gold standard for comparing products by different database and hardware vendors, and are regularly used for marketing purposes [86, 142]. For the last two decades, TPC-C [191] has

[1] Due to the similarities between TPC-A and TPC-B (Section 2.4.1), we omit TPC-A in this study.
[2] This chapter uses material from [186].

been the most widely used OLTP benchmark by the majority of industry and academia. TPC-C consists of simple short-running transactions with frequent updates and less frequent index scans. On the other hand, the benchmark of choice for OLAP workloads is TPC-H [195]. TPC-H observes more complicated long-running read-only queries with frequent index and file scans. The data management stacks, from the database down to hardware, are typically optimized for these two extreme benchmarks.

As Section 2.4 also detailed, the Transaction Processing Performance Council (TPC) introduced the TPC-E benchmark [193] in 2007 in order to represent OLTP workloads more realistically. TPC-E is an OLTP workload that includes transactions for real-time business intelligence combined with client-side requests. It acts in between a typical OLTP and an OLAP benchmark. The design decision for TPC-E was to create a sophisticated OLTP benchmark, having more complicated and longer transactions when compared to TPC-C, relying on the extensive use of non-primary indexes, observing data and access skew, applying integrity and referential constraints, and being less amenable to partitioning.

Both industry and academia are slow at adopting TPC-E. For example, even though the benchmark was standardized seven years ago, all the published results for TPC-E use the same database product (Microsoft SQL Server) [194]. Due to TPC-E's significant differences from the other benchmarks, it is not easy to extrapolate how systems perform when they run TPC-E (and TPC-E-like applications).

Existing experimental studies typically use database benchmarks other than TPC-E. Previous studies of OLTP and OLAP benchmarks, either micro-architectural [3, 106, 161, 177] or profiling [95, 97, 145, 147], provide valuable results. However, they fall short of explaining the behavior TPC-E is expected to exhibit. Recent work that analyzes TPC-E either focuses only on the I/O behavior [33, 104] or reports micro-architectural results on only one type of machine while running TPC-E on a commercial RDBMS and treating the database as a black-box [54]. To date, there is neither an analysis of the TPC-E benchmark on various hardware platforms nor a comprehensive breakdown of the execution time with respect to database engine components.

This chapter performs a detailed study of TPC-E. We characterize where it spends time within an open-source database engine and how it behaves micro-architecturally on two different hardware platforms, one in-order and one out-of-order machine. In parallel, we compare TPC-E to the well-known OLTP benchmarks and observe how TPC's transactional benchmarks have evolved over the years. Then, we discuss what kind of changes in database and hardware systems can be more beneficial for such a workload.

The findings of our study are as follows:

- Our micro-architectural study demonstrates that TPC-E is actually very similar to the previous OLTP benchmarks in terms of its micro-architectural behavior. It highly suffers from L1 instruction misses and exhibits low instructions per cycle (IPC); IPC is smaller than one on a machine that has ability to execute four. Thus, we argue that TPC-E-like workloads need less aggressive processors with a lower instruction issue width on the hardware side. In addition, even though simultaneous multi-threading (SMT) hides some of the stalls caused by instruction misses and almost doubles the IPC, we need more effective solutions like intra-transaction parallelism [36, 145] or computation spreading [14, 30] to better utilize modern processor cores.

- Our profiling study reveals that, within the database engine, TPC-E spends 70% more time inside the lock manager compared to both TPC-B and TPC-C for a configuration with an order of magnitude bigger database size. TPC-E's more complicated schema and transactions make it less straightforward to physically partition a TPC-E database to eliminate its locking overhead due to the significant number of distributed transactions such a design would cause. However, TPC-E can benefit from shared-everything designs that aim to minimize locking with logical [145] or physiological partitioning [147], or systems that rely on optimistic concurrency control [46, 116] to improve system performance.

The rest of the chapter is organized as follows. Section 5.2 surveys the related work. Section 5.3 describes our TPC-E implementation on top of Shore-MT [172] and experimental methodology. Section 5.4 and Section 5.5 present the profiling and micro-architectural analysis, respectively, for TPC-E in comparison with TPC-B and TPC-C. Finally, Section 5.6 summarizes the analysis results, discusses possible design optimizations for both upcoming hardware and storage managers for OLTP systems, and then, concludes.

5.2 Related Work

There is a large body of related work on workload characterization for database workloads. Barrosso et al. [16] investigate the memory system behavior of OLTP and DSS style workloads using TPC-B and TPC-D [192], respectively, both on a real machine and with a full-system simulation. They find that these two types of workloads need different architectural designs in terms of the memory system. Ranganathan et al. [161] use the same workloads as in [16]. However, they only focus on the effectiveness of out-of-order execution on SMPs while running these workloads in a simulation environment. Neither TPC-B nor TPC-D can be representative of TPC-E since TPC-E has much more complicated and longer-running transactions than TPC-B and it is not completely read-only like TPC-D.

Keeton et al. [106] experiment with TPC-C on a 4-way Pentium Pro SMP machine and perform a similar analysis to [16, 161]. Although TPC-C is closer to TPC-E compared to both TPC-B and TPC-D, it still has major differences from TPC-E as described in Section 2.4. Stets et al. [177] perform a micro-architectural comparison between TPC-B and TPC-C. We add TPC-E to this comparison and also analyze what happens within the storage manager.

Ailamaki et al. [3] examine where the time goes on four different commercial DBMSs with a micro-benchmark to have a finer-grain understanding of the memory system behavior of multiprocessors. Hardavellas et al. [69] analyze OLTP, with TPC-C, and DSS, with TPC-H, on both in-order and out-of-order machines by using a simulation environment. Rather than optimizing the hardware for the workloads, these two papers focus on the implications on the DBMS side in order to utilize the underlying hardware more effectively. In this chapter, we consider both the hardware and the DBMS design for optimal TPC-E execution.

Johnson et al. [95, 97] and Pandis et al. [145, 147] provide detailed analysis on where the time goes within the storage manager for typical OLTP benchmarks. Their main aim is to highlight components that become scalability bottlenecks in the existing systems and propose alternative designs that remove those bottlenecks. Here, we also perform the same analysis with TPC-E and discuss which of their techniques can or cannot help TPC-E, and also expose the bottleneck on L1 instruction misses.

There are a few performance analysis papers that use TPC-E. For example, [33, 104] use I/O traces of a production database server running TPC-E in order to study its I/O behavior. In [33] the authors compare the I/O behavior of TPC-C and TPC-E. We do not study the I/O behavior. For our experiments we use memory-resident databases and focus on the micro-architectural behavior. Ferdman et al. [54, 55, 56] present a detailed micro-architectural analysis with many types of workloads on Intel X5670 processors, focusing on the architectural design needs of the scale-out workloads. They provide a comparison between the scale-out workloads and server workloads, like TPC-C and TPC-E. Our analysis uses a very similar methodology while analyzing the OLTP benchmarks micro-architecturally on our Intel X5660 processors and our high-level conclusions corroborate their findings. In addition, we perform such a micro-architectural analysis on different hardware platforms to understand the behavior when we switch from an in-order core to an out-of-order one. Moreover, we also demonstrate which components TPC-E stresses within the storage manager as opposed to a pure micro-architectural study. Finally, Lang et al. [115] use TPC-E to show that a cluster of *wimpy* (low-power Atom-based) nodes is not as energy-efficient as a cluster of traditional server-grade processors (Xeon-based). This chapter does not focus on energy-efficiency.

5.3 Setup and Methodology

Before diving into the analysis results, here we describe the software setup and two servers used for the analysis.

5.3.1 Hardware

We used two servers for our experiments: a Sun UltraSPARC T5220 and a server with two Intel Xeon X5660 processors. Table 5.1 lists the characteristics of each server in detail. The diversity and degree of hardware parallelism on these systems make them good candidates for this study to reflect the behavior of the workloads we evaluate on different types of modern hardware.

Table 5.1: Server properties.

Server	UltraSPARC T2	Intel Xeon X5660
# of Sockets	1	2
# of Cores per Socket	8 (in-order)	6 (OoO)
# of Hardware Contexts	64	24
Clock Speed	1.40GHz	2.80GHz
Memory	64GB	48GB
L3 (shared) size / access latency	-	12MB / 29 cycles
L2 (shared) size / access latency	4MB / 20 cycles	-
L2 (per core) size / access latency	-	256KB / 6 cycles
L1-I (per core) size / access latency	16KB / 3 cycles	32KB / 4 cycles
L1-D (per core) size / access latency	8KB / 3 cycles	32KB / 4 cycles
OS	SunOS 5.10 Generic_141414-10	Ubuntu 10.04 with Linux kernel 2.6.32

5.3.2 TPC-E Implementation

We implement TPC-E in Shore-Kits, which provides a platform to implement benchmarks to be run on Shore-MT (as detailed in Section 2.6). The query plans for the TPC-E transactions are taken from a TPC-E implementation of a major database vendor. As for the index decisions, we initially adapted the indexes from the same kit. Later, however, we had to change some of the indexes in order to optimize performance when running on Shore-MT. For example, Shore-MT's API allows Shore-Kits to use only unclustered indexes, whereas the kit of the commercial database uses clustered ones for the primary indexes. Therefore, the optimal index decisions varied between Shore-Kits and the kit of the commercial database. Due to its large number of tables and longer and more complicated transactions, TPC-E was by far the most difficult benchmark implemented in Shore-Kits.

TPC-E stresses Shore-MT in ways previous benchmarks do not. It pinpointed code-paths, exposing previously undetected bugs and performance bottlenecks. Therefore, it helped us to further improve Shore-MT. For example, Shore-MT had an implementation of forward and backward index scans. But the backward index scans were disabled, because they were causing a large number of deadlocks in some workloads. Debugging and re-enabling backward index scans in Shore-MT improved performance of TPC-E by three orders of magnitude on the Intel server used in this study (Table 5.1).

5.3.3 Software Setup

We chose the most optimal configuration options we determined empirically for all the benchmarks running on top of Shore-MT to make sure that we run them without any obvious scalability bottlenecks and better utilize the hardware resources. In TPC-B we pad the records of `Branch` and `Teller` tables so that a single database page only has a single record. This minimizes false sharing of database pages and avoids latching contention, which can be a fundamental bottleneck for typical shared-everything architectures (Chapter 3). We also enable speculative lock inheritance (SLI) [95] and logging optimizations from Aether [97, 98] to reduce the bottlenecks coming from the lock and log managers, respectively, for the benchmarks that benefit from these techniques.

We use memory-resident databases for our experiments and flush the log to RAM due to not having a suitably fast I/O sub-system. A configuration that allows I/O in our infrastructure might cause an unreasonably slow and highly suboptimal OLTP system, and therefore, unrealistic micro-architectural conclusions.

Furthermore, for TPC-B and TPC-C we spread the requests based on the primary key of the `Branch` and `Warehouse` tables, respectively, to reduce logical lock contention. In order to do that, we picked scaling factors that are equal to the number of hardware contexts available on the machine where a specific experiment is run, since the scaling factor is equal to the number of `Branches` in TPC-B and `Warehouses` in TPC-C. In other words, on the Intel machine we picked a scaling factor of 12 and 24 when hyper-threading is disabled and enabled, respectively, and on the SPARC machine we picked a scaling factor of 64. Unfortunately, for TPC-E, it is not straightforward determining how to spread the requests due to its more complex schema and transactions that do not have correlation based on any primary key column for the majority of the database tables. To be able to run an in-memory database, we picked a database size that contains 1000 customers for TPC-E. We set the *working days* and *scaling factor* parameters to 300 and 500, respectively, which are the default values for these parameters in the TPC-E specification.

5.3.4 Experiments

On the Intel machine, we experiment with two cases; when hyper-threading (HT) is off and when it is on. When hyper-threading is on, the Intel machine supports two hardware contexts running at the same time on one core to be able to overlap the stall time of one of the threads with the execution of the other. This property is analogous to the simultaneous multi-threading (SMT) support in the SPARC machine where each core has support for eight hardware contexts by default, which is actually one of the main design principles of the UltraSPARC T2 architecture.

Before taking any measurements, we start with a newly populated database, make each worker thread in the system execute 1000 transactions to warm-up the caches, and then perform a one-minute run. The tools used to collect the hardware counter values and profiling results during these runs are mentioned in the related sections.

Table 5.2: High-level statistics of each benchmark per scaling factor.

	TPC-B	TPC-C	TPC-E
# of records	~ 10 thousand	~ 600 thousand	~ 117 million
# of heap pages	147	~ 12 thousand	~ 1 million
# of index pages	91	~ 6 thousand	~ 1 million
Average per xct			
# of records accessed	4	36	149
# of row-level locks	10	54	171
# of higher-level locks	10	36	69
# of unique heap pages accessed	4	23	40
# of unique index pages accessed	4	33	33
# of heap pages accessed	7	49	125
# of index pages accessed	4	90	211

5.4 Profiling Analysis

In order to further understand the high-level characteristics of each benchmark, first, we report statistical information collected from the storage manager in Section 5.4.1. Then, in Section 5.4.2, our profiling analysis identifies the components of the storage manager each benchmark spends the most time in.

5.4.1 High-level Analysis

Table 5.2 contains the high-level statistics of each benchmark to further highlight the changes in complexity with each OLTP benchmark standardized by TPC. These statistics are independent of the underlying hardware. We chose a scaling factor of one for each benchmark in this part of the analysis. This corresponds to one `Branch` in TPC-B, one `Warehouse` in TPC-C, and one-thousand `Customers` in TPC-E. For the initial database, we measure the number of records each benchmark has and how many pages it uses in Shore-MT, which uses 8KB pages by default. Then, we use the existing statistic measurements within Shore-MT to see how many records, locks, and pages on average a transaction accesses for each benchmark while performing a run with one worker thread executing transactions.

As expected, Table 5.2 re-emphasizes the complexity increase from TPC-B to TPC-E. TPC-E has several orders of magnitude more records per scaling factor compared to TPC-B and TPC-C, and a much larger database size as the total number of heap and index pages indicates. TPC-B only touches one record per table; hence, it accesses few database locks and pages. TPC-C accesses almost ten times the records TPC-B accesses per transaction in its transaction mix, increasing the number of locks and database pages it accesses as well. Finally, TPC-E performs around four times the record accesses of TPC-C, which is also reflected in the higher number of row-level locks it has to acquire. However, the total number of locks acquired does not increase accordingly since Shore-MT escalates to higher-level locking from row-level

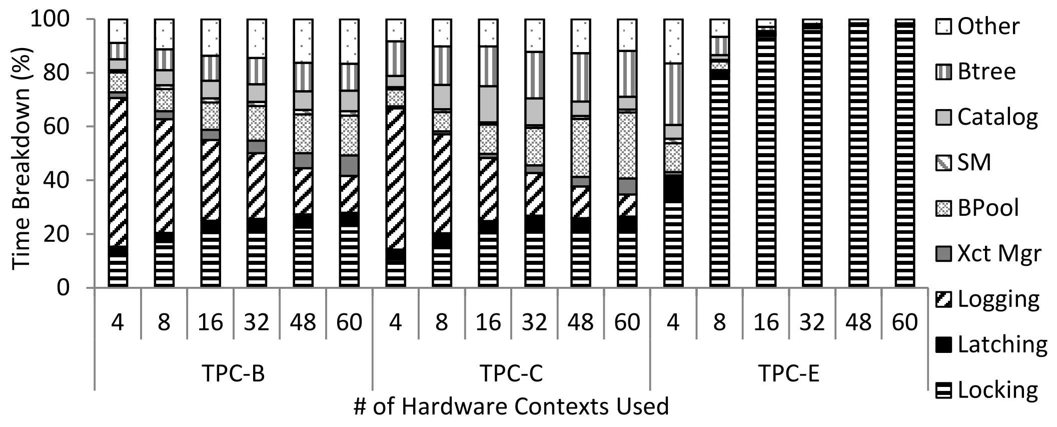


Figure 5.1: Time breakdown as the machine load increases on UltraSPARC T2.

locking when a single transaction accesses more than a threshold of records (the default value is twenty-five in Shore-MT).

Table 5.2 reports two values for the average number of pages accessed in a transaction: the unique number of pages accessed and the total number of pages accessed, which is also the number of times a page is requested from the buffer pool. Such a separation reveals that even though TPC-E accesses more than twice the index pages that TPC-C does, the number of unique index page accesses is the same for both workloads. The main reason for this is TPC-E's extensive index scans (Section 2.4.3). TPC-C does not re-access most of the index pages it touches, while TPC-E does this very frequently for the index leaf pages during its index scans; it sequentially reads an index leaf page and hence frequently reuses that page. This results in TPC-E exhibiting lower L1 data cache miss rates as Section 5.5 shows.

5.4.2 Time breakdown

To get accurate time breakdowns within the storage manager, we use DTrace [48] on the SPARC machine. Figure 5.1 presents the results of the profiling as we increase the machine utilization, i.e., as we run more clients in the system.

Figure 5.1 highlights that the lock manager is one of the components where the OLTP benchmarks spend most of their time in a shared-everything database management system, which corroborates the results of [145]. The lock manager becomes the main bottleneck especially for TPC-E, making it unable to utilize more than eight hardware contexts, while both TPC-B and TPC-C are able to almost fully utilize the machine with smaller database sizes.

Looking at the other components in Figure 5.1 reveals that *Logging* is the next problematic component for TPC-B and TPC-C. It becomes, however, less significant as we increase the system utilization since we adopt the logging optimizations of [97] that benefit from combining logging requests as the number of clients in the system increases. *Btree* and *BPool* (buffer-

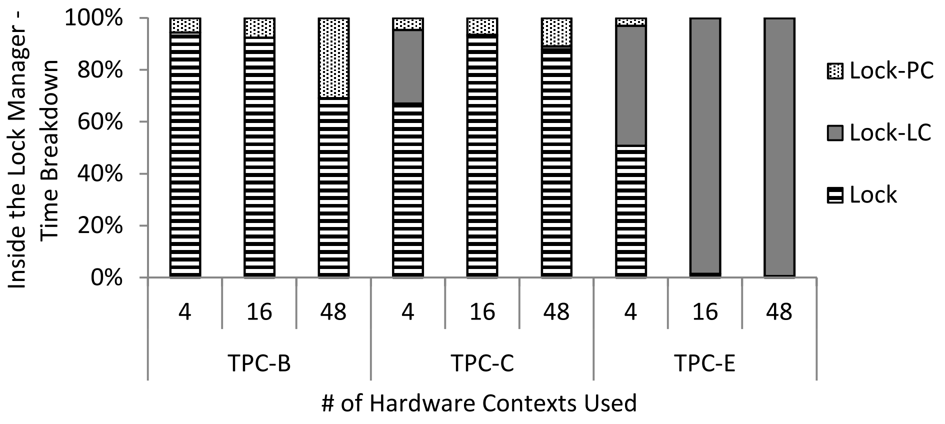


Figure 5.2: Time breakdown inside the lock manager as the machine load increases on UltraSPARC T2.

pool) come after *Locking* and *Logging*, since a transaction's execution is highly dependent on its index operations. The rest of the major components of a storage manager are *Catalog* (metadata manager), *SM* (storage manager API functionality), *Xct Mgr* (transaction manager), and *Latching*; in which none of the workloads spends a major part of their execution time.

Figure 5.2 focuses on the time spent inside the lock manager and shows the time breakdown of sub-categories:

- Physical lock contention, *Lock-PC*, represents the time spent while a thread tries to append its lock request to the linked-list of lock requests for a particular record or table lock.
- Logical lock contention, *Lock-LC*, represents the time spent until a record or table lock is granted after the lock request is appended to the list of requests for this lock.
- Finally, locking, *Lock*, is the time spent on performing the locking operation aside from the waiting time.

TPC-E mainly suffers from logical lock contention (Lock-LC) even though we use a larger database size for it compared to TPC-B and TPC-C. There are three main reasons for this outcome:

- TPC-E observes data and access skew, turning some of the data regions into hotspots (e.g., `Last_Trade` table),
- TPC-E transactions acquire on average more locks since they access a larger number of database records, and
- TPC-E transactions hold the locks they acquire for a longer duration since they are more complicated, longer running, and scan-heavy transactions.

Table 5.3: Number of worker threads used for benchmarks on the two machines.

Server	UltraSPARC T2	Intel Xeon X5660	
		No HT	HT
TPC-B	48	10	18
TPC-C	60	10	18
TPC-E	4	12	24

TPC-B and TPC-C, on the other hand, do not suffer from logical lock contention since the system can properly spread the requests and SLI [95] prevents physical lock contention from becoming problematic, leaving only the actual locking operation as the main time-consuming component within the lock manager.

However, as we will see in Table 5.3, the lock contention is not as problematic when we run TPC-E on the Intel machine, which has faster processors than the SPARC machine. The faster the processor, the faster the lock acquisitions and releases are, and hence the less time spent on lock contention. We come across this fact also when we run TPC-B. When two threads want to access the same `Branch` in a TPC-B database, they first acquire a read lock on the wanted `Branch` during the index probe according to ARIES/IM [129] (the default concurrency control scheme in Shore-MT). Later, when they want to upgrade their read locks to exclusive ones to update the `Branch`, they both wait for each other and they deadlock. While on the SPARC machine we observe such deadlocks, TPC-B runs without deadlocks on the Intel machine since the lock acquisitions are faster. Switching to ARIES/KVL [128], which has stricter concurrency control rules than ARIES/IM, makes this type of deadlocks disappear on the SPARC machine as well.

5.5 Micro-architectural Analysis

While performing a micro-architectural analysis for the OLTP benchmarks, we try to answer the following questions:

- Where do CPU cycles go on different types of modern hardware? Are they wasted on memory stalls or used to retire an instruction?
- Do stalls happen mainly due to instructions or data?
- What are the instruction and data miss rates?
- How much instruction-level (ILP) and memory-level (MLP) parallelism do OLTP benchmarks exhibit?
- What is the effect of simultaneous multi-threading (or hyper-threading)?

All the numbers reported in this section were obtained when the workloads have their peak performance on the corresponding server with their optimal configuration on Shore-MT. Table

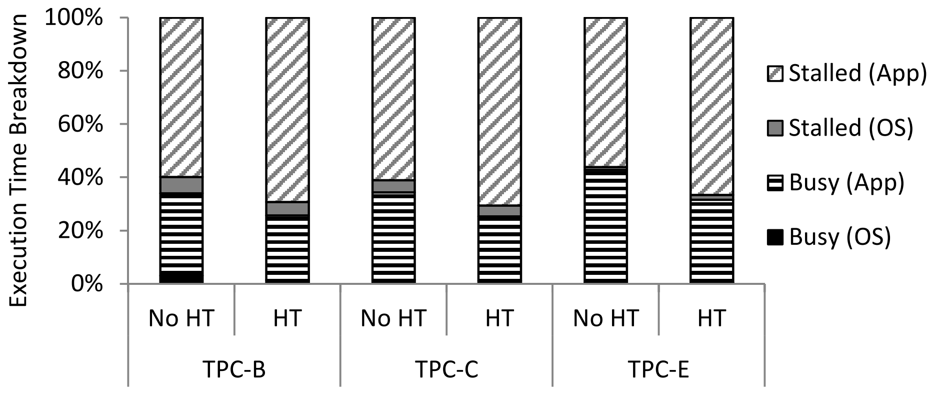


Figure 5.3: Execution time breakdown for three OLTP benchmarks on an OoO processor with and without hyper-threading.

5.3 shows the number of worker threads executing transactions in the system when the peak throughput is achieved for each workload on each server. Adding more worker threads to the system on top of the numbers reported in Table 5.3 causes degradation in throughput, either due to contention on shared records and storage manager objects or over-saturation of the machine being used.

5.5.1 OLTP on an Out-of-Order Processor

This section presents micro-architectural results from the Intel Xeon X5660 processors. We use VTune [89], which provides an API to ease the use of the hardware counters on this machine. We emphasize that the execution time breakdown on a superscalar out-of-order (OoO) processor cannot be precise due to overlapping of different execution components [51]. However, considering the low IPC of the workloads we are experimenting with (Figure 5.6 and Figure 5.8), we can assume that not much work is overlapped. Nevertheless, we draw the execution cycles that might be overlapped side-by-side rather than on top of each other.

Intel Xeon X5660 processors support hyper-threading, running two hardware contexts on one core at the same time. The goal of hyper-threading is to overlap the stall time of one thread with the execution of another. In the following subsections, for each experiment we present results when hyper-threading is disabled and when it is enabled.

Execution time breakdown

Figure 5.3 shows the breakdown of the execution cycles into busy and stall time for the three benchmarks. We count the cycles in which at least one instruction is retired as *busy* and where no instruction is retired as *stalled*.

In Figure 5.3, we see that more than half of the execution time is spent on stalls for all the OLTP benchmarks. While TPC-B and TPC-C show very similar behavior in terms of the percentage

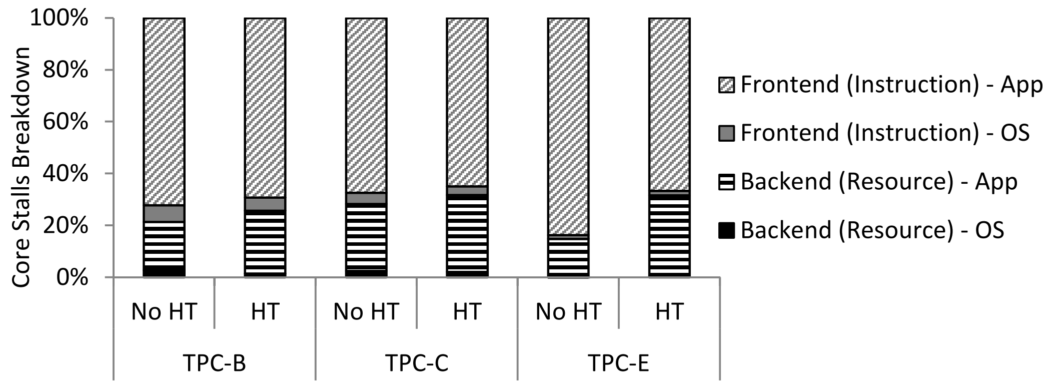


Figure 5.4: Core stalls breakdown for three OLTP benchmarks on an OoO processor with and without hyper-threading.

of *busy* and *stalled* cycles, TPC-E seems to observe fewer *stalled* cycles. This behavior results in a higher IPC value for TPC-E (see Section 5.5.1). As expected, when hyper-threading is enabled, the *stalled* cycles increase in the overall execution time since two threads instead of one share the private L1 and L2 caches, evicting each other's data and instructions from the cache, thus, causing more cache misses.

Figure 5.3 also breaks the execution time into time spent on the operating system operations (*OS*) and application itself (*App*). This separation demonstrates that for our configuration, the *OS* does not contribute much to the overall execution time.

Core stalls

As presented in the previous section, stalls dominate the total execution time of OLTP benchmarks. The estimated breakdown of these stalls into resource, which also includes data, and instruction stalls are given in Figure 5.4. We count resource stalls within a core, mainly stemming from the re-order buffer (ROB) being full, as *backend/resource* stalls while the remaining stalls as *frontend/instruction* stalls. We, again, separate OS and application stalls even though the OS does not contribute significantly to the total stall time.

As Figure 5.4 demonstrates, the main cause of core stalls is the frontend stalls for the OLTP benchmarks. In other words, a core spends most of its execution cycles waiting for instructions, since it cannot find them in its private L1 instruction cache. The percentage of the frontend stalls is higher for TPC-E compared to both TPC-B and TPC-C. We link this behavior to the lower data miss rate of TPC-E (see Figure 5.5), which increases the percentage of stalls for instructions.

In addition, hyper-threading increases the percentage of the backend stalls. Two threads sharing the resources of one core with hyper-threading can increase the hit rate of the instruction cache more than the data cache, because transactions tend to share more instructions than data (as Section 6.6 shows).

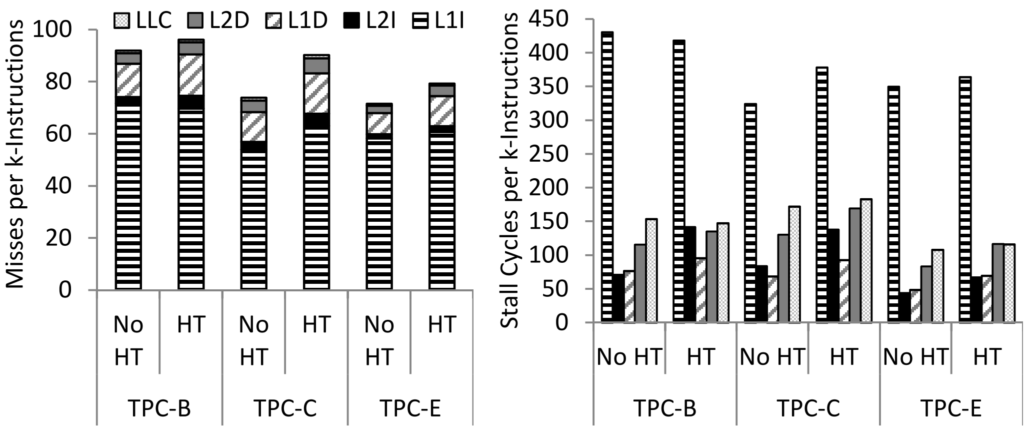


Figure 5.5: Number of misses per 1000 (k-) instructions for three OLTP benchmarks on an OoO processor with and without hyper-threading and the estimated number of cycles spent on these misses.

Data and instruction misses

Figure 5.5 shows the number of misses per k-instructions on the left-hand side and the estimated number of cycles spent on these misses on the right-hand side. As we mentioned before, we demonstrate the cycles spent on various cache misses side-by-side rather than on top of each other because of the unknown overlapping cycles for these misses. We categorize the cache misses into L1 instruction cache misses (*L1I*), L2 instruction misses (*L2I*), L1 data cache misses (*L1D*), L2 data misses (*L2D*), and L3 or last-level cache misses (*LLC*). For stall cycles due to cache misses, we use the expected penalty for that particular miss on the machine being used. For LLC misses, we average the penalty for going to local memory and remote memory.

