EPFL

Thèse n° 10 363

Micro-architectural Analysis of Database Workloads

Présentée le 11 mai 2021

Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Utku SIRIN

Acceptée sur proposition du jury

Prof. K. Aberer, président du jury
Prof. A. Ailamaki, directrice de thèse
Dr S. Chaudhuri, rapporteur
Prof. S. Dwarkadas, rapporteuse
Prof. J. Larus, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

To my parents...

Acknowledgements

In this small section, I would like to thank the people with whom I shared my PhD journey. This journey would not be possible without them.

First and foremost, I would like to thank my advisor, Anastasia Ailamaki, for her great guidance, teaching, care, help, and support. I learned so much from her that she is one of the major figures in my life who shaped my perspective. She is a great advisor, a fascinating teacher, and a true source of inspiration. It is a privilege to be her student, and I am so lucky to be one of them.

Next, I would like to thank my thesis committee members, Surajit Chaudhuri, Sandhya Dwarkadas, and James Larus. Their feedback and comments have greatly improved my thesis. I am so grateful for their time and effort in working with me and providing feedback. I would also like to thank my thesis jury president, Karl Aberer, for his valuable time and effort in organizing and coordinating my thesis exam.

I would like to thank my collaborators, Pinar Tozun, Danica Porobic, Vivek Narasayya, Surajit Chaudhuri, Sandhya Dwarkadas, Foteini Alvanaki, Anisa Llaveshi, Robert West, Nick Koudas, Ahmad Yasin, Angelos-Christos Anadiotis, Raja Appuswamy, Eliezer Levy, Shay Goikhman, Anthony Iliopoulos, Christina Mantonanaki, Jovan Djukic, Mariem Ali Belhaj, Mouadh Hamdi, and Doruk Cetin. I have been so lucky to be surrounded by such amazing people and work with them. They are great minds, incredible scientists, and great friends. Thank you so much for sharing your work, passion, and dedication with me.

When we started working on my first project with Pinar and Danica, I was super-excited. However, I soon realized that the life was harder than I thought, as I was struggling with adapting to a new country, university, and culture. Pinar and Danica were great mentors, who taught me a lot and also helped me in adapting to my new environment. Thank you so much for the great tutoring and the great friendship.

I have done my summer internship at the Data Management, Exploration and Mining (DMX) group at Microsoft Research, Redmond, closely working with Vivek Narasayya and Surajit Chaudhuri. I would like to thank them and the DMX group for the great opportunity and collaboration. It was super-fun and exciting working with you.

Acknowledgements

In December 2017, Sandhya gave a talk at EPFL. The talk was so interesting that I contacted her right away, and we started a collaboration. It was so educating and so much fun to work with her. I am so grateful for her time and support. I learned a lot from her, and I hope to keep doing so.

Anisa and I worked on a summer intern project, which were planned to last three months. Anisa, however, was so passionate about her work that she continued working on the project during the whole next year, until she writes a paper on it and makes it published. She is one of the smartest and most passionate people I know. Thank you for sharing your work and passion with me.

When I first joined DIAS, it was almost immediate that I felt at home. It is a big family with incredibly smart, generous, and fun people: Manos Karpathiotakis, Matthias Olma, Ioannis Alagiannis, Stella Giannakopoulou, Panagiotis Sioulas, Periklis Chrysogelos, Aunn Raza, Viktor Sanca, Iraklis Psaroudakis, Renata Borovica-Gajic, Georgios Psaropoulos, Eleni Tzirita-Zacharatou, Mirjana Pavlovic, Adrian Popescu, Odysseas Papapetrou, Tahir Azim, Darius Sidlauskas, Satya Valluri, Xuesong Lu, Miguel Branco, Thomas Heinis, Erietta Liarou, Cesar Matos, Benjamin Gaidioz, Eleni Zapridou, Bikash Chandra, Srinivas Venkatesh, Ioannis Mytilinis, Haoqiong Bian, Christina Mantonanaki, and Konstantinos Koukas. Thank you so much for always being there, and always supporting me through the hard times. You guys and gals are the best.

I would like to thank our admins, Lionel Sambuc and Stephane Ecuyer, for their great dedication, help, and support. They patiently bore with me. Thank you so much.

I would like to thank Dimitra Tsaoussis Melissargos and Erika Raetz for their great help and care.

For a PhD student, EPFL is a place where you meet your lifelong friends. I would like to thank the EPFL community who made my life at EPFL super-fun and exciting: Miranda Krekovic, Elie Najm, Mario Paulo Drumond, Winston Haaswijk, Dana Kianfar, Lionel Parreaux, Ana Hernandez-Lopez, Radu Cristian Ionescu, and many others. I had such a good time with all of them. Life would be much harder without their friendship.

I would like to thank my theatre group, The Village Players of Lausanne, for sharing their theatre passion with me. Margot Jaggy, Paul-James Hauduroy, Chris Hemmens, and all the others: Thanks so much.

I would like to thank the GirlsCoding organization, Miranda Krekovic, Marta Martinez, Pavle Belanovic, and all the other mentors, for the great initiative and making my life more meaningful and fun.

I would like to thank my friends from Turkey who always supported me and backed me up for my choices in life.

Lastly, I would like to thank my parents, Hatice Oflazoglu Sirin and Hayrettin Sirin. They are the hero of my, their, and many other children's lives. I am so lucky to be their child and grow up with the light in their hearts.

This research has been supported by grants from DIAS Lab, European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa)), Swiss National Science Foundation, Project No.: 200021_146407/1 (Workload- and hardware-aware transaction processing), and HUAWEI Technologies Co., Ltd, P.R. China.

Abstract

Database workloads have significantly evolved in the past twenty years. Traditional database systems that are mainly used to serve Online Transactional Processing (OLTP) workloads evolved into specialized database systems that are optimized for particular types of workloads. Data warehousing applications have led to Online Analytical Processing (OLAP) workloads and real-time analytical processing applications have led to Hybrid Transactional and Analytical Processing (HTAP) workloads.

Similarly, modern hardware has significantly evolved in the past twenty years. Unicore, simple processors with megabytes of main memory have evolved into multi-core, power-limited processors with hundreds of gigabytes of main memory. Furthermore, the processors have complex micro-architectural features such as Single Instruction Multiple Data (SIMD) instructions and complex branch predictors. Advancements in processor technology have led to further evolution of database systems with novel system architectures and query processing paradigms.

We present the micro-architectural behavior of modern database workloads on a modern processor for various categories of workloads and generations of database systems. We examine three main categories of database workloads and study them separately: OLTP, OLAP, and HTAP. We show that OLTP systems spend most of their execution time waiting for instruction-cache or data-cache misses, where the data-cache misses are due to the random data-accesses during the index lookup operation. While using an efficient index structure can significantly reduce the number of data-cache misses, the main micro-architectural bottleneck remains the data-cache misses due to the costly random data-accesses. Hence, OLTP systems should adopt techniques that mitigate the random data-accesses.

OLAP systems spend most of their execution time in data-cache misses, where the data-cache misses are due to high pressure on the memory bandwidth for sequential-scan-heavy queries, and are due to random data-accesses for join-intensive queries. OLAP systems that follow tuple-at-a-time execution models efficiently use the CPU cycles. However, they require executing a significantly larger number of instructions hence are significantly slower than the systems that follow vector-at-a-time and compiled execution models. Therefore, OLAP systems should use efficient execution models and adopt techniques that mitigate data-cache misses.

Abstract

HTAP systems combine OLTP and OLAP systems into a single, unified system, where the OLTP and OLAP systems run on the same hardware and on the same data. Running on the same hardware results in hardware-level interference, where the OLTP throughput significantly drops due to the OLAP side sharing the hardware resources. Running on the same data results in increased OLAP query execution time due to the need for the OLAP side to process the fresh tuples generated by the OLTP side. Therefore, HTAP systems should adopt techniques that mitigate the hardware-level interference and should make sure that the OLAP side uses enough resources to minimize the increased query execution time.

Keywords: Database management systems, micro-architecture, micro-architectural behavior, performance characterization, modern hardware, hardware-software co-design, benchmarking, transactional processing, analytical processing, hybrid transactional and analytical processing.

Résumé

Les charges de travail des bases de données ont considérablement évolué au cours des vingt dernières années. Les systèmes de base de données traditionnels qui sont principalement utilisés pour servir les charges de travail de traitement transactionnel en ligne (OLTP) ont évolué vers des systèmes de base de données spécialisés qui sont optimisés pour des types particuliers de charges de travail. Les applications d'entreposage de données ont conduit à des charges de travail de traitement analytique en ligne (OLAP) et les applications de traitement analytique en temps réel ont conduit à des charges de travail de traitement transactionnel et analytique hybride (HTAP).

De même, le matériel moderne a considérablement évolué au cours des vingt dernières années. Unicore, des processeurs simples avec des mégaoctets de mémoire principale ont évolué vers des processeurs multicœurs à puissance limitée avec des centaines de gigaoctets de mémoire principale. En outre, les processeurs ont des caractéristiques micro-architecturales complexes telles que des instructions SIMD (Single Instruction Multiple Data) et des prédicteurs de branche complexes. Les progrès de la technologie des processeurs ont conduit à une nouvelle évolution des systèmes de bases de données avec de nouvelles architectures système et des paradigmes de traitement des requêtes.

Nous présentons le comportement micro-architectural des charges de travail de base de données modernes sur un processeur moderne pour différentes catégories de charges de travail et générations de systèmes de base de données. Nous examinons trois catégories principales de charges de travail de base de données et les étudions séparément : OLTP, OLAP, et HTAP. Nous montrons que les systèmes OLTP passent la majeure partie de leur temps d'exécution à attendre les échecs du cache d'instructions ou du cache de données, où les échecs du cache de données sont dus aux accès aléatoires aux données pendant l'opération de recherche d'index. Bien que l'utilisation d'une structure d'index efficace puisse réduire considérablement le nombre d'échecs de cache de données, le principal goulot d'étranglement micro-architectural reste les échecs de cache de données en raison des accès aléatoires coûteux aux données. Par conséquent, les systèmes OLTP devraient adopter des techniques qui atténuent les accès aléatoires aux données.

Les systèmes OLAP passent la majeure partie de leur temps d'exécution dans des échecs de cache de données, où les échecs de cache de données sont dus à une pression élevée sur

la bande passante de la mémoire pour les requêtes lourdes d'analyse séquentielle, et sont dus à des accès aléatoires aux données pour les requêtes à jointure intensive . Les systèmes OLAP qui suivent des modèles d'exécution tuple à la fois utilisent efficacement les cycles du processeur. Cependant, ils nécessitent l'exécution d'un nombre beaucoup plus important d'instructions et sont donc beaucoup plus lents que les systèmes qui suivent des modèles d'exécution vectoriels à la fois et compilés. Par conséquent, les systèmes OLAP doivent utiliser des modèles d'exécution efficaces et adopter des techniques qui atténuent les erreurs de cache de données.

Les systèmes HTAP combinent les systèmes OLTP et OLAP en un système unique et unifié, où les systèmes OLTP et OLAP fonctionnent sur le même matériel et sur les mêmes données. L'exécution sur le même matériel entraîne des interférences au niveau matériel, où le débit OLTP diminue considérablement en raison du partage des ressources matérielles du côté OLAP. L'exécution sur les mêmes données entraîne une augmentation du temps d'exécution de la requête OLAP en raison de la nécessité pour le côté OLAP de traiter les nouveaux tuples générés par le côté OLTP. Par conséquent, les systèmes HTAP doivent adopter des techniques qui atténuent les interférences au niveau matériel et doivent s'assurer que le côté OLAP utilise suffisamment de ressources pour minimiser l'augmentation du temps d'exécution des requêtes.

Mots clés : Systèmes de gestion de base de données, micro-architecture, comportement micro-architectural, caractérisation des performances, matériel moderne, co-conception matériel-logiciel, benchmarking, traitement transactionnel, traitement analytique, traitement transactionnel et analytique hybride.

Contents

1 Introduction

1.1 Database Management Systems

Today's world is data-driven. Regular media sources, social media sources, individual people, as well as industrial and community organizations all produce and consume data. This leads to an enormous size for the data and demand to produce and consume the data [1, 36, 37, 38]. Database Management Systems (DBMS) have been at the centre of storing, managing and processing data. As the data size and the demand to produce and consume the data is ever-growing, researchers and developers from academia and industry have proposed numerous techniques and novel DBMS architectures to cope with the ever-growing size of the data and the speed that the data is produced and consumed [8, 34, 40, 54, 105].

One of the most important driving-forces behind novel DBMS architectures has been the advancements in the modern hardware [4]. Modern hardware advancements include the growing-sized main memories and the main-memory bandwidths, complex micro-architectural features such as strong branch prediction features and deep cache hierarchies features, and wide data-parallel execution units such as Single Instruction Multiple Data (SIMD) execution units. The novel hardware-aware DBMS architectures allowed DBMS to use their hardware resources more efficiently, and hence deliver orders of magnitude higher performance [2, 71].

In addition to the modern hardware, the evolving workload demands and characteristics have been the other major driving-force behind novel DBMS architectures. Early relational database systems were mostly driven by Online Transaction Processing (OLTP) workloads, where the workloads are composed of several short-lived transactions that read or write to small amount of records [21, 22]. During the late 1990s, data warehousing applications gained significant attention, where Online Analytical Processing (OLAP) workloads have emerged. OLAP workloads are composed of read-only, long-running queries that process large number of records. Due to the significant differences between the OLAP and OLTP workloads, a specialized set of DBMS have been designed and developed to serve for OLAP workloads [19].

In the past five years real-time analytical processing applications have gained a significant

attention. Real-time analytical processing applications aimed to perform complex OLAP queries on fresh, transactional data. As a result, Hybrid Transactional and Analytical Processing (HTAP) workloads have emerged. HTAP workloads are mix of OLTP and OLAP workloads and are served by HTAP systems designed to efficiently support both OLTP and OLAP workloads [14, 83]. Having designed and developed DBMS that efficiently use their hardware resources and specialized for individual types of workloads allowed DBMS to deliver high performance and cope with the ever-growing size of the data and demand to produce and consume the data.

1.2 Evolution of Modern Processors

Commodity processors have been continuously evolving over the past five decades, in the light of Moore's Law. Initial processor technology focused on improving the single-threaded performance by providing wider instruction-level parallelism (ILP) (through wider-issue super-scalar execution), higher clock frequency rates, and complex micro-architectural features such as highly-accurate branch prediction units and aggressive pipelined execution. During the late 1990s, multiprocessor technology has gained a rise due to the scalability limitations in providing higher ILP and the dissipated heat [82].

Multiprocessor technology has widely been adopted. Numerous applications, such as transaction processing and scientific processing applications, benefited the thread-level parallelism that the multiprocessor technology provides. As a result, multiprocessor technology has lead to popular multi-core processors, where a single Central Processing Unit (CPU) contains multiple communicating cores sharing a coherent main memory. The processor designers gradually increased the number of cores per processor to supply the increasing demand for the thread-level parallelism. Moore's Law coupled with Dennard scaling allowed to have more and more cores on the same CPU. Moore's Law stated that the number of transistors on chip doubles every 18 months. Whereas, Dennard scaling stated that the reduced-sized transistors also consume proportionally reduced power per transistor. As a result, processors accommodated an increasing number of transistors within a similar power budget [24, 77].

In the past five years, the increased core counts have slowed down due to the failure of Dennard scaling. While the number of transistors kept increasing according to Moore's Law, the amount of heat dissipated per transistor failed to decrease. As a result, the increased number of transistors' heat dissipation has hit the maximum level of dissipated heat that the processor silicon can accommodate. The number of cores per processor is more and more flattened, and the applications have stopped benefiting from the exponentially increasing performance freely provided by the hardware [27].

Having hit the power wall, the processor vendors and application developers have considered alternative micro-architectures. In particular, beefy, power-hungry multi-core processors are compared with wimpy, low-power multi-core processors. The goal has been using relatively larger number of low-power cores compared to the number of power-hungry cores, and

benefiting from high thread-level parallelism that the applications exhibit. While this idea has led to several important system architectures [6], wimpy cores failed to gain a wide adoption. The reason has been that, despite the applications exhibit high degree of thread-level parallelism, the scalability of an application is more and more limited as the number of parallel threads it uses is increased. By following Amdahl's law, even a small serialization point has become a performance bottleneck for the system. This required careful and explicit parallelization of many applications, putting a prohibitive software development cost on the programmers [78].

As a result, today's commodity servers are evolved into power-hungry multi-core chips with dozens of cores, where each core contains complex micro-architectural features such as complex branch predictor, deep pipelines, superscalar out-of-order execution units, and deep cache hierarchies. As the processors have evolved into a particular micro-architecture with modest performance improvements over the new generations, software has gained a significant importance in exploiting the modern hardware features to deliver a high performance, and cope with the ever-growing size of the data and the demand to produce and consume the data [4].

1.3 Database Management Systems on Modern Processors

Database systems have long been optimized to efficiently use modern hardware features. Initial work has concentrated on multi-core scalability of traditional, disk-based transaction processing systems, where centralized database system components such as lock and log managers have become bottlenecks inhibiting the multi-core scalability [48, 84, 110]. Later work has concentrated on exploiting large main memory sizes, and eliminating heavy-weight components of disk-based transaction processing systems. This has lead to popular in-memory transaction processing systems that are tightly optimized for modern hardware, providing orders of magnitude higher performance than traditional disk-based systems [25, 54, 106].

Simultaneously, column-stores have gained a significant attention, where specialized analytical processing engines serve for read-only, complex queries over column-wise stored data. Column-stores keep the two-dimensional relational tables column-by-column, rather than traditionally-adopted row-by-row (i.e., record-by-record). This way, column-stores process only the necessary columns, i.e., attributes such as age and name, which allows more efficiently using disk/memory bandwidth and significantly reducing the data processing overheads. Column-stores gained a wide adoption. Most of the major DBMS vendors has now their specialized column-store engines to serve for read-only, analytical processing workloads [2, 17, 40].

1.4 Micro-architectural Behavior of Database Workloads

Micro-architectural behavior defines where the CPU cycles are spent within the core micro-architecture when running a particular workload. The core micro-architecture components include the instruction fetch and decoding units, branch prediction unit, instruction execution units and data load and store units. Early work on understanding the micro-architectural behavior of database management systems have focused on architectural features that are useful for commercial database systems, such as out-of-order execution, on-chip cache sizes and size of the instruction queue [11, 32, 53, 90, 103]. The studies examined the architectural features for a single, commercial, well-established system, when running the two major types of database workloads, OLTP and OLAP, on a hardware simulation environment.

A later work took a step further and examined micro-architectural behavior of several commercial DBMS to identify the micro-architectural behavioral trends that hold across different DBMS running on real hardware [3]. This was the first work that examined micro-architectural behavior across different DBMS and focused on general trends across the DBMS. Later on, several studies have concentrated on traditional, disk-based transaction processing systems running on real hardware, and associated micro-architectural behavioral components, such as instruction-cache stalls, with particular transaction processing system components, such as lock manager [109, 112, 116].

Despite the large body of micro-architectural characterization work for database workloads, existing work has only focused on either single class of a DBMS, e.g., a well-established commercial DBMS, or a single type of database workload, such as OLTP workloads. However, both DBMS and database workloads have gone through a significant evolution over the past 20 years, by following the advancements in the hardware and the application demands. Furthermore, the commodity processor micro-architecture has significantly advanced over the past twenty years. Hence, the micro-architectural behavior of state-of-the-art database workloads remains unclear.

As the processor performance is stagnating due to power limitations, it has been ever more important to understand the micro-architectural behavior in order to efficiently use the micro-architectural resources and extract maximum performance out of the server hardware. This thesis presents micro-architectural analysis of database workloads across different generations of DBMS and categories of database workloads. Its goal is to understand the micro-architectural behavior of the state-of-the-art database workloads, identify general trends and main performance bottlenecks at the software- and hardware-levels, and highlight investigation directions for future database management systems to deliver high performance and efficiently use their hardware resources.

1.5 Thesis Statement and Contributions

The statement of this thesis is as follows.

Thesis Statement

Understanding a database system's interaction with compute and memory resources is key to efficiently using hardware. Database systems spend most of their execution time waiting for instruction-cache or data-cache misses, where data-cache misses are due to high pressure on the memory bandwidth if the data access pattern is sequential, and due to random data-accesses if the data access pattern is random. Workloads with mixed sequential and random data-access patterns also suffer from interference. Hence, database system architects should design and develop data structures and algorithms that are aware of the data access pattern and cache behavior to efficiently use the hardware resources.

Below, is a summary of our contributions in this thesis:

- We examine state-of-the-art micro-architectural analysis methodologies. We compare and contrast conventional cache-miss-based methodology with Intel's recently-proposed Top-down Micro-architecture Analysis Methodology (TMAM). We observed that cache-miss-based methodology under- or over-estimates the execution time due to under- or over-estimating the instruction or data-cache stalls. We adopt Intel's TMAM methodology in our analyses.

- We show that Online Transaction Processing (OLTP) systems spend more than half of the execution time waiting for instruction-cache or data-cache misses. The disk-based OLTP systems suffer mainly from instruction-cache misses, whereas the in-memory OLTP systems suffer from either instruction-cache or data-cache misses. The data-cache misses are due to the random data-accesses during the index lookup operation. While using an efficient index structure can significantly reduce the number of data-cache misses, it does not change the main micro-architectural bottleneck due to the costly random data-accesses.

- We show that, if the system follows vector-at-a-time or compiled execution model, Online Analytical Processing (OLAP) systems spend most of their execution time waiting for data-cache misses. The data-cache misses are due to high pressure on the memory bandwidth for sequential-scan-heavy queries, and are due to random data-accesses for join queries. The OLAP system that follows the tuple-at-a-time execution model efficiently uses the CPU cycles. However, it requires executing a significantly larger number of instructions hence is 1.7 to 5 times slower than the systems that follow the vector-at-a-time and compiled execution models.

- Hybrid Transactional and Analytical Processing (HTAP) systems combine OLTP and OLAP systems into a single, unified system, where the OLTP and OLAP systems run on the same data and on the same hardware. We show that running the OLTP and OLAP systems on the same hardware results in hardware-level interference. The OLTP

throughput is decreased by 22 to 40% due to the OLAP side sharing the hardware resources. Running the OLTP and OLAP systems on the same data results in software-level interference. The query execution time at the OLAP side is increased due to the additional work processing the fresh tuples generated by the OLTP side. We show that, to minimize the increase in the OLAP query execution time, the OLAP component should be allocated enough resources to be able to process the fresh tuples faster than the OLTP component generates them.

1.6 Roadmap

In this section, we present the organization of the thesis.