What we observe is that L1 instruction cache misses dominate both the total number of misses and the total number of cycles spent on those misses for all the OLTP benchmarks. As mentioned previously, enabling hyper-threading increases the total number of misses in general due to more threads sharing the cache resources.

TPC-E exhibits ~35% fewer data misses and almost the same number of instruction misses, regardless of its longer running and more complicated transactions. Since it performs more scan operations, TPC-E can reuse the cache lines for data and instructions it needs more often (as also mentioned in Section 5.4.1).

Instruction- and memory-level parallelism

Finally, Figure 5.6 shows how many instructions per cycle (IPC) these OLTP benchmarks can execute per core on the left-hand side and how many long-latency misses (L2 miss) can be overlapped (MLP) on the right-hand side.

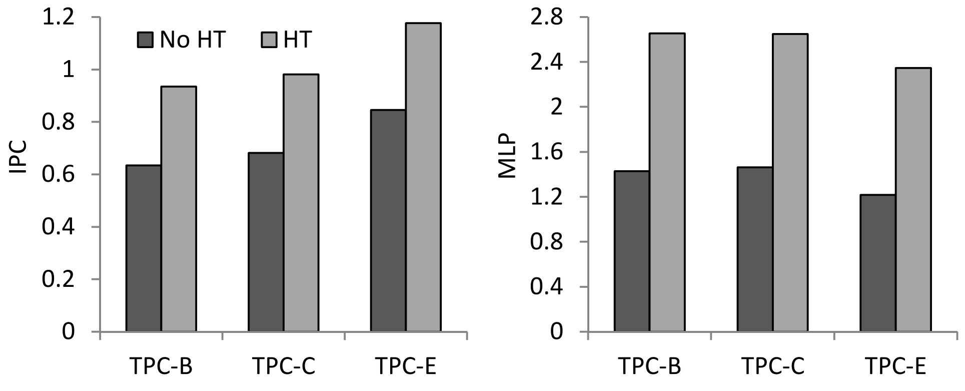


Figure 5.6: Instructions committed per cycle (IPC) and memory-level parallelism (MLP) on an OoO processor with and without hyper-threading.

An Intel Xeon X5660 processor has the ability to retire four instructions per cycle. However, by looking at Figure 5.6, we see that OLTP benchmarks can hardly retire even one instruction per cycle even though enabling hyper-threading provides some benefit. Overall, as the complexity of the benchmark increases, going from TPC-B to TPC-E, the IPC also increases. It is expected that TPC-E has a higher IPC value since it spends less of its execution time on stall cycles compared to the other two workloads (Figure 5.4). Higher IPC stems from TPC-E observing fewer L1 data misses (Figure 5.5) because of its frequent scan operations.

From the MLP values given in Figure 5.6, we also conclude that OLTP benchmarks do not exhibit high MLP. Even though there are 48-entry load-store queues in this processor, OLTP benchmarks do not have more than 2.7 outstanding long-latency misses even when hyper-threading is enabled. While TPC-B and TPC-C observe very similar MLP values, TPC-E exhibits less memory-level parallelism.

5.5.2 OLTP on an In-Order Processor

This section presents micro-architectural results from the Sun UltraSPARC T5220 server. We use the hardware counters on this machine through the `cputrack` command [38], which allows us to count various types of cache misses and number of instructions executed by each thread.

UltraSPARC T2 is an in-order processor that supports simultaneous multi-threading. A core provides support for eight hardware contexts and collocates two hardware contexts in the pipeline in one cycle. Therefore, each of these hardware contexts uses one cycle in every four cycles, aiming to overlap the stall time of other hardware contexts.

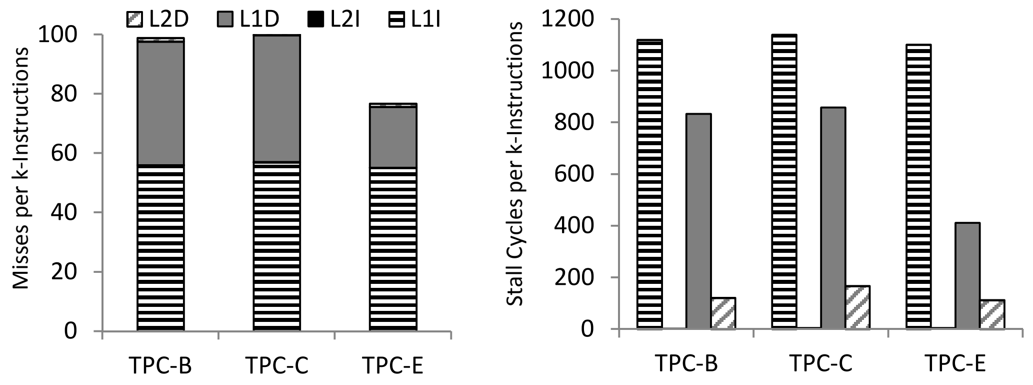


Figure 5.7: Number of misses per 1000 (k-) instructions for the three OLTP benchmarks on an in-order processor with simultaneous multi-threading and the estimated number of cycles spent on these misses.

Data and instruction misses

Figure 5.7 shows the number of misses per k-instructions on the left-hand side and the estimated number of cycles spent on these misses on the right-hand side as in Figure 5.5. On this processor, we also cannot infer the overlapped operations and, as in Figure 5.5, we draw the execution cycles that can be overlapped side-by-side rather than on top of each other. We report L1 instruction cache misses (*L1I*), L2 instruction misses (*L2I*), L1 data cache misses (*L1D*), and L2 data misses (*L2D*). For stall cycles due to misses, we use the expected penalty for that particular miss on this machine.

Similar to the Intel machine, the main source of misses and stall cycles are also L1 instruction cache misses as Figure 5.7 shows. On the other hand, the last-level cache (L2) maintains almost all the instructions for these workloads running on Shore-MT. Due to having smaller L1 data caches and more hardware contexts using the same private L1 cache in a core, L1 data cache misses contribute to a bigger portion of the total stall cycles compared to the Intel machine.

The comparison among the three benchmarks in terms of misses look similar to the comparison we have on the Intel machine (Figure 5.5). The instruction miss numbers are very close to each other for all the workloads and TPC-E has 50% fewer data misses compared to TPC-B and TPC-C.

Instruction-level parallelism

Figure 5.8 shows the IPC values for the three OLTP benchmarks running on UltraSPARC T2. Considering that this is an in-order machine, being able to execute instructions from two hardware contexts in a cycle, the IPC being higher than one shows a more effective use of the hardware resources compared to the Intel machine. While on the Intel machine OLTP

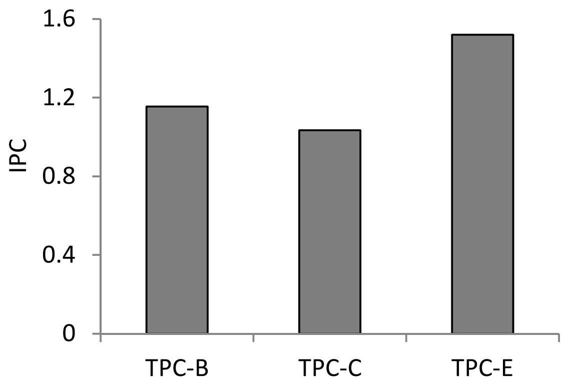


Figure 5.8: Instructions committed per cycle (IPC) on an in-order processor with simultaneous multi-threading.

benchmarks can barely leverage less than half of the instruction issue width, on SPARC they can utilize more than half of it.

5.6 Summary of Results and Conclusion

We present a thorough workload characterization study for TPC-E. We rely on profiling results to determine where the time goes within the storage manager while executing TPC-E on top. Furthermore, we use performance counters to investigate the micro-architectural behavior on two different camps of modern hardware: aggressive out-of-order and lean in-order. We compare TPC-E with previous OLTP benchmarks standardized by TPC, the well-studied TPC-C and the obsolete TPC-B, to better understand what TPC-E-like workloads need from the software and hardware.

As our micro-architectural analysis shows TPC-E has a higher IPC, observes lower miss rates, and spends less of its execution time on memory stalls compared to TPC-B and TPC-C. However, the fact that OLTP benchmarks commonly observe low IPC, spend most of their execution time on memory stalls, and mainly suffer from L1 instruction cache misses still remains. Going from an aggressive out-of-order processor to an in-order processor does not change the micro-architectural characteristics of the OLTP benchmarks much. Therefore, less aggressive processors (with fewer instruction issues) might be preferable for OLTP as previously suggested [54, 161]. On the other hand, we observe that simultaneous multi-threading (or hyper-threading) helps to overlap the stall time caused by cache misses to some extent.

To minimize L1 instruction misses several software and hardware mechanisms might be adopted. Software-side techniques that exploit intra-transaction parallelism [36, 145] divide the transactions into smaller actions and run independent actions in parallel on different nearby cores. Each action has smaller instruction footprint than the entire transaction and a higher chance of fitting its instructions in the L1-I cache. Techniques like STEPS [72, 74], on the other hand, also splits transactions into smaller actions and batches transactions on one core to ensure that the same actions from different transactions are executed one after the other to maximize L1 instruction locality. On the hardware side, as Part III details, computation spreading through thread migration [14, 30, 187] uses multiple cores to execute a transaction and makes newer transactions reuse the instructions brought to the L1-I cache by the older transactions. A more effective solution, however, would be to involve both software and hardware enhancements to minimize the stall cycles due to instructions.

By looking at the time TPC-E spends inside the lock manager, the natural choice would be to partition the database and deploy a shared-nothing design for it. Even though for TPC-B- and TPC-C-like database schemas this would work very well [107, 179], for TPC-E such a design would cause a lot of distributed transactions. There are two main reasons for this:

- Due to its complex schema, not all the TPC-E tables can be correlated with a single database column like the `Branch ID` in TPC-B or `Warehouse ID` in TPC-C.
- The TPC-E transactions access a lot of database records from various tables and perform frequent index scans using secondary indexes.

Therefore, it is not clear based on which columns we should partition TPC-E tables in a way to minimize distributed transactions when we deploy a shared-nothing design.

On the other hand, a shared-everything design based on logical or physiological partitioning like in DORA [145] or PLP (Chapter 3), respectively, might be more beneficial especially for TPC-E-like workloads. Such designs successfully minimize locking and latching overhead within the storage manager and do not suffer from distributed transactions like in a shared-nothing design. In addition, optimistic and multiversion concurrency control schemes [22, 116] may especially help TPC-E-like read-heavy workloads to improve concurrency by avoiding blocking at the time of a potential conflict and rather lazily performing checks at commit time.

To sum up our results, looking at the high-level description and statistics for each benchmark, we see that with each new OLTP benchmark standardized by TPC, we have a significant increase in complexity compared to the previous ones. Within the storage manager, TPC-E stresses the lock manager the most, like its predecessors, although it gets a higher penalty within the lock manager due to logical lock contention on hot database records whereas its predecessors suffer more from physical lock contention. However, regardless of these differences, micro-architecturally, all the OLTP benchmarks that exist today observe very similar behavior.

6 Transactions under the Microscope

As Chapter 5 highlights, OLTP workloads cannot exploit the full capability of modern processors. To better integrate OLTP and hardware in future systems, we first perform a thorough analysis of instruction and data misses, the main causes of memory stalls, using the standardized OLTP benchmarks. We demonstrate which operations and components of a typical storage manager cause the majority of different types of misses in each level of the memory hierarchy on a configuration that closely represents modern commodity hardware. We also observe the impact of data working set size on these misses.

According to our experimental results, L1 instruction misses are an extensive cause of the stall time for OLTP even for data working set sizes as large as 100GB as long as the data fits in memory. As the data working set size grows, the long-latency data misses also become a significant part of the overall stalls. Capacity and compulsory misses coming from the index probe operation are the dominant cause of instruction and data stalls, respectively. During index probe (one of the most common operations in OLTP), the B-tree, lock, and buffer management components of a storage manager are responsible for more than half of the total misses.

Following from the capacity related instruction misses, we also analyze what constitutes the majority of the instruction footprint for different transactions. We observe that, independently of their high-level functionality, transactions running in parallel on a multicore system execute actions chosen from a limited subset of predefined database operations. Performing a memory characterization study using the standardized OLTP benchmarks demonstrates that same-type transactions exhibit at most 6% overlap in their data footprints, whereas there is up to 98% overlap in instructions. [1]

6.1 Introduction

Despite recent advances in transaction processing and computer architecture, traditional online transaction processing (OLTP) exploits modern micro-architectural resources very

[1] This chapter uses material from [184, 187].

poorly (Chapter 5). Most of the execution time (~80%) goes to memory stalls; as a result, on processors that have the ability to execute four instructions in a cycle, which is the most common on modern commodity hardware, OLTP achieves around one instruction per cycle (IPC) [54, 177]. Such under-utilization of micro-architectural features is a great waste of hardware resources.

Several proposals have been made to reduce memory stalls through increasing cache hit rates. These range from cache-conscious data structures and algorithms [32, 58] to sophisticated data partitioning and thread scheduling [154] for data, and from compilation optimizations [136, 159] to advanced prefetching [53, 105] for instructions. Although these techniques reduce data or instruction misses to a great extent, some specifically targeting OLTP workloads and some being more general, none of them has detailed insights on the sources of instruction and data footprint and misses within the storage manager.

In this chapter, we thoroughly analyze the data and instruction misses of an OLTP system to answer the following questions: (1) What types of database operations (*scan*, *index probe*, etc.) and which parts of a storage manager (*locking*, *logging*, etc.) are responsible for various kinds of misses? (2) How sensitive are the results to the data working set size of the workloads? In addition, we characterize the memory behavior of typical OLTP workloads to quantify the shared portion of the instruction and data accesses across different transactions. Our aim is to give insights and hints to researchers and developers who would like to optimize their code and data accesses in order to minimize memory stalls while running OLTP.

Using Pin [124], we extract instruction, data, and function traces from the Shore-MT storage manager [172] while running the OLTP benchmarks standardized by the Transaction Processing Performance Council (TPC) [188]. We replay the traces on a cache configuration that is typical for modern commodity hardware and give miss rates, types, and breakdowns for the main storage manager components as well as the overlaps in instruction and data footprint among transactions at different granularities. Our contributions are listed below:

- We show that the *L1 instruction cache* misses account for a significant part (40-80%) of the overall stall time even when the memory-resident data working set size increases (from 0.1GB to 100GB). The *data misses from the last-level cache* is the next problematic component, especially for the large data set sizes.

- We demonstrate that the cache associativity of typical server hardware is sufficient to minimize the conflict misses for both data and instructions. The *capacity* misses are the single dominant factor in instruction stalls while data misses are mainly *compulsory*.

- We identify the *index probe* operation as the leading component of the cache misses. We also highlight the *B-tree*, *lock*, and *buffer* managers as the storage manager parts that contribute to most of the instruction (~55%) and data (~60%) misses during an index probe.

- Our characterization of the memory behavior of the TPC OLTP benchmarks reveals that same-type transactions exhibit 53% to 98% overlap in their instruction footprint while the data overlap is at most 6%.

The rest of the chapter is organized as follows: Section 6.2 surveys related work in more detail. Section 6.3 describes our experimental methodology. Section 6.4 presents a sensitivity analysis on the data size. Section 6.5 first classifies the most problematic misses into *conflict*, *capacity*, and *compulsory* ones, and then, associates various instruction and data misses into storage manager operations and components. Section 6.6 details typical database operations and presents the findings of our memory characterization study. Finally, Section 6.7 concludes the chapter by summarizing the results and discussing possible solutions to minimize stalls.

6.2 Related Work

There is a large body of related work that analyzes various OLTP workloads from low-level hardware-side analysis, e.g., workload characterization studies, to high-level software-side ones, e.g., time breakdowns.

Previous workload characterization studies [16, 106, 161] investigate OLTP workloads at the micro-architectural level. They all conclude that OLTP cannot exploit aggressive micro-architectural features, wasting most of its time in memory stalls and exhibiting low IPC. More recent workload characterization studies examine the behavior of OLTP workloads on modern commodity hardware ([54], Chapter 5). They show the same high-level conclusions with the older workload characterization studies demonstrating that, after almost 15 years, OLTP still exploits the micro-architectural resources of the most commonly used hardware types today very poorly. Even though these studies highlight the lower level problems of OLTP on modern hardware, all of them consider the data management system as a black-box. There is no clear attribution of the hardware-side problems to the software-side components of a typical OLTP system.

On the other hand, Wenisch et al. [200] attribute the temporal streams in data cache misses to the application components such as various kernel activities, SQL interpreter, storage manager, etc. Here we go one step further and focus only on the storage manager. We map both the data and instruction misses coming from the different levels of the cache hierarchy of a modern commodity server to storage manager components and database operations. In addition, complementary to this work, we investigate the sources of memory access overlaps, within transactions and database operations.

Johnson et al. [95, 97] and Pandis et al. [145, 147] provide time breakdowns for typical OLTP benchmarks showing where they spend the most of their execution time in the storage manager. Their primary goal is to identify components that are scalability bottlenecks on modern hardware and propose alternative design decisions to remove those bottlenecks. We

Table 6.1: Simulated memory hierarchy.

Processor	Intel Xeon E5-2660
# of Sockets	2
# of Cores per Socket	8 (OoO)
# of Hardware Contexts	32
Clock Speed	2.2GHz
Memory / access latency	125GB / 167 cycles (average of remote & local)
L3/LLC (shared) / access latency	20-way 20MB / 19 cycles
L2 (per core) / access latency	8-way 256KB / 8 cycles
L1 (per core) / access latency	8-way 32KB, split I/D / 4 cycles
Core width	4-wide retire and issue

provide similar breakdowns to spot the storage manager components that are responsible for the majority of data and instruction stalls.

Harizopoulos et al. [76] detail where the time goes within the storage manager during a single threaded execution in an OLTP system. They demonstrate that logging, latching, locking, and buffer pool altogether take 75% of the total execution time. VoltDB [199], the commercial version of the H-Store system [179], is designed based on these findings. H-Store specifically aims to increase performance by eliminating all four problematic components with an in-memory shared-nothing system design where each partition has only one worker thread. This chapter aims at providing similarly valuable insights that complement this previous work by mapping cache misses to storage manager components, thereby guiding future software and hardware system designs on how to minimize memory stalls.

6.3 Setup and Methodology

In this chapter, we perform a trace simulation study rather than working with hardware counters on real hardware (unlike what we did in Chapter 5). This allows us to change some of the hardware parameters (like in Section 6.5.1) and have the detailed function call information to map the various cache misses to software components.

Simulator and Traces

We build a custom trace simulator to replay the traces and calculate miss rates on various cache configurations. For this study, we model the memory hierarchy of an Intel Xeon E5-2660 server, see Table 6.1 for details [88].

The data, instruction, and function name traces are collected from Shore-MT using Pin [124], which can instrument x86 binaries. Pin is only able to instrument application level code; therefore, the Pin traces do not include the system-level instructions. To measure the effect of different storage manager components on cache misses, however, the application level trace contains all the necessary information. Moreover, the system time is very low in our setup (application time is 200X more than the system time) since we keep the contention low, run without network communication, and have the data working set size memory-resident throughout the experiments.

Workloads

The traces are collected for three transaction processing benchmarks standardized by TPC [188]—TPC-B [190], TPC-C [191], and TPC-E [193]—while running their workload mix on the Shore-MT storage manager [172]. Except where indicated in Section 6.4, we use 100GB databases. The buffer-pool is set big enough to keep the whole database in memory and the log is flushed to RAM due to not having a suitably fast I/O subsystem. Allowing I/O in our analysis would cause an unreasonable bottleneck considering our infrastructure, and therefore, lead us to unrealistic micro-architectural conclusions. To further make sure we run the most optimal configuration possible, all the logging (Aether [97]) and locking (SLI [95]) optimizations of Shore-MT are enabled.

We run a single worker thread while executing transactions to avoid any possible contention. High contention in our system would cause the worker threads to spin on locks, waiting to acquire them. This would artificially increase the instruction cache hit rate since the spinning code is a short loop (with a small instruction footprint), and give misleading micro-architectural results. Furthermore, cache coherence related data misses would increase under high contention due to extensive data sharing. In this study, we would like to focus on a system where the instruction and data accesses do not exhibit any anomaly due to high contention or data sharing.

Measurements

We collect two trace files for each workload, where each file contains traces of 1000 different transaction instantiations from the workload's transaction mix. One of the trace files from the same workload is run initially to account for cache warm-up. Then, the simulator starts collecting statistics for cache misses while running the other trace file. All the simulated caches use a LRU replacement policy and 64B cache lines.

To calculate the stall cycles due to cache misses, we multiply the number of misses by the expected penalty for that particular miss as given in Table 6.1. For LLC misses, we average the penalty for going to local and remote memory.

In stall time breakdowns, we do not account for the possible overlaps of different execution components that would normally happen on a superscalar out-of-order (OoO) processor [51], like the one this chapter models. Therefore, even though we draw the stall times on top of each other, some are actually hidden either by other stalls or useful execution. For instructions, the decoupled front-end and back-end of a core would be able to hide some of the stalls. For data,

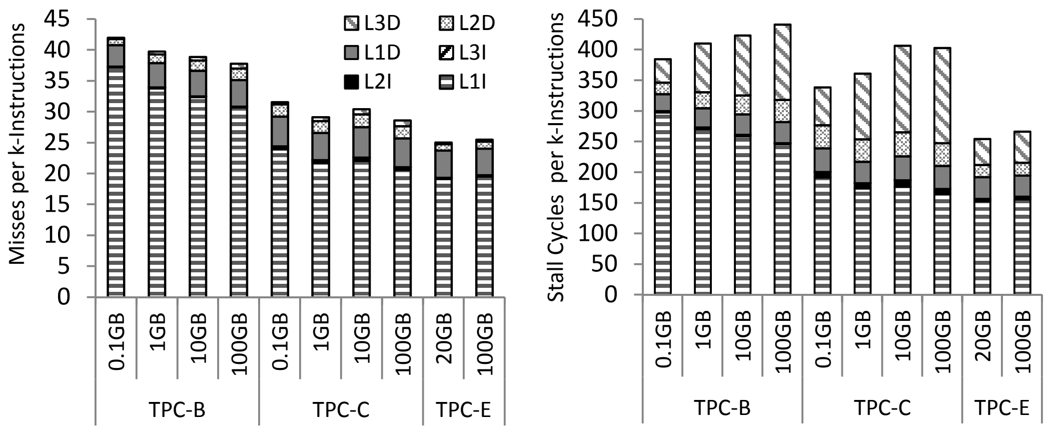


Figure 6.1: Effect of data size on MPKI (left-hand side) and stall time (right-hand side).

out-of-order execution can hide some of the data stalls while prefetching would reduce the effect of some of the data misses. Nevertheless, although such overlaps can reduce the stall time due to misses, the relative breakdown of the software-side components would be similar even with more complex models that would account for the overlaps. Besides, considering the low IPC of the OLTP workloads (Section 5.5), we can also assume that not much of the work is overlapped.

6.4 Sensitivity to Data Size

We initially investigate the effect of increasing data size on the instruction and data misses and stalls coming from different parts of the cache hierarchy. Figure 6.1 shows the misses per 1000 instructions (MPKI) on the left-hand side and the stall time they cause on the right-hand side for all the workloads. We pick scaling factors that populate around 0.1GB, 1GB, 10GB, and 100GB data for both TPC-B and TPC-C. Since a scaling factor of one already creates ~20GB data for TPC-E on Shore-MT, we run TPC-E with 20GB and 100GB data only.

Looking at the MPKI values in Figure 6.1, we see that L1 instruction misses dominate the total number of misses regardless of the data size. The domination of the instruction misses also affects the stall time breakdown as shown in Figure 6.1. Even with 100GB data size, on average 50% of the stalls are because of the L1 instruction misses. On the other hand, L1 and L2 caches, together, are sufficient to keep most of the instruction working set of the workloads we evaluate, keeping the rate of instruction misses from L2 and L3 caches low (at most 2% of the stalls).

Long-latency data misses from L3 caches are the next significant component in the total stall time, even though they form only ~2% of the total MPKI. As expected, L3 data misses increase as we increase the data size and for 100GB data, around 30% of the stalls are due to L3 data misses. On the other hand, L1 and L2 data misses are not as problematic and probably can be overlapped by out-of-order execution with other outstanding data misses or execution of another instruction.

Compared to the other workloads, TPC-E observes fewer data and instruction misses even though the general trends in different types of misses are very similar for all the workloads. This trend corroborates previous results in [54] and Chapter 5, and can be attributed to the increased number of scan operations from simpler workloads, like TPC-B, to more complex ones, like TPC-E (Section 2.4). During a file or an index scan, the routine eventually converges to only fetching the next tuple, which has lower instruction footprint than an index probe operation from B-tree root to leaves. Moreover, a file or an index page is scanned from start to end so almost all the parts of the cache lines brought from a database page are touched leading to lower data MPKI.

We also see a decrease in L1 instruction cache MPKI, especially for TPC-B, as we increase the data size. This might stem from the short loop statements in some of the sub-routines of various database operations that need to iterate more as there are more data. For example, the loop statement in the binary search sub-routine within the index probe operation performs higher number of iterations if there are more data on a particular index page. As a result, the same small instruction working set is executed more frequently at a given time, increasing the chances of finding the required instructions in L1-I and reducing the instruction MPKI.

6.5 Breakdown of Misses

After examining the total MPKI, this section presents why the most problematic misses happen and where they come from within the storage manager. More specifically, we give breakdowns of the instruction and data misses and stalls for each level of the cache hierarchy in three different granularities:

- *3C* miss categories (compulsory, capacity, and conflict),
- database operations (index probe, tuple update, etc.), and
- storage manager components (lock manager, log manager, etc.).

6.5.1 Into Miss Categories

For each workload, Figure 6.2 breaks the instruction and data MPKI of the most problematic misses, which are the L1 instruction and L3 data misses as shown in Section 6.4, into the *3C*-categorization of Hill et al. [82]:

- *Compulsory* misses are the ones that are missed even with an infinitely-sized cache,
- *Capacity* misses are the extra misses a fully-associative cache observes on top of the compulsory misses, and

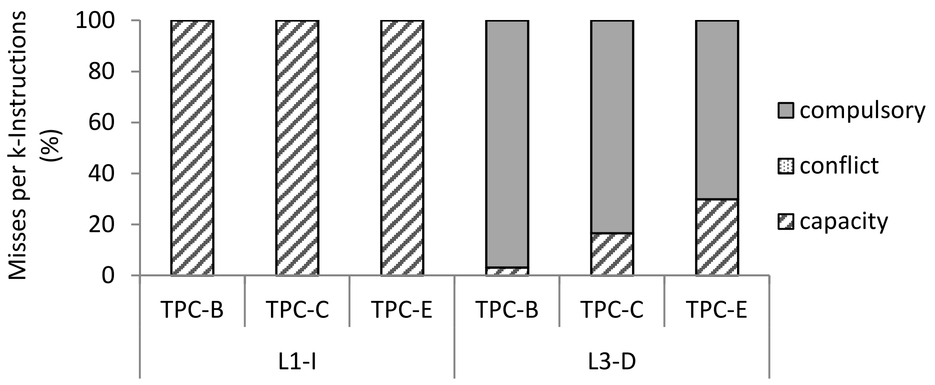


Figure 6.2: L1-I and L3-D misses breakdown into compulsory, capacity, and conflict misses.

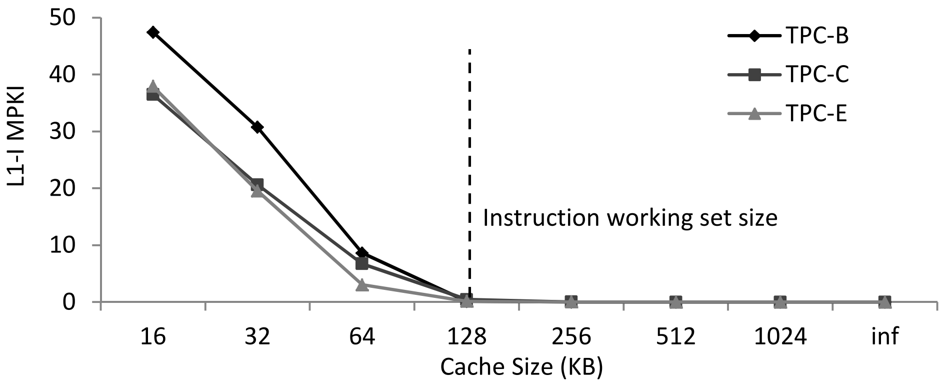


Figure 6.3: 8-way L1-I MPKI as cache size increases.

- *Conflict* misses are the ones that happen due to two addresses mapping to the same cache set and replacing one another due to low cache associativity.

As we can see from Figure 6.2, L1 and L3 cache associativity of the architecture we model, which are 8-way and 20-way, respectively, are sufficient to eliminate all the conflict misses. This leaves the capacity misses as the single cause of all the L1 instruction cache misses, whereas compulsory data misses dominate the total L3 data misses. After the warm-up run with 1000 transaction traces, the infinite instruction cache basically captures all the instructions needed for these workloads. On the other hand, the data working set size is a lot larger than what is accessed in 1000 transactions since the workloads, mostly randomly, access data from a working set of 100GB in our experiments. This explains why there are still many compulsory misses for data even after the warm-up run while we observe none for instructions. If we run longer traces, the percentage of the compulsory misses would be reduced while the capacity misses increase for data as well.