- Chapter 2 presents the background for this thesis. It presents the micro-architectures of today's server processors, and the workload characteristics for the state-of-the-art database workloads: OLTP, OLAP and HTAP workloads. It describes the performance monitoring units (PMU) of today's modern processors, and how to profile performance of a modern processor. Lastly, it presents the related work on the micro-architectural analysis of data-intensive workloads.
- Chapter 3 compares and contrasts the state-of-the-art micro-architectural analysis methodologies: the popular cache-miss-based methodology and Intel's recently-proposed Top-down Micro-architecture Analysis Methodology (TMAM). It concludes that, while the two methodologies provide similar high-level micro-architectural behavior, Intel's TMAM is more powerful as it provides end-to-end execution time breakdown and also accounts for non-memory-stalls. Based on our conclusions in this chapter, we chose Intel TMAM as our micro-architectural analysis methodology, and use it in the rest of this thesis.
- Chapter 4 presents the micro-architectural analysis of OLTP workloads. It presents micro-benchmark as well as standard TPC-C benchmark evaluation. It uses micro-benchmarks for sensitivity analyses that vary the size of the data and the amount of work per transaction. It uses the standard TPC-B and TPC-C benchmarks to confirm that the conclusions drawn using the micro-benchmark apply for a standard benchmark. It further presents the impact of index, transaction compilation, data type and multi-threading on the micro-architectural behavior. It examines the effect of hardware acceleration features and also compares and contrasts the micro-architectural behavior of different Intel micro-architectures. The chapter concludes that OLTP workloads spend most of their execution time in instruction-cache or data-cache misses, where the data-cache misses are due to the random-data-access-heavy nature of OLTP workloads.
- Chapter 5 presents the micro-architectural analysis of OLAP workloads. Unlike OLTP workloads, OLAP workloads are read-only, have highly iterative nature in their computation, and also exhibit both random and sequential data access patterns. The chapter

performs sensitivity analyses by using micro-benchmarks, and also presents standard TPC-H benchmark analysis to confirm that the findings by the micro-benchmark analyses apply to the standard benchmark. The chapter further performs mixed query workload evaluation where workloads with heterogeneous data access patterns are evaluated. The chapter examines software-level optimzations, such as predication, and hardware-level acceleration features, such as SIMD and hyper-threading. The chapter concludes that OLAP workloads spend most of their execution time in data-cache misses, where the data-cache misses are due to either high pressure on the memory bandwidth or to random data-accesses. Although OLAP systems that follow the tuple-at-a-time execution model efficiently use the CPU cycles, they are 1.7 to 4.5 times slower than the OLAP systems that follow vector-at-a-time and compiled execution models due to executing a significantly larger number of instructions.

- Chapter 6 presents HTAP workloads evaluation. HTAP workloads are composed of mixed OLTP and OLAP workloads, where the OLTP and OLAP workloads share both data and hardware. In this chapter, we examine how data and hardware sharing affects the OLTP and OLAP throughput of an HTAP system. The chapter performs micro-benchmark evaluation with workloads that mix basic data-access patterns: sequential and random, as well as the complex CH benchmark evaluation. In this chapter, we examine last-level cache, memory-bandwidth and hyper-thread sharing. We conclude that, while the OLAP throughput does not drop, the OLTP throughput drops by 22 to 40% when OLTP and OLAP workloads share hardware resources. When OLTP and OLAP workloads share data, the OLAP query execution time is minimally affected by the OLTP workload if the OLAP workload is allocated enough resources to process the fresh tuples faster than the OLTP workload generates them.

- Chapter 7 finally presents the conclusions and highlights future investigation directions for modern database systems to efficiently use the hardware resources and deliver a high performance.

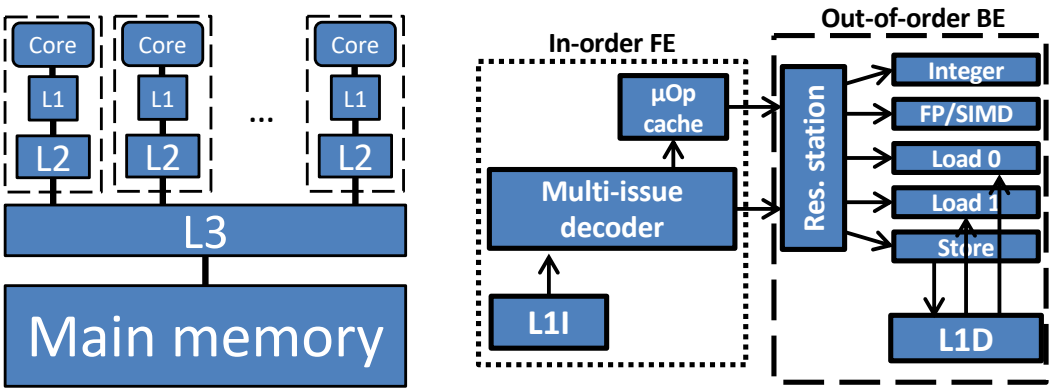
2 Background

This chapter covers the necessary background for the thesis and the related work on micro-architectural analysis of data-intensive workloads. The background includes description of the modern processor micro-architectures, and the descriptions of the modern database workloads: Online Transactional Processing (OLTP), Online Analytical Processing (OLAP), and Hybrid Transactional and Analytical Processing (HTAP) workloads. Lastly, the background describes how to profile a modern processor micro-architecture.

2.1 A Modern Processor Micro-architecture

This section outlines the micro-architecture of today's modern processors. Figure 2.1 shows the simplified block diagram of the Intel Sandy/Ivy Bridge micro-architecture [42]. The left-hand side of the figure presents the multi-core processor organization, whereas the right-hand side of the figure presents the core micro-architecture. The multi-core processor includes multiple cores with private L1 and L2 level of caches, and a shared L3 level of cache. The L3 level of cache is connected to the main memory. The number of clock cycles needed to access to the caches is increased as the level in the cache is increased in the memory hierarchy. For the Intel Sandy/Ivy Bridge micro-architecture, it takes ~4, 12, 30 and 200 clock cycles to access to L1, L2, L3 and main memory, respectively.

The right-hand side of Figure 2.1 zooms into a single core, and presents the core micro-architecture. The core micro-architecture contains two major building blocks, the front-end (FE) and the back-end (BE). FE contains the micro-architectural structures to fetch, decode and issue instructions, which are L1 instruction (L1I) cache, a multi-issue instruction-to-micro-operation(μOp) decoder and a small (1.5KB [42]) μOp cache. The L1I cache is responsible for keeping the instructions that the processor will execute next. The decoder decodes the fetched instructions into μOps. As Intel uses complex instruction set architecture, i.e., CISC, it needs to decode complex instructions into simpler μOps. The decoder is multi-issue, i.e., it is able to decode and deliver multiple instructions in one cycle. Delivering multiple instructions in one cycle enables exploiting instruction-level parallelism, and using the resources of the BE



(a) Multi-core processor organization.

(b) Core micro-architecture.

Figure 2.1 – An out-of-order processor model based on Intel Sandy/Ivy Bridge micro-architecture [42]. FE: front-end, BE: back-end.

more efficiently. Lastly, the μOp cache is responsible for keeping the most recently decoded μOps to avoid re-decoding them.

BE contains the micro-architectural structures to execute the issued μOps, which are a reservation station (RS), several execution units and L1 data (L1D) cache. When a μOp arrives to BE, BE registers the μOp to the RS. RS is then responsible for tracing the operands and dependencies of the μOp, and delivering the μOp to the relevant execution unit. The execution units are responsible for executing the instruction, e.g., adding two integers, or loading an integer to a register. All execution units operate in parallel unless there is a dependency. Parallel execution enables exploiting instruction-level parallelism, and therefore, executing multiple instructions in one cycle. An out-of-order processor can theoretically execute as many instructions as issued in one cycle. Furthermore, every execution unit has its own private buffer. For example, the Sandy/Ivy Bridge micro-architecture contains a 64-entry load and a 36-entry store buffer, which enables buffering up to 64/36 load/store operations. Buffering allows to overlap the same kinds of μOps when some are not ready to complete. For example, if a load operation misses from L1D cache, another load operation can start its execution without waiting for the former load operation to finish. Lastly, L1D cache is incorporated to BE and works cooperatively with the load and store execution units to deliver data from/to memory hierarchy to/from processor. The L1D cache of today's processors contains a set of registers for tracking the outstanding cache misses, i.e., miss status handling registers (MSHRs). MSHRs allow processors to continue their execution when there is a cache miss, and further improve parallelism.

2.2 Online Transactional Processing

A transaction is an indivisible unit of work that should atomically be executed over a database. A transaction can read or write to the database. The atomic execution implies that either all or none of the actions of the transaction should be executed. Online Transaction Processing

(OLTP) systems are used for consistently maintaining a database across multiple, concurrently executing transactions. OLTP systems are responsible for maintaining the *ACID* features, where *A* stands for *Atomicity*, *C* stands for *Consistency*, *I* stands for *Isolation*, and *D* stands for *Durability*. Below we briefly describe each *ACID* property.

- ***Atomicty:*** A transaction is an indivisible unit of work. It should either be fully executed, or not executed at all.
- ***Consistency:*** A transaction should start its execution over a consistent database and finish its execution by leaving the database in another consistent state.
- ***Isolation:*** Concurrently executing transactions should perform their actions as if each transaction performs its actions alone in the database.
- ***Durability:*** Transactional modifications should be durable, i.e., should be persisted to a durable medium such as a disk.

Online Transaction Processing (OLTP) systems are composed of four main components: (i) Buffer manager, (ii) lock manager, (iii) log manager, and (iv) access methods. These components are tightly intertwined to be able to satisfy the *ACID* properties and also provide fast access to the database [35]. We briefly describe each component below.

- ***Buffer manager:*** Traditional OLTP systems keep the data on persistent disk. However, disk access time is orders of magnitude higher than main memory access time. Buffer manager's goal is to keep most frequently used disk pages in main memory to minimize the time needed to access the data.
- ***Lock manager:*** Traditional OLTP systems use a centralized lock manager. The lock manager is responsible for granting the lock acquire/release requests. It maintains the list of worker threads and acquired/released locks, and manages which transactions should be actively executed or blocked based on the lock acquire/release requests.
- ***Log manager:*** Log manager is responsible for logging the transactional modifications to the database to be able to satisfy the durability property. In case of a system failure, e.g., a power outage, the system can recover the database to a consistent state thanks to having the transactional modifications logged to a durable medium such as a disk.
- ***Access methods:*** Access methods are used to access the data fast. An index structure, e.g., a B+tree index, is a commonly used access method in OLTP systems.

2.2.1 In-memory Online Transaction Processing

Commodity servers of the last decade follow two fundamental trends: (1) main-memory becoming cheaper and (2) number of cores increasing exponentially. Simply increasing the

buffer pool size and the number of worker threads in the system to exploit the large main-memory and all the available cores, respectively, lead to marginal gains. Therefore, these two hardware trends have triggered alternative design opportunities for the new generation database systems.

As DRAM prices become cheap enough to buy 1TB main-memory for ~$30K, today it is possible for most OLTP applications to keep all of their data working set in main-memory while running on a commodity server hardware. This has led to the development of various in-memory or main-memory optimized OLTP systems. These systems either manage all the data in main-memory or make sure that the hot data resides in main-memory. Since they manage to eliminate/minimize the disk I/O for the data page accesses, the overheads associated with managing the buffer pool overweigh its benefits [33]. Therefore, the in-memory OLTP systems omit the buffer pool component even though it is essential for the traditional disk-based database systems as it gives the illusion of an infinite main-memory to the system.

On the other hand, in step with Moore's law, the hardware vendors keep providing more and more opportunities for parallelism. Modern servers tend to have multiple multicore processors in the same machine and allow OLTP systems to handle increasing number of transactional requests in parallel. However, the traditional concurrency control mechanisms that ensure isolation among concurrent transactions using a centralized lock manager and two-phase locking are designed at an era where the server hardware were uniprocessors. Therefore, they do not scale on multicores preventing OLTP systems from exploiting the sheer number of cores available to them [84, 120].

In order to achieve better scalability on multicore architectures, in-memory OLTP systems adopt alternative concurrency control mechanisms. These mechanisms can be broadly grouped into two categories based on whether they partition the data or not. The ones that partition the data choose an extreme form of physical partitioning where there is a data partition for each core and a single worker thread for each partition. Systems like VoltDB [107] (or its ancestor H-Store [104]) and the initial version of HyPer [55] deploy this approach. As a result, they can avoid any form of locking within a partition and need to coordinate worker threads only when a transaction requires data from multiple partitions (i.e., in the case of distributed transactions). On the other hand, the systems that prefer avoiding any kind of data partitioning, like Hekaton [61], SAP HANA [64], or the latest version of HyPer [81], rely on optimistic and multiversion concurrency control [13].

In addition to alternative concurrency control mechanisms, in-memory database systems also deploy cache-conscious index structures. They align the index page sizes to the size of a cache line as opposed to the size of a disk page and/or adopt lock-free index page access mechanisms rather than using traditional page latches [67, 114]. Moreover, the in-memory OLTP systems tend to depend on pre-determined stored procedures instead of ad-hoc queries [55, 61, 107] and apply efficient compilation optimization techniques that optimize the instruction stream for a particular transaction [61, 80]. Finally, the new-age in-memory OLTP systems have

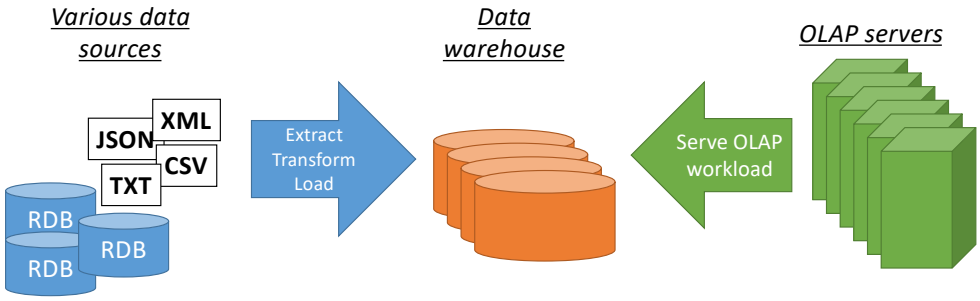


Figure 2.2 – Data Warehouse Architecture (taken from [19]).

codebases that are implemented from scratch. Therefore, they are expected to have a cleaner codebase compared to the traditional disk-based systems where the codebase consists of many branch statements and obsolete code paths due to different release versions spanning several decades of development.

Overall, in-memory OLTP engines deploy lighter storage manager components compared to the traditional disk-based systems aiming to utilize the resources of the modern server hardware in a more effective way.

2.3 Online Analytical Processing

OLTP applications were the main driving force for the invention of relational database systems [21, 22]. During late 90s, data warehousing applications have gained a significant attention and led to Online Analytical Processing (OLAP) systems specialized for OLAP workloads. OLAP workloads are composed of read-only, complex, analytical processing queries that aim to extract valuable information out of the data.

Data warehouses are implemented separately from the transactional databases. Transactional databases usually keep the fresh, up-to-date data. However, executing data warehousing applications on a transactional database would result unacceptably high response time due to the additional work that transactional systems require to keep the data consistent, such as locking and logging. As OLAP workloads are read-only, they are fined-tuned on a separate database to provide fast access to the masses of collected data.

Figure 2.2 presents how data warehouses are typically built. Transactional data is collected from various sources and goes through a procedure called Extract, Transform and Load (ETL). At the end of the ETL procedure the data is ready for the data warehousing applications. Data warehousing applications run complex OLAP queries over the data warehouse. The OLAP queries can be executed on Relational OLAP, i.e., *ROLAP* servers, or Multi-dimensional OLAP, i.e., *MOLAP* servers. Both ROLAP and MOLAP servers provide multi-dimensional view of the data, which is useful for data warehousing applications. While ROLAP servers extend traditional relational databases (RDBs) by using an intermediate layer to provide efficient multi-dimensional view of the data, MOLAP servers directly support the multi-dimensional

storage of the data. Both ROLAP and MOLAP servers use materialized views and optimized indices to provide fast access to massive amounts of data [19].

2.3.1 Column-stores

In early 2000s, column-stores have gained a significant attention. Column-stores relied on the observation that most OLAP queries require accessing only a subset of the attributes of the relational tables that the queries process. Hence, column-stores stored the data column-by-column, as opposed to traditional row-by-row storage. Furthermore, column-stores adopted numerous optimizations such as late materialization, predication and vectorization. Researchers have shown that column-stores can provide orders of magnitude faster performance than traditional OLAP servers thanks to their lean storage layout and the adopted optimizations [2, 17, 18]. As a result, column-stores have gained a wide adoption and become standard for OLAP workloads. Most of the major database system vendors today have a column-store offering as a specialized engine embedded in the main database system [60, 89].

2.3.2 Query Processing Paradigms

When a query is submitted to a database system, the query optimizer parse the query and produce an optimized query execution plan. The query plan is usually expressed in the form of a tree, where every node represents a particular operator, such as a project, select, or join operator. Query processing engine executes the query plan to produce the desired output. There are three main query processing paradigms that modern database systems use: (i) tuple-at-a-time, (ii) vector-at-a-time, and (iii) compiled query processing.

Tuple-at-a-time query processing: In tuple-at-a-time query processing, all query operators use the same abstract class, usually called the Iterator class, that exposes three main methods: `open()`, `next()`, and `close()`. The `open()` method starts the operator's processing. The `next()` method fetches the next tuple to process. And, the `close()` method finalizes the processing of the operator. The `open()` and `close()` functions are called once per operator, whereas the `next()` function is called for every single tuple processed for every operator in the query plan. The tuple-at-a-time query processing paradigm is simple and intuitive as it uses the same abstract class to implement all the query operators. However, it suffers from the high cost of the `next()` function calls, as the `next()` function is called once per tuple, which can be of millions. Most early database systems used the tuple-at-a-time query processing paradigm [29].

Vector-at-a-time query processing: Vector-at-a-time query processing aims to mitigate the high cost of the `next()` function calls. To do so, instead of fetching a single tuple per `next()` function call, vector-at-a-time processing fetches a vector of tuples per `next()` function call. As a result, it significantly reduces the number of time the `next()` function is called and hence reduces the high cost of the `next()` function calls. A vector of tuples is usually an array

of values that is passed from one operator to another. The vector size is a tunable system parameter and is usually in the orders of a few thousands. *VectorWise* has been the pioneering system that follows the vector-at-a-time query processing paradigm, which later on evolved into a commercial system [2, 17].

Compiled query processing: Compiled query processing also aims to mitigate the high cost of the `next()` function calls of the tuple-at-a-time query processing. It fuses multiple query operators into a single meta-query-operator and makes one function call for processing the whole meta-query-operator. As the query plan, which specifies the query operators that are needed to execute the query, is only know at run-time, compiled query processing relies on run-time code generation. After the query optimizer produces the query plan, the compiled query processing paradigm generates the code that will be run to execute the query and compiles the generated code down into the machine code at run-time. The generated and compiled code is run to execute the query and produce the desired output. *HyPer* has been the pioneering system that follows compiled query processing paradigm, which later on evolved into a commercial system [54, 79].

Quickstep: In traditional tuple-at-a-time query processing, the base class is abstract. Hence, the next() function calls are a virtual function calls, which is costlier than regular function calls. A recently proposed system, *Quickstep*, argues that using aggressive function inlining to mitigate the high cost of the `next()` function calls coupled with using efficient parallelization techniques and several filtering-based optimizations enable delivering a similar level of performance to that of the vector-at-a-time query processing paradigm. We examine how costly the `next()` function calls and how useful the filtering-based optimizations for *Quickstep* in Chapter 5 in more detail [85].

2.4 Hybrid Transactional and Analytical Processing

Traditionally OLAP queries run on the data that is collected from various sources as shown in Figure 2.2. This requires an expensive Extract, Transform, Load (ETL) operation on the transactional data. ETL is an expensive operation, and usually is performed once a day/week. Hence, the OLAP queries run on data that is stale. Recently, there is an increasing demand on running real-time analytical queries on fresh, transactional data. Fraud detection, risk analysis, financial trends tracking applications are examples that demand real-time analytical processing [14, 83].

Real-time analytics applications have led to Hybrid Transactional and Analytical Processing (HTAP) systems. HTAP systems combine OLTP and OLAP systems into a single unified framework to run OLAP queries on fresh OLTP data. Hence, HTAP workloads contain both transactional and analytical requests performed on the same database.

There has been several proposed HTAP architectures. The most popular HTAP architecture is the two-copy, mixed format (TCMF) architecture. TCMF architecture keep two copies of

the data, one for OLTP and one for OLAP component. To keep the data consistent across the OLTP and OLAP components, TCMF architecture uses an intermediate data structure, *delta*, that keeps track of the recently modified, fresh tuples. Periodically, TCMF architecture flushes the delta to the OLAP-side to make the OLAP-copy of the data consistent with the OLTP-copy. The OLTP-copy is usually kept in row format as row-wise storage is more optimized for OLTP, whereas the OLAP-copy is usually kept in columnar format as columnar format is more optimized for OLAP. Microsoft SQL Server [62], Oracle [59], and BatchDB [73] use TCMF architecture.

Another popular architecture is single-copy, mixed-format (SCMF) architecture. SCMF architecture keeps a single copy of the data. Furthermore, it keeps an intermediate data structure called *delta* (similar to the TCMF architecture) to keep the fresh, recently-modified OLTP data. OLTP transactions only modify the delta, whereas OLAP queries read both delta and the main copy of the data. The main copy of the data usually kept in columnar format to largely serving for OLAP workloads, whereas the delta structure keeps the data usually in row format. The architecture trades off OLTP performance for reduced memory consumption thanks to keeping a single copy of the data. SAP HANA [95] and MemSQL [100] follow this type of architecture.

Lastly, single-copy, single-format (SCSF) architecture keeps a single-copy of the data and uses only single format, i.e., only row or columnar, both for OLTP and OLAP workloads. It relies on copy-on-write snapshotting or multi-version concurrency control mechanisms to keep multiple versions of the data, through which analytical queries can access to the fresh-most transactional data. HyPer [81] and Caldera [7, 91] follow this type of architecture.

2.5 Profiling A Modern Processor Micro-architecture

Today's processors contain a Performance Monitoring Unit (PMU). PMUs contain a set of registers that the PMU can configure and collect certain types of hardware events such as number of L1 instruction-cache misses. Today's processors feature large number of hardware events. We have observed 422 events on the Ivy Bridge machine that we used in Chapter 3.

VTune is Intel's popular micro-architectural analysis tool that allows accessing the hardware event and defining micro-architectural analysis methodologies. VTune requires the user to identify the set of hardware events that the user is interested in. VTune, then, runs together with the application and collects how many times the identified event has occurs during the application run. Having the application stopped, VTune reports the final numbers for each identified event [44].

Hardware has only a fixed number of registers to collect the event numbers. If the number of events is larger than the number of registers, VTune uses its internal multiplexing algorithm, where VTune collects every event during different periods of the application, and for statistically significant many times. VTune has predefined duration types: very short, short, medium

and long. Having specified the duration type, VTune uses its internally optimized multiplexing algorithm to provide statistically significant times. We have observed that short and medium duration types provide statistically significant results [45].

VTune has a low overhead. It affects the performance of the application, which it concurrently runs with, less than 5% [43].

VTune has several built-in micro-architectural methodologies such as `general-exploration` that adopts Intel's published Top-down Micro-architecture Analysis Methodology (TMAM) [117] and provides end-to-end execution time breakdown inside the core micro-architecture, or `memory-access` that provides memory bandwidth consumption values. The built-in analysis methodologies use a set of events and a set of formulas that uses the set of events to produce a meaningful micro-architectural analysis component, such as the amount of time spent for L1 instruction-cache misses. We study VTune's built-in `general-exploration` in Chapter 3.

2.6 Related Work on Micro-architectural Analysis of Data-Intensive Workloads

There is a large body of related work analyzing the micro-architectural behavior of data-intensive workloads. Barrosso et al. [11] investigate the memory system behavior of OLTP and OLAP style workloads both on a real machine and with a full-system simulation. They argue that these two types of workloads would benefit from different architectural designs in terms of the memory system. Ranganathan et al. [90] perform a similar analysis. However, they only focus on the effectiveness of out-of-order execution on SMPs while running these workloads in a simulation environment. On the other hand, Keeton et al. [53] and Stets et al. [103] experiment only with OLTP benchmarks (TPC-B and TPC-C) on real hardware. These studies agree that OLTP workloads utilize the underlying micro-architectural resources very poorly, wasting most of the execution cycles on memory stalls and exhibiting a low IPC value.

Ailamaki et al. [3] examine where the time goes on four commercial database systems using a micro-benchmark to have a fine-grain understanding of the memory system behavior on multiprocessors, whereas Hardavellas et al. [32] analyze TPC-C and TPC-H on both in-order and out-of-order machines in a simulation environment. These studies focus on the implications for the database systems rather than the hardware to achieve better hardware utilization.

More recent workload characterization studies [111] analyze the OLTP benchmark, TPC-E, and show that micro-architecturally TPC-E behaves very similarly to the TPC-B and TPC-C benchmarks. Ferdman et al. [28] present micro-architectural analysis of a suite of cloud workloads, by concluding that there is a fundamental mismatch among what today's server processors provide and what the cloud workloads demand. These studies corroborate the findings of the previous studies in terms of the inefficient use of the memory hierarchy when

running OLTP. They highlight that the L1-I stalls are the dominant factor in the overall stall time followed by the long-latency data misses.

Yasin et al. [117] introduce Top-Down Micro-architecture Analysis Methodology (TMAM) deployed by Intel VTune as `general-exploration`. Yasin et al. [118] examine Naive Bayes algorithm and its hardware behavior in an Hadoop execution environment. The study shows that software stack, such as the used JVM, and application code efficiency has a significant impact on the overall performance. Kanev et al. [50] examine collective of machines in a Google datacenter running collective of Google datacenter applications. The study shows that the datacenter workload collection spends most of the time waiting for dependent data-cache accesses due to the data-intensive nature of the datacenter workloads. Beamer et al. [12] presents a graph workload analysis and highlights that graph workloads severely under-utilize the memory bandwidth. Sridharan and Patel [102] examine the evaluation of workloads on the popular data analysis language R, over a commodity processor. Awan et al. [9, 10] present a micro-architectural analysis of Spark.

Harizopoulos et al. [33] demonstrate that traditional OLTP systems spend more than half of their execution time inside the buffer pool, latching, locking, and logging components. On the other hand, Wenisch et al. [116] and Tozun et. al [108] tie the micro-architectural behavior of the disk-based OLTP into specific code modules by presenting the breakdown of the cache misses into specific code parts of the traditional OLTP software stack at different code granularities.

Kersten et al. [56] examine vectorized and compiled OLAP engines without getting deep into the micro-architectural behavior. Sompolski et al. [101] present a comparison between vectorized and compiled engines in terms of particular optimizations, such as predication and SIMD.

Despite the large body of existing work on the micro-architectural analysis of data-intensive workloads, the existing work falls short on evaluation the micro-architectural behavior of state-of-the-art database workloads that are composed of different categories of workloads, such as OLTP, OLAP and HTAP, and different generations of machines, such as traditional, well-established systems, ground-up designed, new-generation systems, and academic prototypes. This thesis aims to fill the gap for the micro-architectural analysis of state-of-the-art database workloads by considering each category of the database workloads running on different generations of database systems, separately.

3 Micro-architectural Analysis Methodologies

Today's out-of-order processors provide a rich set of hardware events for micro-architectural analyses. However, database workload characterization studies usually use a simple cache-miss-based micro-architectural analysis methodology (CMBM), which uses only a simple set of events that count the number of cache misses. Intel has recently announced Top-down Micro-architecture Analysis Method (TMAM). TMAM provides instruction-issue-slot-level breakdown by using new hardware events. Hence, it is able to account for an end-to-end execution time breakdown.

In this chapter, we compare the conventional CMBM with TMAM. Our goal is to see whether CMBM is successful enough to account for end-to-end execution time breakdown. The results show that CMBM is inadequate to account for the end-to-end execution time breakdown. CMBM under/over-estimates the number of stall cycles due to not accounting for all types of stalls and/or not being able to account for the overlapping capability that today's out-of-order processors have.

3.1 Introduction

Modern processors provide a performance monitoring unit (PMU) exposing hundreds of hardware events to examine the hardware-software interaction. The hardware events can be used to understand how efficiently a workload uses the processor resources [28]. On the other hand, modern processors are aggressively optimized for speed featuring wide-issue, out-of-order execution engines, deep cache hierarchies, deep pipeline stages and large pipeline buffers. While these features allow processors to exploit instruction- and memory-level parallelism, the increasing complexity of the micro-architecture makes the micro-architectural analysis harder.

Intel has recently announced its Top-down Micro-architecture Analysis Methodology (TMAM) proposing a hierarchical CPU cycles breakdown based on a set of new performance events [117]. TMAM examines every instruction issue slot independently. Hence, it provides an

accurate slot-level breakdown. Although TMAM is an ambitious methodology, it has not been adopted by the studies characterizing the micro-architectural behavior database workloads. The database workload characterization studies usually use a cache-miss-based methodology (CMBM) [98]. While it is conventional knowledge that database workloads spend most of their time in cache misses, it is not clear whether this assumption still holds, and if not what the other reasons for the stalls are when running database workloads.

In this chapter, we compare CMBM with TMAM for Online Transaction Processing (OLTP) workloads. We use one traditional disk-based system, *Shore-MT* [93], and two in-memory systems, *VoltDB* [115] and *HyPer* [39], running the standard TPC-C benchmark. Our goal is to estimate how well CMBM represents the micro-architectural behavior on today's aggressive out-of-order processors compared to TMAM. This chapter shows that:

- CMBM under-estimates stalls at the front-end due to: (i) not being able to account for the true penalty of instruction starvation, and (ii) not being able to account for instruction decoding stalls.
- CMBM under-estimates the back-end stalls for the disk-based system due to not being able to account for L1 hit stalls that cover the stalls due to complex micro-architectural features such as load-store forwarding and 4K aliasing. CMBM slightly over-estimates the back-end stalls for the in-memory systems due to not being able to account for the overlapping capability of today's out-of-order processors.

The rest of the chapter is organized as follows. Section 3.2 describes the micro-architectural analysis methodologies we examine. While Section 3.3 presents the experimental setup and methodology, Section 3.4 compares the analysis methodologies we discuss. Finally, Section 3.5 concludes.

3.2 Micro-architectural Analysis Methodologies

In this section, we explain the two methodologies we compare: CMBM and TMAM. We firstly cover how to estimate the fraction of the time spent for retiring instruction without any stalls, i.e., *Retiring* cycles. Retiring cycles is part of TMAM, but not part of CMBM. Hence, we cover it separately in the first section below. We, then, explain in detail how CMBM and TMAM estimates the stalls.

3.2.1 Retiring

In this section, we describe the *Retiring* time component. Retiring time represents the fraction of the time that CPU spends retiring instructions without any stalls.

Retiring time is found based on the hardware events that count the number of retired μOps

and that count the number of clock cycles. The used formula for Retiring time is as follows.

$$Retiring = \frac{NumberOfRetired\mu Ops}{IssueWidth \times NumberOfClockCycles}$$

The number retired μOps is divided by the multiplication of the issue width and the number of clock cycles. This is because the out-of-order processors can retire up to issue-width many μOps in one cycle. As a result, the Retiring time component describes how many number of μOps are retired, out of the maximum possible number of retired μOps given a certain number of clock cycles.

3.2.2 Cache-Miss-Based Methodology

Component		Description
FE		Stalls at front-end
	L1I	Stalls due to L1I misses
	L2I	Stalls due to L2I misses
	L3I	Stalls due to L3I misses
BE		Stalls at back-end
	L1D	Stalls due to L1D misses
	L2D	Stalls due to L2D misses
	L3D	Stalls due to L3D misses
Branch mispredictions		Stalls due to branch mispredictions

Table 3.1 – Stall cycles breakdown for CMBM for a three-level of cache hierarchy.

This section covers the cache-miss-based methodology (CMBM). Table 3.1 presents stall cycles breakdown for CMBM. CMBM decomposes stall cycles into three main components: front-end (FE) stalls, back-end (BE) stalls and stalls due to the branch mispredictions. It approximates FE stalls by the sum of L1, L2 and L3 instruction (L1I, L2I, L3I) cache miss stalls. It approximates BE stalls by the sum of L1, L2 and L3 data (L1D, L2D, L3D) cache miss stalls[1]. CMBM uses the conventional hardware events that count the number of instruction-/data-cache misses for every level of caches[2]. CMBM uses an estimated cache miss penalty for every level of caches, and multiplies the number of misses with the estimated penalty to obtain how many cycles the processor would stall due to the cache misses for each level of caches. The cache miss penalties are usually available at the optimization reference manual of the profiled processor. For example, Intel's cache miss penalties can be found in page 2-24 and B-42 of [42]. CMBM assumes L1I and L1D hits do not cause any stalls.

Lastly, CMBM estimates branch misprediction stalls based on the conventional hardware

[1] Based on a three level of cache hierarchy.

[2] Note that only L1 cache is split for instruction and data, whereas L2 and L3 caches are unified. However, Intel provides performance events counting the instruction and data misses separately for all the three levels of caches.

Component			Description
FE			Stalls at front-end
	Icache		Stalls due to instruction fetch starvation
	Decoding		Stalls due to inefficiencies at FE units such as decoder and μOp cache
BE			Stalls at back-end
	Dcache		Stalls due to memory accesses
		L1	Stalls due to L1D hits
		L2	Stalls due to L1D misses w/ L2 hit
		L3	Stalls due to L2D misses w/ L3 hit
		DRAM	Stalls due to L3D misses
		Store Buffer	Stalls due to store buffer overflow
	Resource/dependency		Stalls due to instruction dependencies or resource saturation
Branch misprediction			Stalls due to branch mispredictions

Table 3.2 – Stall cycles breakdown for TMAM for a three-level of cache hierarchy.

event counting the number of branch mispredictions. It uses an estimated penalty for a single branch misprediction. It multiplies the number of branch mispredictions with the estimated penalty to estimate how many cycles the processor would stalls for the mispredicted stalls. Branch misprediction penalty is usually available at the optimization reference manual of the profiled processor. For example, Intel's branch mispredicton penalty can be found in page B-47 of [42].

CMBM has been widely used by database workload characterization studies. However, today's wide-issue, out-of-order processors use sophisticated techniques to extract instruction- and memory-level parallelism, and have large pipeline buffers to overlap the stall cycles. Moreover, the true cache miss penalties may vary based on the way that instruction stream interacts with the micro-architecture. For example, if the total number of outstanding misses is larger than the total number of outstanding miss status handling registers, a cache miss can cost more than the estimated penalty. Therefore, simply counting the number of misses per cache level, and multiplying it by a constant pre-estimated penalty can potentially over/underestimate the number of stall cycles. Moreover, stall cycles due to reasons that are other than cache misses, such as resource/data dependencies, might constitute a significant portion of the execution time for different system, workload and configurations. Hence, it is quantify how realistic CMBM is compared to end-to-end execution time breakdown. In the next section, we examine Intel's TMAM which provide end-to-end execution time breakdown.

3.2.3 Top-down Micro-architectural Analysis Methodology

This section describes Top-down Micro-architectural Analysis Methodology (TMAM) [117]. Table 3.2 presents the stall cycles breakdown. TMAM decomposes the stall cycles into three main components at top-level: Front-End (FE) stalls, Back-End (BE) stalls and branch mis-

prediction stalls. TMAM uses Intel's new performance events to count the FE, BE and branch misprediction stalls in slot domain. At every CPU cycle, TMAM categorizes every instruction issue slot either used for retiring an instruction (described in Section 3.2.1), or stalled due to a FE-related issue, or stalled due to a BE-related issue, or stalled due to a branch misprediction.

TMAM breaks FE stalls down into two main components: Icache and decoding stalls. Icache stalls are the stalls due to instruction-cache misses. Decoding stalls are due to the inefficiencies at the FE units such as instruction decoder and μOp cache. TMAM breaks BE stalls down into two main components: Dcache and resource/dependency stalls. Dcache stalls are the stalls due to data-cache misses. Resource/dependency stalls are the stalls due to the dependencies in the instruction stream or saturation of a micro-architectural unit, such as an arithmetic/logic unit. Similar to the FE, BE and branch misprediction stalls, TMAM uses Intel's new performance events to count the Icache, decoding, Dcache and resource/dependency stalls in slot domain to provide a slot-level breakdown. Hence, Retiring, and Icache, decoding, Dcache, resource/dependency and branch misprediction stalls constitute slot-level breakdown of the CPU cycles that are used during the execution of the program. For example, if the processor is 4-issue, then TMAM provides the breakdown of the $4 \times NumberOfClockCycles$ CPU cycles.

TMAM further breaks Dcache stalls down into L1, L2, L3 and DRAM hit, and store buffer overflow stalls. L1 hits are usually free of charge. However, there can be L1 hit stalls due to issues at L1 data cache such as store to load forwarding, 4K aliasing and DTLB misses. L2, L3 and DRAM hit stalls represent the stalls due to L1, L2 and L3 data-cache misses that hit L2, L3 and DRAM, respectively. Store buffer overflow stalls are due to having extensive number of store operations, which overflows the micro-architectural structure called store buffer. In today's processors, store operations are buffered as they are mostly not on the critical path of the program. Store buffer overflow represents the stalls when the store buffer is full and is not able to buffer any more store operations.

3.3 Setup and methodology

This section presents the experimental setup and methodology.

Hardware: We run our experiments on an Intel Xeon processor Ivy Bridge micro-architecture. Table 4.1 shows the details of the micro-architecture. To collect hardware event numbers, we use Intel VTune Amplifier XE 2018 [44]. We use VTune's built-in `general-exploration` analysis that performs TMAM explained in this chapter. In Table A.1, Section A.1 of the Appendix, we provide how each individual component that `general-exploration` analysis reports maps to the component that we use. We disable hyper-threading to obtain more precise hardware sampling values and increase predictability in measurements.

Processor	Intel(R) Xeon(R) CPU E5-2640 v2 (Ivy Bridge)
#Sockets	2
#Cores per Socket	8
Hyper-Threading	Off
Clock Speed	2.00GHz
Memory	256GB
L1I / L1D (per core)	32KB / 32KB 8-cycle miss latency
L2 (per core)	256KB 19-cycle miss latency
L3 (shared)	20MB 167-cycle miss latency

Table 3.3 – Server parameters.

OS: We run all the experiments using RHEL 6.5 with Linux kernel version 2.6.32.

Benchmarks: We use standard TPC-C benchmark [113] with a database of size 20GB for all the experiments.

Analyzed database management systems: We focus on Online Transaction Processing (OLTP) workloads. We analyze one traditional disk-based system, *Shore-MT* [93], and two in-memory OLTP systems, *VoltDB* [115] (Community Edition Version 4.8), and *HyPer* [39] (online demo-version). For all the systems, we use asynchronous logging so that there is no delay due to I/O in the critical path of the execution. We choose these three systems as they are well-known in the community and their design characteristics represent a good variety of today's OLTP systems. *Shore-MT* is a disk-based system possessing large amount of overheads such as buffer pool and disk-oriented index structures. *VoltDB* and *HyPer* are in-memory systems eliminating most of the overheads the disk-based systems possess. Both *VoltDB* and *HyPer* rely on physical data partitioning. While *VoltDB* uses a traditional B-tree with node size tuned to the last-level cache line size [104], *HyPer* implements adaptive radix tree with adaptive compact node sizes [66]. Furthermore, while *HyPer* compiles transactions into the machine code [80], *VoltDB* uses stored procedures without compiling the stored procedures into the machine code.

Measurements: We populate the databases from scratch before each experiment and the data remains memory-resident throughout the experiments. We use memory-mapped I/O for log flushing. Both the worker threads executing the transactions and the client threads generating the transactions run on the same machine. We first start the server process, populate the database, and then start the experiment by launching the clients that generate and submit transactional requests to the database server. For a client submitting transactional requests

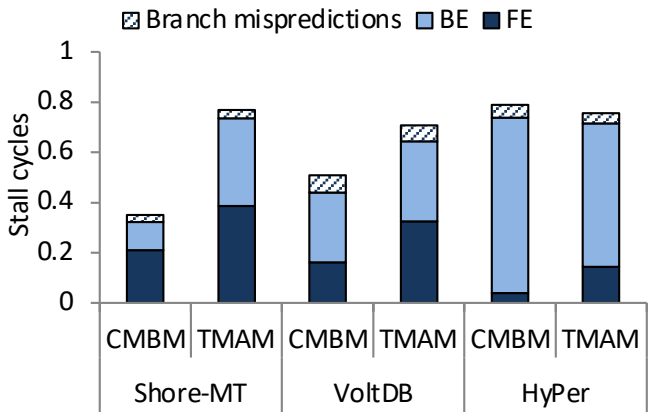


Figure 3.1 – Stall cycles breakdown at top level for CMBM and TMAM. FE: front-end, BE: back-end.

there is one OLTP worker thread satisfying the requests. Less than 1% of the execution time is spent for the client threads, whereas the remaining 99% of the time is spent for the server threads. We profile the database server process by attaching VTune to it during a 120-second benchmark run following a 60-second warm-up period. We repeat every experiment three times and report the average result. We observe a standard deviation of less than 5% for the three repeats.

We run only single-threaded experiments as it has been shown that micro-architectural behavior of OLTP workloads does not change significantly across single- and multi-threaded runs [98].

3.4 Experimental Evaluation

This section presents the experimental evaluation. We firstly compare the high-level micro-architectural behavior that CMBM and TMAM provide. Then, we compare CMBM and TMAM at their front-end (FE) and back-end (BE) breakdowns.

3.4.1 High-level analysis

In this section, we compare CMBM and TMAM at their top-level breakdowns. Our goal is to investigate how well CMBM represents the high-level micro-architectural behavior compared to TMAM.

Figure 3.1 compares the number of stall cycles of CMBM and TMAM. As TMAM provides instruction-issue-slot-level breakdown, the number of stall cycles calculated out of TMAM's formulas are normalized with respect to the number of issue slots times the number of clock cycles, i.e., $IssueWidth \times NumberOfClockCycles$. CMBM, however, provides a cycle-level breakdown. Therefore, the number of stall cycles calculated out of CMBM's formulas are normalized with respect to the number of clock cycles.

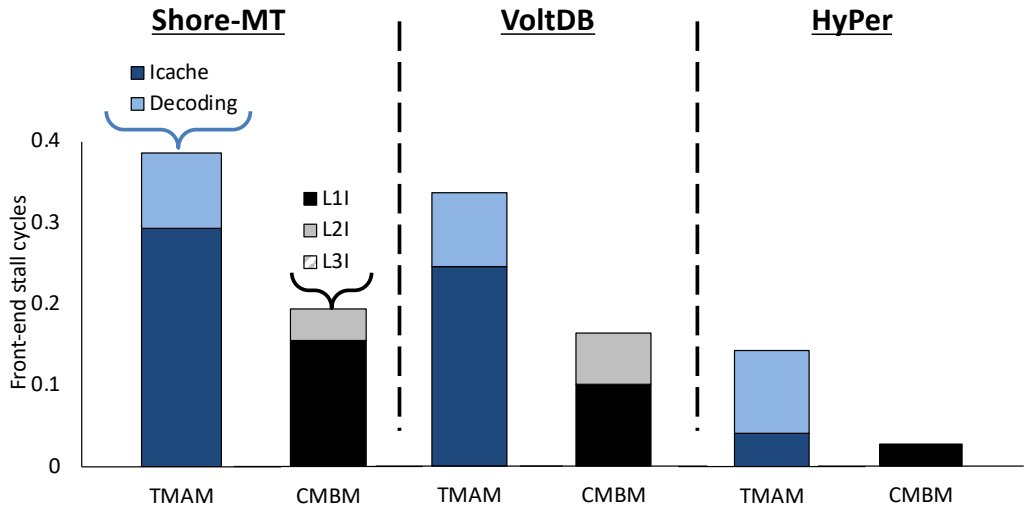


Figure 3.2 – Stall cycles breakdown at FE for TMAM and CMBM.

The figure shows that, for *Shore-MT* and *VoltDB*, CMBM significantly under-estimates the number of stall cycles. This shows that CMBM is not sufficient to represent all the stalls. CMBM slightly over-estimates the number of stall cycles for *HyPer*. To understand the reasons behind these differences, we examine FE and BE breakdowns for CMBM and TMAM in the next two sections.

3.4.2 Front-end stalls

This section compares the FE breakdowns of CMBM and TMAM. Our goal is to identify the reasons for the differences between the two methodologies. We focus on level 2 of TMAM breakdown, where level 1 is the top-level breakdown. Figure 3.2 shows the results. Figure 3.2 contains two bars for each OLTP system we profile: one bar for the breakdown of CMBM and one bar for the breakdown of TMAM. As the two breakdowns have different stall components, the two bars have different legends. We present the legend only for a single system (*Shore-MT*) as it is the same for the other systems.