Finally, Figure 6.3 shows the instruction MPKI for an 8-way L1-I cache as the cache size increases. From Figure 6.3, one can naïvely think that enlarging the L1-I cache should solve the problem of capacity misses since the instruction footprint of the workloads we evaluate

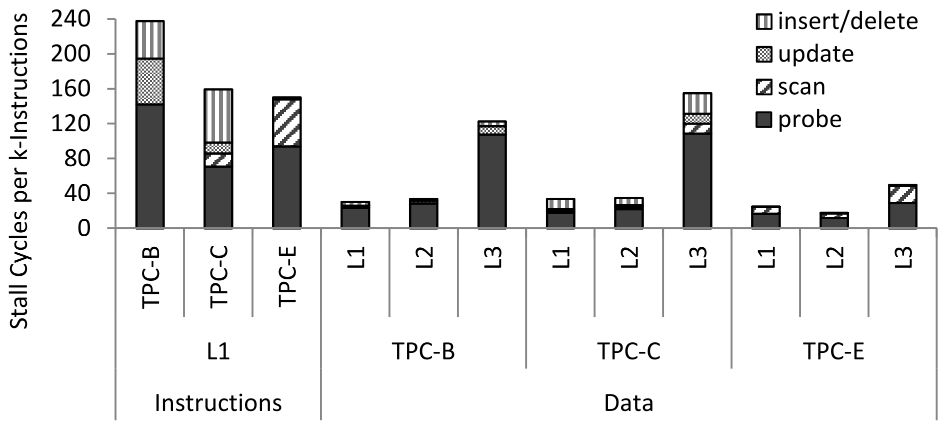


Figure 6.4: Misses breakdown into database operations at each level of the cache hierarchy.

seems to be around 128KB. However, increasing L1-I size also increases the time and energy spent while finding an item in the cache. This, in turn, would affect the clock frequency of a processor. Therefore, despite the growing sizes of L2 and L3 caches, today's typical high-performance processors limit their L1 cache sizes to about 32KB.

6.5.2 Into Operations

In Figure 6.4, we see the instruction and data stalls per 1000 instructions coming from the three levels of the cache hierarchy separated into different database operations. Since there are either none or very few instruction misses coming from L2 and L3 caches for all the workloads, Figure 6.4 has breakdowns only for L1 misses for the instructions.

Figure 6.4 shows that the majority of the misses happen during the index probe operations. This is expected since OLTP workloads do not access many records from a table in their transactions; hence, they highly depend on the index lookups and scans. The index lookups are especially problematic since the code-path is long and complex. It is interleaved with the function calls to many different modules encapsulating code and data from B-tree, lock, and buffer pool management as Section 6.5.3 also shows.

Even though we see several major contributors in instruction stalls, index probe seems to be the dominant operation in data stalls. Update, insert, and delete operations typically access a single tuple, hence a single heap page, whereas during an index probe several index pages are accessed. In the case of index scans, even though initially there is a probe to find the start point for the scan, afterward the same index and heap pages are reused frequently increasing the hit rates.

TPC-B is an update-heavy workload and has no index scans. Therefore, updates and inserts are the only operations causing the stalls for TPC-B after the index probes. Going from TPC-B to TPC-E, however, index scans form a bigger portion of the overall stall time as a result of

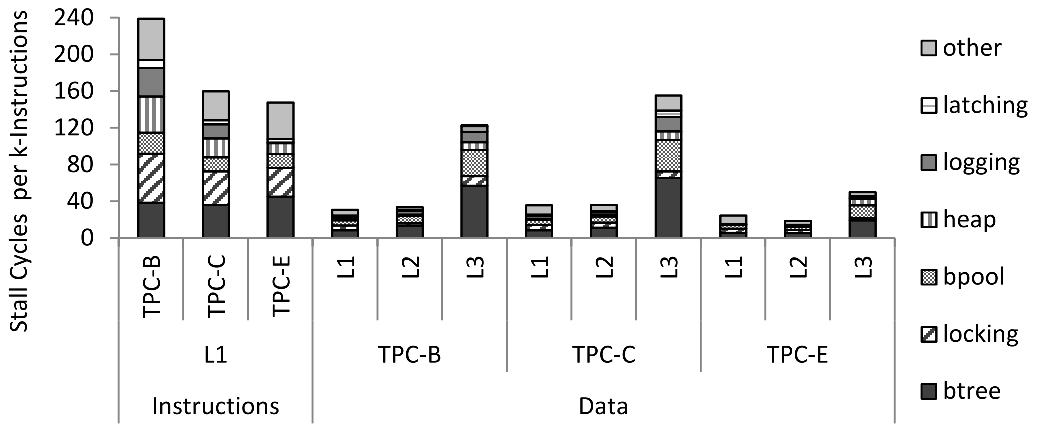


Figure 6.5: Misses breakdown into storage manager components at each level of the cache hierarchy.

increasing number of scan operations. For TPC-E, we do not see many misses due to read-write operations like tuple inserts, deletes, and updates since majority of the transactions (77%) in its transaction mix are read-only (Section 2.4). The trends in the breakdowns, on the other hand, do not change much for different cache levels within each benchmark.

6.5.3 Into Components

Figure 6.5 depicts stalls from different types of caches as does Figure 6.4, but it classifies them into storage manager components rather than database operations. Instruction stalls in L2 and L3 are again omitted since there are either none or very few of them.

Figure 6.5 does not identify a single dominant component as the cause of instruction stalls. B-tree index operations and lock manager together form ~45% of the instruction misses on average. Next component is the buffer pool and heap manager with ~23%. For TPC-B, the heap manager also takes a significant time due to the update and insert heavy nature of this workload. These results corroborate our findings in Section 6.5.2, where we show that the index probe operation is the main cause of the instruction and data stalls. The index probe traverses a B-tree from root to leaves and this process is heavily interleaved with the concurrency control mechanism of databases, which is based on ARIES/IM [129] by default in Shore-MT.

For the data stalls, we see the B-tree and buffer pool as the two significant factors, causing more than half of the data stall time for each of the caches. This result also matches our findings in Section 6.5.2 since the index probe operation requests many B-tree pages from the buffer pool during the traversal.

We also give a breakdown within the basic database operations to see which storage manager components affect the stall time during these operations. Figure 6.6 shows this breakdown

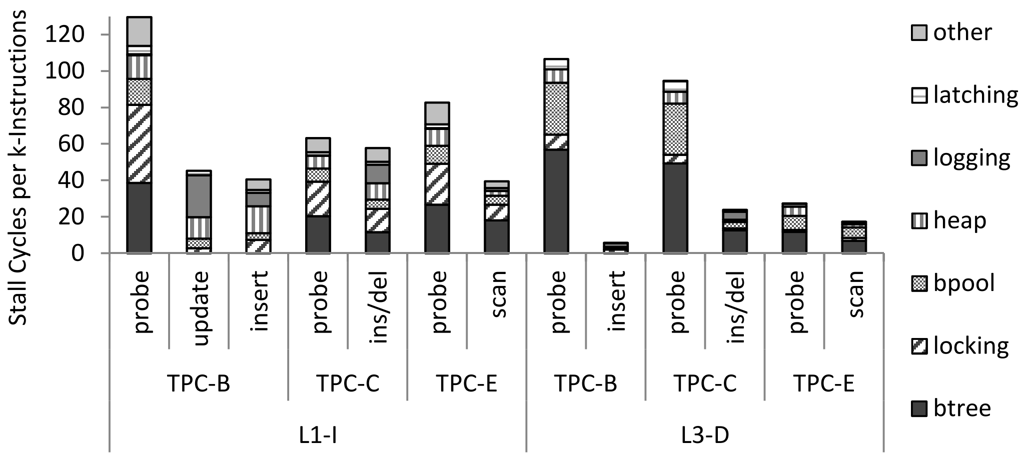


Figure 6.6: Misses breakdown into storage manager components for each database operation.

for the L1 instruction cache and L3 data stalls since Section 6.4 identifies them as the leading causes of the overall stalls. The operations that do not contribute much to the stall time are omitted for simplicity.

Inside the index probe and scan related stall times, we see B-tree, lock manager, and buffer pool as the dominant components for both L1 instruction and L3 data misses. As we have also mentioned above, this result is expected for the index probe operation considering its characteristics. For the update or insert/delete operations, however, logging becomes more significant as well as heap management since these operations modify the heap pages, and therefore, require log updates.

6.6 Inside Transactions

Seeing the problem with the L1 instruction cache misses (Section 6.4) and *capacity* misses from L1-I (Section 6.5.1), we also would like to understand where the majority of the instruction footprint for a transaction comes from and how it differs from transaction to transaction.

Each transaction satisfies a different request in terms of its high-level functionality. However, underneath, transactions execute a series of actions from the same predefined set of database operations, as also used in the breakdowns of Section 6.5.2. These operations dictate the interaction with the storage manager components. This section first details some of the common database operations, and then, investigates the instruction and data overlap across their different instantiations in a workload mix.

6.6.1 Database Operations

Most transactional workloads have five major operations: index probe, index scan, update tuple, insert tuple, delete tuple. In the rest of this section we discuss their main characteristics;

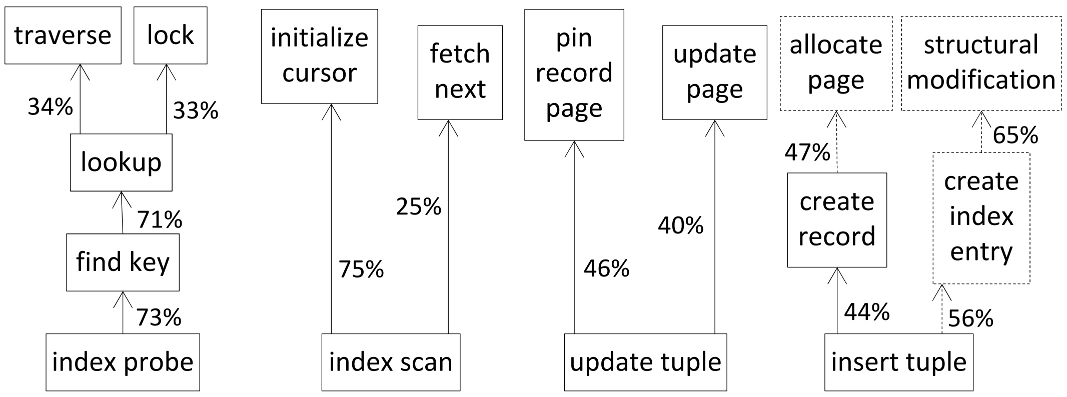


Figure 6.7: The flow graph of common database operations from the TPC-C transaction mix with the percentage of instruction footprints corresponding to each significant code part in these operations. An arrow from A to B with label X% means that X% of A's instruction footprint comes from executing B. The dashed lines indicate the code paths that are not always taken.

we omit delete tuple because of its similarity to insert tuple. To guide the discussion, Figure 6.7 sketches the high-level call flow for each operation including the percentage of the instruction footprint for each significant code path in it. In Figure 6.7, an arrow from box A to box B with label X% indicates that X% of the instruction footprint of A comes from executing routine B. For example, 34% of the instruction footprint of *lookup* comes from executing the *traverse* routine. Solid arrows represent calls that are always made whereas dashed arrows represent calls that are not always made, i.e., they depend on a branch condition. The footprint is measured as the unique 64byte cache blocks requested by each operation when running 1000 transactions from the transaction mix of TPC-C (we basically use one of the trace files mentioned in Section 6.3).

Index Probe

Index probe is the most common operation in transaction processing and is read-only. Its input parameters are an *index identifier* and a *key*. If the key exists in the index, index probe returns the tuple corresponding to the given key value in the index. Otherwise, index probe returns a flag indicating the key is not found. From Figure 6.7, we see that index probe follows a predictable call path. It starts with a call to the storage manager API, *find key*, which calls the *lookup* routine for the corresponding index. Then, it *traverse*s the index pages from top to bottom to find the desired key and interacts with the lock manager to acquire the *lock* for the record that maps the searched key.

Index Scan

Index scan is the other read-only operation used in transactions. It takes as input an *index identifier*, two *key* values for the boundaries of the scan, and two *flag*s indicating the inclusiveness of the boundary keys. It returns the set of tuples mapping to the key values within the given boundaries. As Figure 6.7 shows, index scan has two main parts. *Initialize cursor* first finds the position on the index leaf pages to be used as the starting point for the scan. This routine forms 75% of the instruction footprint of index scan. Then, *fetch next* fetches all the tuples until it reaches the scan's ending boundary. The instruction footprint of this last, tuple fetching code, part is three times smaller than the instruction footprint of *initialize cursor*; *fetch next* has just a short loop that reads the tuples in sequence.

Update Tuple

Update tuple takes as input a *tuple identifier* and the *updated tuple*. Then, it rewrites the part of the data page in the database that corresponds to the tuple identifier. It is a relatively short operation and follows a more predictable execution compared to the other database operations. It has two major routines as shown in Figure 6.7: *pin record page* pins the page that has the tuple to be updated in the buffer pool, and *update page* updates the record and inserts a log entry for the update.

Insert Tuple

Insert tuple takes a *table identifier* and a *tuple* as inputs. *Create record* adds the tuple to one of the data pages that belong to the given table and has sufficient space. *Create index entry* inserts the index entries for this record to all the indexes associated with the table. Figure 6.7 shows that these two routines almost equally contribute to the instruction footprint. Therefore, inserting a tuple to a table that has indexes results in a significantly different instruction stream than inserting a tuple to a table with no indexes. Similarly, if none of the data pages allocated for the given table has space, then a new data page is created (*allocate page*). This process requires almost half of the instructions in *create record*. Further deviation in the instruction stream might happen due to the instructions needed to handle structural modifications in an index (e.g., index page splits, merges, or new index root creation). Such modifications form 65% of all the instructions needed to create an index entry. Overall, the insert tuple operation exhibits the most variety in its instruction flow compared to the other database operations.

6.6.2 Commonalities across Transactions

Considering the database operations transactions share, we expect to see significant overlap in the code executed by different transactions as well as some common data accesses. To quantify this intuition, we analyze the memory behavior of the transactions from TPC-B, TPC-C, and TPC-E. We use one of the files with 1000 transaction traces from Section 6.3 for

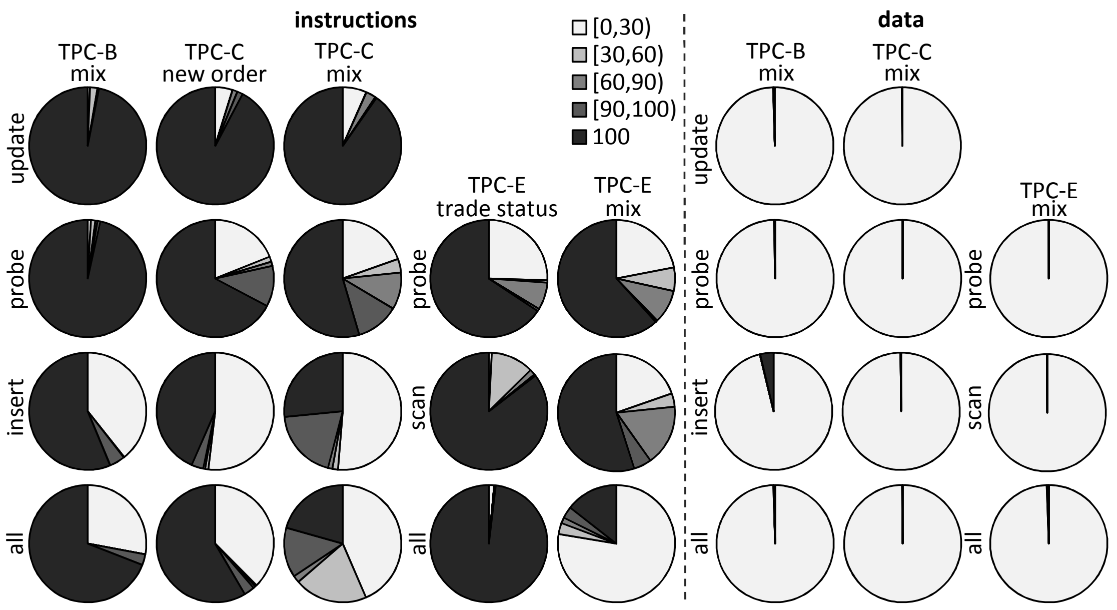


Figure 6.8: Overlaps in instruction and data footprints across different instances of the (1) transactions in a workload mix, (2) same-type transactions, and (3) database operations executed in a particular workload mix or transaction type. Each pie represents the instruction or data footprint for the indicated (1) workload mix, (2) transaction type, or (3) database operation. The legend represents the percentage of the transaction or database operation instances that use the corresponding slice of the overall instruction and data footprint. For example, the darkest slices (*100%*) are for the instructions and data that are executed in all instances, whereas the lightest slices (*[0-30)%*) represent the instructions and data that are common in less than 30% of the instances.

each benchmark. From these traces, we mark each instruction and data cache block accessed by each database operation or transaction. Then, we measure the fraction of transaction or database operation instances that access each cache block. Our goal is to examine instruction and data overlap at three granularities:

- within the whole transaction mix,
- within each transaction of the same type, and
- in each database operation.

Figure 6.8 depicts the highlights of the overall analysis. Each pie-chart represents the whole instruction or data footprint for the indicated workload, transaction, or database operation called within that workload or transaction. Next, we detail the results in Figure 6.8 for instruction and data overlaps.

Instruction Overlap

The left-hand side of Figure 6.8 reports the instruction overlap results. For simplicity, Figure 6.8 only shows the most frequent operations invoked by the most frequent transaction type in the mix in addition to the results from the overall transaction mix for all the workloads. The pie slices group instructions based on the fraction of the database operation or transaction instances accessing them. For example, the darkest slice (*100%*) represents the instructions that are executed in all instances, whereas the lightest slice (*[0-30)%*) represents the instructions that are common in less than 30% of the instances.

Since TPC-B has only one transaction in its mix, Figure 6.8 shows the results only for the workload mix (the leftmost pies). The instruction footprint overlap across all *probe* and *update* operation instances exceeds 90%. The overlap across all *insert* operation instances is at 60%. TPC-B has a single transaction type, `AccountUpdate`, which performs only one type of insert operation; i.e., it inserts a tuple to the `History` table, which has no index. Investigating where that 40% of uncommonly executed code comes from shows that these instructions come from the part of the *insert* operation code that creates a new data page. Even though there are only six `AccountUpdate` instances out of the 1000 that require this routine, the large instruction footprint of the routine (see Figure 6.7) causes a high deviation in the whole instruction stream.

The TPC-C charts show similar trends to TPC-B. Within individual transactions, e.g., `NewOrder`, the instruction overlaps in *probe* and *update* operations are high: at least 70% of the instructions accessed are the same. For the *insert* operation, however, around half of the instructions are not so common. `NewOrder` performs inserts to tables with indexes. This code has more branches compared to TPC-B's `AccountUpdate` since it also needs to execute the routine for creating an index entry (see Figure 6.7).

Since `NewOrder` forms almost half of the TPC-C transaction mix, the charts for each operation in the mix are similar to the charts for `NewOrder`. The slight differences are due to the different tables accessed by the transactions in the mix. For example, `Payment`, which together with `NewOrder` contributes to 88% of the mix, inserts a tuple to a table with no indexes. Therefore, the instructions for creating an index entry are not common in the overall mix. Furthermore, the degree of overlap is lower in the whole transaction mix (third column, fourth row in Figure 6.8) compared to the individual operations. This is expected since *probe* is the only database operation code shared by all TPC-C transactions.

Since almost 80% of the TPC-E mix is read-only, Figure 6.8 presents the results for *probe* and *scan* for TPC-E. TPC-E has 10 transaction types in its mix, twice the number of TPC-C, and the most frequent transaction, `TradeStatus`, accounts for only 19% of the mix. Therefore, the instruction overlap is less in the overall TPC-E mix (fifth column, fourth row in Figure 6.8) compared to the other two benchmarks. However, among same-type transactions instruction overlap is still significant; different `TradeStatus` instances observe a 98% instruction overlap.

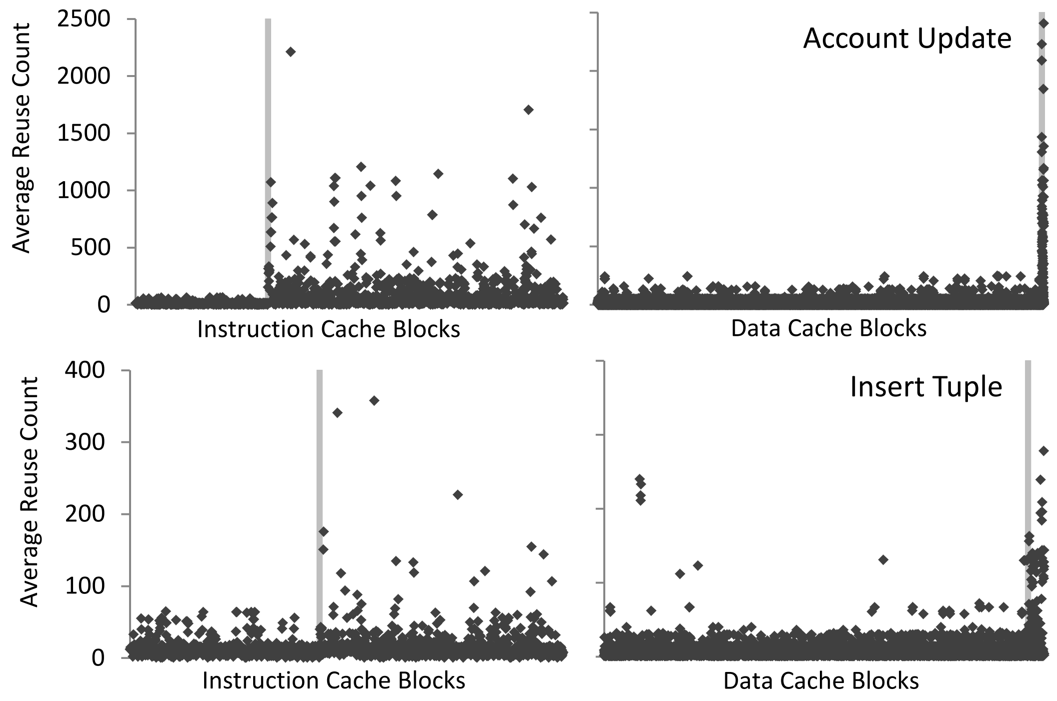


Figure 6.9: The average number of accesses to each memory address per instance of the TPC-B's `AccountUpdate` transaction and insert tuple operation. The x-axis places the addresses in the order of their commonality across different transaction instances in the workload. The addresses to the right of the vertical light-gray line are the ones that are used in all instances.

Data Overlap

After studying instruction overlap, we also examine the data commonality across different transactions and database operations. The right-hand side of Figure 6.8 presents data overlap results for the transaction mixes only since the conclusions are the same for individual transaction types. Figure 6.8 clearly depicts that the overlap in data is very low, at most 6%.

The dataset used while collecting the traces is around 100GB for each workload. Therefore, there is almost no overlap for the data that represent database records or lower-levels of the indexes. On the other hand, investigating the sources of the few, very frequently used data shows that metadata information, lock manager, buffer pool structures, and index root pages are commonly accessed (mostly read) across different transactions. Such data mainly stem from the tables that are accessed in all the transactions of a workload's mix, e.g., the `Warehouse` table in TPC-C, or used by all the instances of a particular database operation, e.g., the inserts to the `History` table in TPC-B.

6.6.3 Average Reuse in an Instance

Figure 6.8 demonstrates the instruction reuse frequency across different instances of transactions or database operations. However, it does not indicate how frequently a memory address is reused within each instance. Therefore, we also measure the average per instruction and per data address accesses within one instance of each transaction and database operation. Figure 6.9 shows the results for the `AccountUpdate` transaction and the *insert tuple* operation in TPC-B. The results for TPC-C and TPC-E and the other database operations share similar trends.

Figure 6.9 omits the address labels on the x-axis, but places the addresses based on their frequency across different transaction instances (from left to right the frequency increases). The addresses on the right of the light-gray vertical line appear in all the instances. Figure 6.9 highlights that the frequently reused addresses across transaction and operation instances are also frequently reused within each instance.

6.7 Conclusions

Recent studies emphasize that there is still a clear mismatch between what modern hardware offers and what OLTP systems need. Memory stalls dominate the overall execution time, and in turn OLTP performance deteriorates and the underlying hardware remains largely underutilized.

We conduct a detailed trace simulation of instruction and data misses in OLTP benchmarks modeling the memory hierarchy of one of the most commonly used hardware types. The experimental results link the most important memory-related stall types to software components in the storage manager, and quantify the effect of increasing the data size. More specifically, our results demonstrate that the capacity related L1 instruction misses are the main cause of the stalls even when working with large memory-resident data sets followed by the compulsory long-latency misses. The index probe operation, which is the most frequent routine for OLTP workloads, is the fundamental cause of both data and instruction misses coming from different levels of the cache hierarchy. The misses coming from the B-tree, lock, and buffer pool management are the essential factor in the misses observed during an index probe.

On the other hand, we also perform a memory characterization study for the same benchmarks. The goal of this study is to better understand the similarities or differences of the instruction and data footprint across transactions to be able to get further insights on improving cache locality for OLTP applications. This study demonstrates that:

- Transactions exhibit high instruction overlap because of the common database operations they execute, especially among same-type transactions. This offers an opportunity to achieve better L1-I cache locality by scheduling transactions in a way that would enable instruction reuse across transactions based on their common actions.

- The percentage of the data that is common across transactions is very low due to infrequent reuse of the database tuples. The few frequently used data are small-sized and mostly read-only. Accordingly it may be possible to *pin* them in the caches to improve data cache locality.

- The cache blocks that are highly common across different transaction instances tend to be more frequently reused within each instance. Therefore, any technique for improving cache locality for the common instructions and data across different instances also has potential to improve cache locality within each transaction instance.

To achieve a more graceful integration of hardware and software for OLTP systems, both of the layers should become more aware of each other. On the software side, reducing code complexity in the components mentioned above and designing more cache-friendly index structures are crucial. On the hardware side, as the next part (Part III) of this thesis also shows, dedicating several close-by cores for specific transaction operations can help reducing the capacity misses while exploiting instruction commonality across transactions as well as create opportunities for hardware specialization.

Chasing Instructions Part III

7 Boosting Instruction Cache Reuse in OLTP

The previous part highlighted that the performance of online transaction processing workloads suffers from instruction stalls; the instruction footprint of a typical transaction exceeds by far the capacity of an L1 cache, leading to ongoing cache thrashing. Several proposed techniques remove some instruction stalls in exchange for error-prone instrumentation of the code, or a sharp increase in the L1-I cache unit area and power. Others reduce instruction miss latency by better utilizing a shared L2 cache. This chapter presents two hardware mechanisms that change the way we traditionally schedule transactions to minimize L1 instruction misses. Both of the mechanisms exploit the observation that OLTP transactions exhibit significant intra- and inter-thread overlap in their instruction footprint (Section 6.6).

SLICC is a programmer-transparent and low-cost technique that migrates threads spreading their instruction footprint over several L1 caches. Under SLICC, a transaction's first iteration prefetches the instructions for the subsequent iterations or similar subsequent transactions. SLICC reduces instruction misses by 60% on average for TPC-C and TPC-E, thereby improving performance by 70%.

On the other hand, even though SLICC is promising for high core counts, it performs sub-optimally or hurts performance when running on few cores. Therefore, this chapter also presents STREX, another programmer-transparent hardware technique that exploits typical transaction behavior to improve instruction reuse in the L1 caches. STREX time-multiplexes the execution of similar transactions dynamically on a single core so that instructions fetched by one transaction are reused by all other transactions executing in the system as much as possible. Both SLICC and STREX dynamically slice the execution of each transaction. Experiments show that, when compared to baseline execution on 2 – 16 cores, STREX consistently improves performance (48% on average) while reducing the number of L1 instruction and data misses by 36% and 13% on average, respectively. [1]

[1] This chapter uses material from [13, 14, 15].

7.1 Introduction

As discussed in Chapter 5, existing server infrastructures are not tailored well to the needs of OLTP applications, with memory stalls accounting for 80% of the overall execution cycles, most of which are due to first-level instruction cache misses. Transactions of canonical OLTP systems are randomly assigned to worker threads, each of which usually runs on one core of a modern multicore system. The instruction footprint of a typical transaction does not fit into a single L1-I cache, thus, thrashing the cache and incurring high instruction miss rates. Although L2 and L3 caches are growing in size, today's technology and CPU clock cycle constraints prevent deploying L1-I caches larger than 32KB.

Several works propose to alleviate instruction stalls using hardware [30, 53, 91, 158] or software [74, 159] techniques. Existing techniques effectively reduce the number of L1-I misses or the associated penalty when running OLTP workloads, but in return either require error-prone instrumentation to the software code base or employ hardware prefetching tables that more than double the area devoted to the L1-I cache units. Others reduce instruction miss latency by better utilizing the aggregate L2 cache capacity.

As Section 6.5.1 demonstrates, the instruction footprint of a typical OLTP transaction fits comfortably in the aggregate L1-I cache capacity of modern multicore or many-core chips. Provided that there is sufficient code reuse (Section 6.6.2), spreading the footprint of transactions over multiple L1-I caches has potential to reduce instruction cache misses. Therefore, this chapter initially proposes SLICC (Self-Assembly of Instruction Cache Collectives), a hardware technique that utilizes thread migration to minimize instruction misses for OLTP workloads. SLICC divides the instruction footprint of a transaction into smaller code segments and spreads them over multiple cores, so that each L1-I cache holds part of the instruction footprint. As part of this process the L1-I caches self-assemble to form a *collective* that reduces the instruction misses for this transaction and other similar ones. SLICC exploits intra- and inter-thread instruction locality in two orthogonal ways:

- A thread looping over multiple code segments spread over multiple caches observes a lower miss rate (as opposed to a conventional system in which each segment would evict the others from the cache), thereby avoiding thrashing.
- A preamble thread effectively prefetches and distributes common code segments for subsequent threads, thereby reducing the total miss rate.

As execution progresses, old cache collectives are naturally disassembled and new ones are formed to hold the footprints of new transactions.

On the other hand, as this chapter also shows, SLICC is not as effective and may *hurt* performance when the footprint of all concurrently running transactions exceeds the available aggregate L1-I capacity. While the number of cores on-chip in main-stream servers is expected to grow in the future, the number of cores per application may not always be sufficient. Data center design and deployment and application trends influence the available per application core count.