The figure shows that TMAM's Icache stalls are significantly higher than the sum of the L1, L2 and L3 instruction stalls of CMBM for all the three systems. This shows that CMBM underestimates the instruction starvation stalls. This can be due to several reasons such as that the instruction prefetcher might not be working optimally such that the L1 instruction hits have a certain amount of penalty rather than being a zero-penalty operation as CMBM assumes. Furthermore, CMBM is not able to account for about 10% of Decoding time. As a result, CMBM significantly underestimates the overall time spent at FE.

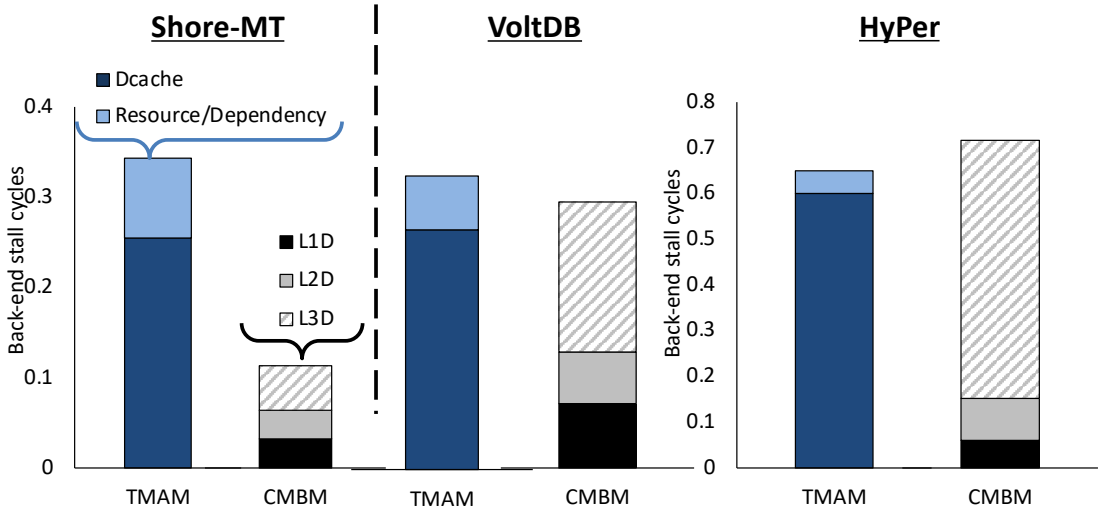


Figure 3.3 – Stall cycles breakdown at BE for CMBM and the optimized TMAM. FE: front-end, BE: back-end, misc: miscellaneous.

3.4.3 Back-end stalls

This section compares CMBM with TMAM at the BE. Figure 3.3 presents the results. We focus on level 2 of TMAM breakdown, where level 1 is the top-level breakdown. Figure 3.3 follows the same format Figure 3.2 follows. It has two bars per system, presenting the legend only once (for *Shore-MT*). The figure shows that TMAM's Dcache stalls are higher than the sum of L1, L2 and L3 data-cache misses of CMBM for *Shore-MT*. The reason is that CMBM does not include the L1 hit stalls component that covers the stalls due to complex micro-architectural features such as load-store forwarding and 4K aliasing.

TMAM's Dcache stalls are a little lower than the sum of L1, L2 and L3 data-cache stalls of CMBM for *VoltDB* and *HyPer*. The reason is that TMAM accounts for the overlapping capability of today's processors. L1 hit stalls constitute a smaller fraction of the Dcache stalls for *VoltDB* and *HyPer*. *Shore-MT* is a traditional disk-based system, whereas *VoltDB* and *HyPer* in-memory optimized OLTP systems with significantly higher throughput than *Shore-MT*. Hence, *VoltDB* and *HyPer*'s data access patterns are less complicated as they do not deal with heavy-weight disk-based meta-data such as centralized lock manager's lock table and complex buffer manager meta data. Hence, L1 hit stalls are less of a problem for *VoltDB* and *HyPer*. Nevertheless, as we aim to profile different generations of database system, it is important for the micro-architectural methodology we use to be robust against profiling different categories of systems, which renders TMAM more suitable for our goal.

3.5 Conclusion

This chapter compares the micro-architectural analysis methodologies: conventional cache-miss-based methodology (CMBM) and Intel's recently announced Top-down Micro-architecture

Analysis Methodology (TMAM). The results show that CMBM under-estimates the stalls at front-end due to: (i) not being able to account for the true penalty of instruction starvation, and (ii) not being able to account for instruction decoding stalls. CMBM under- or over-estimates the back-end stalls. CMBM under-estimates the back-end stalls for the disk-based system due to not being able to account for L1 hit stalls. CMBM slightly over-estimates the back-end stalls for the in-memory systems due to not being able to account for the overlapping capability of modern processors.

As CMBM under-estimates the stalls at front-end stalls, and is not robust against different database management systems at back-end, we conclude that CMBM is inadequate to represent the end-to-end execution time breakdown for different generations of database management systems. Hence, we choose TMAM as our micro-architectural analysis methodology for the rest of the thesis.

4 Online Transactional Processing Workloads

Micro-architectural behavior of traditional disk-based online transaction processing (OLTP) systems has been investigated extensively over the past couple of decades. Results show that traditional OLTP systems mostly under-utilize the available micro-architectural resources. In-memory OLTP systems, on the other hand, process all the data in main-memory, and therefore, can omit the buffer pool. Furthermore, they usually adopt more lightweight concurrency control mechanisms, cache-conscious data structures, and cleaner codebases since they are usually designed from scratch. Hence, we expect significant differences in micro-architectural behavior when running OLTP on platforms optimized for in-memory processing as opposed to disk-based database systems. In particular, we expect that in-memory systems exploit micro architectural features such as instruction and data caches significantly better than disk-based systems.

This chapter sheds light on the micro-architectural behavior of in-memory database systems by analyzing and contrasting it to the behavior of disk-based systems when running OLTP workloads. The results show that despite all the design changes, in-memory OLTP exhibits very similar micro-architectural behavior to disk-based OLTP systems: more than half of the execution time goes to memory stalls where instruction-cache misses or the long-latency data misses from the last-level cache (LLC) are the dominant factors in the overall execution time. Even though ground-up designed in-memory systems can eliminate the instruction-cache misses, the reduction in instruction stalls amplifies the impact of LLC data misses. As a result, only 30% of the CPU cycles are used to retire instructions, and 70% of the CPU cycles are spent for stalls for both disk-based and in-memory OLTP.

4.1 Introduction

Recent years have witnessed the rise of in-memory or main-memory optimized OLTP systems [25, 76, 115]. Traditional OLTP engines are disk-based since they are designed in an era where the server hardware had a main-memory size in megabytes. Today, however, a server hardware with 1TB main-memory is a commodity. Therefore, the database management systems (DBMSs) are able to process the data working set of most OLTP applications in memory. This

has led various vendors and researchers to design brand new OLTP engines optimized for the case where the hot dataset resides in memory [55, 61, 64, 107].

In-memory OLTP systems have significant differences compared to disk-based systems. First, since the data working set resides mostly in memory, in-memory OLTP systems omit the buffer pool component, which acts as the virtual memory of a DBMS and is, therefore, essential for the disk-based systems. Then, they tend to adopt more lightweight concurrency control mechanisms to avoid the scalability bottlenecks that arise due to traditional centralized locking. They also opt for cache-conscious indexes instead of the disk-optimized B-trees. Finally, since their codebases are written from scratch, they tend to have lighter storage engines.

OLTP benchmarks are famous for their suboptimal micro-architectural behavior. There is a large body of work that characterizes OLTP benchmarks at the micro-architectural level [11, 28, 53, 90, 103, 108, 111]. They all conclude that OLTP exhibits high stall time (> 50% of the execution cycles), and a low instructions-per-cycle (IPC) value (< 1 IPC on machines that can retire up to 4 instructions in a cycle) [28]. The instruction-cache misses are the main source of the stall time, while the next contributing factor is the long-latency data misses from the last-level cache (LLC) [108].

All the previous workload characterization studies, however, run the OLTP benchmarks on a disk-based OLTP engine. Considering the lighter components, cache-friendly data structures, and cleaner codebase of in-memory systems, one expects them to exhibit better cache locality (especially for the instruction cache) and less memory stall time. Due to the distinctive design features of the in-memory systems from the disk-based ones, however, it is not straightforward to extrapolate how OLTP benchmarks behave at the micro-architectural level when run on an in-memory engine solely by looking at the results of previous studies.

In this chapter, we perform a detailed analysis of the micro-architectural behavior of the in-memory OLTP systems. More specifically, we compare three in-memory OLTP systems (an in-memory OLTP engine of a popular commercial vendor, a ground-up designed in-memory OLTP system, and an open source OLTP engine, Silo [114]) to two disk-based OLTP systems (a popular commercial DBMS and an open source OLTP engine, *Shore-MT* [93]). We examine execution time breakdown at the hardware-level, normalized throughput, and memory bandwidth consumption while running simple micro-benchmarks as well as the more complex TPC benchmarks (TPC-B and TPC-C) [113]. Our analysis demonstrates the following:

- Despite all the design differences, in-memory OLTP spends more than half of the execution time waiting for instruction-cache or data-cache misses, similar to disk-based OLTP.
- The disk-based OLTP system and its in-memory OLTP engine suffer mainly from instruction-cache misses. Even though the in-memory optimized OLTP engine delivers significantly higher throughput due to the optimizations it uses, the main micro-architectural bottle-

neck remains the instruction-cache misses both for the disk-based OLTP system and its in-memory OLTP engine.

- Ground-up designed in-memory OLTP systems do not suffer from instruction-cache misses. However, they spend more than half of their execution time in data-cache misses due to the random data-accesses the OLTP workloads do. As a result, their CPU utilization characteristics remain close to the disk-based OLTP system.

The rest of the chapter is organized as follows. Section 4.2 describes the experimental methodology. Section 4.3 and Section 4.4 present the analysis results with a micro-benchmark and TPC benchmarks, respectively. Section 4.5 analyzes the effects of transaction compilation, index structures, and data types, whereas Section 4.6 investigates the impact of multithreading on the micro-architectural behavior. Section 4.7 present the analysis of the memory bandwidth consumptions. Section 4.8 analyzes the acceleration features such as hyper-threading, turbo-boost and hardware prefetchers. Section 4.9 compares two latest generations of Intel's successive micro-architectures. Finally Section 4.10 concludes.

4.2 Setup and Methodology

The experiments presented in this chapter are executed on real hardware and performance is measured using event counters as opposed to hardware simulators since we are not investigating the impact of changing some of the hardware parameters on the micro-architectural behavior. The rest of this section details the setup and methodology for our study.

Hardware: We run experiments on a modern commodity server with Intel's Broadwell processors. Table 4.1 shows the architectural details of this server. To collect numbers about various hardware events and break down the time spent in specific code modules, we use Intel VTune Amplifier XE 2018 [44], which provides an API for lightweight hardware counter sampling. We disable hyper-threading and turbo-boost to obtain more precise hardware sampling values and increase predictability in measurements.

OS & Compiler: We use Ubuntu 16.04.6 LTS and gcc 5.4.0 on the Broadwell server.

Benchmarks: We run two types of benchmarks: micro-benchmarks and TPC benchmarks [113]. Our goal is to perform sensitivity analysis and have a more detailed understanding of the systems using the micro-benchmark, while the experiments using the TPC benchmarks serve to give an idea about the behavior of the systems when running well-known real-world applications.

The micro-benchmark uses a randomly generated table with two columns (`key` and `value`) of the type Long. It has two versions: read-only and read-write. The read-only version reads N random rows from the table, whereas the read-write version updates N random rows. Both versions use an index lookup operation on the randomly picked `key` value to reach the row to be read or updated. We also use a modified version of the micro-benchmark where we use

Processor	Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell)
#Sockets	2
#Cores per Socket	14
Hyper-threading	Off
Turbo-boost	Off
Clock Speed	2.40GHz
Bandwidth (per socket)	66 GB/s
L1I / L1D (per core)	32KB / 32KB 16-cycle miss latency
L2 (per core)	256KB 26-cycle miss latency
LLC (shared)	35MB 160-cycle miss latency
Memory	256GB

Table 4.1 – Server Parameters

strings of 50 bytes for both columns to quantify the impact of data type on micro-architectural utilization in Section 4.5.2. As for the TPC benchmarks, we use TPC-B and TPC-C. We do not study range lookups as we focus on TPC-B- and TPC-C-like random point queries.

Analyzed Systems: We analyze three in-memory OLTP systems: the in-memory OLTP engine of a closed-source commercial vendor (*DBMS M*), an open-source commercial OLTP system (*DBMS N*) and an open-source OLTP engine (*Silo* [114]).

We pick these three systems as they are well-known in the community and their design characteristics represent a good variety among today's in-memory OLTP systems. While *DBMS M* adopts multiversioned concurrency control, *DBMS N* uses physical data partitioning, and *Silo* uses optimistic concurrency control. *DBMS M* implements both hash index and a variant of cache-conscious B-tree index similar to [67, 68]. *DBMS N* uses a variant of a self-balancing binary search tree, red-black tree. *Silo* implements Masstree, a highly parallel in-memory index structure [75]. For *DBMS M*, we use the B-tree index. Moreover, *DBMS M* use transaction compilation techniques for the stored procedures, whereas *DBMS N* and *Silo* do not.

In order to gain better insights about the differences between the in-memory and disk-based OLTP systems, we also include two disk-based systems: a popular, commercial system (*DBMS D*) and the open-source *Shore-MT* [93] storage manager.

To implement benchmarks, we use the SQL frontend of the commercial systems, *DBMS D*, *DBMS M* and *DBMS N*, and *Silo*'s benchmarks in C++, *Shore-MT*'s Shore-Kits suite that provides an environment to implement benchmarks for *Shore-MT* in C++.

For all the systems, we use asynchronous logging. Therefore, there is no delay due to I/O in the critical path of the transaction execution.

Measurements: We populate the databases from scratch before each experiment and the data remains memory-resident throughout the experiment. In the following sections, we indicate the database sizes used in each experiment before discussing the results. In our experiments, both the database server process executing the transactions and the client processes generating the transactions run on the same machine. We first start the server process, populate the database, and then start the experiment by simultaneously launching all clients that generate and submit transactional requests to the database server.

We profile the database server process by attaching VTune to it during a 120-second benchmark run following a 60-second warm-up period. We repeat every experiment three times and report the average result.

In terms of micro-architectural efficiency, our goal is to observe how well each system exploits the resources of a single core regardless of the parallelism in the system. Therefore, all the experiments except for the ones in Section 4.6 and Section 4.7, use a single worker thread executing the transactions of the corresponding benchmark.

The choice of a single worker thread also eliminates contention due to several threads trying to access the shared data in the case of non-partitioned systems and distributed transactions in the case of partitioning-based systems. This way we avoid possible misleading micro-architectural conclusions. For example, high contention for a shared data page could lead to multiple threads spinning on a latch for that data page, thus artificially increasing the cache hit ratio.

We use one client to generate request in the single-threaded experiments. *Shore-MT*, *DBMS D*, *Silo*, and *DBMS M* assign one worker thread per client. *DBMS N*, on the other hand, generates one worker thread per data partition, so we configure it to have only one partition. From VTune, we filter the hardware counter results particularly for the identified worker thread excluding the other threads that are responsible for background tasks, e.g., communication between the server and client, parsing transactions, etc.

In multi-threaded experiments (Section 4.6 and Section 4.7), we use multiple clients to generate requests for all systems. For *DBMS N*, we also use multiple data partitions and ensure that all transactions access only a single partition. For each system, we gradually increase the number of clients, and profile the execution with the number of clients that give the highest aggregate throughput. From VTune, we filter hardware counter results for each worker thread separately and report their average.

VTune: We use Intel VTune 2018. We use VTune's built-in general-exploration analysis that performs Intel's Top-down Micro-architectural Analysis Methodology (TMAM) explained in Chapter 3. We use VTune's built-in memory-access analysis to measure the consumed memory bandwidth. As we numa-localize our experiments on a single socket, we report average bandwidth per-socket values. We use VTune's built-in advanced-hotspots analysis to perform function call trace breakdown.

We provide an overview of Intel's TMAM explained in Chapter 3. Each instruction issue slot is categorized into one of two components: retiring and stalling. A retiring slot is a slot where the slot is used for retiring an instruction. A stalling slot is a slot where the slot stalls, i.e., has to wait due to a particular issue. Ideally, all issue slots would be used for retiring. Stalling slots are further decomposed into five components: (i) branch misprediction, (ii) Icache, (iii) decoding, (iv) Dcache and (v) resource/dependency stalls. Branch misprediction stalls are the slots that stall due mispredicted branch instructions. Today's processors use a hardware unit called branch predictor; it predicts the outcome of a branch instruction (i.e., an if() statement) and speculatively executes instructions per the predicted branch direction and/or target. If the processor then realizes the prediction is not correct, it undoes whatever it has been doing and starts executing the correct set of instructions. This cost is defined as the branch misprediction stalls and can be very costly, as it requires canceling a large amount of work. Icache stalls are the slots that stall due to instruction-cache and instruction translation lookaside buffer misses. Decoding stalls are the slots that stall due to sub-optimal micro-architectural implementation of the instruction decoding unit. Dcache stalls are the slots that stall due to data-cache misses. Resource/dependency stalls are the slots that stall due to resource and/or data dependencies. For example, if two instructions require using the same arithmetic-logic unit, one has to wait for the other. This time is identified as the resource/dependency time. Or, if an instruction's operand depends on the result of another instruction, the instruction with the dependent operand has to wait for the other instruction to finish. This time is identified as the resource/dependency time.

4.3 Micro-benchmark

Before performing an analysis using the community standard TPC benchmarks, we devise a sensitivity study on the micro-architectural behavior of in-memory OLTP systems using the micro-benchmark. The goal of this study is to answer the following questions:

- Where do CPU cycles go when running in-memory OLTP? Are they wasted on memory stalls or used to retire instructions?
- Where do memory stalls come from? Are they mainly due to instructions or data for in-memory OLTP?
- What is the impact of the database size on the above metrics?
- Does the amount of work done per transaction affect the results and, if yes, how?

To answer these questions, we break the analysis into two parts. The first part (Section 4.3.1) varies the database size by varying the number of rows in the table while keeping the amount of work done per transaction constant. On the other hand, the second part (Section 4.3.2) varies the amount of work done per transaction by increasing the number of rows read in a transaction while keeping the database size constant.

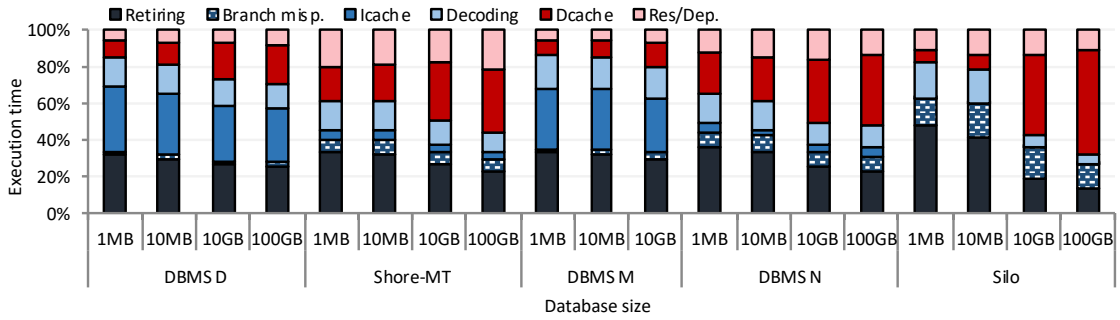


Figure 4.1 – Execution time breakdowns as we increase the database size when running the read-only micro-benchmark.

Both parts examine execution time breakdowns at the hardware-level and normalized throughput values as explained in Section 4.2.

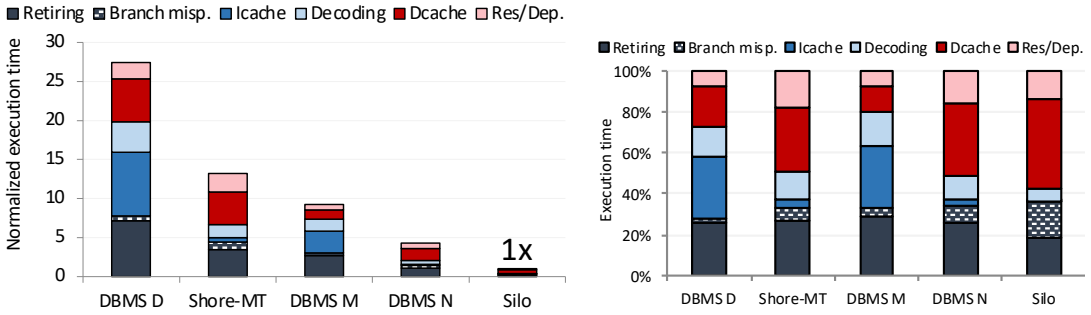
4.3.1 Sensitivity to Data Size

To investigate the impact of database size on the micro-architectural behavior, we populate databases of size 1MB, 10MB, 10GB, and 100GB. *DBMS M* has the limitation of 32GB on the maximum size of a database. Hence, we use maximum of 10GB of database for *DBMS M*. As the micro-architectural and performance behavior of *DBMS M* do not change significantly as the database size varies (see Figure 4.1 and Table 4.2), we rely on the results for 10GB of data to interpret *DBMS M*'s micro-architectural and performance behavior for a large data size. Then, we collect hardware events as the systems run the micro-benchmark with a single transaction type that just reads/updates one random row after an index probe operation. While the results for the read-only version of the micro-benchmark are in the following sub-sections, the results for the read-write version of the micro-benchmark are in Section A.2.1 of the Appendix.

Execution Time Breakdown at the Hardware-Level

Figure 4.1 shows the breakdown of the execution times, as the database size is increased. The retiring cycles ratios are similar for databases of sizes 1MB and 10MB since the data working set mainly fits in the last-level cache (LLC), which is 35MB (see Table 4.1). As we increase the database size to 10GB and 100GB, the retiring times are decreased since the data working set no longer fits in caches and the long-latency data misses become more significant. All the retiring times are less than 30% for a database of size 10GB and 100GB for all the systems. Hence, in-memory OLTP systems spend most of their time in stalls similar to disk-based systems.

Shore-MT is bound by Dcache and resource/dependency stalls. *Shore-MT* is a disk-based system. Existing work on *Shore-MT* has long shown that *Shore-MT* is Icache-stalls-bound [98, 99, 108, 111]. We have examined the existing work on *Shore-MT* and observed that all machines used in the existing work is Intel's, version 2, Ivy Bridge micro-architecture. We, on



(a) Normalized execution time breakdowns. Execution times are normalized to Silo's execution time.

(b) Execution time breakdowns streched to 100%.

Figure 4.2 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for 10GB of data when running the read-only micro-benchmark.

the other hand, use a later-generation version 4 micro-architecture, Broadwell, which is the slightly improved version of the version 3 Haswell micro-architecture. Intel has announced an important micro-architectural improvement on the instruction fetch unit of the Haswell micro-architecture [31]. As Hammarlund et al. [31] specifies, "State-of-the-art advances in branch prediction algorithms enable accurate fetch requests to run ahead of micro-operation supply to hide instruction TLB and cache misses.". Hence, instruction fetch unit keeps supplying instructions even though there is an instruction-cache miss, which allows overlapping the instruction-cache miss latency with useful work. As a result, *Shore-MT*, being a long-standing-Icache-stalls-bound system has become Dcache- and resource/dependency-stalls-bound. We compare Ivy Bridge and Broadwell for all the systems we analyze in Section 4.9.

Shore-MT spends 40% of its time in Dcache and resource/dependency stalls even when the data size is small. We examine *Shore-MT*'s call stack and observe that most of the Dcache and resource/dependency stalls are due to meta-data processing such as looking into the hash table that keeps track of the buffer pool pages and acquiring a lock. As the data size exceeds the LLC size, *Shore-MT* becomes more and more Dcache-bound due to its working set not fitting into the LLC and suffering from expensive LLC misses.

DBMS D and *DBMS M* suffer from high Icache stalls. However, *DBMS M*'s throughput is three times that of *DBMS D*'s (see Table 4.2). Hence, the optimizations that *DBMS M* adopts help in delivering a throughput higher than *DBMS D* delivers. Figure 4.2 (left) shows the breakdown of the normalized execution time, where the execution times are normalized to *Silo*'s execution time. The figure shows that *DBMS M* is faster than *DBMS D* for several reasons. First, it retires significantly fewer instructions than *DBMS D*. Second, it suffers significantly less from Icache stalls. Lastly, it suffers from Dcache stalls significantly less than *DBMS D*. Overall, these improvements enable an execution time that is 3x lower. Nevertheless, both *DBMS D* and *M* suffer mainly from Icache stalls. Figure 4.2 (right) presents the breakdown of the execution times stretched to 100%. Both *DBMS D* and *M* spend 30% of their time waiting for Icache stalls, thus showing the severe effects of instruction-cache misses for the commercial disk-based system and its in-memory OLTP engine.

Figure 4.2 (left) shows that *DBMS D* spends 7.2 units of time executing instructions and 8.2 units of time waiting for Icache stalls. *DBMS M* spends 2.7 units of time executing instructions and 2.8 units of time waiting for Icache stalls. Hence, though *DBMS M* spends less time waiting for Icache stalls than *DBMS D* spends, *DBMS D* and *M* spend a similar amount of time waiting for Icache stalls per instruction executed: 8.2/7.2 = 1.14 for *DBMS D* and 2.7/2.8 = 1.04 for *DBMS M*. This finding corroborates with the finding in Figure 4.2 (right), where both *DBMS D* and *M* spend a similar percentage of their time waiting for Icache stalls.

DBMS N suffers from high Dcache and resource/dependency stalls. *DBMS N* is an in-memory system, which eliminates the heavy weight disk-based system components such as buffer pool and lock manager. Nevertheless, it suffers significant amount of Dcache and resource/dependency stalls for small data sizes, similar to *Shore-MT*. We examine *DBMS N*'s call stack and observe that the Dcache and resource/dependency stalls are largely due to the meta-data processing that *DBMS N* uses for setting up and instantiating the transactions. As the data size is increased, *DBMS N* becomes more and more Dcache-stalls-bound due to its working set not fitting into the LLC and suffering from expensive LLC misses.

DBMS N is ~2x faster than *DBMS M* as shown by Figure 4.2 (left). This is due to the reduced number of executed instructions and to the elimination of the Icache stalls. *DBMS M* spends 2.7 units of time executing instructions, whereas *DBMS N* spends 1.1 units of time executing instructions. *DBMS M* spends 2.8 units of time waiting for Icache stalls, whereas *DBMS N* spends 0.14 units of time waiting for Icache stalls. Hence, *DBMS N* executes ~60% fewer instructions and spends time waiting for Icache stalls 95% less than *DBMS M* spends. As a result, *DBMS N* delivers ~2x higher throughput than *DBMS M* delivers.

Silo has high retiring time for small data sizes. *Silo* is an in-memory system eliminating the meta-data processing that disk-based systems would require. Moreover, *Silo* is a kernel OLTP engine that has transactions hard-coded in C++. Hence, it does not suffer from the meta-data that would require instantiating and scheduling user requests as *DBMS N* does. As a result, it does not suffer from Dcache stalls when the data size is small, and it has high retiring time for small data sizes. As the data size exceeds the LLC size, *Silo* becomes mostly Dcache-stalls-bound such that *Silo*'s retiring time is the lowest among all the systems we analyze. The reason is, once again, that *Silo* eliminates the meta-data processing that *Shore-MT* and *DBMS N* do. As a result, *Silo*'s data access pattern mostly includes what the micro-benchmark dictates. As the micro-benchmark randomly reads one row per transaction, it results in large number of LLC misses, and hence the Dcache stalls.

Silo also suffers, in addition to the Dcache stalls, from a significant amount of branch misprediction. This is due to the in-node searches that *Silo* performs during the index traversal [1]. Although *DBMS N* also performs an index traversal during its transaction processing, branch misprediction stalls only surface up on *Silo* due (i) to *Silo* being a kernel OLTP engine that does not perform the meta-data processing to setup and instantiate transactions that *DBMS N*

[1] Silo uses linear search as its in-node search algorithm.

	1MB	10MB	10GB	100GB
DBMS D	1	1	1	1
Shore-MT	3.0	2.9	2.1	1.6
DBMS M	2.8	3.0	3.0	-
DBMS N	7.8	7.8	6.4	3.9
Silo	75.4	62.9	27.5	19.3

Table 4.2 – Normalized throughput as we increase the database size. Throughputs are normalized with respect to the throughput of *DBMS D* for each database size individually.

performs, and (ii) to *Silo* using an efficient index structure. We examine *DBMS N*'s call stack and observe that *DBMS N* spends 35% of its time setting up and instantiating transactions. It makes function calls such as `processInitiateTask()`, `xfer()` (dequeues user requests) and `coreExecutePlanFragments()`. *DBMS N* spends most of the remaining 65% of its time performing the index lookup. We examine *DBMS N* and *Silo*'s index structure in Section 4.3.2.

Figure 4.2 shows that *Silo* uses significantly fewer instructions than *DBMS N* uses and suffers from significantly fewer Dcache stalls. As a result, it is 5x faster than *DBMS N* is. This shows that new-generation in-memory systems can be even faster by reducing the number of instructions they use and by reducing the amount of Dcache stalls. We examine *Silo* and *DBMS N*'s call stacks and observe that *Silo*'s reduced Dcache stalls are mostly due to the fact that *Silo* uses a more efficient index structure. We compare the index structures used by *DBMS N* and *Silo* in Section 4.3.2 in more detail.

We observe that all the systems suffer 10-15% decoding stalls. Decoding stalls are the penalties in the instruction decoding unit of the processor. As Intel relies on a Complex Instruction Set Computer (CISC) type of microprocessor design, it decodes instructions into micro-operations. A small amount of decoding stalls show that decoding stalls do not constitute a significant problem for OLTP workloads.

Normalized Throughput

Table 4.2 shows normalized throughputs for each system as the database size is increased. The throughput values are normalized with respect to the throughput of *DBMS D* for each database size individually. The relative performance between *DBMS D* and *M* remains stable. This is because both *DBMS D* and *M* are Icache-stalls-bound, and hence the increased data size does not significantly affect their performance. *Shore-MT*, *DBMS N* and *Silo*'s relative performance, on the other hand, is decreased as the data size is increased. This is because *Shore-MT*, *DBMS N* and *Silo* are Dcache and resource/dependency-stalls-bound. The increased data size results in a more substantial drop in their throughput than *DBMS D* and M. As a result, *Shore-MT*, *DBMS N* and *Silo*'s throughput get close to *DBMS D* and *M* as the data size is increased.

All the in-memory systems are faster than the disk-based systems for all the data sizes. This shows that the optimizations that in-memory systems implement significantly help improving the throughput. *DBMS M*, despite suffering from Icache stalls is faster than *Shore-MT* for 10GB

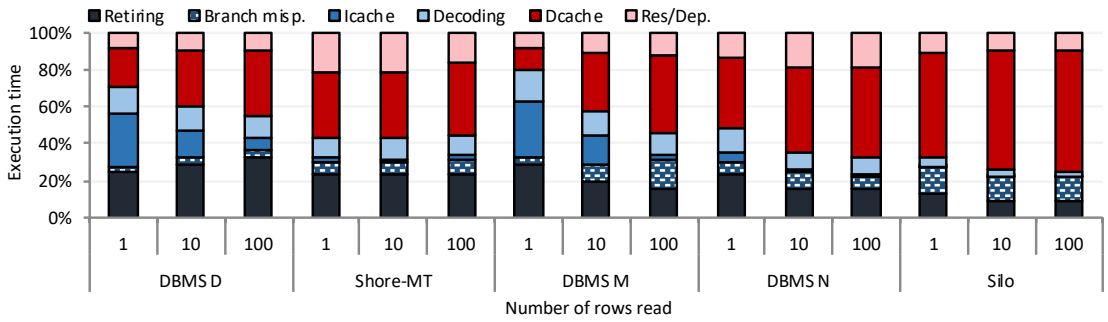


Figure 4.3 – Execution time breakdowns as we increase the amount of work done per transaction with a database of size 100GB (10GB for DBMS M).

of database. As *Shore-MT* mainly suffers from Dcache stalls, this shows the severe negative effects of disk-based systems' index structure overhead, such as large index and buffer pool pages.

DBMS M is 50% slower than *DBMS N* is. This is because *DBMS M* executes larger number of instructions and suffers from significantly larger amount of Icache stalls than *DBMS N*. As shown by Figure 4.2 (right), *DBMS M* executes 2.5x larger number of instructions and spends 20x more time waiting for Icache stalls. Hence, *DBMS M* can improve its performance by reducing the number of executed instructions, and by using the instruction caches more efficiently. *Silo* is the fastest system we have profiled. One reason for that is *Silo* is a kernel OLTP engine that hard-codes transactions in C++. Hence, it does not suffer from the cost of setting up and instantiating the transactions. *DBMS N*, on the other hand, is an end-to-end SQL-based OLTP system. Another reason is that *Silo* uses an efficiently implemented index structure, Masstree [75]. As a result, it is 4.9x faster than *DBMS N* for 100GB of database. We examine the inefficient index structure issue of *DBMS N* in Section 4.3.2 in more detail.

Summary

Relative throughput of OLTP systems widely vary among different categories of OLTP systems. However, CPU cycles utilization of all the OLTP systems are low. *DBMS D* and *M* suffer from Icache stalls. *Shore-MT*, *DBMS N* and *Silo* eliminate the Icache stalls. The reduced Icache stalls cause Dcache stalls to surface up, which *Shore-MT*, *DBMS N* and *Silo* suffer from. The Dcache stalls are mostly due to the random-data-access nature of the workload, in addition to the meta-data processing overhead for *Shore-MT* and *DBMS N*. As a result, *Shore-MT*, *DBMS N* and *Silo* spend only 30% of the CPU cycles retiring instructions similar to *DBMS D* and *M*.

4.3.2 Sensitivity to Work per Transaction

To investigate the impact of the amount of work per transaction on the micro-architectural behavior, we increase the number of rows that a transaction accesses from 1 to 10 and then to 100. We perform these experiments with 100GB dataset for all the systems except *DBMS M*.

	1 row	10 rows	100 rows
DBMS D	1	1	1
Shore-MT	1.6	0.7	0.7
DBMS M	3.0	3.7	3.9
DBMS N	3.9	2.1	1.7
Silo	19.3	8.0	6.0

Table 4.3 – Normalized throughput as we increase the amount of work done per transaction with a database of size 100GB (10GB for *DBMS M*). Throughputs are normalized to *DBMS D*.

We use 10GB of database for *DBMS M* due to its 32GB of maximum database size limitation.

In the following sub-sections, we present the results for the read-only version of the micro-benchmark. The results for the read-write version of the micro-benchmark can be found in Section A.2.2 of the Appendix.

Figure 4.3 shows the breakdown of the execution times for each system, as the amount of work per transaction is increased. As we increase the amount of work per transaction, *DBMS D* and *M* suffer less and less from Icache stalls. The repetitive behavior within a transaction leads to a better instruction-cache locality. As a result, Icache stalls are decreased as the amount of work per transaction is increased. As we increase the work done per transaction, *DBMS D*'s and *M*'s Dcache stalls are increased. As the transactions access more distinct rows, they make larger number of random data-accesses. This leads to a higher data-miss rate hence to higher Dcache stalls.

Shore-MT, *DBMS N* and *Silo* have slightly increased Dcache stalls as the amount of work per transaction is increased. *Shore-MT*, *DBMS N* and *Silo* are Dcache-stalls-bound even for reading 1 row per transaction. Hence, the increased instruction locality does not make a significant difference in terms of the micro-architectural behavior of *Shore-MT*, *DBMS N* and *Silo*. Nevertheless, Dcache stalls are slightly increased as the number of rows read per transaction is increased, similarly to the *DBMS D* and *M*.

Table 4.3 shows the normalized throughputs. *DBMS M*'s relative throughput is increased as the number of rows per transaction is increased. *DBMS M* becomes even faster than *DBMS N*, as the number of rows per transacton is increased to 10 and 100. This is due to the increased instruction locality that *DBMS M* benefits from. As the number of rows read per transaction is increased, *DBMS M* suffers from Icache stalls less and less. As a result, *DBMS M* outperforms *DBMS N*. This shows the severe negative effect of instruction-cache misses for *DBMS M* and how highly *DBMS M* benefits from the increased instruction locality.

DBMS M spends ~30%, ~15% and ~3% of its time waiting for Icache stalls for reading 1, 10 and 100 rows. Hence, *DBMS M*'s Icache stalls are reduced by 50% when the number of rows per transaction is increased from 1 to 10, and further reduced by 80% when the number of rows per transaction is increased from 10 to 100. *DBMS N* spends ~5%, ~1.5% and ~1% of its time waiting for Icache stalls. Hence, *DBMS N*'s Icache stalls are reduced by 70% when the

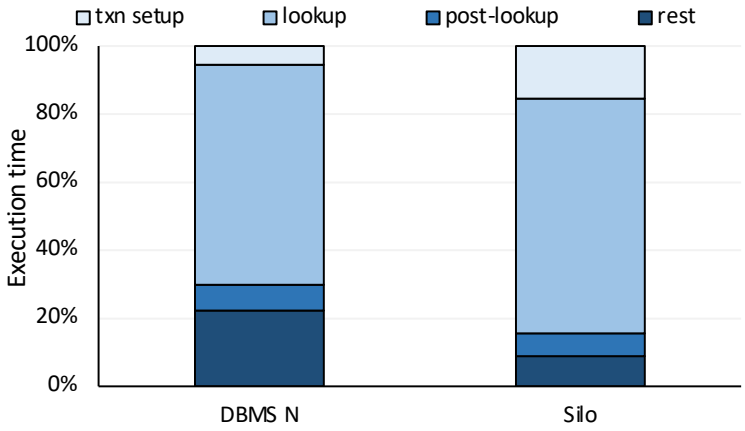


Figure 4.4 – Function call trace breakdown for *DBMS N* and *Silo* when running the micro-benchmark that reads 100 rows per transaction.

number of rows per transaction is increased from 1 to 10, and further reduced by 35% when the number of rows per transaction is increased from 10 to 100. Therefore, *DBMS N* also benefits from the increased instruction-cache locality, similar to *DBMS M*. However, as the percentage of the time *DBMS N* spends on Icache stalls is small ($\leq$ 5%), its delivered throughput is not increased as much as *DBMS M*'s throughput is increased.

Shore-MT, *DBMS N* and *Silo*'s relative throughput to *DBMS D* is decreased as the number of rows per transaction is increased. This is because *DBMS D* benefits from the increased instruction locality as the number of rows is increased, similar to *DBMS M*.

DBMS N's relative throughput with respect to *Silo* is decreased as the number of rows is increased. *DBMS N*'s relative throughput with respect to *Silo* is: 4.9, 3.8 and 3.5 for 1, 10 and 100 rows, respectively. This is because *DBMS N* executes less and less the code to setup and instantiate the transactions as the amount of work per transaction is increased. As a result, its throughput gets closer and closer to *Silo*, which minimally suffers from the work required to setup and instantiate the transactions. Nevertheless, *Silo* is 3.5x faster than *DBMS N* even when *DBMS N* mostly eliminates the transaction setup overheads. We examine this throughput difference in more detail in the following section, Section 4.3.2.

Execution Time Breakdown at the Software-Level

As discussed in the previous section, *DBMS N* is 3.5x slower than *Silo*, even when probing 100 rows per transaction (see Table 4.3). We break down the execution time of *DBMS N* and *Silo* at the software-level when probing 100 rows. Figure 4.4 shows the results. Both OLTP systems have four main categories of functions: (i) transaction setup, (ii) lookup, (iii) post-lookup and (iv) rest. The transaction setup component includes the work that requires setting up and instantiating the transaction. The lookup category is usually a single function that performs the index traversal. The post-lookup is the work that requires the value from the found leaf-level node to be returned. This requires type-checking for *DBMS N* as it performs different

tasks for different data types such as BIGINT or BYTEARRAY. *Silo*'s post-lookup work decodes the values found after the lookup, as it encodes all the keys and values in a string and uses string as the universal key and value type. The rest category is the rest of the work that the system requires be performed such as sending the result back to the transaction, obtaining the target table, achieving the target index, etc.

The figure shows that both systems spend the largest portion of their time performing the index traversal. This shows that the main reason for the performance difference between *DBMS N* and *Silo* is the inefficient index structure that *DBMS N* uses. We further examined *DBMS N*'s index data structure and index traversal algorithm. We saw that *DBMS N* uses a red-black tree as its index structure. Red-black trees are self-balancing binary search trees, where the number of elements per node is 1. Hence, at every level of the tree, red-black tree performs a single comparison. As every node of the tree resides in a random memory location, red-black tree is subject to a data-cache miss at every level during the index traversal.

Silo, on the other hand, uses Masstree, which is a variant of B-tree. As all the B-trees, Masstree's nodes have a particular node size and fanout. It is 15 for *Silo*. Hence, instead of 1, it keeps 15 elements per node. As the tree depth drops exponentially with the node size, *Silo*'s index is much more shallow than *DBMS N*'s red-black tree. Therefore, Masstree is subjected to a significantly fewer data-cache misses. Furthermore, Masstree software prefetches the node's data blocks by injecting a software prefetch instruction during the index traversal. As a result, *Silo* is subject to a single data-cache miss for the entire node it accesses at every level of the tree, making *Silo* significantly faster than *DBMS N*. This shows that, despite the overheads of a real-life system, the efficiency of the used index structure is still one of the most critical factors in defining the performance characteristics of an in-memory OLTP system.

Finally, *DBMS N*'s rest component is significantly higher than that of *Silo*. This is because *DBMS N*, being a real-life system, executes more functions to provide the end-to-end response.

The used data type for keys is important for the index lookup operation, as the index lookup operation performs a significant number of key comparisons during the traversal of the index. *Silo* uses string-encoded keys. *Silo*'s used index structure, Masstree, combines B-tree and trie index structures, where every node of the trie structure is a separate B-tree. Masstree slices the string-encoded keys into pieces of eight bytes and performs a separate B-tree search for every eight bytes of the key. Silo uses the the <, > and == operators to perform the key comparisons between the eight-byte-long pieces of the keys. *DBMS N* stores the keys based on the data types specified by the schema. For the integer keys, it uses its internal `IntsKey` key type. For string keys, it uses its internal `GenericKey` key type. Different key types use different comparison operators. For integers, *DBMS N* uses the <, > and == operators. For strings, it uses the C library function `strncmp`.

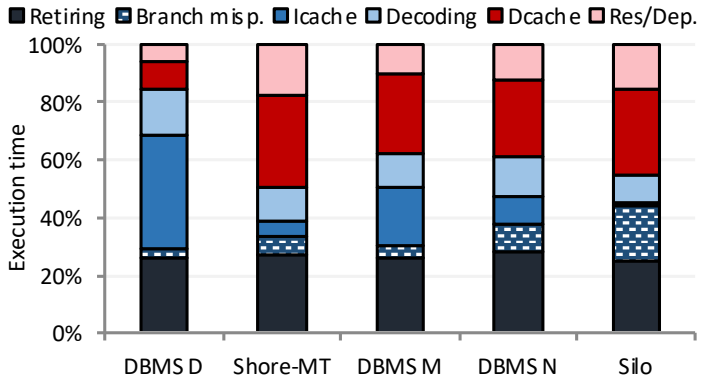


Figure 4.5 – Execution time breakdowns while running TPC-B.

	TPC-B	TPC-C
DBMS D	1	1
Shore-MT	1.7	1.2
DBMS M	2.2	3.1
DBMS N	5.4	2.7
Silo	18.5	20.0

Table 4.4 – Normalized throughput for TPC-B and TPC-C with a database of size 100GB (32GB for *DBMS M*). Throughputs are normalized to *DBMS D*.

4.4 TPC Benchmarks

Section 4.3 performs a sensitivity analysis using a simple micro-benchmark to gain a fine-grained understanding of the in-memory OLTP systems compared to the disk-based ones at the micro-architectural level. This section investigates the behavior of the same systems while running the more complex and community standard TPC-B (Section 4.4.1) and TPC-C (Section 4.4.2) benchmarks. All the experiments in this section use a database of size 100GB, except *DBMS M* for which we use the maximum allowed database size of 32GB. Similar to Section 4.3, we analyze the execution time breakdowns and normalized throughput values.

4.4.1 TPC-B

TPC-B is an update-heavy benchmark that simulates a banking system. `AccountUpdate` is its only transaction type, which updates one row each in three tables, `Branch`, `Teller`, and `Account`, and appends a row to the `History` table.

Figure 4.5 shows the execution breakdowns and Table 4.4 shows normalized throughputs for TPC-B. *DBMS D*, *Shore-MT*, *DBMS N* and *Silo* all suffer less from Dcache stalls compared to the micro-benchmark that reads 1 row per transaction (see Figure 4.1). This is mainly because TPC-B has better data locality compared to the micro-benchmark. When running the micro-bench- mark, we randomly probe rows from a 100GB table, which includes more than one billion rows. On the other hand, TPC-B first probes one of the ~ 20K `Branches` randomly.