- Current data center design trends are toward consolidating more virtual machines on servers as this increases utilization, improves security and energy efficiency, and reduces costs and management overhead [26, 85, 198].
- A data center may run multiple OLTP workloads, each with different transaction types.
- While L1-I capacities remain cycle-time limited, OLTP transaction instruction footprints are increasing. Transactions are becoming more complex and thus larger as a result of additional functionality such as data analytics (e.g., WebSphere [84], WebLogic [141]), or more complex logic.

Accordingly, it is also desirable to avoid SLICC's performance cliff and to develop an instruction stall reduction technique that is effective irrespective of the number of available cores.

The temporal code overlap among similar OLTP transactions (Section 6.6.2) also motivates STREX, a transaction scheduling mechanism that exploits inter-transaction locality by grouping and synchronizing the execution of similar transactions into time slices. During each time slice, a *lead* transaction brings into L1-I an instruction code segment that all other transactions ought to reuse. Ideally, when the transactions within a group overlap perfectly, only the lead transaction incurs all necessary misses, the misses that a transaction would incur on a conventional system anyhow. As a result of STREX's time slicing, the remaining transactions find the instructions they need in L1-I.

This chapter has the following contributions and organization:

- Based on the observation that concurrent transactions on a multicore server exhibit significant code overlap (Section 6.6), Section 7.3 presents the design of SLICC, a transaction scheduling mechanism that spreads the execution of a transaction over multiple cores to both exploit aggregate L1-I capacity and enable instruction reuse.
- In order to avoid the drawbacks of SLICC under insufficient core counts, Section 7.4 proposes STREX, a transaction scheduling mechanism that batches transactions on a single core to execute their common code segments one after the other, also exploiting instruction overlap across transactions.
- Section 7.5 prototypes SLICC and STREX on an x86 multicore simulator and evaluates them using the TPC-C [191] and TPC-E [193] benchmarks. The evaluation demonstrates that STREX reduces L1-I miss rates (~36%) and improves performance (35-55%) regardless of the core counts. On the other hand, while SLICC hurts performance under low core counts, it outperforms STREX (by ~16% on average) when there are enough cores to spread the instruction footprint of the workloads.

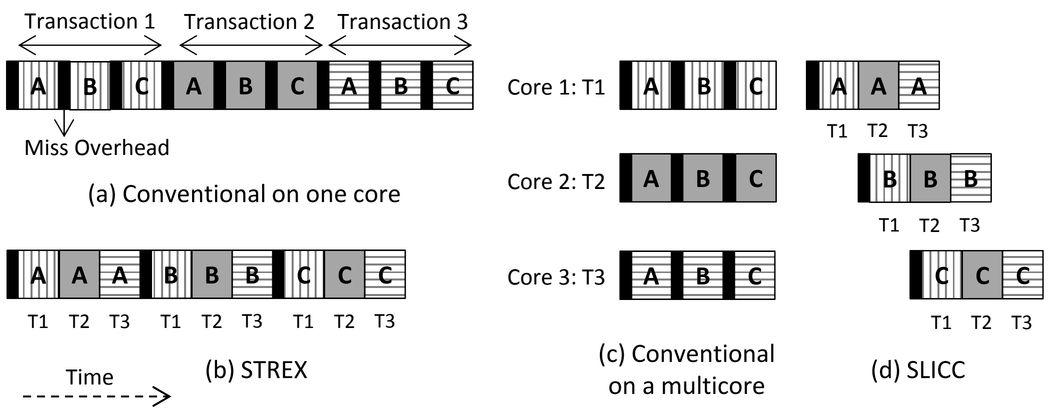


Figure 7.1: Ways of scheduling transactions.

Finally, Section 7.2 discusses how transaction behavior can be potentially exploited to improve instruction reuse in caches while giving a high-level overview for SLICC and STREX. Section 7.6 reviews related work and Section 7.7 presents our conclusions.

7.2 Exploiting Instruction Overlap

Transactions are composed of *actions* that in turn may execute several basic functions. Basic function examples include probing and scanning an index, inserting a tuple to a table, updating a tuple, etc. No matter how different the output or high-level functionality of one transaction is from another, all database transactions are composed of a subset of such basic functions, repeated several times for different inputs (as detailed in Section 6.6).

This section describes three ways of scheduling similar transactions: the conventional way and the two techniques that this chapter proposes to exploit instruction commonality across transactions.

Conventional

Figure 7.1(a) & (c) show how three transactions executing exactly the same code parts would execute under a conventional OLTP system on one core and on multiple cores, respectively. The example transactions execute the code segments A, B, and C in order. Each segment fits in L1-I, but any two segments exceed its capacity. When these transactions execute in a conventional system, they take turns thrashing the cache since each executes segments A through C in order independent of the other transactions. Thus, each segment incurs an overhead due to instruction cache misses.

SLICC

As long as there are enough cores so that the aggregate L1-I capacity can hold all code segments, a transaction can migrate to the core whose L1-I cache holds the code segment the transaction is about to execute. For example, as Figure 7.1(d) shows, the *lead* transaction can execute segment A first on core 1, then migrate to core 2 where it would execute segment B, then migrate to core 3 where it would execute segment C. Transactions 2 and 3 can follow in a pipelined fashion, finding segments A, B, and C, in cores 1, 2, and 3, respectively. While transaction 1 incurs an overhead when fetching the segments for the first time, the other transactions do not.

STREX

Figure 7.1(b) shows another way of improving L1-I utilization where the first, *lead*, transaction executes segment A incurring an overhead as previously. However, instead of proceeding to execute segment B, transaction 1 context switches allowing, in turn, transactions 2 and 3 to execute instead. Transactions 2 and 3 find segment A in L1-I and thus incur no overhead due to misses. Once all three transactions execute the first segment, execution proceeds to segment B and so on.

7.3 Self-Assembly of Instruction Cache Collectives

SLICC exploits intra- and inter-thread locality.

- It virtually increases the L1-I cache capacity observed by a thread; thus, it improves locality within a thread.
- It pipelines similar threads, such that one thread fetches instruction cache blocks that are reused by many threads.

7.3.1 SLICC Design

SLICC is a dynamic hardware thread scheduling and migration algorithm that is programmer transparent. SLICC attempts to partition on-the-fly the instruction footprint of transactions into several segments where each segment fits in the L1-I cache, but two segments do not fit together. Ideally:

- a thread will migrate to another core when it starts touching a different segment and
- the destination core will already have the segment cached.

In the steady state, each SLICC core has a running thread and a hardware queue of waiting threads. Using a naïve load-balancing strategy, newly arrived threads are scheduled to the

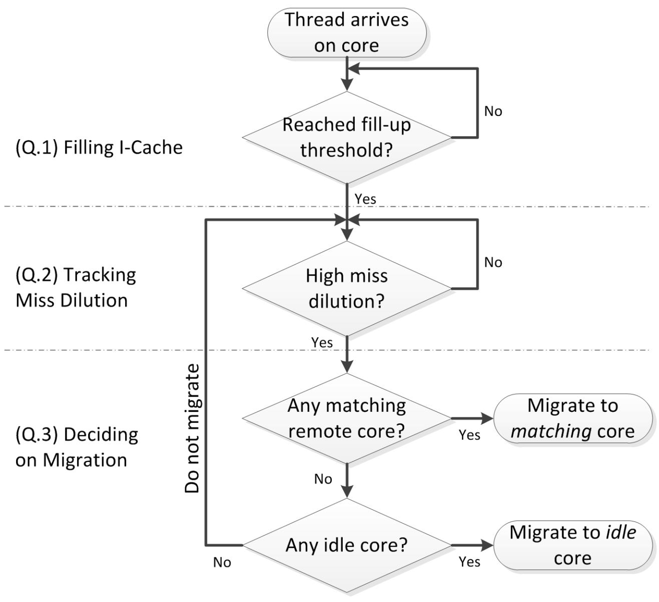

Thread arrives on core
Reached fill-up threshold?
No
Yes
(Q.1) Filling I-Cache
(Q.2) Tracking Miss Dilution
High miss dilution?
No
Yes
(Q.3) Deciding on Migration
Do not migrate
Any matching remote core?
Yes
Migrate to *matching* core
No
Any idle core?
No
Yes
Migrate to *idle* core

Figure 7.2: SLICC's thread migration algorithm.

least congested core (i.e., the core with the least number of waiting threads). A SLICC agent at each core continuously monitors execution locally in order to determine

- whether the local cache is filled-up with useful instruction blocks,
- if so, whether these blocks are useful to the current thread and for how long,
- and where to migrate if needed.

Figure 7.2 summarizes the execution stages of a thread on a core until it migrates or completes execution. The following subsections details each of these stages.

(Q1) Is the cache full with useful blocks?

As a thread starts executing on a core it may experience many misses. If the cache contains a segment that may be useful for other threads, it is best to migrate the current thread to another core. Otherwise, it is best to allow the current thread to load a new segment in the cache. SLICC uses a *cache full* detection heuristic to make this decision. Initially, all caches are *empty*. To detect whether a cache has been filled up with a *segment*, SLICC counts the number of misses using a resettable, saturating miss counter (*MC*) local to each core. When the number of misses exceeds the threshold, *fill-up_t*, the cache is considered *full*. In the long run, all MCs will saturate, preventing new segments from being cached effectively due to premature thread migration. To create opportunities for loading new segments, SLICC resets the MC when the core's thread queue becomes empty. The currently cached blocks are not

Table 7.1: SLICC thresholds.

fill-up_t	the number of misses that indicates the cache is full
dilution_t	the miss-hit ratio that triggers migration
matched_t	the number of most recently missed cache blocks that must be found in remote caches before migration

flushed, so if a subsequent thread requires the same segment it will still find it there. However, a thread touching a new segment will be given the opportunity to cache it.

(Q2) Are the current cache contents useful to this thread and for how long?

When running a thread on a *full* cache, SLICC tries to determine whether the thread is going over the cached segment, or whether it is about to move to a new segment. For this purpose SLICC measures *miss dilution*, that is, the recent frequency of misses (detailed in Section 7.3.2). If miss dilution is low, then SLICC predicts that thread is only temporarily diverting away from the cached segment. Since the thread will converge again soon, it is best not to migrate to benefit from the forthcoming instruction reuse. If miss dilution is high, then SLICC predicts that the thread is moving to a different segment. If it continues execution on this core it will evict useful cache blocks, which could be reused by other threads. SLICC predicts that it might be better to migrate the thread elsewhere.

(Q3) Where to migrate to?

Ideally, SLICC would migrate a thread to a cache that has the thread's next segment. SLICC attempts the following in order:

- If the thread is going to touch a code segment that is available on another core, the thread migrates there.
- Otherwise, the thread migrates to an idle core, if any.
- Otherwise, the thread stays put.

In the last case, migrating the thread would incur overhead and would evict remotely cached segments that may be useful for other threads. SLICC opts for incurring the instruction misses locally, avoiding the migration overhead.

To detect which, if any, remote cache has the next segment, SLICC uses a short sequence of most recent misses, predicting that they form the preamble of the next segment. Conceptually, once SLICC decides to try to migrate a thread, it searches all remote L1-I caches for these recently missed tags. Section 7.3.2 explains how this search can be implemented including an incremental method that uses the existing coherence protocol responses.

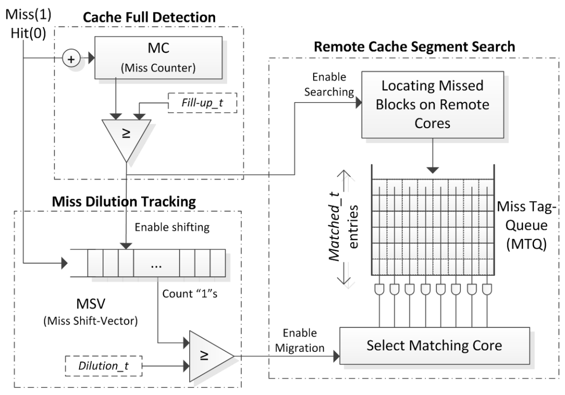


Figure 7.3: SLICC architecture.

7.3.2 Implementation Requirements

Table 7.1 summarizes SLICC's thresholds and Figure 7.3 shows that SLICC's implementation comprises:

- a cache full detector,
- a miss dilution tracker, and
- a remote cache segment search unit.

SLICC uses hardware thread migration, and thus, interacts with the OS as Section 7.3.4 explains in more detail. The three aforementioned units, described subsequently, track all cache accesses, including speculative ones.

Cache full detection

A $log_2(L1I\ cache\ blocks)$ wide saturating miss counter (MC) continuously counts the number of misses. When MC saturates at a value of *fill-up_t*, SLICC assumes that the cache has now captured a full segment and may trigger migrations accordingly. We experimentally found that using a value on the order of $\frac{cache\ size}{2}$ for the *fill-up_t* threshold works reasonably well, with little sensitivity to the exact value of this parameter.

Miss dilution tracking

It is not always beneficial to migrate threads immediately after a cache becomes full or when a thread incurs a few misses. SLICC must predict whether the thread is only temporarily diverging due to conditional control flow or whether it is moving to a completely different segment. Furthermore, since threads have to miss for a few blocks before migrating (*matched_t* tags must be located on a remote cache), a few useful cache blocks may be evicted, creating gaps in the exiting segment and causing a corresponding number of misses for subsequent threads. Finally, a thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path after being selected for migration.

SLICC handles these cases by considering the frequency of instruction misses; it restricts migration to the cases when a thread starts to miss more often. If the thread is moving to a new segment, it will incur more misses than hits. SLICC counts the number of misses in a window of recent accesses. When this count is above the dilution threshold, *dilution_t*, migration is enabled. The miss shift-vector (*MSV*) is a 100-bit FIFO shift vector recording the hit/miss history for the last 100 cache accesses (enabled when cache is filled-up). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reaches a threshold (*dilution_t*), SLICC enables migration. SLICC resets the *MSV* with every migration.

Searching remote cache segments

When SLICC decides to migrate a thread, it has to determine which cache, if any, contains the segment the thread is executing. To do so, SLICC records recently missed tags in the Missed Tag Queue (*MTQ*), which is a *matched_t* entry FIFO of n-bit entries, where n is the number of cores. A logic-1 on bit index C for MTQ entry i indicates that the ith recently missed cache block was cached at core C. Thus, by ANDing all bits at index C we know whether core C holds all the recently missed cache blocks. This information does not have to be exact or accurate, since it is used by a prediction mechanism. SLICC gathers this information *incrementally* as misses occur and stores it in the MTQ. The remote cache segment search is distributed and the decision is made locally by the core we migrate from. A directory coherence protocol could report the complete or partial sharing vector for misses that are tracked by the MTQ.

Alternatively, or if the coherence protocol is snoop-based, SLICC could broadcast the missed tags as they occur and explicitly request that remote cores identify themselves. On snoop coherence systems, these requests can piggyback on the existing snoop requests. Searching remote L1-I caches requires extra bandwidth on the remote caches that is proportional to the number of missed tags and cores.

To avoid this bandwidth overhead, we use an approximate cache signature in the form of a partial-address bloom filter that supports evictions [151]. As Figure 7.4 illustrates, the partial-address bloom filter uses a part of the whole cache line addresses as the key values. More specifically, a key value is the least significant bits of the address starting from the index bits. The number of bits to be used in the key depends on the size of the partial-address bloom filter. Every core maintains such a filter, representing a superset of the currently cached blocks. Once migration is triggered in a core, that core checks whether the last *matched_t* missed cache lines is in the other caches through checking the partial-address bloom filters of those caches. This way, SLICC avoids interfering with the actual cache requests of the remote caches.

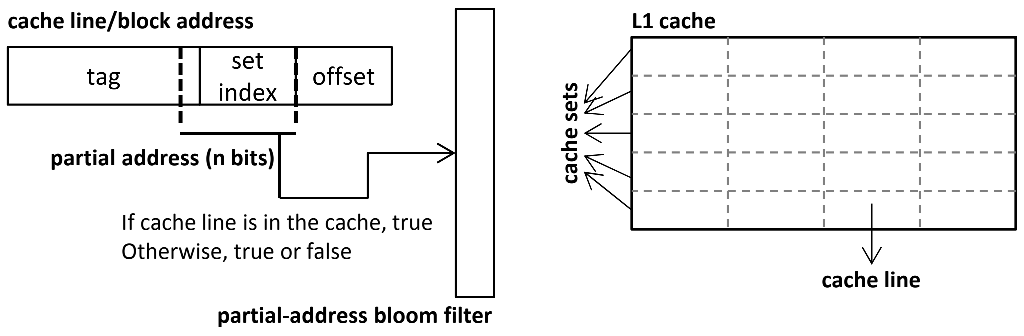


Figure 7.4: Partial-address bloom filter.

In Section 7.5.2, we evaluate the trade-off between the bloom filter's accuracy versus its size. We find that for a 32KB cache, a 256B bloom filter is sufficient.

If no matching remote cache is found, SLICC attempts to find an idle core. SLICC either broadcasts a request for idle cores to report, or piggy-backs this information on the responses received during the miss tag search phase. Thread migrations are relatively infrequent (every 3.2K instructions on average), reducing the relative overhead of remote cache segment and idle core searching.

7.3.3 Exploiting Transaction Type Information

As Section 6.6.2 shows, the instruction footprint overlap is higher among threads of the same transaction type. Therefore, SLICC forms teams of similar/same-type transactions on-the-fly. In order to perform the detection of such transactions in a programmer-transparent manner, SLICC uses a hardware preprocessing phase to assign types to threads as they launch. SLICC exploits the observation that in OLTP the first few instructions executed are the same for same-type transactions, while they differ across different-type transactions. SLICC only needs to know when a new transaction is launched. A middle-ware layer assigns transactions in groups to a core devoted for this purpose (*scout core*) initially. There, each thread executes a few tens of instructions, while the instruction addresses are hashed. The resulting values are used as thread type identifiers.

For each transaction instance SLICC records a unique numerical ID, a type ID, and an arrival timestamp. The timestamp of a team is that of its oldest transaction. The oldest team is scheduled, without preemption if possible. We intuitively design a scheduling algorithm that maximizes the core utilization and reduces the queuing delay of threads. Team sizes differ and for an N-core architecture we categorize them into

- *large* (1.5× to 2× N threads),
- *medium* (0.5× to 1.5× N threads), and
- *small* (less than 0.5 × N threads) teams.

Cores are time-multiplexed among teams. When large teams are scheduled, they are allowed to execute on all cores. Medium size teams are limited to half the resources ($0.5 \times N$ cores). Threads of a small team are treated as stray threads, and are not grouped. Rather, stray threads are scheduled, individually, to idle cores, or in parallel with a medium team. When a team of threads completes execution, SLICC resets all *MC*s, *MTQ*s, and *MSV*s.

7.3.4 Support for Thread Migration

To allow for queuing threads, the thread migration performed in SLICC transfers architectural register files as in Thread Motion [160]. The thread's context is saved in the L2 cache closest to the target core and is then retrieved at the target core. This minimizes the set-up time for the thread. Since modern commercial processor technologies (e.g., Intel Virtualization (VT) [197] and AMD Secure Virtual Machine (SVM) [10]) provide hardware support for thread migration, minimal modifications are required to make the migration process transparent to higher software layers.

Canonical OS kernels are responsible for assigning threads to cores. Hardware support for thread migration that is transparent to higher layers avoids any software overhead. Otherwise, the OS scheduler must be informed about these migrations. An alternative is a hybrid system in which hardware mechanisms provide counters and migration acceleration, while leaving the policy choice to software. This enables easier integration between existing schedulers and platforms with virtualization support.

7.4 Stratified Transaction Execution

STREX dynamically detects the points at which a transaction ought to context-switch in order to keep inter-transaction execution synchronized, thereby maximizing instruction cache reuse. If a transaction executes for a long time, it will end up evicting cache blocks that other transactions could have reused. If a transaction executes for a short time, the overhead of context switching and of a potential increase in contention in the data caches will overwhelm performance. For these reasons, context switching at regular intervals would perform sub-optimally at best. An optimal *synchronization algorithm* must rely on dynamic information: a transaction should be allowed to execute as long as it benefits from data and instruction locality, however, it should not be allowed to evict any blocks that would be useful for other transactions. Moreover, the costs of context switching over the benefits of the increase in instruction reuse must be amortized.

Breaking down the instruction footprint of several random transactions into smaller chunks will not generally result in identical code segments. Optimally scheduling those chunks in order to maximize instruction locality is akin to job scheduling, an NP-complete problem [57]. However, an inexpensive algorithm that performs well exists for the simpler case of a team of same-type transactions.

7.4.1 STREX Synchronization Algorithm

An *optimal* algorithm for STREX to schedule transactions could have been possible if we had had a team of identical transaction instances. However, different instances of the same-type transactions have identical instruction streams very rarely due to data dependencies (Section 6.6.2). This section presents the synchronization algorithm STREX uses to improve instruction cache reuse for the general case of multiple, non-identical transactions. Since even same-type transactions diverge at runtime, the first transaction in a team, the *lead*, may not touch all blocks that the other, subsequent threads need. Hence, *non-lead* transactions should also be allowed to fetch new cache blocks.

Based on the above intuitions the STREX synchronization algorithm is as follows:

- Given a pool of transactions, STREX groups transactions of the same type into *teams*. STREX places each team into a hardware thread queue in an available execution core. Then, it flags the first transaction in the queue as the *lead*.
- STREX synchronizes transaction execution using a per-core $phase_{ID}$ counter. As a transaction touches an instruction block, it tags the block with the current $phase_{ID}$ value no matter whether the access was a hit or a miss. Whenever the *lead* resumes execution, it increments the $phase_{ID}$ counter.
- STREX continuously monitors *victim* cache blocks. Upon encountering a victim block tagged with the current $phase_{ID}$ value, STREX context switches the current executing transaction and places it at the end of the thread queue. The next ready transaction resumes execution.
- If the *lead* transaction terminates, the next thread in the queue becomes the *lead*.
- Threads keep running in a round robin order until they all complete execution.
- Once all the threads in a team complete execution, the core becomes available for another team to execute.

7.4.2 Implementation

STREX's implementation requires the following components per core:

- thread execution queue,
- a $phase_{ID}$ counter,
- a $phase_{ID}$ tag per L1-I cache block,
- a victim block monitoring unit,

- a thread context switching unit, and
- STREX's control logic.

STREX tags all cache blocks with $phase_{ID}$ values. These $phase_{ID}$s can be maintained separately in a table (PIDT) to avoid impacting the L1-I design and latency. The *PIDT* contains a $phase_{ID}$ entry per cache block and is accessed in parallel with the L1 tag and data arrays. This work uses 8-bit $phase_{ID}$ tags and an 8-bit, modulo $phase_{ID}$ counter per core. The area overhead of the *PIDT* is small as it uses only eight additional bits per cache block. A PIDT does not contain any address tags or any additional block related information.

STREX groups similar transactions into teams by examining the address of the header instructions similar to SLICC (Section 7.3.3). The maximum number of transactions that a team can have ($team\_size$) is fixed system-wide. STREX assigns teams in the arrival order of the oldest thread in a team. When transactions that are not part of a team (*stray* transactions) become the oldest, they are scheduled individually.

Section 7.5.5 shows that by controlling the maximum allowed team size, STREX can trade-off overall throughput and per transaction latency. Software database management scheduling schemes that batch transactions exhibit a similar trade-off [74, 170].

STREX context switches threads by saving and restoring their architectural state to/from the L2 cache slice nearest to the core, also similar to what SLICC does (Section 7.3.4). Like SLICC, STREX requires support for hardware scheduling of multiple threads. Several proposals exist for implementing hardware-level thread scheduling and context switching (e.g., [167]). STREX serves as additional motivation for further investigating how hardware-level thread scheduling ought to be supported.

7.4.3 Effect on Regular Execution

This section discusses some of the implications of STREX for corner cases and its overhead.

Forward progress guarantees

STREX's effectiveness is limited by the amount of temporal overlap available across transactions of the same team. More precisely, the *lead* transaction has the largest impact on locality. For example, in a scenario where the *lead* transaction has minimal temporal overlap with the rest of the team, only the *lead* thread will make forward progress, while the others will have to wait until the *lead* finishes. There is no possibility of starvation as the *lead* is guaranteed to finish, and in the worst possible scenario, the rest of the threads will become *leads* in order. Since the *lead* always starts execution with a new $phase_{ID}$, it has the highest authority to evict cache blocks. If other threads do not touch these blocks and try to evict them, they will be context switched too early. Yet, with the workloads evaluated in this chapter, this scenario

Table 7.2: Workloads setup.

TPC-C	10 warehouses, 1 GB, Wholesale supplier
TPC-E	1000 customers, 20 GB, Brokerage house
MapReduce	Hadoop 0.20.2, Mahout 0.4 library, Wikipedia page articles (12 GB)

has never happened due to the inherent temporal overlap across transactions of the same type. An extension to STREX might investigate placing lower limits on the amount of forward progress a thread should make before context switching.

Context switching overhead

STREX incurs an overhead for context switching among team members. The architectural state of each transaction has to be saved and restored. In this work, thread contexts are saved in the second level cache to avoid thrashing L1-D. STREX amortizes this overhead by improving instruction and data locality, which result in overall throughput improvement (see Section 7.5.4). A portion of the physical address space is reserved for storing thread contexts. For the workloads studied, context switches are sufficiently infrequent that the overhead of saving and restoring context is never a significant fraction of the overall execution time (as Section 8.3.8 will also show). An implementation may choose to enforce a minimum number of instructions or cycles that a transaction ought to execute before a context switch is allowed.

7.5 Evaluation

The evaluation of SLICC and STREX is organized as follows:

- Section 7.5.2 studies the configuration parameters for SLICC.
- Section 7.5.3 demonstrates the impact of SLICC and STREX on instruction and data misses on varying core counts.
- Section 7.5.4 shows the throughput improvement of SLICC and STREX in comparison with a next-line prefetcher [173] and a state-of-the-art instruction prefetcher (PIF [53]).
- Section 7.5.5 investigates the trade-off between transaction latency and overall throughput under SLICC and STREX.
- Section 7.5.6 discusses the hardware cost of SLICC and STREX.

7.5.1 Methodology

Table 7.2 lists the workloads used. TPC-C [191] and TPC-E [193] run on top of the scalable open-source storage manager Shore-MT [172]. The client-driver and the database execute on the same machine and the buffer-pool is configured to keep the whole database in memory. The experiments use a 1.2 billion instructions sample from these workloads. We also perform experiments with the MapReduce workload [42, 148] as configured for the experiments in [54]. MapReduce has a relatively small instruction footprint and serves to demonstrate that the proposed scheduling mechanisms are robust in that they do not hurt performance for workloads that do not have similar behavior to OLTP. The MapReduce workload divides the input dataset across 300 threads, each performing a single map/reduce task. For clarity, the discussion focuses on TPC-C and TPC-E with MapReduce being included only where absolutely necessary.

Conventional operating systems lack support for thread context switching or migration at the hardware level. To work around this limitation, the experiments replay x86 execution traces, modeling the timing of all events, and maintaining the original thread sequence. The traces for TPC-C and TPC-E include both *user* and *kernel* activity, collected using QTrace [183], an instrumentation extension to the QEMU full-system emulator [17]. For MapReduce, Pin [124] was used to extract execution traces.

We evaluate the three scheduling mechanisms illustrated in Figure 7.1 as well as two instruction prefetchers:

- *Baseline*, the conventional transaction scheduling mechanisms, where each transaction starts and finishes its execution on one core without any interruption provided that no context-switching occurs due to I/O, waiting for locks, etc.
- *STREX* (Section 7.4), which time-multiplexes a batch of transactions on the same core to enable instruction reuse among the transactions in the batch.
- *SLICC* (Section 7.3), which spreads the computation of transactions over several cores to localize common instructions to caches without any software hints.
- *Next-line* ([173]) is the next-line prefetcher, which is used in most commodity hardware as an instruction prefetcher.
- *PIF* ([54]) is a state-of-the-art instruction prefetcher based on temporal streaming that has near-optimal coverage.

All the evaluated mechanisms are prototyped using the Zesto x86 multicore architecture simulator [122]. The Zesto simulator is a well-known infrastructure that has been used in other computer architecture studies (e.g., [66, 121, 203]).

Table 7.3 details the baseline architecture. With *N* cores, the baseline architecture has *N* hardware contexts with the OS making thread scheduling decisions. SLICC or STREX form teams over a pool of up to 30 virtual contexts. Unless otherwise noted, each core maintains a thread queue of up to ten threads. STREX forms teams of up to ten threads, whereas SLICC forms teams of up to *2N* threads. Throughput is measured as the inverse of the number of

Table 7.3: Simulated system parameters.

Processing Cores	*N* OoO cores, 2.5GHz, 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ, BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Cache	32KB, 64B blocks, 8-way, 3-cycle load-to-use 32 MSHRs, MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, *N* slices, 16-cycle hit latency, 64 MSHRs
Interconnect	2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks, 8B Bus Width, Open Page Policy

Latencies	CAS(10), RCD(10), RAS(35), RC(47.5)
	WR(15), WTR(7.5), RTRS(1), CCD(4), CWD(9.5)

cycles required to execute all transactions. The experiments also report the misses per kilo (1000) instructions for instruction (I-MPKI) and data (D-MPKI).

7.5.2 Exploring SLICC's Parameter Space

SLICC utilizes three thresholds for its thread migration decisions: *fill-up_t*, *matched_t*, and *dilution_t* (Table 7.1). It also depends on a partial-address bloom filter to reduce the overhead of remote cache segment searching. This section explores the parameter space for *SLICC*'s thresholds while measuring their impact on overall performance and how the bloom filter size affects its accuracy.

fill-up_t and matched_t

As defined in Section 7.3, *fill-up_t* sets the threshold for the initial fill-up period for an L1-I cache, during which instructions are brought in until the cache is almost *full*. When the miss counter (*MC*) is lower than *fill-up_t*, a thread is not allowed to migrate. On the other hand, *matched_t* sets the minimum number of tags that should be found on a remote cache before a thread migrates to it. Larger *matched_t* limits migration, while smaller values trigger too frequent migrations. To simplify the parameter search space, we first keep the *dilution_t* value at zero, and explore the parameter space of *fill-up_t* and *matched_t*. In addition, we assume zero-overhead to search for remote tags. We later model an actual search mechanism.