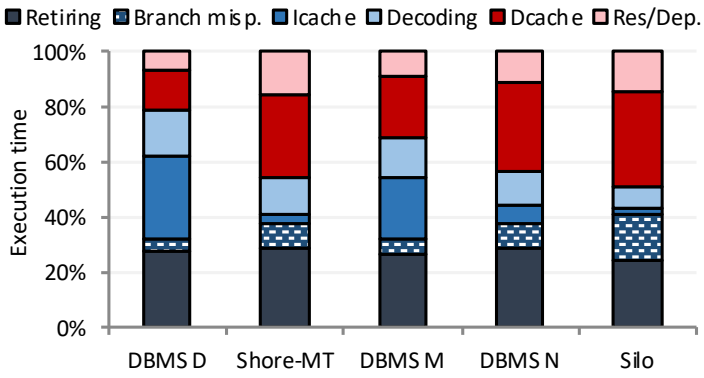


Figure 4.6 – Execution time breakdowns while running TPC-C.

Then, it probes one of the ~ 200K `Tellers` and one of the ~ 2 billion `Accounts`. Finally, it inserts on row into the `History` table. Hence, the probability of re-accessing the same branch or teller as well as the same `History` table page is quite high compared to re-accessing a row from the micro-benchmark's single large table.

DBMS M suffers less from Icache stalls and more from Dcache stalls for TPC-B compared to the micro-benchmark. *DBMS M* relies on a multi-version concurrency control mechanism, where updates create new versions. As TPC-B is an update-heavy benchmark, *DBMS M* requires creating new version for every transactions and hence perform more random data-accesses during the traversal of the version chain. This results in higher degree of Dcache stalls for TPC-B compared to the read-only micro-benchmark. We confirm this hypothesis by the read-write version of the read-only micro-benchmark discussed in Section A.2.1 in the Appendix.

The normalized throughput values follow a similar trend to the micro-benchmark. All the in-memory systems are faster than the disk-based systems. Among the in-memory systems, *DBMS N* is faster than *DBMS M* thanks to not suffering from Icache stalls, and *Silo* is faster than *DBMS N* thanks to its efficient engine components and not performing the work required to set up and instantiate the transactions.

4.4.2 TPC-C

After investigating the micro-architectural behavior of the systems using TPC-B, this section focuses on the more complex TPC-C benchmark. TPC-C models a wholesale supplier with nine tables and five transaction types (2 of which are read-only and form 8% of the benchmark mix). In terms of the database operations, the TPC-C transactions contain probes, inserts, updates, and joins covering a richer set of operations than TPC-B. Therefore, we expect a different behavior for TPC-C than TPC-B.

Figure 4.6 shows the breakdown of the execution times and Table 4.4 shows normalized throughputs for TPC-C. The micro-architectural behavior follows a similar trend to the micro-benchmark and TPC-B. While *DBMS D* and *M* are Icache-stalls-bound, *Shore-MT*, *DBMS N* and *Silo* are Dcache- and resource/dependency-stalls-bound. All the systems suffer less from

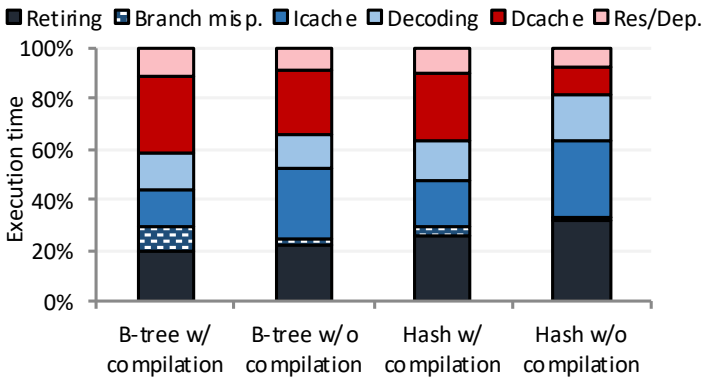


Figure 4.7 – Execution time breakdowns for different index structures with and without compilation optimizations while running the micro-benchmark.

the Dcache stalls compared to the micro-benchmark thanks to the workload locality that TPC-C has. As a result, they have slightly higher retiring time than the micro-benchmark.

The normalized throughput values follow a similar trend to the micro-benchmark. All the in-memory systems are faster than the disk-based systems. Among the in-memory systems, *Silo* is faster than *DBMS M* and *DBMS N*.

DBMS N's relative throughput is less for TPC-C compared to the micro-benchmark and TPC-B. We examined *DBMS N*'s call stack and observed that *DBMS N*'s time spent in serializing/de-serializing tuples is significantly increased for TPC-C compared to the micro-benchmark. *DBMS N* keeps every tuple in its own format (byte array) and deserializes/serializes the data during transaction processing. As TPC-C requires in-transaction processing of the tuples, such as incrementing the order ID for the new order transaction, *DBMS N*'s relative throughput is decreased when running complex benchmark compared to when running a simple micro-benchmark.

Shore-MT's relative throughput is less for TPC-C compared to the micro-benchmark and TPC-B. We examined *Shore-MT*'s call stack and observed that *Shore-MT*'s time spent on B-tree search and lock manager are significantly increased for TPC-C than it is for the micro-benchmark. This highlights *Shore-MT*'s index structure and meta-data processing overheads becoming more prominent for a complex benchmark. This is also visible in Table 4.3, where *Shore-MT*'s throughput is lower than *DBMS D* for probing 10 and 100 rows per transaction.

4.5 Index and Compilation Optimizations, and Data Types

This section analyzes the impact of index and compilation optimizations the in-memory systems adopt, as well as the impact of the data types, at the micro-architectural level. Among the systems used in this study, *DBMS M* is the only one that allows enabling/disabling the compilation optimizations and using two different index structures; hash index and a variant of cache-conscious B-tree index similar to [67, 68]. Therefore, while we use *DBMS M* for

	Micro-bench.	TPC-C
B-tree w/ comp.	1	1
B-tree w/o comp.	0.2	0.2
Hash w/ comp.	1.8	1
Hash w/o comp.	0.3	0.2

Table 4.5 – Normalized throughput for different index structures with and without compilation. Throughputs are normalized to using B-tree w/ compilation.

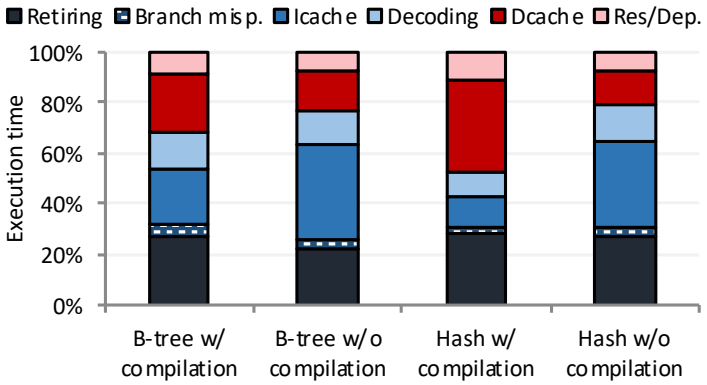


Figure 4.8 – Execution time breakdown for different index structures with and without compilation while running TPC-C.

analyzing the impact of index and compilation optimizations, we experiment with all the three in-memory systems (*DBMS M*, *DBMS N*, and *Silo*) to quantify the effect of different data types.

4.5.1 Impact of index type and compilation

To quantify the impact of the type of index and compilation on the micro-architectual utilization, we start with the read-only variant of the micro-benchmark. Figure 4.7 presents the breakdown of the execution times and Table 4.5 presents the normalized throughput values. The results for the read-write version of the micro-benchmark can be found in Section A.2.3 of the Appendix. We use the version of the micro-benchmark where we access 10 rows per transaction from the 10GB dataset. Transaction compilation has a significant effect on the Icache stalls, which results in ~ 50% reduction in the Icache stalls regardless of the index type. Transaction compilation enables many optimizations in the instruction stream. It can eliminate the virtual function calls. It can inline templated function calls. It can eliminate type checkings and certain branches. Lastly, it allows the compiler to employ its own optimizations more aggressively as it reduces the whole task into a generated code file.

Table 4.5 shows that transaction compilation improves the throughput by 5-6x. B-tree has more Dcache stalls than the Hash index when transactions are not compiled. This is expected as B-tree requires multiple levels of random lookups, whereas Hash index usually requires one or two random lookups. When the transactions are compiled, B-tree and Hash index have similar ratios of Dcache stalls; even though Hash index is 80% faster than B-tree.

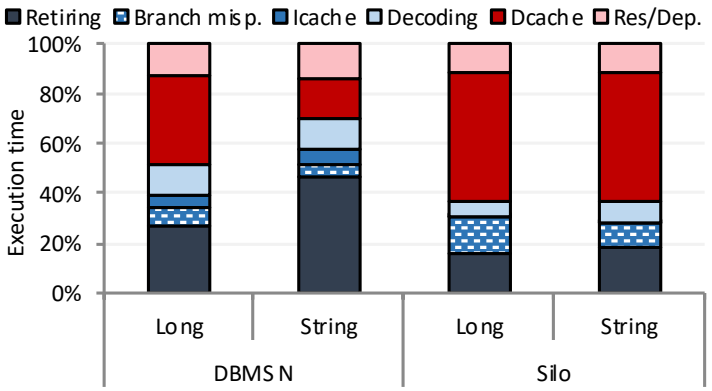


Figure 4.9 – Execution time breakdowns for String and Long data types while running the micro-benchmark.

	Long	String
DBMS N	1	1
Silo	1	0.7

Table 4.6 – Normalized throughput for String and Long data types while running the micro-benchmark. Throughputs with string data type is normalized to throughputs with long data type.

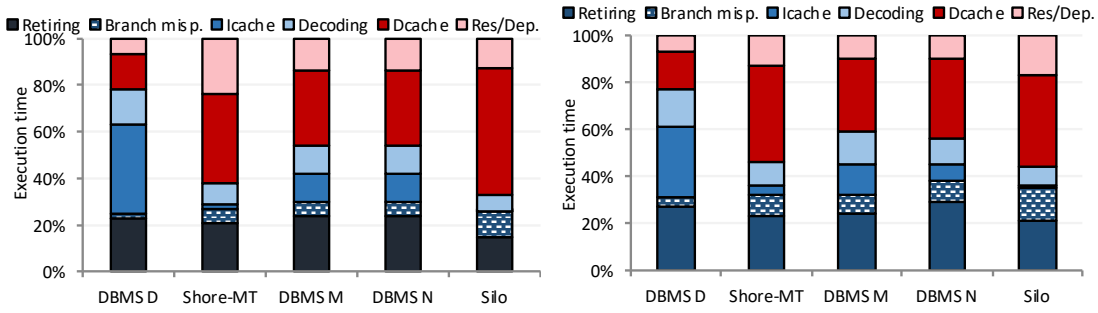
We repeat the experiment above using the TPC-C benchmark. Figure 4.8 shows the breakdown of the execution times and Table 4.5 shows the normalized throughput values. Once again, compilation optimizations reduce Icache stalls significantly for both index types. Moreover, transaction compilation improves *DBMS M*'s throughput by 5x for both B-tree and Hash index types. For Dcache stalls, since the TPC-C benchmark requires fewer random data reads compared to the micro-benchmark, we do not observe a significant difference in Dcache stalls for B-tree and Hash index.

4.5.2 Impact of data type

To quantify the impact of different data types on micro-architectural utilization, we use the read-only version of the micro-benchmark where we probe 1 row per transaction over a 100GB database. The results for the read-write version of the micro-benchmark can be found in Section A.2.3 of the Appendix. We modify the micro-benchmark to use two 50 bytes string columns instead of two long columns in the table and compare the two versions.

Figure 4.9 presents the breakdown of the execution times. *DBMS N* suffers less from Dcache stalls for string compared to long. This is expected as string processing operations usually have high spatial locality. We examined *DBMS N*'s function call stack, and observed that string comparison code constitutes a larger fraction of the execution time with less Dcache stalls.

Silo has a similar micro-architectural behavior for long and string data types. This is because *Silo*'s index structure, Masstree combines B-tree and trie index structures, where every node



(a) Micro-benchmark.

(b) TPC-C.

Figure 4.10 – Execution time breakdowns for the multi-threaded experiments while running the micro-benchmark (left) and TPC-C (right).

of the trie structure is a separate B-tree. Masstree slices the keys into pieces of eight bytes and does a separate B-tree search for every eight bytes of the key, while traversing the overall trie structure. As a result, using a long, 50-byte of string key does not make a significant difference in terms of the data access pattern during the key comparisons, except that a 50-byte key is sliced into a larger number of pieces and hence require more work to perform the index search.

Keeping B-tree within a trie structure allows skipping the upper levels of the trie structure for keys with long common prefixes (such as http URLs). The keys we use, however, do not have such a feature. Hence, Masstree search boils down to multiple levels of B-tree searches.

Table 4.6 presents the throughput values for the string data type that is normalized to the long data type. The results show that *Silo* delivers lower throughput for string than it delivers for long, whereas *DBMS N*'s throughput remains the same for the long and string data type. This is because, unlike *Silo*, *DBMS N* is able to exploit the spatial locality of string processing. As a result, the increased work due to using strings is balanced out with the higher spatial locality of string search.

4.6 Impact of Multi-threading

This section analyzes the effect of running multiple server side threads on the micro-architectural behavior. The single-threaded experiments aim to present an idealized case for all the systems since it avoids cache invalidations due to data sharing across different worker threads or misleading artificially high IPC values due to threads spinning under possible contention. On the other hand, multi-threaded experiments aim to investigate a more realistic scenario where systems are loaded with multiple threads executing transactions from multiple clients.

Figure 4.10 shows the breakdown of the execution times, and Table 4.7 presents the normalized throughput values while running the read-only version of the micro-benchmark when reading 1 row (left) and TPC-C (right) benchmark. We use a database of size 100GB in both of the

	Micro-bench.	TPC-C
DBMS D	1	1
Shore-MT	2.3	1
DBMS M	3.3	1.5
DBMS N	4.7	2.8
Silo	27.8	26.7

Table 4.7 – Normalized throughput for the multi-threaded experiments. Throughputs are normalized to *DBMS D*.

experiments for all the systems except *DBMS M*. We use 10GB of database for the micro-benchmark, and 32GB of database for the TPC-C benchmark for *DBMS M*. We observe that normalized throughput values and the micro-architectural behavior of the individual systems is similar to the single-threaded executions when running the micro-benchmark. *DBMS D* mainly suffers from the Icache stalls, whereas *Shore-MT*, *DBMS N* and *Silo* mainly suffer from the Dcache stalls. *DBMS M* suffers less from Icache and more from Dcache stalls when running multi-threaded than when running single-threaded execution. Therefore, when the system is loaded with multiple threads, Dcache stalls and the sources of Dcache stalls, such as the index lookup operation, gain more importance for *DBMS M*.

Micro-architectural behavior follows similar trends for TPC-C for the multi-threaded execution compared to the single-threaded execution. *DBMS D* largely suffers from Icache stalls, and *Shore-MT*, *DBMS N* and *Silo* mainly suffer from the Dcache stalls. *DBMS M*, once again, suffers less from the Icache and more from the Dcache stalls than it does for the single-threaded execution, showing that Dcache stalls and sources of Dcache stalls gain more importance *DBMS M* for the multi-threaded execution.

4.7 Memory Bandwidth Consumption

This section presents the consumed memory bandwidth for the five OLTP systems we examine for the sensitivity to data size and work per transaction micro-benchmarks and TPC-C benchmark. We measured both single-threaded and multi-threaded consumed memory bandwidth. We observed that the consumed single-threaded bandwidth is always less than 1 GB/s for all the systems. Hence we omit the single-threaded bandwidth results, and focus on the multi-threaded ones.

4.7.1 Data Size Micro-benchmark

Table 4.8 presents the consumed bandwidth for the increasing data size. We observe that all the systems consume significantly less memory bandwidth than the maximum available bandwidth. While the maximum available bandwidth is 66GB/s, the maximum consumed bandwidth is 8.3GB/s by *Silo* for 100GB of data size.

DBMS D, *Shore-MT* and *DBMS M* consume significantly less amount of bandwidth than *DBMS*

	1MB	10MB	10GB	100GB
DBMS D	0	0	0	0
Shore-MT	0	0	2	2
DBMS M	0	0	1	-
DBMS N	5.2	5.1	6.2	6.2
Silo	0	0	8.2	8.3

Table 4.8 – Consumed bandwidth in GB/s as we increase the database size for multi-threaded execution when reading 1 row per transaction.

	1 row	10 rows	100 rows
DBMS D	0	2	3
Shore-MT	2	2	2.5
DBMS M	1	7	11
DBMS N	6.2	8.5	8.4
Silo	8.3	8.3	8.4

Table 4.9 – Consumed bandwidth in GB/s as we increase the amount of work per transaction for multi-threaded execution for a database of size 100GB.

N and *Silo*. *DBMS D* and *M* suffer from Icache stalls, which prevents them from stressing the memory bandwidth. As a result, their consumed bandwidth values are very low. *Shore-MT* suffers from Dcache stalls for 10GB and 100GB. As being a disk-based system, it nevertheless is significantly slower (see Table 4.2) than *DBMS N* and *Silo*. As a result, it stresses the memory bandwidth only modestly.

DBMS N has relatively high bandwidth consumption for 1 and 10MB of data. This is due to *DBMS N*'s meta-data processing for setting up and instantiating the transactions. As the data size is increased the consumed bandwidth is also increased. *Silo* consumes no memory bandwidth for 1 and 10MB of data as the data is mostly cache-resident. *Silo* consumes the highest bandwidth among the systems we analyze for 10 and 100GB of data. This is expected as *Silo* does not perform the work that disk-based systems perform and the meta-data processing during the transaction setup and instantiation. As a result, it delivers the highest relative throughput (see Table 4.2) and stresses the memory bandwidth the highest. Nevertheless, *Silo*'s maximum consumed bandwidth is significantly less than the maximum available bandwidth of 66GB/s. This shows that OLTP systems generate only modest amount of memory traffic, and hence severely under-utilize the memory bandwidth.

4.7.2 Work per Transaction Micro-benchmark

Table 4.9 shows the consumed bandwidth as we increase the amount of work per transaction. We use 100GB of database for all the systems, except *DBMS M* and 10GB of database for *DBMS M*. The consumed bandwidth is increased as the amount of work per transaction is increased for all the systems. This increase is more pronounced for *DBMS M*. This follows *DBMS M*'s significantly increased throughput as the amount of work per transaction is increased (see

	TPC-C
DBMS D	0
Shore-MT	2.6
DBMS M	0
DBMS N	2.6
Silo	5.3

Table 4.10 – Consumed bandwidth in GB/s for TPC-C benchmark for multi-threaded execution.

Table 4.3). As the amount of work per transaction is increased, *DBMS M* suffers less and less from Icache stalls. As a result, its relative throughput and consumed bandwidth is significantly increased.

The increase is also observable for *DBMS D* and *DBMS N*: They both stress the memory bandwidth increasingly higher, as the amount of work per transaction is increased. The consumed bandwidth is significantly less for *DBMS D* than it is for *DBMS N*. *DBMS D* is significantly slower than *DBMS N*; hence, even when probing 100 rows per transaction, it lightly stresses the memory bandwidth.

Shore-MT and *Silo*'s consumed bandwidths are only modestly increased. This is because *Shore-MT* and *Silo* are OLTP engines that hard code transactions in C++. Hence, the increased amount of work per transaction stresses the memory bandwidth at a similar level per unit of a time. As a result, the consumed bandwidth remains mostly stable as the amount of work per transaction is increased.

Overall, despite the increased amount of work per transaction, all the OLTP systems we examine consume only a modest fraction of the maximum available bandwidth. While the maximum available bandwidth is 66GB/s, the maximum consumed bandwidth is 11GB/s by *DBMS M* when reading 100 rows per transaction.

4.7.3 TPC-C

In this section, we examine the amount of consumed bandwidth for TPC-C benchmark for a database of size 100GB, except for *DBMS M*. We use 32GB of database for *DBMS M*. Table 4.10 shows the results. The consumed bandwidth values are less for all the systems compared to the micro-benchmark. This is expected as TPC-C transactions are more complex and has more workload locality, and hence require more on-chip computation rather than stressing memory bandwidth.

DBMS D and *M*, being Icache-stalls-bound systems, consume very low memory bandwidth. *Shore-MT*, *DBMS N* and *Silo*, being Dcache- and resource/dependency-stalls-bound, consume certain amount of memory bandwidth. *Silo*, being the fastest OLTP system we analyze, consumes the highest amount of memory bandwidth. Nevertheless, all the systems we analyze consumes bandwidth that is significantly below the maximum available bandwidth. While

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.2	1.3	1.2	1.3
Silo	1.4	1.6	1.4	1.7

Table 4.11 – Normalized throughput values for hyper-threading evaluation. ST: The throughput of running two hyper-threads on the same physical core is normalized to the throughput of running one hyper-thread on the physical core. MT: The throughput of running 28 hyper-threads on 14 physical cores of a same socket is normalized to the throughput of running 14 hyper-threads on the same 14 physical cores.

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.3	1.1	1.2	1.2
Silo	1.2	1.1	1.2	1.1

Table 4.12 – Normalized throughput values for turbo-boost evaluation. Hyper-threading is turned off. ST: The throughput of running single thread having turbo-boost turned on is normalized to the throughput of running single thread having turbo-boost turned off. MT: The throughput of running 14 threads on a same socket having turbo-boost turned on is normalized to the throughput of running 14 threads on the same socket having turbo-boost turned off.

the maximum available bandwidth is 66GB/s, the highest consumed bandwidth is 5.3GB/s by *Silo*.

4.8 Acceleration Features

In this section, we examine three acceleration features that today's processors provide: hyper-threading, turbo-boost and hardware prefetchers. We present normalized throughput numbers.

We use *DBMS M* and *Silo* when running the read-only micro-benchmark while probing 1 row per transaction, and when running the TPC-C benchmark for single- and multi-threaded executions. We use 10GB of database for the micro-benchmark and 32GB of database for TPC-C when profiling *DBMS M*. We use 100GB of database for *Silo*.

4.8.1 Hyper-threading

Table 4.11 shows normalized throughput values for hyper-threading evaluation. For single-threaded execution, the normalized throughput shows the throughput improvement when running two threads on the same physical core compared to running a single thread on a single physical core. For multi-threaded execution, the normalized throughput shows the throughput improvement when running 28 threads on 14 physical cores compared to running 14 threads on 14 physical cores (assuming that 14 threads deliver the highest throughput). We

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.0	1.0	1.0	1.0
Silo	1.0	1.0	1.0	1.0

Table 4.13 – Normalized throughput values for prefetcher evaluation. Hyper-threading and turbo-boost is turned off. ST: The throughput of running a single thread having prefetchers turned on is normalized to the throughput of running a single thread having prefetchers turned off. MT: The throughput of running 14 threads on a same socket having prefetchers turned on is normalized to throughput of running 14 threads on the same socket having prefetchers turned off.

use the DBMS's and/or OS's relevant configuration interface to bind the threads to a single socket and allocate memory locally.