Figure 7.5 reports the throughput and L1-I MPKI (misses per 1000 instructions) relative to *Baseline* as a function of *fill-up_t* and *matched_t*. The *fill-up_t* values shown correspond to fractions of the L1-I cache capacity (512 cache blocks): $\frac{1}{4}$, $\frac{1}{2}$, and 1. The *matched_t* range shown is $2-10$; larger *matched_t* values further degrade performance.

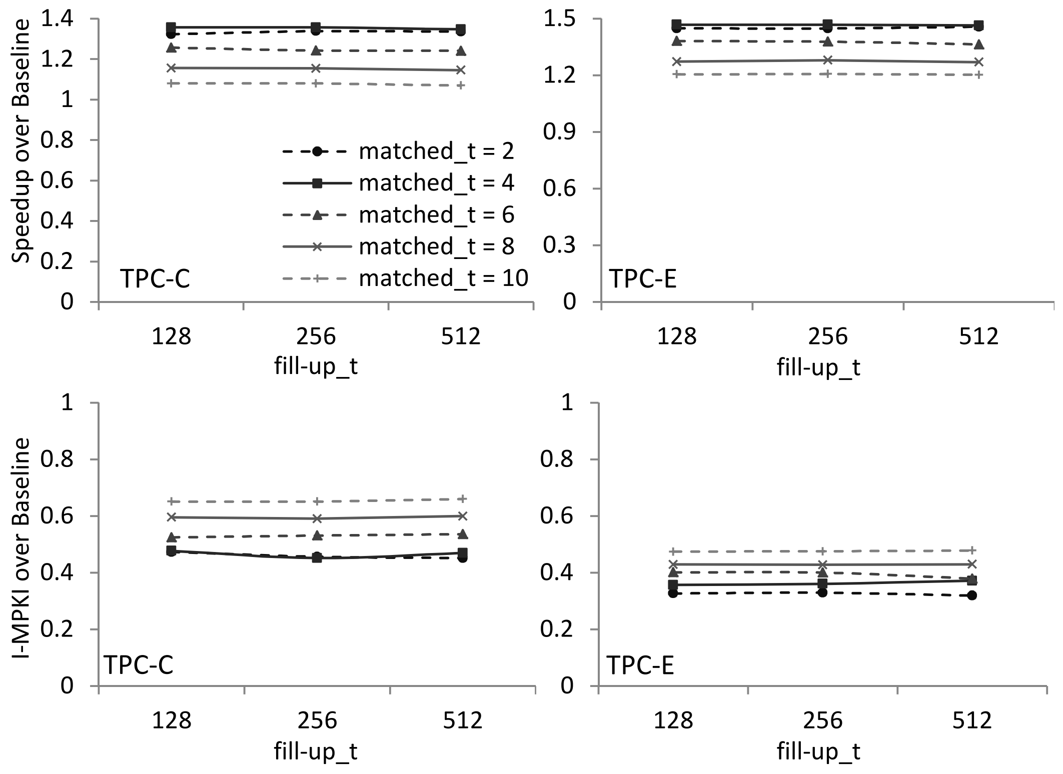


Figure 7.5: Relative performance of *SLICC* as a function of *fill-up_t* and *matched_t* thresholds (value 1 on Y-axes represents *Baseline*).

The results show that *SLICC* is not very sensitive to the different values of *fill-up_t*. *Fill-up_t* is actually a proxy for warming-up the caches; it affects only the first migration from a core. Thus with more migrations, the effect of *fill-up_t* diminishes. On the other hand, Figure 7.5 demonstrates that for *matched_t* values larger than four, performance benefits drop.

dilution_t

Dilution_t is the minimum number of misses in the last 100 accesses to allow migration. It tends to restrict migration to the cases when more frequent misses are observed by a thread. Using a small value for *dilution_t* triggers more frequent migrations. Using too large a value for *dilution_t* reduces migration overhead, but with a possible I-MPKI increase since it results in partial cache thrashing. Figure 7.6 shows L1-I MPKI and throughput of *SLICC* relative to *Baseline* for *dilution_t* values 1 through 30 when *fill-up_t = 256* and *matched_t = 4* (best configuration from Figure 7.5). As *dilution_t* increases, instruction misses are reduced improving performance up to a point. Afterward, larger *dilution_t* leads to fewer migrations, less overhead, but higher I-MPKI. There is a trade-off between reducing instruction misses and reducing migration overhead. Beyond *dilution_t* values of 28 (TPC-C) and 24 (TPC-

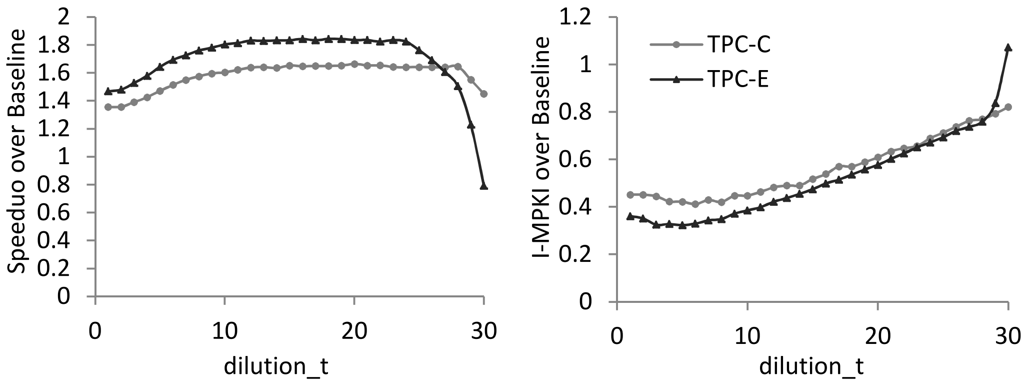


Figure 7.6: Relative performance of *SLICC* as a function of *dilution_t* (value 1 on Y-axes represents *Baseline*).

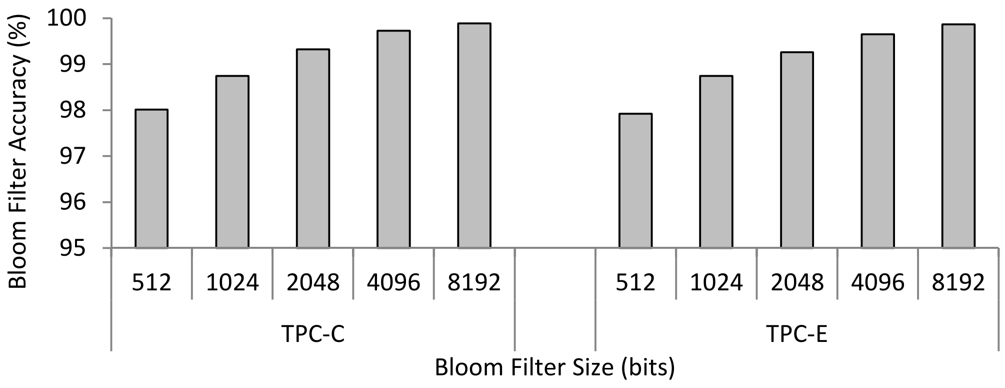


Figure 7.7: *SLICC*'s partial-address bloom filter accuracy with respect to bloom filter size.

E), although the overall MKPI is reduced, the performance degrades due to more limited migration. At even higher *dilution_t* values, migrations cease and performance drops below *Baseline*.

Bloom filter accuracy

Section 7.3.2 explains that using a partial-address bloom filter reduces the overhead of remote cache segment searching. Figure 7.7 shows the accuracy of bloom filters of different sizes. The smallest bloom filter requires 512 bits to support evictions for a 32KB cache, with 64B blocks, and 512-sets. Accuracy is measured for all cache accesses and an access is accurate if the bloom filter and the cache agree on whether this is a hit or a miss. The trend is similar for TPC-C and TPC-E.

In the remaining parts of this evaluation, we use *dilution_t = 10*, *fill-up_t = 256*, and *matched_t = 4* as well as a bloom filter of size 2K-bits since its effect on performance is less than 0.5% (99.3% accuracy).

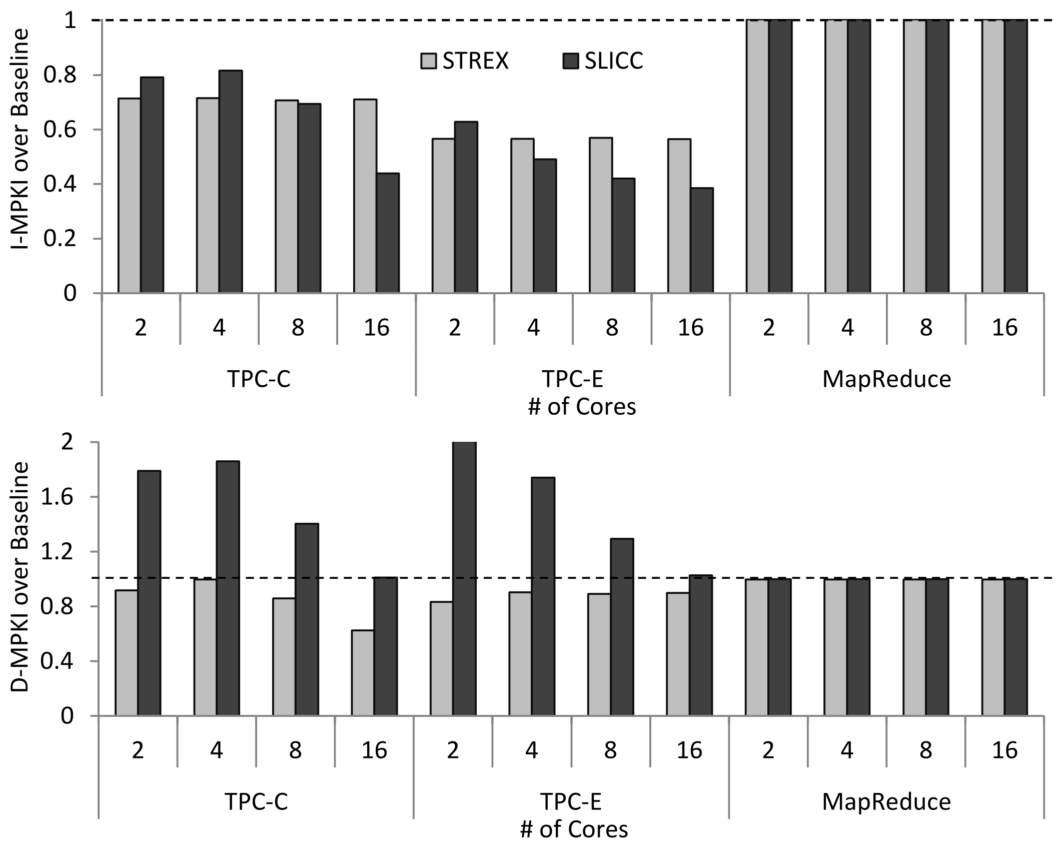


Figure 7.8: Effect of *SLICC* and *STREX* on L1 instruction and data misses as the number of available cores increase. Y-axes plot the number of misses per 1000 instructions (MPKI) normalized over *Baseline* (=1 on Y-axis).

7.5.3 L1 Miss Rate

Figure 7.8 reports the relative L1 I-MPKI and D-MPKI incurred by *SLICC* and *STREX* over *Baseline* for two to 16 cores. For MapReduce, *SLICC* and *STREX* do not affect the I- and D-MPKI as context switches rarely occur. The next section shows that performance is virtually identical as well.

STREX consistently reduces I-MPKI over *Baseline* with the I-MPKI remaining practically constant (the variation is less than 2%) no matter how many cores are available. *STREX* reduces I-MPKI by an average of 29% and 44% for TPC-C and TPC-E, respectively, over *Baseline*. *STREX* also improves data locality by synchronizing their execution. The same-type transactions tend to access the same metadata and locks for the same tables, as well as the same index roots during index probes, and they tend to do so in the same sequence. *STREX* enables reuse for such shared data items across transactions. For 16 cores, *STREX* reduces D-MPKI by 37% and 11% for TPC-C and TPC-E, respectively.

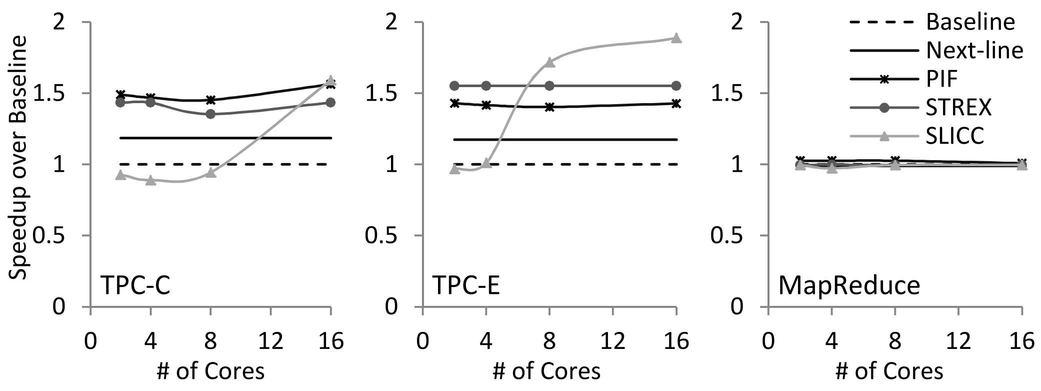


Figure 7.9: Throughput of SLICC and STREX compared to state-of-the-art instruction miss reduction techniques as the number of available cores increases. Y-axes plot the speedup over *Baseline* (=1 on Y-axis).

On the other hand, *SLICC* reduces more instruction misses as more cores become available. When there are less than eight or four cores for TPC-C and TPC-E, respectively, *STREX* outperforms *SLICC* in improving instruction cache locality. However, *SLICC* is more effective under high core counts; i.e., when there are enough cores to spread the instruction footprint of the transactions. Moreover, *SLICC* always increases D-MPKI. Most of the increase in D-MPKI, however, is for stores. Stores form 45% of total memory accesses in our experiments, while loads are nearly unaffected. This means that the increased D-MPKI with *SLICC* can be easily overlapped through out-of-order execution on the architecture we simulate. Therefore, *SLICC* improves performance as a result of reduced I-MPKI even though it slightly increases D-MPKI when there are enough cores available (as the next section shows).

7.5.4 Throughput

This section compares *SLICC* and *STREX* to *Baseline*, *Next-line* prefetcher, and the state-of-the-art instruction prefetcher *PIF*. Figure 7.9 reports overall throughput normalized over the throughput of *Baseline* with the corresponding number of cores.

STREX consistently improves throughput over *Baseline* and *Next-line* by an average of 35–55% and by 20–32%, respectively, for 2–16 cores. Contrary to *SLICC*, *STREX* is insensitive to the number of cores and always improves performance.

SLICC either degrades or barely improves performance over *Baseline* for TPC-C with up to eight cores and for TPC-E with up to four cores. For the same configurations the *Next-line* prefetcher consistently outperforms *SLICC*. However, *SLICC* outperforms *STREX* and does so considerably when there are at least eight and 16 cores, respectively, for TPC-E and TPC-C. With 16 cores, *SLICC* is 11% and 21% faster than *STREX* for TPC-C and TPC-E, respectively.

The results of Figure 7.9 are an upper bound for *PIF*'s performance as the experiment models *PIF* with a 100% hit rate L1-I cache. Demand traffic is generated for cache blocks that would

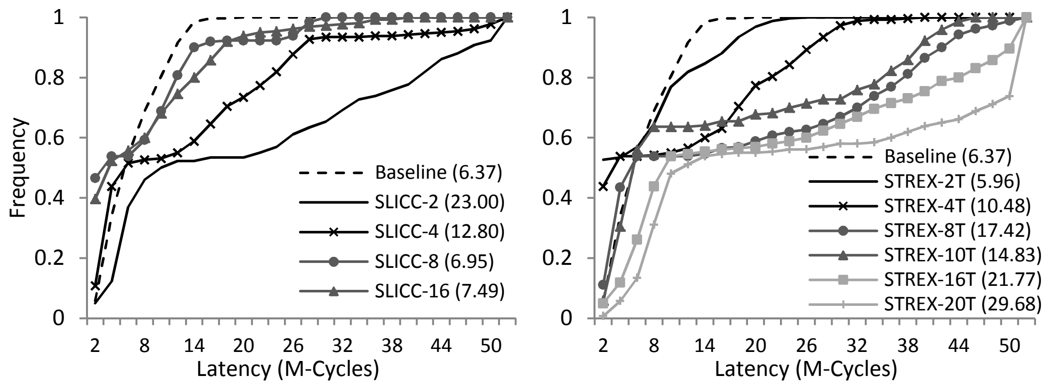


Figure 7.10: TPC-C transaction latency distribution as a function of *team_size* for *STREX* and of core count for *SLICC*.

have otherwise missed on a real cache, thus partially modeling the contention that PIF would incur. This is an optimistic 100% accurate prefetcher that issues perfectly timely requests. The actual PIF prefetcher may fail to prefetch in some cases, over-prefetch in others, and not always manage to completely hide the miss latency. For 2–16 cores, *STREX* achieves on average 95% of *PIF*'s performance for TPC-C, and outperforms *PIF* by 9% for TPC-E, with less than 2% of the storage overhead. On the other hand, for 16 cores, *SLICC* is within 2% of *PIF*'s performance for TPC-C and 21% faster than *PIF* for TPC-E, with only 2.4% of PIF's storage requirements (see Section 7.5.6) per core.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains unaffected with all techniques.

7.5.5 Transaction Throughput vs. Latency

STREX improves the overall throughput, but may increase transaction latency due to transaction batching. By adjusting the maximum number of transactions per team (*team_size*), it is possible to control this trade-off as Figure 7.10 suggests. Figure 7.10 shows the distribution of transaction latencies when running TPC-C for *Baseline*, *STREX*, and *SLICC*. For *STREX* the figure reports latency distributions as a function of *team_size*, noted as STREX-xT, where x is a team size in the range of two to 20. In all preceding experiments teams had up to ten threads. With *STREX*, the transaction latency is independent of the core count; hence, Figure 7.10 shows latencies for 16 cores only. For *SLICC*, however, the figure reports latency distributions as a function of core count, noted as SLICC-x where x is a core count in the range of two to 16. The legends on the two graphs report in parentheses the average per distribution latencies.

A transaction's latency is the number of cycles elapsed from the moment it enters the transaction queue until it completes execution. For *STREX*, as $team\_size$ increases, the distribution tends to move toward longer transaction latencies. Figure 7.11 shows the corresponding relative throughput for TPC-C and TPC-E demonstrating that throughput also increases with

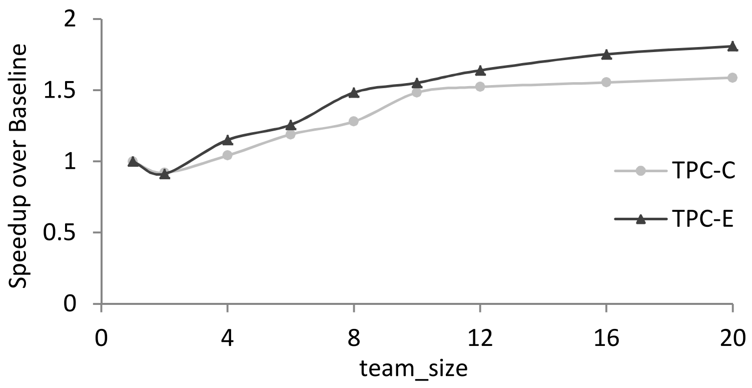


Figure 7.11: *STREX* throughput over *Baseline* for a range of *team_size* values.

the *team_size*. With up to 20 threads per team, throughput improvements are the highest at 59% and 80% over *Baseline* for TPC-C and TPC-E, respectively. It would be straightforward to make the *team_size* configurable by the system, which can then set *team_size* according to its specific needs. Figure 7.10 shows that with *SLICC*, transaction latencies become shorter as the number of cores increases as expected.

7.5.6 Hardware Cost

Table 7.4 details the cost of SLICC's and STREX's hardware components.

Section 7.3.2 describes the hardware components Table 7.4 gives for SLICC except for the *thread queue*, which holds threads waiting for cores. Each thread queue entry contains a unique numerical ID, a pointer to the threads' context, and a core ID. The thread queues can be local to each core, or centralized to one core. The table shows the cost for a centralized queue. Fewer entries are required when the queues are local to each core. The *team management table* is responsible for forming teams of similar threads. Each entry consists of: a unique numerical ID, a type ID, a team ID, index within a team, and a timestamp. The team management table is best thought of as being centralized, since every core needs to know which cores are assigned to which teams. We can either have one centralized copy or per core copies that are kept coherent. For this work we simulated a centralized copy at one of the cores and modeled the necessary traffic. On each core, a SLICC agent is responsible for managing the thread queue. The thread queue is a circular FIFO buffer and the first entry is executed until it migrates, completes, or gets blocked for I/O. On the latter case, the thread is moved to the end of the queue. With an over-provisioned thread queue of 30 threads and a copy of the team management table, per core, SLICC requires a maximum of 966 bytes in addition to logic. None of the logic operations for SLICC are on the critical path of transaction execution.

On the other hand, STREX utilizes two main units: a team formation unit and a thread scheduler unit. The team formation unit is used to group similar transactions into teams as in SLICC. In this work STREX searches through a window of 30 threads. The *team management*

Table 7.4: Hardware space cost of SLICC and STREX.

Cache Monitor Unit		
	SLICC	**STREX**
Missed-Tag Queue (MTQ)	60-bits (16-core, *matched_t* = 4)	NA
Miss Shift-Vector (MSV)	100-bits	NA
Cache Signature (Bloom Filter)	2K-bits	NA
Total	2208 bits (276 Bytes)	0 bits
Thread Scheduler		
	SLICC	**STREX**
Thread Queue	30-entries (12-bits ID, 48-bits pointer to thread context, 4-bits core ID)	20-entries (12-bits ID, 48-bits pointer to thread context, 1-bit *lead* flag)
$phase_{ID}$ Counter	NA	8-bits
Auxiliary $phase_{ID}$ Table	NA	8-bit per cache block (512 cache blocks)
Total	1920 bits (240 Bytes)	5324 bits (665.5 Bytes)
Team Formation		
	SLICC	**STREX**
Team Management Table	60-entries (12-bits ID, 32-bits timestamp, 4-bits type ID, 4-bits team ID, 8-bits team index)	30-entries (12-bits ID, 32-bits timestamp, 4-bits type ID, 4-bits team ID, 8-bits team index)
Total	3600 bits (450 Bytes)	1800 bits (225 Bytes)
Grand Total	7728 bits (966 Bytes)	7124 bits (890.5 Bytes)

table maintains information about threads until they are dispatched to a core and its entries consist of the same information as in SLICC. As detailed in Section 7.4.1, the thread scheduler unit is responsible for incrementing the $phase_{ID}$ counter, tagging cache blocks with the current $phase_{ID}$ value, keeping track of the *lead* thread, monitoring instruction cache block victims, and context switching threads. The thread queue is a circular FIFO buffer. Each entry consists of a unique ID, a pointer to the thread's context in the L2 cache, and a *lead* flag bit. The size of the thread queue should be the maximum value allowed for the *team_size* configuration parameter. Most experiments set *team_size* to 10 with 20 being the maximum considered. Assuming one team management table per core, the total storage required per core by STREX is 890.5 bytes in addition to the logic.

7.6 Related Work

There have been several hardware and software proposals for reducing instruction stalls that are applicable to OLTP workloads such as instruction prefetching [52, 53, 101, 105, 161], computation spreading [30], and transaction batching [74].

Instruction prefetching is a well-studied research area. Stream buffers [101, 161] are simple to implement in hardware, but they provide relatively low instruction coverage. More sophisticated prefetchers [52, 53] utilize bookkeeping structures to record encountered instruction streams, and to replay them when part of the stream is touched again. Their structures increase area and energy. Moreover, prefetching, unless 100% accurate, increases miss traffic for fetching blocks that are never touched prior to being evicted. PIF [53] was reported to achieve near-optimal instruction coverage. Section 7.5 compares SLICC and STREX with PIF and shows that their performance is competitive while their hardware space cost is 40× lower than PIF's. SHIFT [105] is a recent proposal that aims to minimize the space cost of PIF through sharing the instruction stream history across cores, which also exploits the observation of high temporal code overlaps across concurrent threads in a system. Nevertheless, any prefetching technique is orthogonal to the scheduling mechanisms this chapter proposes. For example, STREX and SLICC can avoid many of the misses that PIF has to incur, thus possibly reducing the storage, power, and bandwidth overhead of PIF. PIF could reduce execution time for the initial transactions, thus improving performance when used in conjunction with STREX or SLICC. Therefore, there is potential to investigate the combination of these proposals.

Chakraborty et al. show a high-degree of redundancy in instruction fragments across threads concurrently running on multiple cores [30]. They propose CSP, which employs thread migration to distribute the dissimilar instruction code segments and group the similar ones together. For system code, which is commonly used by multiple threads, CSP fragments and distributes the code across a group of dedicated cores. CSP then migrates threads to these dedicated cores to execute system code. When threads are done, they return back to their original cores to resume execution for the user-level code. Thus CSP is limited to fragmenting OS code, losing opportunities of fragmentation within user code. SLICC borrows ideas from CSP, however, generalizes thread migration to include interleaved user-OS code fragmentation points. In addition, thread migration in SLICC is managed by the hardware, while with CSP, the OS performs the migrations.

STEPS [74], on the other hand, is a software solution whose approach is identical in spirit to STREX. STEPS relies on manual code instrumentation, which is a cumbersome task that requires a high level of expertise, is prone to many errors as it is manual, and results in code that is not portable since it is platform dependent. A slightly improved version, autoSTEPS, automates several components of the instrumentation process.

7.7 Conclusions

OLTP workloads suffer from high instruction miss stalls on high-end server processors since their transaction instruction footprints are by far larger than current L1-I caches, thus leading to ongoing cache thrashing. To exploit the significant temporal instruction overlap among similar transactions, this chapter presents two programmer-transparent scheduling mechanisms to increase instruction reuse in the caches. While SLICC adaptively spreads the execution of a transaction over multiple cores through thread migration, STREX time-multiplexes transactions on one core. They both enable reuse of common instructions by localizing them to cores. As a result, they improve performance over conventional transaction scheduling and exhibit competitive performance to state-of-the-art prefetchers despite significantly lower space cost. When the available aggregate L1 instruction cache capacity is enough to spread a workload's instruction footprint, SLICC outperforms STREX, whereas under low core counts STREX should be the choice of scheduling.

8 Transaction-aware Instruction Chasing

The previous chapter (Chapter 7) aims to maximize instruction cache locality through two hardware mechanisms and surveys related work that propose either software- or hardware-side solutions to the same problem. However, exploiting hardware resources based on the hints given by the software-side has not been widely studied for data management systems. This chapter presents ADDICT, a software-guided hardware mechanism that schedules transactions in a way to maximize the instruction cache locality.

ADDICT is based on the same observation that inspired the two hardware mechanisms in the previous chapter: concurrent transactions exhibit high instruction commonality (Section 6.6). However, ADDICT initially performs a profiling step to determine the most frequent actions of database operations, whose instruction footprint can fit in an L1 instruction cache, and assigns a core to execute each of these actions. Then, it schedules each action on its corresponding core. This way, it requires less hardware complexity and leads to more precise scheduling decisions.

Our prototype implementation of ADDICT reduces L1 instruction misses by 85% and the long latency data misses by 20% compared to the conventional way of scheduling transactions. As a result, ADDICT leads up to a 50% reduction in the total execution time for the evaluated workloads. Furthermore, it is 20% and 35% faster than SLICC and STREX, respectively, on average. [1]

8.1 Introduction

As discussed in the previous part, several workload characterization studies show that micro-architectural resources are severely underutilized when running online transaction processing (OLTP) applications [54, 177, 186] (and also Chapter 5). Up to 80% of the execution cycles go to memory stalls [54]. As a result, on modern processors, OLTP barely achieves one instruction per cycle (IPC), far below the processors peak capability of four IPC.

[1] This chapter uses material from [187].

Previous work on reducing memory stall time for data management systems aimed at reducing cache miss rates, focusing primarily on improving locality and cache utilization for data rather than for instructions. Proposals range from cache-conscious data structures and algorithms [32, 58] to sophisticated data partitioning and thread scheduling [154] on the software-side, whereas hardware techniques mainly target data prefetching [175].

However, as we have shown in Part II, for traditional transaction processing systems, the stall time due to L1 instruction misses is at least as problematic as long-latency data misses from the last-level cache. Improving code layout by writing better code or by compilation optimizations [159] does improve instruction cache utilization, but does so by mainly reducing conflict misses. However, it is capacity misses that dominate L1 instruction misses on today's most commonly used server hardware (Section 6.5.1); the instruction footprint of a transaction is too big to fit in the L1 caches, thus thrashing L1-I and leading to very lengthy stalls.

Chapter 7 proposes two hardware mechanisms, STREX and SLICC, which address capacity instruction misses in OLTP. STEPS [72, 74] is a software mechanism with the same goal as STREX and SLICC. These proposals are motivated by the observation that threads executing transactions in parallel on a multicore server *execute a significant amount of common code* (Section 6.6). To be able to reuse the common instructions already brought into L1, STEPS [74] and STREX [15] time-multiplex a batch of threads on the same core, whereas SLICC [13, 14] spreads the computation of a transaction to several cores to localize the common instructions to specific caches. Nevertheless, STREX and SLICC are completely oblivious to software and miss the opportunity to more precisely improve instruction locality through software guidance. STEPS, on the other hand, is a pure software technique designed to run only on a single-core and requires significant manually-aided instrumentation. Furthermore, all three techniques increase average transaction latency and STREX and STEPS increase the potential of deadlocks due to extensive batching and context-switching.