We observe that hyper-threading is modestly useful for *DBMS M*, whereas it is significantly useful for *Silo*. Hyper-threading is the most useful when there is long latency data stalls that can easily be overlapped. As *Silo* highly suffers from Dcache stalls, hyper-threading provides a more significant speedup for *Silo*. We also observe that the improved throughput is higher for multi-threaded execution than it is for single-threaded both for *DBMS M* and *Silo*. This is likely due to the increased sharing of the data structures at the last-level cache when running concurrently on the multiple cores.

4.8.2 Turbo-boost

Table 4.12 shows normalized throughput values for turbo-boost evaluation. We present the throughput values with turbo-boost turned on normalized to the ones with turbo-boost turned off.

We observe that both *DBMS M* and *Silo* modestly benefit from turbo-boost. Turbo-boost provides the highest speedups when the computation is arithmetic-operation-heavy rather than memory-access-bound as it is the case for OLTP. As a result, both systems only modestly benefit from turbo-boost feature.

4.8.3 Hardware prefetchers

Table 4.13 shows normalized throughput values for data prefetchers evaluation. We present the throughput values with prefetchers enables normalized to the ones with prefetchers disables. There are four hardware prefetchers that today's server processors provide: L1 next line, L1 streamer, L2 next line and L2 streamer prefetchers [42]. We disable them all and enable them all.

We observe that prefetchers have no visible effect on the OLTP system performance. OLTP workloads are random-data-access-heavy and have low spatial locality. As a result, the streamer prefetchers might not be providing a visible performance gain when enabled. *DBMS*

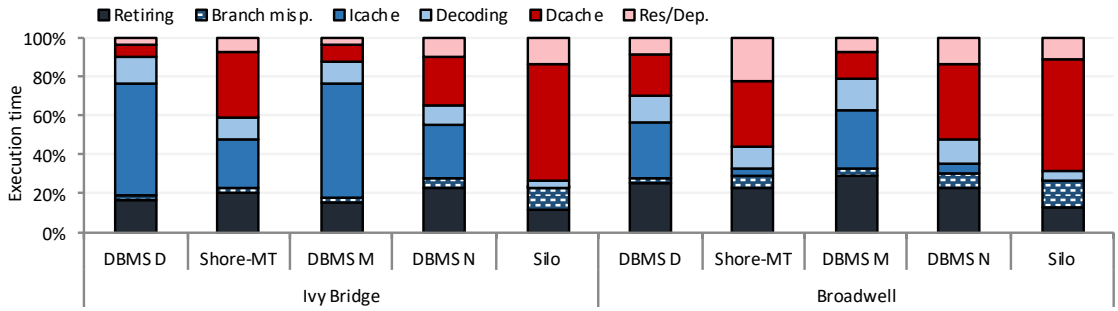


Figure 4.11 – Execution time breakdowns for successive Intel micro-architectures when running the read-only microbenchmark where every transaction randomly reads 1 row.

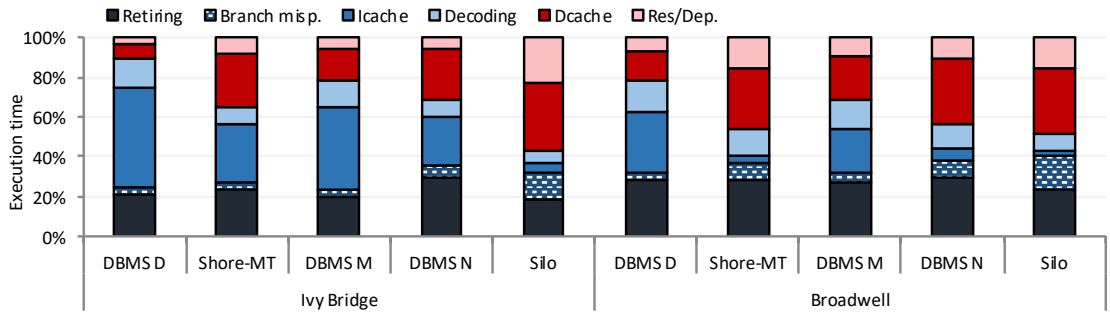


Figure 4.12 – Execution time breakdowns for successive Intel micro-architectures when running TPC-C.

M mainly suffers from Icache stalls. Hence, the improvement that the next line prefetchers bring is likely to be negligible. *Silo* uses software prefetching to prefetch consecutive cache lines that belongs to the same index node, during its index traversal. Hence, the disabled next line prefetcher is likely not creating an observable effect on *Silo*'s throughput.

4.9 Ivy Bridge vs. Broadwell

In this section, we compare two Intel generations in terms of their micro-architectural behavior when running the microbenchmark that randomly reads 1 row from a database of 100GB and the TPC-C benchmark. We use 10GB of a database for the micro-benchmark, and 32GB of a database for TPC-C when profiling *DBMS M*. We examine the Intel Xeon v2 line Ivy Bridge micro-architecture and Intel Xeon v4 line Broadwell micro-architecture. We choose these two generations as there is a major micro-architectural change from Intel Xeon v2 line Ivy Bridge to Intel Xeon v3 line Haswell micro-architecture, especially in the instruction fetch unit of the processors [31] (see Section 4.3.1, paragraph 2). As OLTP systems are known to severely suffer from Icache stalls, we examine how effective the instruction fetch unit improvement for OLTP systems. We use Broadwell micro-architecture in this analysis. Broadwell is slightly improved version of the Haswell micro-architecture.

Figure 4.11 and 4.12 show the results. We observe that there is a significant change in the micro-

	Micro-bench.	TPC-C
DBMS D	2.0	1.6
Shore-MT	1.6	1.6
DBMS M	2.5	1.4
DBMS N	1.6	1.5
Silo	1.1	1.3

Table 4.14 – Throughput on Broadwell normalized to throughput on Ivy Bridge.

architectural behavior of the OLTP systems across the two processor generations. On the Ivy Bridge micro-architecture, all the systems except *Silo* mainly suffer from Icache stalls. On the Broadwell micro-architecture, *Shore-MT* and *DBMS N*'s main micro-architectural bottlenecks shift from Icache stalls to Dcache stalls. Similarly, *DBMS D* and *M* suffer significantly less from the Icache stalls on the Broadwell micro-architecture compared to the Ivy Bridge micro-architecture. This shows that the advances in the instruction fetch unit of the processor significantly help in reducing the Icache stalls.

Our finding on Ivy Bridge vs. Broadwell corroborates the recent work of Yasin et al. [119] where SPEC benchmarks are evaluated across Ivy Bridge and Skylake (the generation after Broadwell) micro-architectures. Yasin et al. also show that the improvement on the Skylake micro-architecture, which inherits the improvements from the Broadwell micro-architecture significantly reduces the Icache stalls.

Table 4.14 shows the throughput values on the Broadwell machine normalized to the throughput values on the Ivy Bridge machine. As expected, the Broadwell machine delivers significantly higher throughput than the Ivy Bridge machine thanks to its micro-architectural improvements.

Existing work on micro-architectural analysis of OLTP systems has mostly used an Ivy Bridge or an earlier micro-architecture generation. As a result, their conclusions were mostly referring to the high Icache stalls of disk-based and in-memory OLTP systems [98, 99, 108, 111]. We take the existing work one step ahead, and provide conclusions on one of the latest generations of Intel micro-architectures.

4.10 Conclusion

In this chapter, we perform a detailed micro-architectural analysis of the in-memory OLTP systems contrasting them to the disk-based OLTP systems. The results demonstrates that in-memory OLTP systems spend most of their time in stalls similarly to the disk-based OLTP systems despite all the design differences and lighter storage manager components of the memory-optimized systems. The disk-based and the in-memory engine of the disk-based OLTP systems suffer mainly from Icache stalls. Although the optimizations that the in-memory engine uses are helpful in significantly improving the delivered throughput, the main micro-architectural bottleneck remains the same. Ground-up designed in-memory OLTP systems

eliminate the Icache stalls; however, due to the random data-accesses during the index traversal, they spend more than half of their execution time waiting for Dcache stalls. While using an efficient index structure can significantly reduce the amount of Dcache stalls, Dcache stalls remain the main micro-architectural bottleneck due to the costly random data-accesses.

5 Online Analytical Processing Workloads

In Chapter 4, we have shown that online transaction processing (OLTP) systems spend most of their execution time waiting for instruction-cache or data-cache misses. Online analytical processing (OLAP) workloads exhibit a completely different computing pattern. OLAP workloads are read-only, bandwidth-intensive, and include various data access patterns. With the rise of column-stores, they run on high-performance engines that are tightly optimized for modern hardware. Consequently, the micro-architectural behavior of modern OLAP systems is unclear.

In this chapter, we present a micro-architectural analysis of OLAP workloads. Unlike the OLTP workloads, the OLAP workloads do not suffer from instruction-cache misses. The commercial row-store and its column-store extension efficiently use the CPU cycles; however, they require executing a significantly larger number of instructions hence are 2 to 56 times slower than the column-stores that follow tuple-at-a-time, vector-at-a-time, and compiled execution models. The column-store that follows the tuple-at-a-time execution model also efficiently uses the CPU cycles; however, it also requires a significantly larger number of instructions than the column-stores that use vector-at-a-time and compiled execution models, hence they are 1.7 to 5 times slower than.

Column-stores that use vector-at-a-time and compiled execution models spend most of their execution time waiting for data-cache misses. The data-cache stalls are due to the stress on the memory bandwidth for sequential-scan-heavy queries, and are due to long-latency data-cache misses caused by random data-accesses for join-intensive queries. Concurrently executing scan-intensive and join-intensive queries can improve the utilization, but it creates interference in the shared resources, which results in sub-optimal performance.

5.1 Introduction

Online analytical processing (OLAP) is an ever-growing, multi-billion dollar industry. To extract valuable information from their data, many industrial and community organizations rely on fast and efficient analytical processing. Micro-architectural behavior reveals the limitations of

and opportunities for efficiently using modern hardware resources, hence enables delivering high performance. Research has shown that OLAP systems can improve performance by orders of magnitude by more efficiently using the modern hardware resources [74].

Chapter 4 has shown that, despite being aggressively optimized for modern hardware, in-memory OLTP systems spend most of the time in instruction-cache and/or data-cache misses. OLAP workloads exhibit a completely different computing pattern. Unlike the update-heavy OLTP workloads, OLAP workloads are read-only. Therefore, they do not require a concurrency control and logging mechanism or a complex buffer pool for synchronizing the modified pages on disk. OLAP workloads are arithmetic-operation- and bandwidth-intensive. They process large amounts of data with various data access patterns including both sequential and random data-accesses.

With the rise of column-stores [2, 40], researchers proposed a diverse set of query processing paradigms (vectorized [15, 85] vs. compiled query processing [79]), and system prototypes (Proteus [52], Typer, and Tectorwise [56]). Many database systems, such as SQL Server, Oracle, and DB2, support a column-store extension [59, 60, 89]. column-stores operate only on the columns that are necessary for the query, thus utilize memory bandwidth more efficiently. They process columns in tight, hardware-friendly execution loops that are optimized for the efficient use of the CPU cycles.

Hence, the micro-architectural behavior of modern OLAP systems is unclear. This chapter performs a detailed micro-architectural analysis of OLAP workloads running on modern hardware. We profile six OLAP systems: a commercial row-store, the column-store extension of the commercial row-store, a column-store that follow tuple-at-a-time execution model, two column-stores that follow vector-at-a-time execution model, and a column-store that follows compiled execution model. We evaluate the execution time breakdown at the hardware-level, memory bandwidth consumption, and normalized execution time. In this chapter, we show the following:

- The commercial row-store and its column-store extension efficiently use the CPU cycles. However, they require a significantly larger number of instructions than the column-stores that use tuple-at-a-time, vector-at-a-time and compiled execution models. As a result, they are 2 to 56 times slower than the column-stores that use these models.
- The column-store that follows the tuple-at-a-time execution model efficiently uses CPU cycles; however, it requires a significantly larger number of instructions than the column-stores that follow vector-at-a-time and compiled execution models. As a result, it is 1.7 to 5 times slower than the column-stores that follow vector-at-a-time and compiled execution models.
- The column-stores that follow vector-at-a-time and compiled execution models spend most of their execution time waiting for data-cache misses due to either the stress on the memory bandwidth or random data-accesses. The scan-intensive queries stress the

memory bandwidth, and the join-intensive queries make a large number of random data-accesses.

- The column-stores that follow vector-at-a-time or compiled execution model saturate the memory bandwidth before saturating the number of cores when running a scan-intensive query, whereas saturate the number of cores before saturating the memory bandwidth when running a join-intensive query. Concurrently executing scan- and join-intensive queries enables the saturation of both the number cores and the memory bandwidth. However, this creates interference in the shared last-level cache and memory bandwidth and hence results in a sub-optimal performance.

The rest of the chapter is organized as follows. In Section 5.2, we present the experimental setup and methodology. In Section 5.3, 5.4 and 5.5, we present the projection, selection and join micro-benchmark analyses. In Section 5.6, we present the analysis of TPC-H queries. In Section 5.7, we present mixed query workload analysis. In Section 5.8, 5.9, 5.10 and 5.11, we present the analyses of predication, SIMD, hardware prefetchers and hyper-threading/turbo-boost. Lastly, in Section 5.12, we present the conclusions.

5.2 Setup & Methodology

In this section, we present the experimental setup and methodology.

Benchmarks: We use micro-benchmarks and TPC-H queries [113]. We use projection, selection, and join micro-benchmarks as they constitute the basic SQL operators. All the systems use the hash join algorithm when running the join micro-benchmark.

All the micro-benchmarks use the TPC-H schema. The projection micro-benchmark does a single SUM() over a set of columns from the lineitem table. Hence, it performs aggregation after projecting the relevant columns. We vary the number of columns from one to four. We use *l_extendedprice*, *l_discount*, *l_tax* and *l_quantity* columns. We add the projected columns inside the SUM(). We call the projection micro-benchmark that does a SUM() over n columns a projection query with the degree of n.

The selection micro-benchmark extends the projection query with the degree of four with a WHERE clause of three predicates over three columns of the lineitem table: *l_shipdate*, *l_commitdate* and *l_receiptdate*. It varies the selectivity of each individual predicate from 10% to 50% and 90%. The join micro-benchmark does a join over two tables, followed by a projection. The small-sized join micro-benchmark joins the supplier and nation tables, it and does a SUM() over the addition of *s_acctbal* and *s_suppkey*. The medium-sized join joins the partsupplier and supplier tables, and it does a SUM() over the addition of *ps_availqty* and *ps_supplycost*. The large-sized join joins the lineitem and orders table, and it does a SUM() over the addition of the four columns that the projection query with the degree of four uses. The joins cover the common case of having a join followed by an aggregation.

Processor	Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell)
#sockets	2
#cores per socket	14
Hyper-threading	Off
Turbo-boost	Off
Clock speed	2.40GHz
Per-core bandwidth	12GB/s (sequential) 7GB/s (random)
Per-socket bandwidth	66GB/s (sequential) 60GB/s (random)
L1I / L1D (per core)	32KB / 32KB 16-cycle miss latency
L2 (per core)	256KB 26-cycle miss latency
L3 (shared)	(inclusive) 35MB 160-cycle miss latency
Memory	256GB

Table 5.1 – Broadwell server parameters.

We profile a large subset of TPC-H queries on *DBMS V*. We chose *DBMS V* for this purpose, as *DBMS V* is the highest performing real-life system we use. We categorize the TPC-H queries based on their micro-architectural behavior. We then choose six representative queries and continue with the cross-system analysis. Our selection of the queries corroborates with the queries used by [56].

The TPC-H queries cover a large set of use cases that are common across the OLAP workloads: For example, Q3 has joins with selections over the dimension tables; and Q4 has joins with low hit-rate which enables bloom filters to be used. Our examinations have shown that the cost of a join operation is orders of magnitude higher than the cost of projection, aggregation, and selection operations. Hence, the execution times of the queries with joins are usually dominated by the execution time of the join. The categorization of TPC-H queries also shows that the execution times of most TPC-H queries are dominated by joins, where the size of the build-side table and the hit-rate of the join operation are the determining factors on the micro-architectural behavior of the queries.

Hardware: We conduct our experiments on an Intel Broadwell server. Table 5.1 presents the server parameters. As the Broadwell micro-architecture does not support AVX-512 instructions, we conduct the SIMD experiments on a separate Skylake server. The Skylake server has a similar execution engine but a different memory hierarchy from the Broadwell server. The Skylake server has a significantly larger L2 cache (1 MB), a smaller non-inclusive L3 cache (16MB), a smaller per-core (10 GB/s) and a larger per-socket (87 GB/s) sequential access bandwidth. It has a similar per-core and per-socket random access bandwidth.

We use Intel's Memory Latency Checker (MLC) [47] to measure cache access-latencies and

maximum single/multi-core and random/sequential-access bandwidth.

OLAP systems: We examine (i) a commercial row-store, *DBMS R*, (ii) the column-store extension of the commercial row-store, *DBMS C*, (iii) an open-source column-store that follow tuple-at-a-time execution model, *Quickstep* [85], (iv) a closed-source column-store that follow vector-at-a-time execution model, *DBMS V*, (v) an academic column-store prototype that follow vector-at-a-time execution model, *Tectorwise* [56], (vi) an academic column-store prototype that follow compiled execution model, *Typer* [56]. We chose these six systems as each represents a different category of a system and execution model.

OS & Compiler: We use Ubuntu 16.04.6 LTS and gcc 5.4.0 on the Broadwell server, and Ubuntu 18.04.2 LTS and gcc 7.4.0 on the Skylake server.

VTune: We use Intel VTune 2018 on the Broadwell server, and VTune 2019 on the Skylake server. We use VTune's built-in general-exploration (uarch-exploration on VTune 2019) analysis for the execution time breakdown at the hardware-level, which performs Intel's Top-down Micro-architectural Analysis Methodology (TMAM) that is covered in Chapter 3. We use VTune's built-in memory-access analysis to measure the consumed memory bandwidth. As we numa-localize our experiments on a single socket, we report average bandwidth per-socket values. We use VTune's built-in hotspots and advanced-hotspots analyses to perform function call trace breakdown.

We provide an overview of Intel's TMAM explained in Chapter 3. Each instruction issue slot is categorized into one of two components: retiring and stalling. A retiring slot is a slot where the slot is used for retiring an instruction. A stalling slot is a slot where the slot stalls, i.e., has to wait due to a particular issue. Ideally, all issue slots would be used for retiring. Stalling slots are further decomposed into five components: (i) branch misprediction, (ii) Icache, (iii) decoding, (iv) Dcache and (v) resource/dependency stalls. Branch misprediction stalls are the slots that stall due mispredicted branch instructions. Today's processors use a hardware unit called branch predictor; it predicts the outcome of a branch instruction (i.e., an if() statement) and speculatively executes instructions per the predicted branch direction and/or target. If the processor then realizes the prediction is not correct, it undoes whatever it has been doing and starts executing the correct set of instructions. This cost is defined as the branch misprediction stalls and can be very costly, as it requires canceling a large amount of work. Icache stalls are the slots that stall due to instruction-cache and instruction translation lookaside buffer misses. Decoding stalls are the slots that stall due to sub-optimal micro-architectural implementation of the instruction decoding unit. Dcache stalls are the slots that stall due to data-cache misses. Resource/dependency stalls are the slots that stall due to resource and/or data dependencies. For example, if two instructions require using the same arithmetic-logic unit, one has to wait for the other. This time is identified as the resource/dependency time. Or, if an instruction's operand depends on the result of another instruction, the instruction with the dependent operand has to wait for the other instruction to finish. This time is identified as the resource/dependency time.

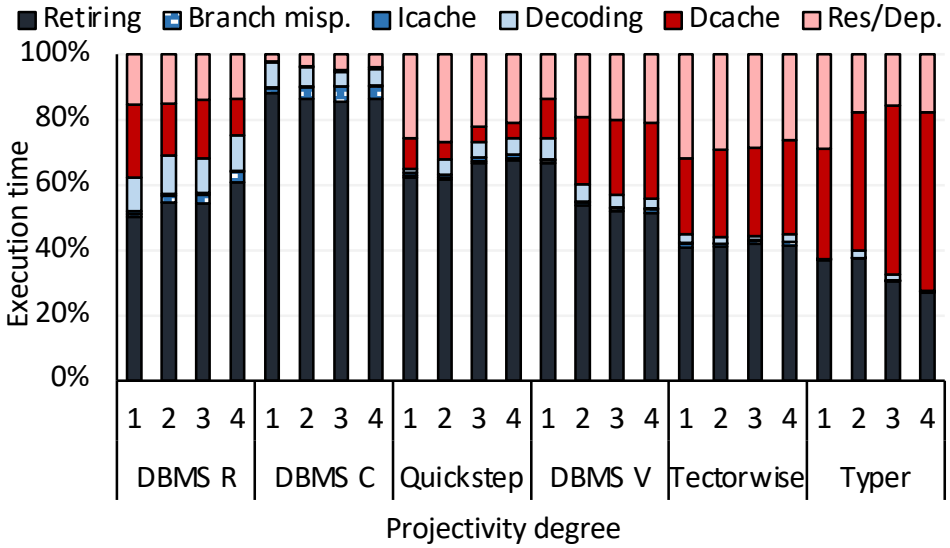


Figure 5.1 – Execution time breakdown for the projection micro-benchmark for single-threaded execution.

Measurements: For every experiment, we first populate the database. We use a one-minute warmup period, followed by a three-minute VTune profiling period. We disable hyper-threading (HT) and turbo-boost (TB), as they jeopardize VTune counter values [45]. We examine HT and TB separately, in Section 5.11.

We numa-localize every experiment by using Linux's numactl command. We do single- and multi-threaded experiments. For the multi-threaded experiments, we use the number of threads that provides the lowest execution time. We choose a scaling factor of 70 (the database of 70GB) for all the experiments as it makes 5GB/core to process; this is large enough for out-of-cache experiments. We normalize execution times to *DBMS V*'s execution time as *DBMS V* is the highest performing real-life system we use.

We generate statistics before profiling each database. For a more fair comparison, we disable compression for all the systems. We test the compression on *DBMS V* when it runs the TPC-H benchmark, and we see that it increases the execution time for 18 of the 22 queries. For the remaining 4 queries, it decreases the execution time less than 15%.