The goal of this chapter is to better exploit the L1 caches when running transactions based solely on hints from the software-side. The traditional way of scheduling transactions considers each as one big, monolithic task. Therefore, the granularity of tasks assigned to run on a core is too coarse, which leads to cache thrashing due to the large instruction footprint of the scheduled task. This work proposes to reduce the granularity of task-to-core assignment by scheduling the actions of common database operations. This approach bridges the gap between a transaction's instruction footprint and the L1 capacity.

To assign finer-grained tasks to cores while running transactions, we design ADDICT, a transaction scheduling mechanism that chases instruction cache locality. ADDICT first segments a database operation into smaller actions, where the instruction footprint of each action fits in a single L1 instruction cache. Then, it assigns specific cores for each of these actions and migrates the transactions over multiple cores using core assignment decisions that aim to maximize instruction locality for each action.

The contributions and the organization of this chapter are as follows:

- Based on the insights from Section 6.6, Section 8.2 describes the design of ADDICT, a transaction scheduling mechanism that views transactions as a composition of the actions from the database operations they execute.

- Section 8.3 evaluates our prototype of ADDICT and shows that ADDICT reduces L1 instruction cache misses by 85%, while also reducing the long-latency data misses from the last-level cache by 20%. Even though ADDICT slightly increases L1 data cache misses and average transaction latency, the improved instruction locality leads to 45% and 15% gains in total execution time on average on shallow and deep cache hierarchies, respectively. ADDICT also outperforms STREX (by 35%) and SLICC (by 20%).

Finally, Section 8.4 surveys related work and Section 8.5 concludes.

8.2 ADDICT

Section 6.6 emphasizes that transactions exhibit high instruction commonality whereas the data commonality is low. Based on this finding, we design an alternative method to schedule transactions to maximize instruction cache locality. ADDICT, an advanced instruction chasing mechanism for transactions, departs from the traditional way of scheduling transactions, which sees a transaction as one big task. ADDICT rather considers a transaction as a combination of the database operations it calls and migrates transactions over cores based on the actions their operations are about to execute.

ADDICT consists of two steps, which are detailed in the subsequent subsections.

- Step 1 (Section 8.2.1) determines the migration points in each database operation.

- Step 2 (Section 8.2.2) spreads the execution of a transaction over multiple cores based on the migration points picked in the previous step.

Step 2 is always dynamic since it orchestrates transaction execution during the actual run, whereas Step 1 can be either static or dynamic depending on the application's needs.

8.2.1 Finding Migration Points

To be able to determine when and where to move a transaction at run-time, ADDICT first needs to decide on the migration points in each database operation. ADDICT picks these points separately for each transaction type since the code paths each database operation takes might change based on the tables accessed in a particular transaction, as we observe in Section 6.6.2.

Algorithm 4 Finding migration points.
Inputs: list of transactions and database operations.
Output: a list of instruction sequences that indicate the migration points picked for each database operation invoked by each transaction.

```
1:  $m \rightarrow$ keeps possible migration points
2:  for instruction access $addr$ in workload do
3:      if a transaction entry/exit then
4:          empty the L1-I cache
5:          if transaction entry then
6:              $xct$ = current transaction type
7:      else if a database operation entry/exit then
8:          empty the L1-I cache
9:          if operation entry then
10:             $op$ = current operation
11:             create empty $sequence$
12:         else
13:             $m[xct][op][sequence]++$
14:     else if $addr$ request requires an eviction then
15:         empty the L1-I cache
16:         $sequence.append(addr)$
17: return the $sequence$ with the highest value for each $m[xct][op]$
```

Algorithm

Algorithm 4 shows the details of ADDICT's initial step, which finds the migration points for a workload. It takes a list of *indicators* to identify the transactions and database operations in the workload. These indicators can be function names or instruction addresses that correspond to the entry and exit points of the transactions or operations.

In lines 1-16 of Algorithm 4, ADDICT records the sequences of instructions that cause an eviction from each database operation invoked in a particular transaction type as migration point candidates. In parallel, it collects the occurrence count for each of these sequences. Since ADDICT aims to migrate transactions at the granularity of actions from database operations that can fit in an L1-I cache, it resets the L1-I cache upon transaction or database operation entry and exit points in this step. After collecting the candidates, ADDICT picks the most frequent sequence of instructions for each database operation from each transaction type as migration points (line 17 in Algorithm 4).

Example

In line 17 of Algorithm 4, ADDICT has information similar to the following in map *m*:

```
1) xct1→insert→0x8b5f5f 0x899397→9
```

```
2) xct1→insert→0x9bd97f 0x8b5fbf 0x94ffde→1

3) xct2→probe→0x98560e 0x8d97bc→10

4) xct2→update→0x9557f0→5
```

Each entry in m correspond to a migration point sequence candidate from a particular database operation called within a particular transaction type. The entries also keep the number of times the corresponding migration point sequence is observed during the profiling phase. For example, (1) from above indicates that the migration point sequence `0x8b5f5f 0x899397` are the sequence of instructions that cause L1-I cache evictions in nine instances of the *insert* operation called from xct1.

ADDICT goes over this information to figure out the most frequent sequence of migration points. In this example, these are

- (1) for *insert* operation in xct1,
- (3) for *probe* operation in xct2, and
- (4) for *update* operation in xct2.

Migration points in (2) represent a corner case in the *insert* tuple operation since they only appear once among all instances of xct1. For *probe* and *update* operations in xct2, however, there are no alternative migration points to the ones in (3) and (4). If there are multiple sequences of migration points that are the most frequent for an operation, ADDICT picks one of them randomly. However, we do not observe such cases for the workloads we evaluate in Section 8.3.

Implementation

There are several ways of deploying Algorithm 4 in practice. Adopting ADDICT as a pure dynamic approach requires integrating Algorithm 4 with the actual workload run. ADDICT can perform this step as a part of the ramp-up time (a few seconds) without making any specialized scheduling decisions for transactions and then switch to migrating transactions based on the information collected in this step. On the other hand, Step 1 of ADDICT can be static and performed *a priori* as well. In this case, ADDICT would migrate transactions over the dedicated cores as soon as the real workload run starts.

In this step, ADDICT detects cache-sized chunks from each database operation. Therefore, given an empty L1-I cache, ADDICT should track the instructions that cause cache evictions within each database operation. To track such instructions at run-time, ADDICT can use either the hardware counters on the target hardware or mechanisms like informing memory operations [83]. Upon a transaction/operation entry/exit or eviction, ADDICT must flush the

L1-I contents to reset the instruction cache and determine the next cache-sized code chunk in the current operation.

In addition, within the storage manager, there might be functions/routines where one should avoid migrating. For example, migrating within short-critical sections or lock acquisitions/releases would increase the duration of these routines. Therefore, Algorithm 4 can take additional input that indicates such functions and avoid picking migration points within these functions.

8.2.2 Migrating Transactions

After determining the migration points, ADDICT applies its scheduling principles during regular transaction execution. Since it picks the migration points separately for each transaction, it batches same-type transactions to maximize instruction cache locality. Furthermore, while processing a batch, ADDICT adjusts the core assignments based on the needs of the application, i.e., it assigns more cores to a migration point if it is more frequently used.

Algorithm

Algorithm 5 shows the core assignment and transaction migration principles of ADDICT's Step 2. Algorithm 5 takes as input the migration points found by Algorithm 4. It first assigns cores to each of the migration points (lines 1-14). Then, it migrates transactions based on the core assignments (lines 16-31).

Lines 1-14 of Algorithm 5 handle the core assignments on the target hardware. As in Algorithm 4, ADDICT considers each transaction separately. Therefore, each transaction takes $core_0$ as their entry core (lines 3-6). For the remaining core assignments, ADDICT incrementally assigns a unique core ID to each database operation in a transaction (lines 7-10) and its corresponding migration points (lines 11-14). Section 8.2.2 describes how ADDICT handles the cases where the number of migration points does not exactly match the number of available cores. Algorithm 5 omits these details for simplicity.

Lines 16-31 of Algorithm 5 perform the actual transaction execution. To maximize cache locality, in lines 16-17, same-type transactions from the list of client requests form a batch. The batch size is equal to the number of available cores on the current hardware to avoid increasing average transaction latency drastically. Then, for each instruction to be executed, ADDICT checks whether the transaction should migrate to another core based on the prior core assignment decisions (lines 20-26). If destination core ID has a different value than the current core ID of the transaction being executed, ADDICT migrates the transaction provided that there is an available destination core for the current migration point (lines 27-31).

To ensure the instruction stream is on a path that matches the migration points sequence in the input, ADDICT also tracks the previous migration addresses for each migration point.

It migrates a transaction upon encountering a migration point only if that transaction has already executed the previous migration point in the sequence (line 25). An instruction address might be used several times during the execution of a database operation. However, it might lead to migration only if it is called through a specific path. Therefore, ADDICT must check for such order dependencies in the migration sequence.

Algorithm 5 Migrating transactions.
Input: migration points (output of Step 1) m.

```
1: $cores$ → keeps core assignments
2: $prev$ → keeps previous migration point
3: for each transaction type $xct$ in $m$ do
4:     $core = 0$, $op = 0$, $prev = 0$
5:     $addr$ = entry instruction for $xct$
6:     $cores[xct][op][addr] =< core, prev >$
7:     for each operation $op$ in $m[xct]$ do
8:         $core++$, $prev = 0$
9:         $addr$ = entry instruction for $op$
10:        $cores[xct][op][addr] =< core, prev >$
11:        for each migration address $addr$ in $m[xct][op]$ do
12:            $core++$
13:            $cores[xct][op][addr] =< core, prev >$
14:            $prev = addr$
15:
16: for each transaction type $xct$ in the list of requests do
17:     group $num\_cores$ transactions of type $xct$ in $batch$
18:     for each core do $m_{xct} = cores[xct]$, $op = 0$, $prev = 0$
19:     for each transaction $t$ in $batch$ do
20:         for each instruction access $addr$ in $t$ do
21:             $core_{dest} = core_{curr}$
22:             if $addr$ is in $m_{xct}$ then
23:                 $op = addr$, $prev = 0$
24:             if $addr$ is in $m_{xct}[op]$ then
25:                 if $prev == m_{xct}[op][addr].prev$ then
26:                     $core_{dest} = m_{xct}[op][addr].core$, $prev = addr$
27:             if $core_{dest} != core_{curr}$ then
28:                 if $core_{dest}$ is available then
29:                     migrate $t$ to $core_{dest}$
30:                 else
31:                     steal an idle core from another migration point or wait in the work queue of $core_{dest}$
```

Example

Let's assume that Algorithm 5 takes as input the output of the example in Section 8.2.1, which is:

```
xct1→insert→0x8b5f5f 0x899397→9

xct2→probe→0x98560e 0x8d97bc→10

xct2→update→0x9557f0→5
```

At line 15 of Algorithm 5, $cores$ would have the assignments given below:

xct1→<core$_0$,0>

xct1→insert→<core$_1$,0>

xct1→insert→0x8b5f5f→<core$_2$,0>

xct1→insert→0x899397→<core$_3$,0x8b5f5f>

xct2→<core$_0$,0>

xct2→probe→<core$_1$,0>

xct2→probe→0x98560e→<core$_2$,0>

xct2→probe→0x8d97bc→<core$_3$,0x98560e>

xct2→update→<core$_4$,0>

xct2→update→0x9557f0→<core$_5$,0>

After deciding on the core assignments, ADDICT starts batching transactions. Let's assume that it initially batches requests of xct1 and one of the transactions in that batch has the following instruction sequence:

```
xct1_entry_instr ...  insert_entry_instr ...  0x899397 0x89939c 0x89939e ...
0x8b5f5f 0x8b5f62 ...  0x899397 ...
```

Upon xct1 and *insert* operation entry, ADDICT migrates the transaction to core$_0$ and core$_1$, respectively. When the instruction 0x899397 is accessed for the first time, since its previous migration point, 0x8b5f5f, is not yet encountered, ADDICT keeps the transaction on the same core. When the transaction uses the instruction 0x8b5f5f, since it is the first migration point in the *insert* operation, ADDICT migrates the transaction to core$_2$. When the instruction 0x899397 is reused, since it comes after a migration to core$_2$ due to 0x8b5f5f, ADDICT now migrates the transaction to core$_3$.

Load Balancing

Algorithm 5 presents a simplified version of the actual ADDICT algorithm as it just assigns one core per migration point. In a typical OLTP workload running on modern server hardware, there are

- database operations that are more frequently used than others and
- more or fewer cores than the number needed by a transaction.

We describe how ADDICT deals with such cases below.

More migration points than cores: If the migration points for a transaction require more cores than what is available in the system, ADDICT starts ignoring the internal migration points in less frequent database operations starting from the last migration point. For example, in Section 8.2.2, if there were only four cores in the system, there would not be any cores assigned to `0x9557f0` in *update* and `0x8d97bc` in *probe* for xct2. `0x9557f0` in *update* is ignored prior to `0x8d97bc` in *probe* since the *update* operation occurs less often (5 vs. 10 in Section 8.2.1:Example). `0x8d97bc` in *probe* is ignored next since there are no more internal migration points to ignore in *update*. In our experiments in Section 8.3, this situation arises for some TPC-C and TPC-E transactions.

If there are too few cores available for a workload, e.g., if the number of cores is even less than the number of operations executed by a transaction type, ADDICT can either fall back to traditional scheduling or switch to a scheduling technique that optimizes instruction locality for a single-core (e.g., STREX (Section 7.4), STEPS [74]).

Fewer migration points than cores: When a transaction requires fewer cores than what is available on the machine, which is the common case in the era of multisocket multicores, ADDICT distributes the remaining cores based on the frequency of operations. For example, in Section 8.2.2, if there were ten cores in the system, there would be two cores assigned to each migration point in the *probe* operation since it is more frequent than *update*. The remaining core would be given to the entry point of *update*.

In the case of having enough cores to assign to the migration points from multiple transactions, ADDICT can run multiple batches of transactions in parallel.

Dynamic reassignment of cores: After the initial core assignments, ADDICT deploys a dynamic approach. Whenever the destination core of migration is not available, i.e., occupied by another transaction (line 31 of Algorithm 5), there are two options:

- if there are any idle cores that belong to another migration point, ADDICT re-assigns one of these idle cores to the current migration point and

- if there are no idle cores, then the transaction waits in the work-queue of the destination core.

Implementation

We design ADDICT to be a software-guided hardware mechanism. We think of the migration points picked by Step 1 of ADDICT as the software hints used by Step 2 of ADDICT on the hardware side. Therefore, while Step 1 can use the already existing hardware features of modern hardware, Step 2 requires some additional features from the hardware side. These additional features stem from two requirements:

- keeping track of the migration points and
- performing fast and exact thread migrations.

To be able to decide when and where to migrate a transaction, each core must keep the list of migration points for that transaction as well as an indicator for the current database operation and the previous migration point. ADDICT distinguishes both database operations and migration points using instruction addresses. Therefore, we can calculate ADDICT's space cost mainly based on the space cost of an instruction address. If we distinguish instructions based on their unique cache block addresses during program execution, then 58bits would be enough for an instruction on a server with 64B cache blocks and 64bit memory address space (most common case for modern servers). Keeping the current database operation and the previous migration point would require 116bits per core. For each migration point, we need to map a <*database operation, migration point*> pair to a <*core id, previous migration point*>. In this mapping, except for the *core id* value, the other three values are instructions. We can keep the *core id*s as 8bit integers since 8bits already give us 256 distinct values. As a result, 182bits would be enough to keep a migration point. This way, a core can keep up to 40 migration points in less than 1KB of space, which is a feasible space cost per core on most server hardware as also stated in Section 7.5.6.

On the other hand, the hardware cost of the thread migrations is mainly algorithmic, which Section 7.3.4 also discusses in detail. We estimate the time required per thread migration to be ~90 cycles—the cost of writing/reading a thread's state (e.g., the register values, last program counter, etc.) to/from the last-level-cache (~6 cache lines).

Deploying ADDICT as a pure software mechanism would be less straightforward than our design. Dictating which transactions should run on which cores is harder and less efficient on the software side. Modifying the context-switching code in the current platform in order to perform fast context-switches, like STEPS does [74], would help to some extent. However, this still does not guarantee that threads are going to migrate exactly to the cores ADDICT wants them to migrate. The functions that set a thread's core affinity (e.g., `pthread_setaffinity_np` in the POSIX library) only work well provided that the destination core is idle. Otherwise, the OS scheduler schedules the thread to one of the underutilized cores automatically. To prevent such undesirable migrations and cache thrashing, ADDICT requires a more drastic design change on the software-side if a software-only design is more desirable. Deploying an execution model similar to staged databases [73, 75] and assigning stages to each database operation would allow us to pin each stage to a core, send requests to each stage's work queue, and give ADDICT more control over the core affinities.

Effect on database components

Under ADDICT, a transaction goes through the same database components as it does under traditional scheduling. ADDICT only involves multiple cores in the execution of a transaction. However, it does not change what a transaction executes. Therefore, ADDICT's migrations have no effect on ACID properties, concurrency control mechanisms, or the logging subsystem. In addition, since ADDICT does not batch more transactions than the number of available cores in the system, it does not change the data contention patterns.

For the cases outside the regular workload run, such as recovery or database population, ADDICT can either fallback to traditional scheduling or find new migration points for the specific operations or routines executed during such periods of execution.

8.3 Evaluation

The evaluation demonstrates:

- the stability of the migration points ADDICT picks across different numbers of transaction instances in Section 8.3.2,
- ADDICT's effect on instruction and data misses at different levels of the memory hierarchy in Section 8.3.3,
- ADDICT's impact on performance in Section 8.3.4,
- the effect of changing server load on ADDICT's performance in Section 8.3.5,
- ADDICT's behavior with simultaneous multithreading Section 8.3.6,
- ADDICT's effectiveness under deeper cache hierarchies in Section 8.3.7, and
- ADDICT's overhead in Section 8.3.8.

8.3.1 Setup and Methodology

Since ADDICT is a software-guided hardware mechanism (Section 8.2.2), the evaluation uses full timing simulation. We collect x86 execution traces from transactions using Pin [124]. We

Table 8.1: Simulated system parameters.

Processing Cores	16 OoO cores, 2.5GHz, 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ, BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Caches	32KB, 64B blocks, 8-way, 3-cycle load-to-use 32 MSHRs, MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, 16 banks, 16-cycle hit latency, 64 MSHRs
Interconnect	2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks, 8B Bus Width, Open Page Policy

Latencies	CAS(10), RCD(10), RAS(35), RC(47.5)
	WR(15), WTR(7.5), RTRS(1), CCD(4), CWD(9.5)

replay these traces on the Zesto x86 multicore architecture simulator [122], modeling the timing of all events. Table 8.1 details the hardware parameters in our simulation.

The traces are extracted from three standard transaction processing benchmarks [188] - TPC-B [190], TPC-C [191], and TPC-E [193] - while running their workload mix after a warm-up period on the Shore-MT storage manager [172]. Scaling factors are set big enough to have a 100GB dataset right after database population, and the buffer-pool is configured to keep the whole database in memory. To run the most scalable configuration for all the benchmarks, we enable all the logging [97] and locking [95] optimizations of Shore-MT. Since we simulate 16 cores, there are 16 worker threads executing transactions during the trace collection.

We compare *ADDICT* against three transaction scheduling mechanisms:

- *Baseline*, the traditional transaction scheduling, where each transaction starts and finishes its execution on one core provided that no context-switching occurs due to I/O, waiting for locks, etc.,
- *STREX* (Section 7.4), which time-multiplexes a batch of transactions on the same core to enable instruction reuse among the transactions in the batch, and
- *SLICC* (Section 7.3), which spreads the computation of transactions over several cores to localize common instructions to caches without any software hints.

We implement all four scheduling mechanisms on the Zesto simulator. Except for *Baseline*, all the mechanisms rely on batching same-type transactions. ADDICT picks a batch size that is equal to the number of available cores by default. Therefore, except for Section 8.3.5, the batch size is 16 in our experiments.

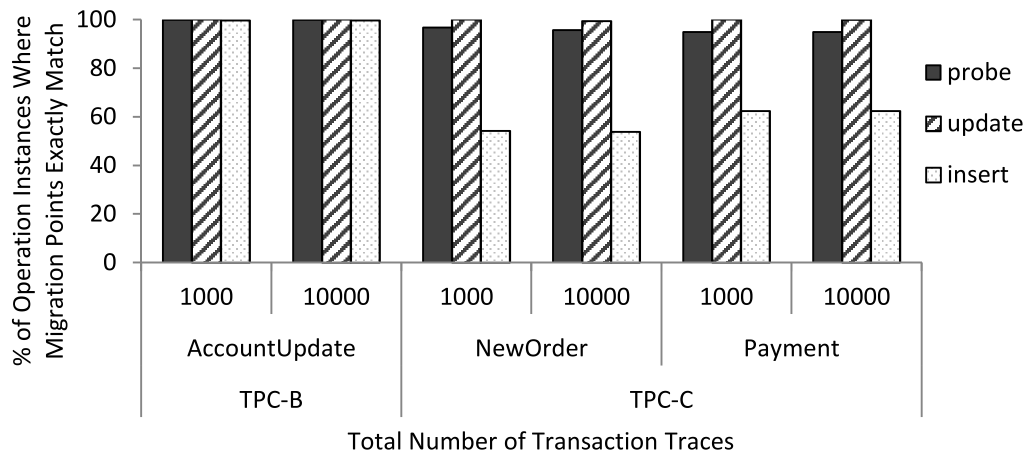


Figure 8.1: Percentage of database operation instances where the migration points picked by ADDICT have an exact match as we increase the total number of transaction instances.

We collect 11000 transaction traces for each workload. The initial step of ADDICT (Algorithm 4) uses the first 1000 traces (from 1 to 1000) to determine the migration points. Section 8.3.2 uses all the traces after the first 1000 (from 1001 to 11000), whereas the rest of the sections use the next batch of 1000 traces (from 1001 to 2000) while evaluating the different scheduling mechanisms.

8.3.2 Migration Points

As Section 8.2.1 describes, ADDICT picks the most common migration point sequences among all possible migration points for a transaction type. In our experimental evaluation, ADDICT determines the migration points based on a run with 1000 transaction traces (Section 8.3.1). This section investigates the *stability* of these migration points across all the instances of a transaction. It also shows how stability changes as we drastically increase the total number of transaction instances. A transaction instance has *stable* migration points if ADDICT's core migration selection algorithm, when ran directly on this transaction instance alone, picks migration points that match the migration points chosen by ADDICT during the initial profiling phase using the first 1000 transaction instances. For brevity, Figure 8.1 shows the results only for TPC-B's `AccountUpdate` and TPC-C's `NewOrder` and `Payment` transactions. The results are very similar for the other transaction types.

Except for the *insert tuple* operation in TPC-C, the migration points ADDICT determines for each database operation is stable in at least 90% of all the transactions. As Section 6.6.1 notes, *insert tuple* is the operation that has the most variety in its instruction stream across different instantiations. Therefore, it is expected that even the most frequent migration sequence for *insert tuple* does not satisfy almost half of the instances for some transaction types.

Furthermore, Figure 8.1 shows that the percentage of stability of the migration points stays the same when we move from 1000 to 10000 traces. This demonstrates that the 1000 transaction traces is sufficient enough to capture the differences across multiple instantiations of a transaction type for the workloads we evaluate. Therefore, the rest of the experiments in this section use 1000 transaction traces that is different from the 1000 transaction traces used for determining the migration points (see Section 8.3.1).

8.3.3 Instruction and Data Misses

This section quantifies ADDICT's impact on the instruction and data misses at the various cache hierarchy levels. More specifically, this section measures the number of instruction and data misses per 1000 instructions (MPKI) at the L1-I, L1-D, and L2 caches as we run the workloads with different scheduling techniques. Figure 8.2 reports the MPKI values for *ADDICT*, *STREX*, and *SLICC* normalized over the MPKI values from the *Baseline*.

L1-I

As Figure 8.2 illustrates, all scheduling mechanisms reduce the L1-I misses. However, *ADDICT* is more effective in reducing the instruction misses compared to the two hardware-only techniques. Specifically, ADDICT reduces instruction misses by 85% on average over *Baseline*, whereas the reduction is 20% and 60% with *STREX* and *SLICC*, respectively. *ADDICT* makes more precise scheduling decisions while chasing instruction locality for transactions because of the software-guidance.

TPC-B benefits the most from ADDICT since its transaction mix has only one transaction type. The migration points picked for TPC-B are suitable for all transactions. Therefore, after the initial set of transactions the instructions are spread over the various instruction caches and remain mostly resident for all other transactions. For TPC-C and TPC-E, however, if the new batch of transactions is of a different type than the ones in the previous batch, the non-overlapping instruction footprint must be first loaded in the instruction caches by the first few transactions.

L1-D

The L1-D MPKI results in Figure 8.2 show that the techniques that are based on computation spreading, *SLICC* and *ADDICT*, hinder data locality. When a transaction migrates from one core to another, it leaves its data behind. Therefore, *SLICC* and *ADDICT* increase data misses by 40% and 25% on average over the *Baseline*, respectively. *STREX*, on the other hand, leads to constructive data sharing for the few overlapped read-only data cache blocks (see Section 6.6.2).

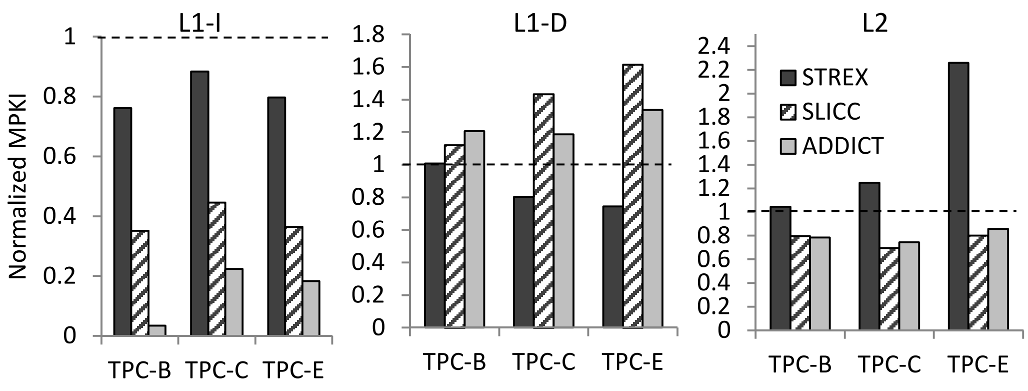


Figure 8.2: ADDICT's impact on instruction and data misses. Y-axes show the number of misses per 1000 instructions (MPKI) normalized over *Baseline* (=1 on Y-axis).

Section 6.4 shows that the data misses OLTP suffers from the most are the long-latency data misses from the last-level cache. These misses result in off-chip accesses that require a trip to main-memory. Modern out-of-order (OoO) processor cores are capable of hiding the latency of a few additional L1 data misses that end up being serviced by the on-chip memory hierarchy. Moreover, it is harder to overlap L1 instruction miss stalls compared to L1 data misses on a modern superscalar OoO processor, like the one we model (Section 8.3.1). Therefore, the slight increase in L1-D MPKI does not outweigh the benefits of reducing the L1-I MPKI as long as we avoid increasing the misses from the last-level cache. Section 8.3.4 supports this claim.

L2

ADDICT and *SLICC* both reduce the L2 MPKI by ~20%, whereas *STREX* increases it by 50% on average. Due to batching transactions on one core, *STREX* runs more transactions concurrently, which increases the stress on the requests to the last-level cache. However, *STREX* still improves the performance as Section 8.3.4 shows, emphasizing the importance of reducing the instruction misses once again. On the other hand, since all the techniques batch the same type of transactions, they access the same tables concurrently. Therefore, the reduction in L2 MPKI for *ADDICT* and *SLICC* stems from the constructive sharing of the read-only metadata information and higher-levels of the B-tree indexes for the same tables.

8.3.4 Performance Impact

This section measures how performance varies with ADDICT. It uses two performance metrics:

- total execution time to complete all traces and
- average time to complete a single transaction.

Figure 8.3 presents the results.

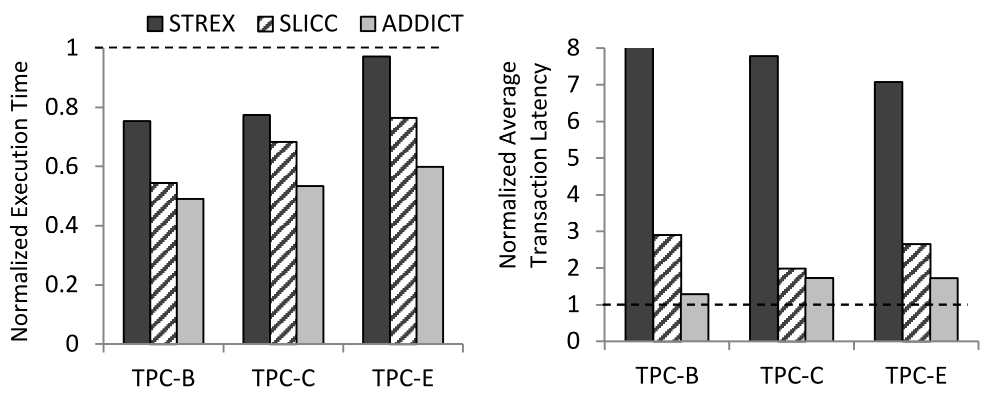


Figure 8.3: Impact of different scheduling techniques on performance; total execution cycles to complete 1000 transaction traces (left-hand side) and average transaction latency (right-hand side). Y-axes are normalized over *Baseline* (=1 on Y-axis).

Total execution cycles

Figure 8.3 shows that *ADDICT* reduces the total execution time by 45% over the *Baseline*. ADDICT is better than *STREX* and *SLICC*, which respectively improve performance by 17% and 35% on average over the *Baseline*. ADDICT manages to better utilize the instruction caches boosting instruction cache locality (see Figure 8.2).

Latency

While *STREX*, *SLICC*, and *ADDICT* reduce the total execution time and improve throughput, they all depend on transaction batching. As a result, they increase the average transaction latency in all the workloads. However, *ADDICT* exhibits the lowest transaction latency overhead compared to *STREX* and *SLICC*, increasing average transaction latency by 60% over the *Baseline*, whereas the latency increase is $7-8\times$ by *STREX* since it overloads cores with multiple transactions.

8.3.5 Effect of Changing Loads

By default, ADDICT picks a batch size that is equal to the number of available cores in the system. This section investigates ADDICT's behavior under different batch sizes, in parallel observing the effect of changing server load on ADDICT. Figure 8.4 reports how well ADDICT reduces the total execution cycles and L1-I misses as a function of batch size, i.e., the number of concurrent transactions in the system, from two (lightly-loaded system) to 32 (heavily-loaded system).

Figure 8.4 shows that while the reduction in L1-I MPKI remains the same, the total execution time improves for larger batch sizes. This is expected since the transactions from the previous batch might prefetch the instructions needed for the current batch. Therefore, ADDICT's

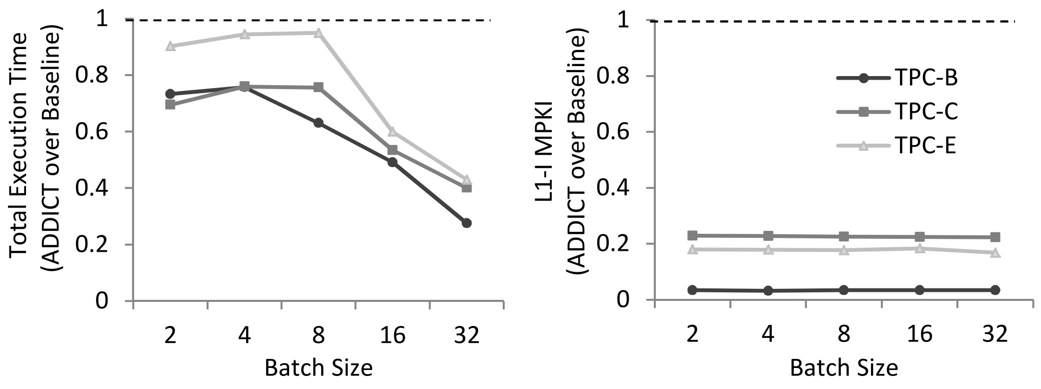


Figure 8.4: Impact of changing server load (or batch size) on ADDICT; total execution cycles to complete 1000 transaction traces (left-hand side) and instruction cache misses per 1000 instructions – L1-I MPKI – (right-hand side). Y-axes are normalized over *Baseline* (=1 on Y-axis).

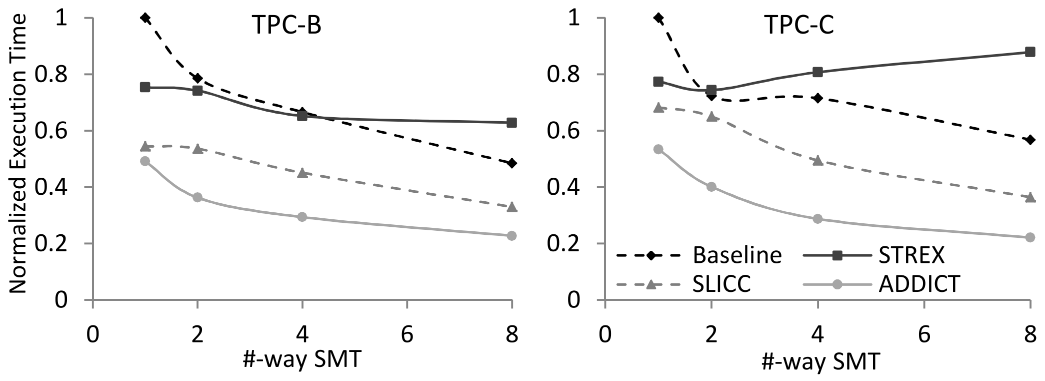


Figure 8.5: Behavior of different scheduling mechanisms with simultaneous multithreading (SMT). Y-axes plot the total execution cycles to complete 1000 transaction traces normalized over *Baseline* with no SMT (=1 on Y-axis) for the corresponding benchmarks.

effect on L1-I MPKI does not change as we increase the batch size. On the other hand, as we increase the batch size more transactions exploit the improved L1-I locality at a time. As a result, the reduction in the total execution time increases starting from a batch size of 8.

8.3.6 With Simultaneous Multithreading

So far, the experiments in Part III have simulated cores that support one hardware context (no simultaneous multithreading). This section investigates how the behavior of the evaluated scheduling techniques changes as we increase the number of hardware contexts available in a core. Figure 8.5 plots the total execution cycles for the TPC-B and TPC-C benchmarks as the different scheduling mechanisms run on hardware with 1-, 2-, 4-, and 8-way simultaneous multithreading (SMT). To keep the simulated hardware fully utilized, the number of transactions *Baseline*, *SLICC*, and *ADDICT* concurrently execute is equal to the total number of

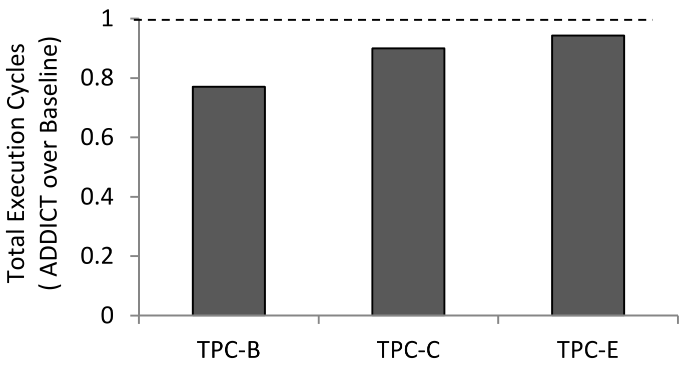


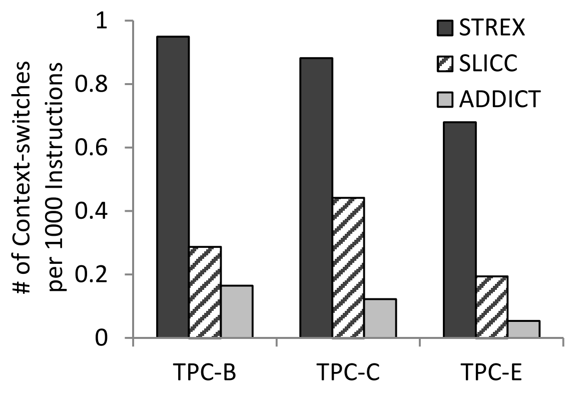
Figure 8.6: Impact of deeper cache hierarchies on ADDICT. Y-axis plots normalized ADDICT values over *Baseline* (=1 on Y-axis).

hardware contexts available at that time. For example, with 4-way SMT, all three mechanisms run 64 transactions at the same time provided that there are enough requests. In the case of *SLICC* and *ADDICT*, this means picking a batch size of 64 transactions. On the other hand, the machine load is kept the same in the case of *STREX*. *STREX* already executes 16 transactions concurrently on one core with no-SMT. Increasing this number with increasing SMT degree would drastically overload the machine. In order to keep the total load constant, *STREX* batches 16, 8, 4, and 2 transactions per hardware context when run with 1-, 2-, 4-, 8-way SMT, respectively.

SMT, in general, has a positive impact over all the scheduling mechanisms except for *STREX* when running TPC-C. An 8-way SMT almost halves the total execution cycles for *Baseline*, *SLICC*, and *ADDICT* since it helps in overlapping the various memory-related stall times. However, the reduction in the execution cycles is not proportional to the degree of SMT since running multiple transactions on the same core simultaneously also stresses the caches. For *STREX*, since we do not enforce all the hardware contexts in a core to pick the same transaction type while batching transactions, cores might be executing different transaction types simultaneously. Since TPC-B has only one transaction type, this does not cause performance degradation for *STREX*. However, in the case of the TPC-C benchmark, an 8-way SMT increases the total execution time by 14% even though *STREX* still outperforms the *Baseline* with no-SMT.

8.3.7 On Deeper Memory Hierarchies

This section considers a deeper memory hierarchy, which is representative of certain popular modern chip multiprocessors. More specifically, the experiments of this section introduce an additional 256KB per core L2 cache with 7 cycles of access latency. The previously considered shared L2 now appears as a shared L3 and the L1 caches remain the same. Figure 8.6 shows the total execution cycles for *ADDICT* normalized over the *Baseline*.



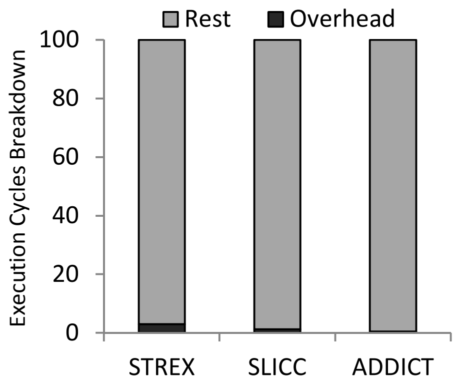


Figure 8.7: Number of context-switches/thread migrations per 1000 instructions (left-hand side) and execution cycles breakdown (right-hand side).

The reduction in L1-I MPKI and LLC(=L3) MPKI are similar to those for the shallower memory hierarchy (Figure 8.2). ADDICT remains effective at improving overall performance. As expected, the overall performance improvements are lower compared to the results with the shallower memory hierarchy since now the penalty for an L1-I cache miss is lower; the new L2 cache now handles some of the instruction cache misses. Considering that Shore-MT has an instruction footprint of 128KB-256KB, most L1-I misses are now served by the 256KB L2 cache, which effectively keeps the whole instruction footprint. However, the instruction footprint for commercial database management systems would be higher than the instruction footprint of Shore-MT.

8.3.8 Overhead

All three hardware mechanisms we evaluate have one major run-time overhead: they require additional context-switches either due to time-multiplexing transactions on a single core (*STREX*) or thread migrations across multiple cores (*SLICC* and *ADDICT*). Figure 8.7 compares the three mechanisms in terms of this overhead. More specifically, we first measure the number of times they context-switch transactions per 1000 instructions. Then, we report the contribution of this overhead to total execution cycles.

As the graph on the left-hand side of Figure 8.7 shows, *ADDICT* achieves its better performance through fewer migrations compared to both *STREX* and *SLICC*; 85% and 60%, respectively. Therefore, its run-time overhead due to context-switches is lower compared to the other two mechanisms. Nevertheless, the graph on the right-hand side of Figure 8.7 shows the execution cycles breakdown averaging the results for all the workload runs. It demonstrates that none of the mechanisms suffer due to the additional context-switches they incur. Even in the case of *STREX*, only 3% of the overall cycles go to these context-switches (labeled *Overhead*).

8.3.9 Summary

The evaluation shows that ADDICT is able to make effective decisions on the migration points for a variety of transaction types. Therefore, it significantly reduces the instruction misses since it optimizes transaction scheduling to maximize instruction locality. ADDICT encounters 85% fewer instruction misses for typical OLTP benchmarks compared to traditional scheduling. As a result, it reduces the total execution time by 45% under shallower cache hierarchies and 15% under deeper cache hierarchies. In addition, it incurs lower run-time cost and performs better than the current state-of-the-art hardware scheduling mechanisms for transactions (e.g., STREX and SLICC (Chapter 7)).

8.4 Related Work

There is a large body of work on reducing instruction stalls through improving instruction cache locality. Here we survey the ones that target OLTP workloads specifically.

Smart static or dynamic compilation techniques [159] can optimize the code layout to minimize the conflict misses. However, as Section 6.5.1 shows, even if we minimize the conflict misses with code optimization techniques, there is a significant amount of capacity misses that we have to reduce for more efficient OLTP execution.

On the other hand, instruction prefetching proposals designed for OLTP-like applications have emerged from simple stream buffers [161] to highly sophisticated stream predictors [53] that trade simplicity for accuracy. For example, PIF [53] requires ~40KB of extra storage per core. Therefore, modern commodity servers still prefer the low-cost next-line prefetcher, which sequentially fetches the memory addresses [173], for L1-I. Nevertheless, both the code optimization and instruction prefetching techniques are orthogonal to ADDICT and can be combined with it.

In addition to these techniques, there is a line of recent work that aims to improve instruction locality through exploiting the code commonality among concurrent transactions. These span proposals from batching transactions and time-multiplexing their execution on one core, STEPS [74] and STREX [15] (Section 7.4), to spreading the computation of transactions across multiple cores, computation spreading [30] and SLICC [14] (Section 7.3). Similarly to ADDICT, they all rely on the initial/leader thread to miss the instructions it needs as it would during traditional transaction execution, and the rest of the threads to reuse the instructions already brought into cache(s) by the initial thread. However, except for STEPS, they are all oblivious to software. They cannot prevent migrations or context switches during lock acquisitions or releases. In addition, even though their hardware costs are low, ADDICT minimizes the space and functionality required by the pure hardware techniques since it determines its migration decisions through software hints. On the other hand, STEPS is unable to exploit multicore hardware and requires cumbersome code modifications to be able to perform fast context switching at the software level. Finally, they all increase the average latency to execute a transaction even though they improve the overall throughput.

ADDICT aims to achieve the best of both SLICC and STEPS: spread the computation of a transaction over multiple cores to enable an ample cache capacity for instructions and get the insights for when and where to migrate transactions from the software-side to better localize the instructions in L1-I. In parallel, ADDICT reduces the migration costs and the transaction latency incurred by the two techniques.

8.5 Conclusions

L1 instruction miss stalls are the main cause of the hardware underutilization when running transaction processing applications on today's hardware. To overcome this problem, we design ADDICT. ADDICT assigns cores to the actions of each database operation in each transaction at a granularity that matches the size of the L1 instruction cache being used. It dynamically spreads the execution of transactions over multiple cores based on the core assignments to maximize the locality for instructions. Our evaluation shows that ADDICT's efforts in improving the instruction cache locality offer great potential in terms of performance and hardware utilization because of the high reuse frequency of instructions both within one and across different transactions and database operations.

We envision ADDICT as a task scheduler on emerging heterogeneous many-core processors where cores are specialized for various database functionalities. In such a setting, ADDICT can also guide developers while making decisions about the granularity at which each database operations should be specialized. Finally, in addition to OLTP workloads, ADDICT can benefit any application that suffers from instruction stalls and has concurrent requests executing a series of actions from a predefined set.

9 Future Directions and Concluding Remarks

As the scale of the data management applications and collected data continue to grow, it becomes even more crucial to get the best of the available and emerging computer architecture technology for effective, fast, and user-friendly data management. The insights and techniques described in this dissertation contribute toward this goal, but there is more to accomplish. Most proposals with similar goals still consider the hardware and software separately from each other; i.e., they propose changes for only one part of the whole stack. The long term solution within this ecosystem, however, depends on both the software and hardware sides becoming more aware of each other's capabilities and needs. ADDICT (Chapter 8) is definitely one of the initial steps in this direction. Further steps require interdisciplinary collaborations; especially involving people from the data management, computer architecture, and compiler communities. The data management systems should know which hardware they run on and its advantages and disadvantages for various data management tasks, hardware should frequently get hints from the data management systems to improve its predictions at the micro-architectural level and improve the resource utilization, and compilers should help in providing efficient communication primitives between the software and hardware.

9.1 Hardware Specialization

Hardware specialization is an area that has recently re-gained its popularity in the context of data management applications. Where Moore's Law prevented the rise of the database machines back in the 80s [25, 44], today the idea of building processors specialized for particular data management operations has become more appealing. There are several factors in this outcome, which mainly help in justifying the change at the hardware side from an economic point of view:

- halt of exploiting Moore's Law for free due to power concerns and emerging dark silicon [50, 68],
- increasing availability and usage of reconfigurable hardware [87, 133], and

- growing scale of the applications that run on the cloud [67].

As a result, there are several hardware/software co-design proposals both from industry and academia for data management applications. Widx [110] accelerates the index lookup routine in a hash-join algorithm, whereas Wu et al. [202] design an instruction set and build a collection of ASICs (application-specific integrated circuits) for data analytics operations in general. On the other hand, works like Bionic DBMS [93] and Catapult [156] aim to integrate field-programmable gate arrays (FPGAs) alongside commodity servers to accelerate some frequent routines in OLTP and web search, respectively, where these routines are offloaded to the corresponding FPGAs in the system. Finally, Oracle's RAPID [140] is a hardware-software co-design project, which targets building hardware for large-scale data analytics applications focusing on energy efficiency.

The above examples indicate that there is an increasing demand for building hardware for faster and more energy-efficient data management. Even though it might not be feasible to design a whole chip specifically for a data management application, building hardware components that accelerate frequent operations from an application might be, especially if the accelerator can benefit similar operations from other applications as well. With this discussion in mind, next we revisit the applicability of alternative scheduling mechanisms proposed in Part III.

9.2 Other Applications to Benefit from Alternative Scheduling

Part III proposed and evaluated three scheduling mechanisms (SLICC, STREX, and ADDICT) that target minimizing instruction cache misses while running transactions. It also described the hardware cost for these techniques concluding that the cost is feasible. However, to be able to justify the time and financial costs to build any hardware functionality, one must ensure a large scale of applications this particular functionality benefits. Part III already demonstrated the impact of the proposed mechanisms for OLTP applications, which already has an ample scale [63]. However, there might be other applications that can potentially benefit from such scheduling mechanisms.

In order to successfully adopt SLICC, STREX, and ADDICT to other applications, the target application should have the following properties in common with the transaction processing applications:

- suffering from first-level instruction misses due to a large instruction footprint and
- managing concurrent requests that exhibit high instruction overlap among each other since they are composed of operations from a predefined set of operations.

Looking at these properties, we can point to a few more large-scale applications that can exploit our scheduling mechanisms. For example, Ferdman et al. [55] show that some of the

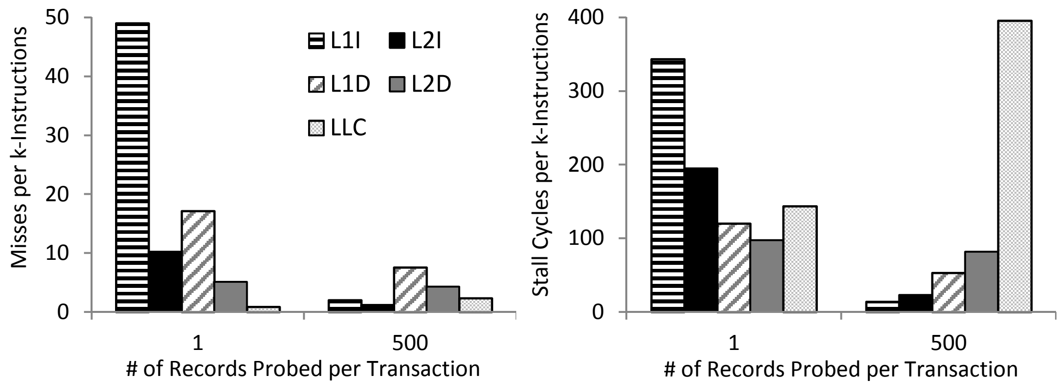


Figure 9.1: Misses per 1000 instructions for the in-memory OLTP system VoltDB and the estimated number of cycles spent on these misses.

typical cloud applications (media streaming, data serving, etc.) also observe high instruction miss rates and as any data management application their requests execute a series of predefined operations. Meisner et al. [126] introduce the concept of online data-intensive (OLDI) applications, which share the typical characteristics of an online transaction processing application except for the fact that they have to deal with large volumes of data in one request. Such applications span from social networking sites to web mail and they can also potentially take advantage of the alternative scheduling mechanisms described in Part III.

Furthermore, we have used the Shore-MT storage manager [172] throughout this thesis, which is not optimized for main-memory. Transaction processing systems optimized for main-memory usually eliminate the buffer pool component and also adopt more lightweight concurrency control schemes. Therefore, they have a smaller instruction footprint compared to a disk-based system. Figure 9.1 shows the results of an opportunity study for such systems. The left-hand side of Figure 9.1 has the number of misses per 1000 instructions from different levels of the memory hierarchy of an Intel Sandy Bridge server and the right-hand side of Figure 9.1 plots the estimated stall cycles for these misses. We use VoltDB [199] in these experiments and run a micro-benchmark that just probes (performs an index lookup) N records from a database of size 100GB. Where probing 500 records in a transaction stresses purely the storage manager part of the system, probing 1 record also stresses the other layers such as query parsing, communication with client threads, etc. [1] The difference between the two cases indicates that even though from the storage manager side VoltDB does not suffer from instruction-related stalls, it does so from other layers of the system. Therefore, scheduling mechanisms that target minimizing instruction misses can be applied to these other layers.

[1] Shore-MT does not have such additional layers. Shore-Kits is linked as a static library to the executable and does not contribute to the overall instruction footprint heavily.

9.3 Thesis Summary

In this thesis, we thoroughly investigated the dominant sources of hardware underutilization when running transaction processing applications. The analysis demonstrated that:

- At the level of the whole machine, conventional storage managers fail to exploit explicit/horizontal parallelism due to sheer number of unbounded critical sections a transaction has to go through;
- whereas within a core, the large instruction footprint of transactions leads to poor instruction locality causing OLTP applications to exhibit poor implicit/vertical parallelism.

Based on the findings above: We, first, designed a shared-everything transaction processing system that logically partitions the physical data accesses, where the partitions can be dynamically adjusted in a lightweight manner upon workload changes. We refer to this type of partitioning as physiological partitioning (PLP). To achieve its goal, PLP replaces the single-rooted index structure with a multi-rooted one, ensures each database record is only reachable through a single index root, and involves multiple worker threads in the execution of a transaction where each thread handles actions requiring data this thread is responsible from. Through regulating more predictable data accesses to both database records and pages, PLP eliminates the vast majority of the unbounded critical sections (due to locking and latching) from transaction execution within a shared-everything infrastructure.

Then, we designed three alternative scheduling methods (two hardware-only and one software-guided) for transactions that aim at maximizing instruction cache locality by exploiting the instruction commonality across concurrent transactions. SLICC adaptively spreads the execution of a transaction over multiple cores through thread migration and enables an ample L1 instruction cache capacity for a transaction, while STREX time-multiplexes a batch of transactions on the same core and does not depend on the aggregate cache capacity in the system. ADDICT, on the other hand, migrates transactions based on hints collected via a pre-profiling step from a sample workload run. Therefore, ADDICT requires fewer modifications at the hardware side in terms of additional functionality and data structures required to enable SLICC and STREX.

As a result of departing from traditional transaction scheduling, which considers each transaction as an indivisible unit of work, this thesis allows better data and instruction locality for transactions at the level it is needed. PLP achieves better thread-to-data access locality in a transaction processing system and exploits it to eliminate the pessimistic ways of protecting the shared data. SLICC, STREX, and ADDICT localize the instructions to first-level caches and exploit intra- and inter- instruction overlap for transactions. We advocate that the more parallel and heterogeneous the hardware gets with each generation, the more beneficial such finer-grained task scheduling mechanisms will become.

Bibliography

[1] Kiran J. Achyutuni, Edward Omiecinski, and Shamkant B. Navathe. Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases. In *SIGMOD*, pages 125–136, 1996. 4.6

[2] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM TODS*, 12(4):609–654, 1987. 3.5.2

[3] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999. 1.3, 2.3.2, 5.1, 5.2

[4] Anastasia Ailamaki, David J. DeWitt, and Mark D. Hill. Walking Four Machines By The Shore. In *CAECW*, 2001. 2.6

[5] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001. 2.3.2

[6] Anastasia Ailamaki, David J. DeWitt, and Mark D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *VLDB J.*, 11(3):198–215, 2002. 2.6

[7] Anastasia Ailamaki, Ryan Johnson, Ippokratis Pandis, and Pınar Tözün. Toward Scalable Transaction Processing: Evolution of Shore-MT. *PVLDB*, 6(11):1192–1193, 2013. 2.6

[8] Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Scaling Up Analytical Queries with Column-stores. In *DBTest*, pages 8:1–8:6, 2013. 1.3

[9] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, pages 1103–1114, 2014. 2.3.2

[10] AMD. Secure Virtual Machine Architecture Reference Manual, 2005. http://www.mimuw.edu.pl/ vincent/lecture6/sources/amd-pacifica-specification.pdf. 7.3.4

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS*, pages 483–485, 1967. 1.3

[12] Anon. et al. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, 1985. 2.4.1

[13] Islam Atta, Pınar Tözün, Anastasia Ailamaki, and Andreas Moshovos. Reducing OLTP Instruction Misses with Thread Migration. In *DaMoN*, pages 9–15, 2012. 1, 8.1

[14] Islam Atta, Pınar Tözün, Anastasia Ailamaki, and Andreas Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *MICRO*, pages 188–198, 2012. 5.1, 5.6, 1, 8.1, 8.4

[15] Islam Atta, Pınar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *ISCA*, pages 273–284, 2013. 1, 8.1, 8.4

[16] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA*, pages 3–14, 1998. 1.5, 5.2, 6.2

[17] Daniel Bartholomew. QEMU: A Multihost, Multitarget Emulator. *Linux J.*, 2006(145):3–, 2006. 7.5.1

[18] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. In *SIGFIDET*, pages 107–141, 1970. 3.3.1

[19] Peter Beaumont. The truth about Twitter, Facebook and the uprisings in the Arab world, 2011. http://www.theguardian.com/world/2011/feb/25/twitter-facebook-uprisings-arab-libya. 1.1

[20] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *MICRO*, pages 319–330, 2004. 3.7

[21] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent Cache-Oblivious B-trees. In *SPAA*, pages 228–237, 2005. 3.5.2

[22] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control—Theory and Algorithms. *ACM TODS*, 8(4):465–483, 1983. 3.5.2, 5.6

[23] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. 3.5.2

[24] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005. 2.3.2, 3.6

[25] Haran Boral and David J. DeWitt. Parallel Architectures for Database Systems. chapter Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pages 11–28. IEEE Press, Piscataway, NJ, USA, 1989. 9.1

[26] Bridget Botelho. Virtual machines per server: A viable metric for hardware selection?, 2008. http://itknowledgeexchange.techtarget.com/server-farm/virtual-machines-per-server-a-viable-metric-for-hardware-selection/. 7.1

[27] Eric A. Brewer. Towards Robust Distributed Systems (abstract). In *PODC*, pages 7–7, 2000. 3.2.3

[28] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In *SIGMOD*, pages 383–394, 1994. 2.6

[29] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*, pages 181–190, 2001. 3.5.2

[30] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-Fly. In *ASPLOS*, pages 283–292, 2006. 2.3.2, 5.1, 5.6, 7.1, 7.6, 8.4

[31] Shimin Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD*, pages 73–86, 2009. 3.5.1, 4.3.3

[32] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. In *SIGMOD*, pages 157–168, 2002. 2.3.2, 3.3.5, 3.5.2, 6.1, 8.1

[33] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Record*, 39:5–10, 2010. 5.1, 5.2

[34] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6): 377–387, 1970. 1.1

[35] E. F. Codd. Relational Database: A Practical Foundation for Productivity. *CACM*, 25(2): 109–117, 1982. 1.1

[36] Christopher B. Colohan, Anastasia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. Optimistic Intra-Transaction Parallelism on Chip Multiprocessors. In *VLDB*, pages 73–84, 2005. 5.1, 5.6

[37] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, pages 268–279, 1985. 2.3.2

[38] cputrack. cputrack. http://docs.oracle.com/cd/E19253-01/816-5165/6mbb0m9ct/index.html. 5.5.2

[39] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3:48–57, 2010. 2.3.1, 3.1, 3.2.3, 3.5.1, 4.1, 4.6

[40] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. Thread Cooperation in Multicore Architectures for Frequency Counting Over Multiple Data Streams. *PVLDB*, 2:217–228, 2009. 3.5.1

[41] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP*, pages 33–48, 2013. 2.3.1, 3.5.2

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 107–113, 2004. 7.5.1

[43] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *IEEE J. Solid-State Circuits*, pages 256–268, 1974. 1.2

[44] David J. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems. In *ISCA*, pages 182–189, 1978. 9.1

[45] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-i Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE TKDE*, 2(1): 44–62, 1990. 1.4, 2.3.1, 3.1, 3.2.3, 3.5.1, 4.1

[46] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013. 2.3.1, 2.3.2, 5.1

[47] Donko Donjerkovic, Yannis E. Ioannidis, and Raghu Ramakrishnan. Dynamic Histograms: Capturing Evolving Data Sets. In *ICDE*, page 86, 2000. 4.6

[48] DTrace. DTrace. http://dtrace.org. 3.4.1, 5.4.2

[49] EMC. The Digital Universe in 2020, 2012. http://www.emc.com/leadership/digital-universe/2012iview/executive-summary-a-universe-of.htm. 1.1

[50] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, pages 365–376, 2011. 1.2, 9.1

[51] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *ASPLOS*, pages 175–184, 2006. 5.5.1, 6.3

[52] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal Instruction Fetch Streaming. In *MICRO*, pages 1–10, 2008. 2.3.2, 7.6

[53] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive Instruction Fetch. In *MICRO*, pages 152–162, 2011. 6.1, 7.1, 7.5, 7.6, 8.4

[54] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012. 2.3.2, 5.1, 5.2, 5.6, 6.1, 6.2, 6.4, 7.5.1, 8.1

[55] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Quantifying the Mismatch between Emerging Scale-Out Applications and Modern Processors. *ACM TOCS*, 30(4):15:1–15:24, 2012. 1.5, 5.2, 9.2

[56] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. A Case for Specialized Processors for Scale-Out Workloads. *IEEE MICRO*, 34(3):31–42, 2014. 5.2

[57] M. R. Garey, D. S. Johnson, and Ravi Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976. 7.4

[58] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-Conscious Frequent Pattern Mining on Modern and Emerging Processors. *VLDB J.*, 16(1):77–96, 2007. 6.1, 8.1

[59] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast Incremental Maintenance of Approximate Histograms. *ACM TODS*, 27:261–298, 2002. 4.6

[60] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, pages 102–111, 1990. 2.3.2

[61] Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003. 3.5.2

[62] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. Foster B-trees. *ACM TODS*, 37(3): 17:1–17:29, 2012. 2.6

[63] Colleen Graham, Bhavish Sood, Hideaki Horiuchi, and Dan Sommer. Market Share: Database Management System Software, Worldwide, 2009. http://www.gartner.com/DisplayDocument?id=1044912. 1.1, 9.2

[64] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 1.1, 2.1, 1

[65] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4: 105–116, 2010. 2.3.2

[66] Fazal Hameed, Lars Bauer, and Jörg Henkel. Dynamic Cache Management in Multi-Core Architectures through Run-time Adaptation. In *DATE*, pages 485–490, 2012. 7.5.1

[67] James R. Hamilton. Internet Scale Storage. In *SIGMOD*, 2011. 9.1

[68] Nikos Hardavellas. The Rise and Fall of Dark silicon. *USENIX*, 37(2):7–17, 2012. 9.1

[69] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastasia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007. 1.3, 3.7, 5.2

[70] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *ISCA*, pages 184–195, 2009. 3.7

[71] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE MICRO*, 31(4):6–15, 2011. 1.2

[72] Stavros Harizopoulos and Anastasia Ailamaki. STEPS Towards Cache-Resident Transaction Processing. In *VLDB*, pages 660–671, 2004. 2.3.2, 5.6, 8.1

[73] Stavros Harizopoulos and Anastasia Ailamaki. StagedDB: Designing Database Servers for Modern Hardware. *IEEE DEBull*, 28(2):11–16, 2005. 8.2.2

[74] Stavros Harizopoulos and Anastasia Ailamaki. Improving Instruction Cache Performance in OLTP. *ACM TODS*, 31(3):887–920, 2006. 5.6, 7.1, 7.4.2, 7.6, 8.1, 8.2.2, 8.2.2, 8.4

[75] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastasia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, pages 383–394, 2005. 2.3.2, 8.2.2

[76] Stavros Harizopoulos, Daniel J. Abadi, Sam Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008. 1.4, 3.1, 3.5.1, 4.1, 6.2

[77] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, pages 132–141, 2007. 2.3.1, 3.1, 3.2.3

[78] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 1.2, 1, 2.3.2

[79] Melanie Herschel, Yannis Tzitzikas, Selcuk Candan, and Amelie Marian. Exploratory search: New name for an old hat?, 2011. http://wp.sigmod.org/?p=1183. 1.1

[80] Anthony J.G. Hey, Stewart Tansley, and Kristin M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. 1.1

[81] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41: 33–38, 2008. 3.2

[82] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE TOCS*, 38 (12):1612–1630, 1989. 6.5.1

[83] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM TOCS*, 16(2):170–205, 1998. 8.2.1

[84] IBM. WebSphere Software. http://www-01.ibm.com/software/websphere. 7.1

[85] IBM. Shrinking 3900 Distributed Servers to 30 Linux Mainframes, 2007. http://www-03.ibm.com/press/us/en/pressrelease/21945.wss. 7.1

[86] IBM. IBM Breaks Double Digit Performance Barrier With 10 Million Transactions Per Minute, 2010. http://www-03.ibm.com/press/us/en/pressrelease/32328.wss. 1.1, 2.4.2, 5.1

[87] IBM Netezza. IBM Netezza Data Warehouse Appliances. http://www-01.ibm.com/software/data/netezza. 9.1

[88] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2012. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. 1.3, 6.3

[89] Intel. Intel VTune Amplifier XE Performance Profiler, 2013. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/. 5.5.1

[90] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *CACM*, 57, 2014. 1.1

[91] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA*, pages 60–71, 2010. 7.1

[92] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. B-Tree Concurrency Control and Recovery in Page-Server Database Systems. *ACM TODS*, 31:82–132, 2006. 3.3.5, 3.5.2

[93] Ryan Johnson and Ippokratis Pandis. The bionic DBMS is coming, but what will it look like? In *CIDR*, 2013. 9.1

[94] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines. In *DaMoN*, pages 35–40, 2008. 1.4

[95] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP Scalability Using Speculative Lock Inheritance. *PVLDB*, 2(1):479–489, 2009. 1.4, 2.6, 3.2.2, 3.4.1, 3.5.1, 5.1, 5.2, 5.3.3, 5.4.2, 6.2, 6.3, 8.3.1

[96] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, pages 24–35, 2009. 1.3, 1, 2.1, 2.3.1, 2.6, 3.1, 3.2, 3.4.1, 3.5.1, 3.7, 4.1, 4.5.1

[97] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A Scalable Approach to Logging. *PVLDB*, 3:681–692, 2010. 1.4, 2.6, 3.2, 3.2.2, 3.4.1, 3.5.1, 5.1, 5.2, 5.3.3, 5.4.2, 6.2, 6.3, 8.3.1

[98] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.*, 21:239–263, 2012. 3.2.3, 3.5.1, 5.3.3

[99] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Eliminating Unscalable Communication in Transaction Processing. *VLDB J.*, 23(1):1–23, 2014. 1.3, 1.4, 2.6, 3.7

[100] Evan Jones, Daniel J. Abadi, and Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD*, pages 603–614, 2010. 3.5.1

[101] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ISCA*, pages 364–373, 1990. 7.6

[102] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A Scalable Lock Manager for Multicores. In *SIGMOD*, pages 73–84, 2013. 2.6

[103] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008. 3.1

[104] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC*, pages 119–128, 2008. 5.1, 5.2

[105] Cansu Kaynak, Boris Grot, and Babak Falsafi. SHIFT: Shared History Instruction Fetch for Lean-Core Server Processors. In *MICRO*, pages 272–283, 2013. 2.3.2, 6.1, 7.6

[106] Kimberly Keeton, David A. Patterson, Yong Qian He, Raphael C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*, pages 15–26, 1998. 1.3, 1.5, 2.3.2, 5.1, 5.2, 6.2

[107] Alfons Kemper and Thomas Neumann. HyPer – A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011. 2.3.1, 3.1, 3.5.1, 4.1, 5.6

[108] Hideaki Kimura, Goetz Graefe, and Harumi Kuno. Efficient Locking Techniques for Databases on Modern Hardware. *ADMS*, 2012. 3.5.1

[109] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014. 2.3.2

[110] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*, pages 468–479, 2013. 9.1

[111] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc processor. *IEEE MICRO*, 25(2), 2005. 1.1

[112] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010. 2.3.2

[113] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM TODS*, 6(2):213–226, 1981. 3.5.2

[114] Tirthankar Lahiri, Vinay Srihari, Wilson Chan, N. MacNaughton, and Sashikanth Chandrasekaran. Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. In *VLDB*, pages 683–686, 2001. 2.3.1, 3.5.1

[115] Willis Lang, Jignesh M. Patel, and Srinath Shankar. Wimpy Node Clusters: What About Non-Wimpy Workloads? In *DaMoN*, pages 47–55, 2010. 5.2

[116] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4), 2011. 1.3, 2.3.1, 3.5.2, 5.1, 5.6

[117] Juchang Lee, Yong Sik Kwon, F. Farber, M. Muehle, Chulwon Lee, C. Bensberg, Joo Yeon Lee, AH. Lee, and W. Lehner. SAP HANA Distributed In-Memory Database System: Transaction, Session, and Metadata Management. In *ICDE*, pages 1165–1173, 2013. 2.3.1

[118] Mong-Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards Self-Tuning Data Placement in Parallel Database Systems. In *SIGMOD*, pages 225–236, 2000. 4.6

[119] Justin Levandoski, David Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, pages 302–313, 2013. 2.3.1, 3.5.2

[120] Sam Lightstone, Maheswaran Surendra, Yixin Diao, Sujay S. Parekh, Joseph L. Hellerstein, Kevin Rose, Adam J. Storm, and Christian Garcia-Arellano. Control Theory: a Foundational Technique for Self Managing Databases. In *ICDE Workshops*, pages 395–403, 2007. 4.4.3

[121] Song Liu, Brian Leung, Alexander Neckar, Seda Ogrenci Memik, Gokhan Memik, and Nikos Hardavellas. Hardware/Software Techniques for DRAM Thermal Management. In *HPCA*, pages 515–525, 2011. 7.5.1

[122] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *ISPASS*, pages 53–64, 2009. 7.5.1, 8.3.1

[123] Dave Lomet, Rick Anderson, T. K. Rengarajan, and Peter Spiro. How the Rdb/VMS Data Sharing System Became Fast. Technical Report CRL-92-4, DEC, 1992. 3.2.3

[124] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005. 6.1, 6.3, 7.5.1, 8.3.1

[125] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*, pages 183–196, 2012. 2.3.2, 3.5.2

[126] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, pages 319–330, 2011. 9.2

[127] MemSQL. MemSQL. http://www.memsql.com/. 2.3.1

[128] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*, pages 392–405, 1990. 3.3.5, 5.4.2

[129] C. Mohan and Frank Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *SIGMOD*, pages 371–380, 1992. 3.3.5, 3.5.2, 5.4.2, 6.5.3

[130] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *SPAA*, pages 253–262, 2005. 3.2.1

[131] Anirban Mondal, Masaru Kitsuregawa, Beng Chin Ooi, and Kian-Lee Tan. R-tree-based Data Migration and Self-Tuning Strategies in Shared-Nothing Spatial Databases. In *GIS*, pages 28–33, 2001. 4.6

[132] Gordon Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38 (6), 1965. 1.2

[133] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *PVLDB*, 2 (1):910–921, 2009. 9.1

[134] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. The LHAM log-structured history data access method. *VLDB J.*, 8:199–221, 2000. 3.5.2

[135] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In *OSDI*, pages 511–524, 2014. 2.3.1

[136] Thomas Neumann and Viktor Leis. Compiling Database Queries into Machine Code. *IEEE DEBull*, 37(1):3–11, 2014. 2.3.2, 6.1

[137] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark (TATP), 2009. http://tatpbenchmark.sourceforge.net/. 1, 2.5, 2.6, 3.2.2, 3.2.2, 4.2

[138] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *ASPLOS*, pages 2–11, 1996. 1.2

[139] Pat O'Neil, Betty O'Neil, and Xuedong Chen. Star Schema Benchmark (SSB), 2009. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF. 2.6

[140] Oracle. RAPID. https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:14. 9.1

[141] Oracle. Oracle WebLogic Suite 11g White Paper, 2009. http://www.oracle.com/partners/en/047014.pdf. 7.1

[142] Oracle. SPARC Supercluster with 27 SPARC T3-4 Servers Demonstrates World Record Performance on TPC-C Benchmark, 2010. http://www.oracle.com/us/solutions/performance-scalability/t3-4-tpc-c-12210-bmark-190934.html. 1.1, 2.4.2, 5.1

[143] Oracle. Oracle Real Application Clusters White Paper, 2013. http://www.oracle.com/technetwork/database/options/clustering/rac-wp-12c-1896129.pdf?ssSourceSiteId=ocomen. 2.3.1, 3.5.1

[144] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *DaMoN*, pages 8:1–8:7, 2014. 2.6

[145] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-Oriented Transaction Execution. *PVLDB*, 3(1):928–939, 2010. 1.4, 2.3.1, 2.3.1, 2.6, 3.1, 3.1.2, 3.2.2, 3.3.1, 3.4.1, 3.5.1, 4.1, 5.1, 5.2, 5.4.2, 5.6, 6.2

[146] Ippokratis Pandis, Pınar Tözün, Miguel Branco, Dimitris Karampinas, Danica Porobic, Ryan Johnson, and Anastasia Ailamaki. A Data-Oriented Transaction Execution Engine and Supporting Tools. In *SIGMOD*, pages 1237–1240, 2011. 2.3.1, 3.6, 3.6

[147] Ippokratis Pandis, Pınar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page Latch-Free Shared-Everything OLTP. *PVLDB*, 4(10):610–621, 2011. 1.4, 2.6, 1, 1, 5.1, 5.2, 6.2

[148] PARSA. Data Analytics Benchmark with Hadoop MapReduce Framework, 2012. http://parsa.epfl.ch/cloudsuite/analytics.html. 7.5.1

[149] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):85–96, 2011. 2.3.1, 3.1

[150] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012. 2.3.1, 3.1, 3.2.3, 4.6

[151] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *ICS*, pages 189–198, 2002. 7.3.2

[152] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *PVLDB*, 7(2):121–132, 2013. 2.6

[153] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012. 1.4, 3.1, 3.2.3

[154] Danica Porobic, Erietta Liarou, Pınar Tözün, and Anastasia Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *ICDE*, pages 688–699, 2014. 1.4, 2.3.1, 2.6, 3.5.2, 3.6, 2, 6.1, 8.1

[155] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *PVLDB*, 6(9):637–648, 2013. 1.3, 2.6

[156] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*, pages 13–24, 2014. 9.1

[157] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable Workload-Aware Data Placement for Transactional Workloads. In *EDBT*, pages 430–441, 2013. 4.6

[158] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *ISCA*, pages 381–391, 2007. 7.1

[159] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA*, pages 155–164, 2001. 2.3.2, 6.1, 7.1, 8.1, 8.4

[160] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. In *ISCA*, pages 302–313, 2009. 7.3.4

[161] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *ASPLOS*, pages 307–318, 1998. 1.5, 2.3.2, 5.1, 5.2, 5.6, 6.2, 7.6, 8.4

[162] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, pages 78–89, 1999. 3.3.5, 3.5.2

[163] Jun Rao and Kenneth A. Ross. Making B+-trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000. 2.3.2, 3.3.5, 3.5.2

[164] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating Physical Database Design in a Parallel Database. In *SIGMOD*, pages 558–569, 2002. 4.1, 4.6

[165] Elaine Ellis Reddy. Occupy Gezi: How Twitter Facilitated a Social Movement in Turkey, 2014. http://blog.gnip.com/occupy-gezi-twitter/. 1.1

[166] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *EuroSys*, pages 17–30, 2011. 1.3

[167] Daniel Sanchez and Christos Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *ASPLOS*, pages 311–322, 2010. 7.4.2

[168] Caetano Sauer, Goetz Graefe, and Theo Härder. An empirical analysis of database recovery costs. In *RDSS*, pages 1–6, 2014. 2.6

[169] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Rafiq Taha, and Umar Farooq Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *PVLDB*, 7(12):1035–1046, 2014. 2.3.1

[170] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+Trees on Many-Core Processors. *PVLDB*, 4(11):795–806, 2011. 3.5.2, 7.4.2

[171] ShoreMT. Shore-MT and Shore-Kits Code Repositories, . https://bitbucket.org/shoremt. 2.6

[172] ShoreMT. Shore-MT Official Website, . http://diaswww.epfl.ch/shore-mt/. 1.3, 2.6, 3.4.1, 4.5.1, 5.1, 6.1, 6.3, 7.5.1, 8.3.1, 9.2

[173] Alan Jay Smith. Sequentiality and Prefetching in Database Systems. *ACM TODS*, 3, 1978. 7.5, 7.5.1, 8.4

[174] Stephen Somogyi, Thomas F. Wenisch, Nikos Hardavellas, Jangwoo Kim, Anastasia Ailamaki, and Babak Falsafi. Memory Coherence Activity Prediction in Commercial Workloads. In *WMPI*, pages 37–45, 2004. 1.4, 3.1, 3.7

[175] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-Temporal Memory Streaming. In *ISCA*, pages 69–80, 2009. 2.3.2, 3.7, 8.1

[176] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org/. 1.5

[177] Robert Stets, Kourosh Gharachorloo, and Luiz André Barroso. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*, pages 37–48, 2002. 1.3, 1.5, 2.3.2, 5.1, 5.2, 6.1, 8.1

[178] Michael Stonebraker. The Case for Shared Nothing. *IEEE DEBull*, 9:4–9, 1986. 1.4, 2.3.1, 3.1, 3.2, 3.2.3, 3.5.1

[179] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007. 2.3.1, 3.1, 3.1.1, 3.2.3, 3.3.2, 3.5.1, 4.1, 5.6, 6.2

[180] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column Oriented DBMS. In *VLDB*, pages 553–564, 2005. 2.3.2, 3.6

[181] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM CSUR*, 30:70–119, 1998. 3.5.2

[182] TokuDB. TokuDB. http://www.tokutek.com/. 2.3.1

[183] Xin Tong, Jack Luo, and Andreas Moshovos. QTrace: An Interface for Customizable Full System Instrumentation. In *ISPASS*, pages 132–133, 2013. 7.5.1

[184] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. OLTP in Wonderland – Where do cache misses come from in major OLTP components? In *DaMoN*, pages 8:1–8:6, 2013. 1

[185] Pınar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. Scalable and Dynamically Balanced Shared-Everything OLTP with Physiological Partitioning. *VLDB J.*, 22(2):151–175, 2013. 1, 1

[186] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored. In *EDBT*, pages 17–28, 2013. 1, 2.6, 2, 8.1

[187] Pınar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. ADDICT: Advanced Instruction Chasing for Transactions. *PVLDB*, 7(14), 2014. 5.6, 1, 1

[188] TPC. Transaction Processing Performance Council. http://www.tpc.org. 1.5, 1.6, 1.7, 1, 2.4, 2.3, 6.1, 6.3, 8.3.1

[189] TPC-A. TPC Benchmark A Standard Specification, 1994. http://www.tpc.org/tpca. 2.4.1

[190] TPC-B. TPC Benchmark B Standard Specification, 1994. http://www.tpc.org/tpcb. 2.4.1, 2.6, 3.2.2, 6.3, 8.3.1

[191] TPC-C. TPC Benchmark C Standard Specification, 2010. http://www.tpc.org/tpcc. 2.4.2, 2.6, 3.2.2, 5.1, 6.3, 7.1, 7.5.1, 8.3.1

[192] TPC-D. TPC benchmark D standard specification, 1998. http://www.tpc.org/tpcd. 5.2

[193] TPC-E. TPC Benchmark E Standard Specification, 2014. http://www.tpc.org/tpce. 1.7, 2.4.3, 2.6, 5.1, 6.3, 7.1, 7.5.1, 8.3.1

[194] TPC-E. TPC-E Top Ten Performance Results, 2014. http://www.tpc.org/tpce/results/tpce_perf_results.asp. 5.1

[195] TPC-H. TPC Benchmark H Standard Specification, 2014. http://www.tpc.org/tpch. 2.6, 5.1

[196] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, pages 18–32, 2013. 2.3.1, 3.5.2

[197] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer*, pages 48–56, 2005. 7.3.4

[198] VMWare Inc. Enabling End-to-End Virtualization Solutions for Mid-market and Enterprise Customers. http://www.vmware.com. 7.1

[199] VoltDB. VoltDB. http://www.voltdb.com. 2.3.1, 3.1, 6.2, 9.2

[200] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal Streams in Commercial Server Applications. In *IISWC*, pages 99–108, 2008. 6.2

[201] Eugene Wu and Samuel Madden. Partitioning Techniques for Fine-Grained indexing. In *ICDE*, pages 1127 –1138, 2011. 4.6

[202] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *ASPLOS*, pages 255–268, 2014. 9.1

[203] Doe Hyun Yoon, Min Kyu Jeong, Michael Sullivan, and Mattan Erez. The Dynamic Granularity Memory System. In *ISCA*, pages 548–559, 2012. 7.5.1

[204] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. Technical report, MIT, CMU, Department of Computer Science, 2014. 2.3.1, 3.5.2

[205] Paul Zubulake and Sang Lee. *The High Frequency Game Changer: How Automated Trading Strategies Have Revolutionized the Markets*. Wiley Trading. John Wiley&Sons, 2011. 1.1

Pınar Tözün

Ph.D. in Computer Science
École Polytechnique Fédérale de Lausanne
CH–1015, Lausanne, Switzerland
pinar.tozun@epfl.ch
http://www.pinartozun.com

RESEARCH INTERESTS

Efficient data management on modern hardware

ACADEMIC BACKGROUND

Ph.D. in Computer Science, 2009 – 2014 **École Polytechnique Fédérale de Lausanne**

Thesis: Transactions Chasing Scalability and Instruction Locality on Multicores
Advisor: Prof. Anastasia Ailamaki

Diploma in Computer Engineering, 2005 – 2009 **Koç University, Istanbul, Turkey**

Department Rank: 1st

HONORS & AWARDS

- SAP Student Travel Grant, **EDBT/ICDT** 2013
- Best Demonstration Award, ACM **SIGMOD** Conference, 2011
- Google Eurosys Conference Grant and Travel Award for Women in CS, **Eurosys** 2010
- EPFL I&C School Fellowship Student, 2009-2010
- Vehbi Koç Honorary Award, for having SPA over 3.5, in Semester Fall 2005, Spring 2006, Fall 2006, Spring 2007, Fall 2007, Spring 2008
- Visiting Student, Massachusetts Institute of Technology, Spring 2006, in recognition of outstanding performance in Math 106 and Physics 101 classes in Koç University
- Full Scholarship from Vehbi Koç Foundation, founder of the Koç University, and Turkish Government during Bachelor's education for being among the top 200 students in the University Entrance Exam

PUBLICATIONS

- A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, I. Psaroudakis. "How to Stop Underutilization and Love Multicores." In Proceedings of the International Conference on Data Engineering (**ICDE**), 2015. **Tutorial**
- P. Tözün, I. Atta, A. Ailamaki, A. Moshovos. "ADDICT: Advanced Instruction Chasing for Transactions." In Proceedings of the Very Large Databases Endowment (**PVLDB**), Vol. 7(14), 2014.
- I. Psaroudakis, T. Kissinger, D. Porobic, T. Ilsche, E. Liarou, P. Tözün, A. Ailamaki, W. Lehner. "Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries." In Proceedings of the International Workshop on Data Management on New Hardware (**DaMoN**), 2014.
- A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, I. Psaroudakis. "How to Stop Underutilization and Love Multicores." In Proceedings of the ACM **SIGMOD** International Conference on Management of Data, 2014. **Tutorial**
- D. Porobic, E. Liarou, P. Tözün, A. Ailamaki. "ATraPos: Adaptive Transaction Processing on Hardware Islands." In Proceedings of the International Conference on Data Engineering (**ICDE**), 2014.
- A. Ailamaki, R. Johnson, I. Pandis, P. Tözün. "Toward Scalable Transaction Processing – Evolution of Shore-MT." In Proceedings of the Very Large Databases Endowment (**PVLDB**), 2013. **Tutorial**
- P. Tözün, B. Gold, A. Ailamaki. "OLTP in Wonderland -- Where do cache misses come from in major OLTP components?" In Proceedings of the International Workshop on Data Management on New Hardware (**DaMoN**), 2013.
- I. Atta, P. Tözün, X. Tong, A. Ailamaki, A. Moshovos. "STREX: Boosting Instruction Cache Reuse in

OLTP Workloads through Stratified Transaction Execution." In Proceedings of the International Symposium on Computer Architecture (**ISCA**), 2013.

- P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, A. Ailamaki. "From A to E: Analyzing TPC's OLTP Benchmarks - The obsolete, the ubiquitous, the unexplored." In Proceedings of the International Conference on Extending Database Technology (**EDBT**), 2013.
- P. Tözün, I. Pandis, R. Johnson, A. Ailamaki. "Scalable and Dynamically Balanced Shared-Everything OLTP with Physiological Partitioning." In the International Journal of Very Large Databases (**VLDBJ**), 2013.
- I. Atta, P. Tözün, A. Ailamaki, A. Moshovos. "SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads." In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (**MICRO**), 2012.
- D. Porobic, I. Pandis, M. Branco, P. Tözün, A. Ailamaki. "OLTP on Hardware Islands." In Proceedings of the Very Large Databases Endowment (**PVLDB**), Vol. 5(12), 2012.
- I. Atta, P. Tözün, A. Ailamaki, A. Moshovos. "Reducing OLTP Instruction Misses with Thread Migration." In Proceedings of the International Workshop on Data Management on New Hardware (**DaMoN**), 2012.
- I. Pandis, P. Tözün, R. Johnson, A. Ailamaki. "PLP: Page Latch-free Shared-everything OLTP." In Proceedings of the Very Large Databases Endowment (**PVLDB**), Vol. 4 (10), 2011.
- I. Pandis, P. Tözün, M. Branco, D. Karampinas, D. Porobic, R. Johnson, A. Ailamaki. "A Data-oriented Transaction Execution Engine and Supporting Tools." In Proceedings of the ACM **SIGMOD** International Conference on Management of Data, 2011. **Best Demonstration Award**.
- H. Jula, P. Tözün, G. Candea. "Communix: A Collaborative Deadlock Immunity Framework." In Proceedings of the International Conference on Dependable Systems and Networks (**DSN**), 2011.

WORKING EXPERIENCE

7/2010 – 11/2014 **École Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**

Member of the Data-Intensive Applications and Systems (DIAS) laboratory led by Prof. Anastasia Ailamaki working on exploiting modern hardware resources more effectively for data management systems.

5/2012 – 8/2012 **Oracle Labs** **Redwood Shores, CA**

Summer Intern at Oracle Labs mentored by Brian Gold and Eric Sedlar working on memory characterization of traditional transaction processing systems.

2/2010 – 7/2010 **École Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**

Member of the Distributed Programming Laboratory (LPD) led by Prof. Rachid Guerraoui working on conflict avoidance in software transactional memory.

9/2009 – 2/2010 **École Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**

Member of Dependable Systems Laboratory (DSLAB) led by Prof. George Candea working on distributed deadlock immunity.

6/2008 – 8/2008 **University of Twente** **Enschede, Netherlands**

Summer Intern at the Software Engineering Group led by Prof. Mehmet Akşit, mentored by Dr. Gürcan Güleşir and Prof. Lodewijk Bergmans working on evolvable behavior specifications during the development of complex software systems.

7/2007 – 8/2007 **Veripark** **Istanbul, Turkey**

Summer Intern developing applications and web interfaces using JSP and .NET.

CONFERENCE PRESENTATIONS AND INVITED TALKS

- "How to Stop Underutilization and Love Multicores." At ACM **SIGMOD** International Conference
- "HPTS Next-Generation Panel." At High Performance Transaction Systems (**HPTS**) Workshop, 2013.
- "Toward Scalable Transaction Processing." At the International Conference on Very Large Databases (**VLDB**), 2013.
- "OLTP in Wonderland." At the International Workshop on Data Management on New Hardware (**DaMoN**), 2013.
- "Transactions Chasing Instruction Locality on Multicores" At Technical University of Dortmund (**TU Dortmund**), 2013.
- "From A to E: Analyzing TPC's OLTP Benchmarks." At the International Conference on Extending Database Technology (**EDBT**), 2013.
- "A Case for Thread Migration." At the Biennial Conference on Innovative Data Systems Research (**CIDR**), 2013.
- "Transactions on Modern Hardware." At **Koç University**, 2013.
- "Reducing OLTP Instruction Misses with Thread Migration." At the International Workshop on Data Management on New Hardware (**DaMoN**), 2012.
- "Critical Sections through the Looking Glass." At High Performance Transaction Systems (**HPTS**) Workshop, 2013.
- "PLP: Page Latch-free Shared-everything OLTP." At the International Conference on Very Large Databases (**VLDB**), 2011.

PROFESSIONAL ACTIVITIES

Program committees:	TinyToCS 2013, IMDM 2014, CIKM 2014, SIGMOD 2015 (Demo)
Reviewer:	The VLDB Journal (VLDBJ)

TEACHING

École Polytechnique Fédérale de Lausanne:

Spring 2013	TA in undergraduate course *Introduction to Database Systems* Instructor: Prof. Anastasia Ailamaki (anastasia.ailamaki@epfl.ch).
Fall 2011	TA in undergraduate course *Introduction to Programming* Instructor: Prof. Ronan Boulic (ronan.boulic@epfl.ch).
Spring 2011	TA in graduate course *Advanced Databases* Instructor: Prof. Christoph Koch (christoph.koch@epfl.ch)
Fall 2010	TA in undergraduate course *Introduction to Programming* Instructor: Prof. David Lindelöf (david.lindelof@epfl.ch)

Koç University:

Spring 2008	TA in undergraduate course *Introduction to Programming* Instructor: Prof. Serdar Taşıran (stasiran@ku.edu.tr)
Spring 2007	TA in undergraduate course *Structure and Interpretation of Computer Programs* Instructor: Prof. Deniz Yüret (dyuret@ku.edu.tr)

Voluntary Teaching:

Spring 2008	One of the organizers of the weekly film interpretation and discussion sessions with female prisoners in Bakırköy Women Penitentiary in Istanbul, Turkey.

Spring & Fall 2006, 2007	Organizer and one of the instructors of the mathematics course in Koç University Volunteers – Children Who Can Think Project, where sixth grade students from neighborhood elementary schools are brought to Koç University and attend weekly lessons and activities held by the university students.

LANGUAGES

Turkish	Native
English	Fluent
German	Intermediate
French	Beginner

PERSONAL

- Member of the mentorship program in Koç University (Fall 2007 & 2008), which helps the new students to adapt the undergraduate life and education.
- President (2007-2008), Secretary (2006-2007), Web Designer (2008-2009), and Member (2005-2009) of the Koç University Cinema Club.
- Participant in the Short Film Workshop given by Selim Evci, founder of IFSAK and organizer of the AKBANK Short Film Festival in Turkey (2006).
- Player in TED Zonguldak Basketball Team (1997 – 2002).

REFERENCES

Prof. Anastasia Ailamaki	École Polytechnique Fédérale de Lausanne (**EPFL**), Lausanne, Switzerland Title: Professor Email: anastasia.ailamaki@epfl.ch
Prof. Andreas Moshovos	**University of Toronto**, Toronto, Canada Title: Professor Email: moshovos@eecg.utoronto.ca
Dr. Philip A. Bernstein	**Microsoft Research**, Redmond, WA Title: Distinguished Scientist Email: philbe@microsoft.com
Shel Finkelstein	**Independent**, Los Altos, CA Email: shelfin68@yahoo.com
Eric Sedlar	**Oracle Labs**, Redwood Shores, CA Title: Vice President and Technical Director Email: eric.sedlar@oracle.com
Prof. Ryan Johnson	**University of Toronto**, Toronto, Canada Title: Assistant Professor Email: ryan.johnson@cs.utoronto.ca
Prof. Christoph Koch	École Polytechnique Fédérale de Lausanne (**EPFL**), Lausanne, Switzerland Title: Professor Email: christoph.koch@epfl.ch

Last update: October 2014.