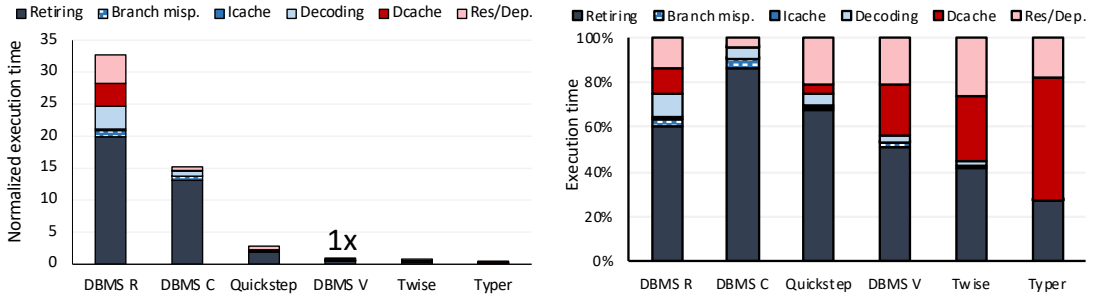
We do hardware prefetcher experiments in Section 5.10 by modifying the relevant model-specific register (msr) of the processor [41].

5.3 Projection

We present the projection micro-benchmark. Figure 5.1 and 5.3 show the breakdown of the execution times, Table 5.2 presents normalized execution times, and Table 5.3 presents consumed memory bandwidth values for single- and multi-threaded executions.

	Single-threaded				Multi-threaded			
	p1	p2	p3	p4	p1	p2	p3	p4
R	56	39.3	35.5	32.8	43	30.1	24.5	22
C	13.6	14.4	15.4	15.3	9	10.1	10.3	10
Qs	2.7	2.7	2.7	2.8	1.7	1.8	1.7	1.8
V	1	1	1	1	1	1	1	1
Tw	0.5	0.6	0.7	0.8	0.6	0.7	0.7	0.7
Ty	0.5	0.4	0.5	0.5	0.6	0.7	0.7	0.7

Table 5.2 – Normalized execution times for projection micro-benchmark for single- and multi-threaded executions. The execution times are normalized to *DBMS V*.



(a) Normalized execution time breakdowns. The execution times are normalized to*DBMS V*.

(b) Execution time breakdowns streched to 100%.

Figure 5.2 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for the projection query with projectivity of degree 4 for single-threaded execution.

DBMS R* & *C spend most of their execution time retiring instructions. They are also 10 to 56 times slower than *DBMS V* is. As the retiring time is proportional to the number of retired instructions, it shows that, to execute the same workload, *DBMS R* and *C* require a number of instructions significantly larger than *DBMS V* requires. Figure 5.2 presents the breakdown of the normalized execution times (left) and the breakdowns of the execution times stretched to 100% (right) for each system for the projection query with a projectivity of degree 4. The figure shows that *DBMS R* and *C* are slower than *DBMS V* is, mainly due to the large number of instructions they execute.

Quickstep spends most of its execution time retiring instructions. It is also 1.7 to 2.8 times slower than *DBMS V* is. Hence, similarly to *DBMS R* and *C*, *Quickstep* is slower than *DBMS V* is, mainly due to using a larger number of instructions. We examine *Quickstep*'s function-call trace for the projection query of degree four to understand the reasons for executing a larger number of instructions. *Quickstep* spends 50% of its time in `getUntypedValue()` and 8.1% of its time in `next()` function. It spends the remaining time inside a functor that performs aggregation. `getUntypedValue()` function performs null/boundary checking, whereas `next()` is used to increment the processed tuple ID.

Quickstep follows tuple-at-a-time execution model with aggressive function inlining, where

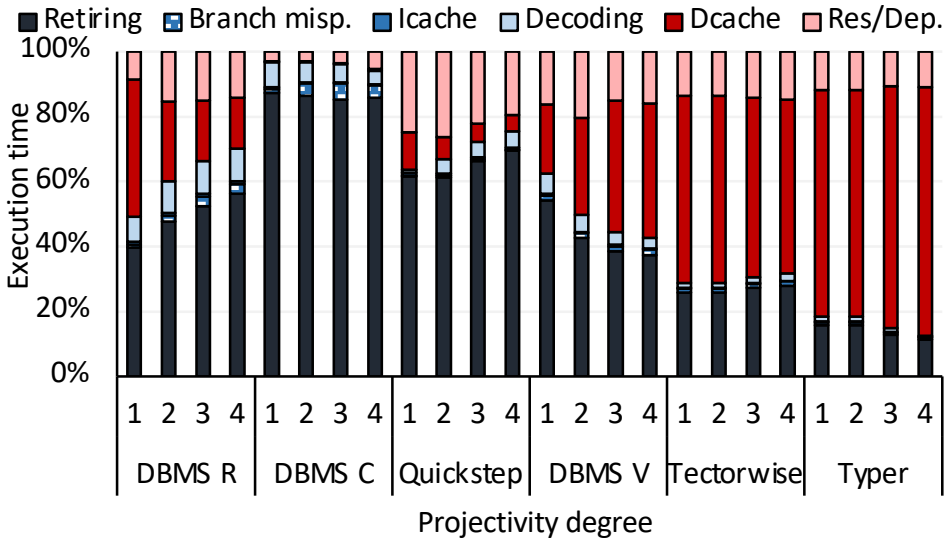


Figure 5.3 – Execution time breakdowns for the projection micro-benchmark for multi-threaded execution.

	Single-threaded				Multi-threaded			
	p1	p2	p3	p4	p1	p2	p3	p4
R	0	0	0	0	44.6	34.7	27.3	23.6
C	0	0	0	0	0.9	0.6	0.3	0.3
Qs	0	0.1	0.9	0.7	21.6	23.5	24	24.2
V	2.8	2.6	2.4	2.2	38.6	43.2	43.8	45.6
Tw	7	6.8	5	4.9	62.9	62.4	61.5	61.1
Ty	8.6	10.5	9.6	10.1	62.8	63	62.9	62.8

Table 5.3 – Consumed bandwidth in GB/s for projection micro-benchmark for single- and multi-threaded executions.

at every iteration of the aggregation it makes inlined function calls of `getUntypedValue()` and `next()` per tuple. Although inlined function calls are not as expensive as regular or virtual function calls, they require extra work. `getUntypedValue()` makes five further inlined function calls, `getAttributeValue()`, `getAttributeId()`, `hasAttributeWithId()`, `IdInRange()` and `elementIsNull()`, which requires even more work, hence making the aggregation 1.7 to 2.8 times slower than *DBMS V*.

DBMS V is twice as slow as *Tectorwise* at the projectivity of degree one for single-threaded execution. As *DBMS V* and *Tectorwise* implement a similar execution model, the difference highlights the work that a full-fledged, real-life system need to perform compared to an academic prototype.

Tectorwise* & *Typer have the same performance at the projectivity of degree one. As the projectivity increases, *Typer* outperforms *Tectorwise* at single-threaded execution. This is because, as the projectivity increases, *Tectorwise* suffers more from the materialization over-

head. Whereas, *Typer* follows a compiled execution model that does not suffer from the materialization overhead.

Typer's and *Tectorwise*'s relative execution times are the same at the multi-threaded execution, as they are both memory-bandwidth-bound. Table 5.3 shows that both *Tectorwise* and *Typer* saturate the memory bandwidth at the multi-threaded execution. We also examine *Typer*'s and *Tectorwise*'s function-call traces. They both spend almost 100% of their time inside the aggregation function.

Single vs. Multi-threaded Execution: *DBMS C* and *Quickstep* have the same execution time breakdowns for the single- and multi-threaded executions. Table 5.3 shows that *DBMS C* has very low single- and multi-threaded bandwidth consumption, which explains the similar micro-architectural behavior. *Quickstep* has a low single-threaded, yet significant multi-threaded bandwidth consumption. Nevertheless, its bandwidth stress is not sufficiently high to change the micro-architectural behavior.

DBMS V, *Tectorwise* and *Typer* have higher Dcache stalls when running multi-threaded compared to when running single-threaded. *DBMS V* consumes a large fraction of the memory bandwidth, which results in the higher Dcache stalls. *Tectorwise* and *Typer* fully consume the memory bandwidth, which results in the highly pronounced Dcache stalls.

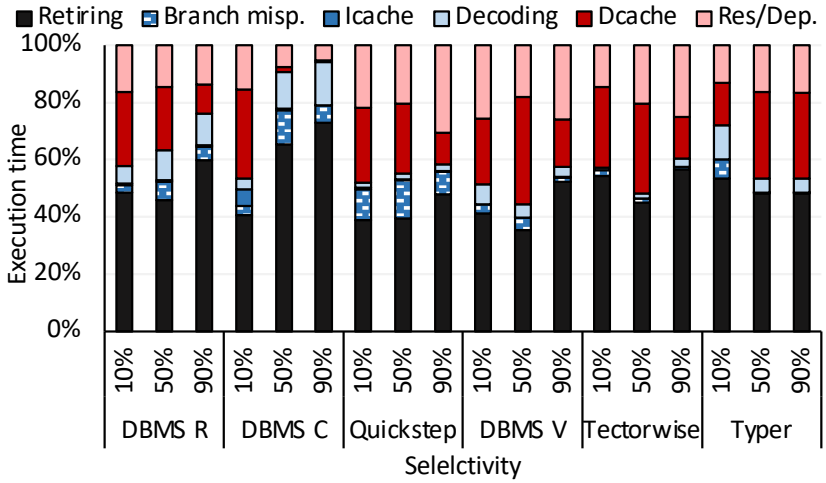


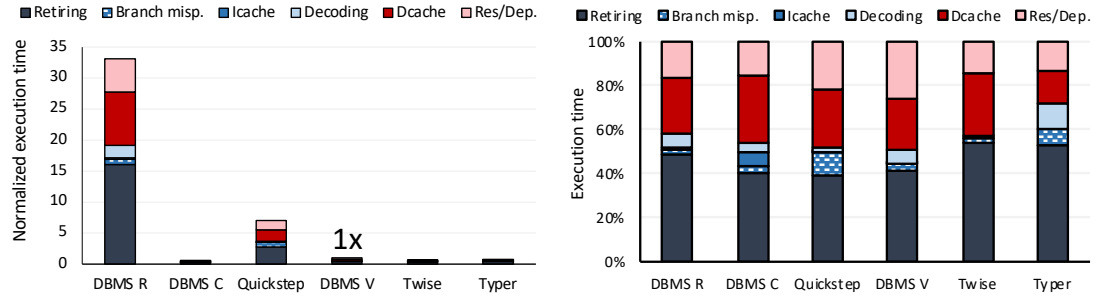
Figure 5.4 – Execution time breakdown for the selection micro-benchmark for single-threaded execution.

5.4 Selection

We present the selection micro-benchmark. Figure 5.4 and 5.6 show the breakdown of the execution times, Table 5.4 shows normalized execution times, and Table 5.5 shows consumed memory bandwidths for single- and multi-threaded executions. We use the highest performing version of the branched vs. branch-free implementations for *Tectorwise* and *Typer*. This version

	Single-threaded			Multi-threaded		
	10%	50%	90%	10%	50%	90%
R	33.1	18.9	18.5	21.1	13.1	14.6
C	0.6	3.9	10.9	0.7	2.6	8.2
Qs	7.1	5.5	6.4	3.4	3.4	4.4
V	1	1	1	1	1	1
Tw	0.7	0.7	0.7	0.4	0.7	0.7
Ty	0.8	0.7	0.4	0.4	0.9	0.7

Table 5.4 – Normalized execution times for selection micro-benchmark for single- and multi-threaded executions. The execution times are normalized to *DBMS V*.



(a) Normalized execution time breakdowns. The execution times are normalized to *DBMS V*.

(b) Execution time breakdowns streched to 100%.

Figure 5.5 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for the selection query with 10% selectivity for single-threaded execution.

is the branched version for *Typer* at 10% selectivity and the branch-free version for all the other cases.

DBMS R is 13.1 to 33.1 times slower than *DBMS V* is. It spends 50% of its time retiring instructions. To understand the relationship between the execution time and the number of retired instructions, we plot the breakdown of the normalized execution times (left) and the breakdown of the execution times stretched to 100% (right) in Figure 5.5 for 10% of selectivity and single-threaded execution. The figure shows that *DBMS R* is slower than *DBMS V* is, mainly due to the execution of a significantly larger number of instructions than *DBMS V*.

DBMS C is 40% faster than *DBMS V* and consumes a large amount of memory bandwidth at 10% selectivity. As the selectivity is increased, *DBMS C* becomes significantly slower than *DBMS V* and consumes less and less bandwidth.

DBMS C keeps its columns in 1MBs of blocks together with some meta-data information per block such as minimum, maximum and count values. In the case of low selectivities, it simply scans the meta-data and skips the blocks that are not necessary to scan. Hence, it has a very low execution time for 10% selectivity. We confirm our hypothesis by using micro-benchmarks that use only meta-data information and by obtaining similar results.

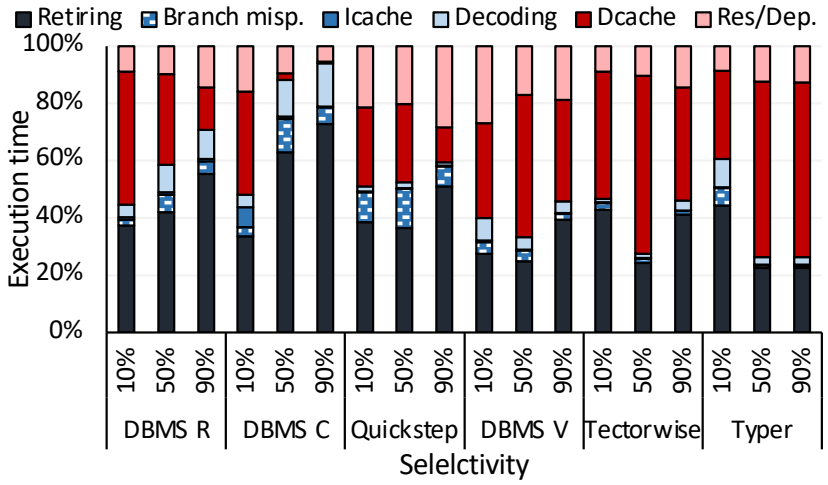


Figure 5.6 – Execution time breakdowns for selection micro-benchmark for multi-threaded execution.

	Single-threaded			Multi-threaded		
	10%	50%	90%	10%	50%	90%
R	0.7	0	0	46	36.4	23.2
C	3	0	0	34	12.8	2.6
Qs	0	0	0	9.2	18.3	10.8
V	0.6	1.9	1.9	12.6	44.6	39.7
Tw	3	6.8	4.7	50.4	62.7	58.7
Ty	3	9	8.4	51.7	62.8	62.8

Table 5.5 – Consumed bandwidth in GB/s for selection micro-benchmark for single- and multi-threaded executions.

Quickstep is 3.4 to 7.1 times slower than *DBMS V* is. It spends 40% of its time retiring instructions, and 20% of its time waiting for Dcache stalls for 10% and 50% of selectivities. Hence, *Quickstep* is slower than *DBMS V* is, due to the execution of a significantly larger number of instructions and to inefficiencies in its data-access primitives. We examine *Quickstep*'s function-call trace to understand the reasons for using large number of instructions. *Quickstep* spends 40.8% of its time in `BitVector::firstOne()`, and 15.5% in `BitVector::setBit()`. *Quickstep* uses these two functions to avoid processing the tuples that are already filtered out by the previous predicates in a conjunctive condition. `BitVector::firstOne()` relies also on a C++ construct `__builtin_clz()` to count leading zero bits of an integer. As the predicate evaluation, by itself, is a simple condition-check operation, a library function-call together with bitvector manipulations take a large fraction of *Quickstep*'s time on selection processing.

Tectorwise uses selection vectors to avoid processing already filtered tuples. A selection vector keeps the IDs of the tuples that should be evaluated for the second and onwards predicates. As selection vectors require a single cache-resident lookup, they are likely to be more efficient than bitvectors. *Quickstep* could benefit from it.

The breakdown of Dcache stalls shows that ~70% of the stalls are due to 4K Aliasing that comes from `DateLit:<` operator, which is used for date comparison. 4K Aliasing occurs when the memory addresses of successive load and store operations are aliased by 4K. In this case, hardware fails to perform the *store-to-load forwarding* optimization. This causes a five-cycle penalty and can be significant if it happens frequently. 4K Aliasing can be solved by aligning the data blocks to 32 bytes, or by changing offsets between input and output buffers [46].

DBMS V* & *Tectorwise have a 30% performance gap, except for 10% selectivity at the multi-threaded execution. *DBMS V* scales the worst at 10% selectivity. This is due to the scalability limitations of the exchange operator that *DBMS V* relies on. The exchange operator statically creates a number of producer and consumer threads and, in the case of uneven distribution of the tuples, suffers from load imbalance. As, at lower selectivities, the uneven load is likely higher, *DBMS V* scales worse at lower selectivities. *Tectorwise* uses morsel-driven parallelism that scales better under uneven loads [65].

We also examine *Typer*'s and *Tectorwise*'s function-call traces. They both spend almost 100% of their time inside the filter and aggregation functions.

Single vs. Multi-threaded Execution: At 10% selectivity, *DBMS C* stresses the memory bandwidth thanks to its fast meta-data processing technique. As a result, *DBMS C* has higher Dcache stalls for multi-threaded execution than single-threaded execution. For 50% and 90% selectivities the micro-architectural behavior is similar for the single- and multi-threaded executions due to low bandwidth stress. Similar to *DBMS C*, *Quickstep*'s micro-architectural behavior is the same for single- and multi-threaded executions. *DBMS V*, *Tectorwise*, and *Typer* all suffer significantly higher from Dcache stalls at the multi-threaded execution than the single-threaded execution, as they all highly stress the memory bandwidth.

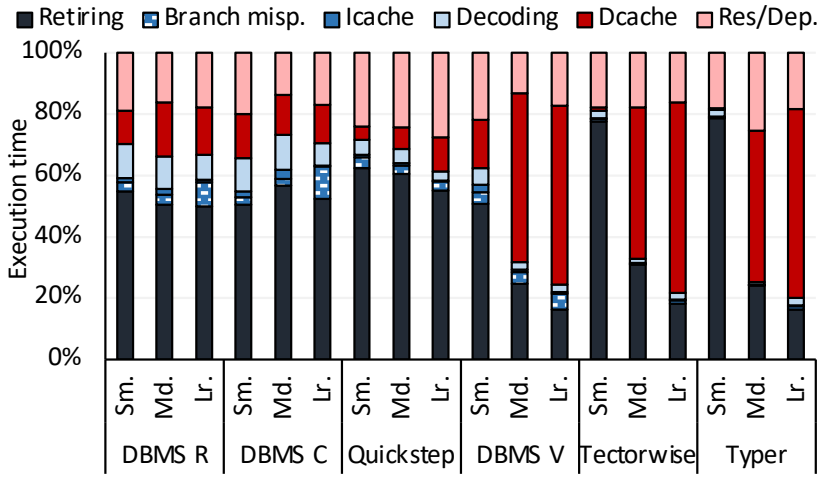
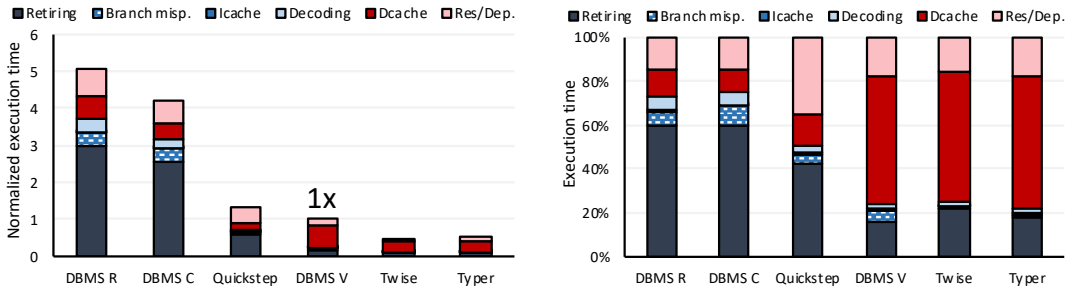


Figure 5.7 – Execution time breakdown for the join micro-benchmark for single-threaded execution.

	Single-threaded			Multi-threaded		
	Sm.	Md.	Lr.	Sm.	Md.	Lr.
R	7.2	6.8	6.1	2.0	5.3	4
C	7.8	3.8	4.8	2.1	4.5	3.5
Qs	1.9	1.7	1.1	0.4	1.3	0.8
V	1	1	1	1	1	1
Tw	0.2	0.7	0.6	0.1	0.4	0.5
Ty	0.3	0.7	0.6	0.1	0.5	0.5

Table 5.6 – Normalized execution times for join micro-benchmark for single- and multi-threaded executions. The execution times are normalized to *DBMS V*.



(a) Normalized execution time breakdowns. The execution times are normalized to *DBMS V*.

(b) Execution time breakdowns streched to 100%.

Figure 5.8 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for the large-sized join query for single-threaded execution.

5.5 Join

We present the join micro-benchmark. Figure 5.7 shows the breakdown of the execution times for single-threaded execution. Table 5.6 shows normalized execution times, and Table 5.7 shows consumed memory bandwidths for single- and multi-threaded executions. We omit the breakdown of the execution times for multi-threaded execution as it is the same as that of single-threaded. All systems use the hash join algorithm.

DBMS R* & *C are 2 to 7.2 times slower than *DBMS V* is. They spend most of the execution time retiring instructions. To understand the relationship between the execution time and the number of retired instructions, in Figure 5.8, we plot the breakdown of the normalized execution times (left) and the breakdown of the execution times stretched to 100% (right) for the large-sized join. As the figure shows, *DBMS R* and *C* are slower than *DBMS V* is, mainly due to the execution of a significantly larger number of instructions.

Quickstep is 10% slower and 20% faster than *DBMS V* for single- and multi-threaded executions for the large-sized join. The reason is that *Quickstep* converts the hash join into a *Filter Joins (FJ)* if (i) the probe-side join key is unique, and (ii) if no attribute is required from the probe side in the result of the join. In this case, FJ builds an *Exact Filter (EF)*, rather than a hash table, on the build side. An EF is a bitvector where every build key corresponds to a single

	Single-threaded			Multi-threaded		
	Sm.	Md.	Lr.	Sm.	Md.	Lr.
R	0	0	0	2.6	30.2	12.1
C	0	0	0	0.4	18.6	3.1
Qs	0	0	0	0	9.6	13.8
V	0	1	0.9	0	8.5	17.3
Tw	0	0	1.3	0	15.4	23.1
Ty	0	0	1.2	0	11.5	21.3

Table 5.7 – Consumed bandwidth in GB/s for join micro-benchmark for single- and multi-threaded executions.

bit. FJ then probes the EF to decide whether a tuple from the probe side should pass the join.

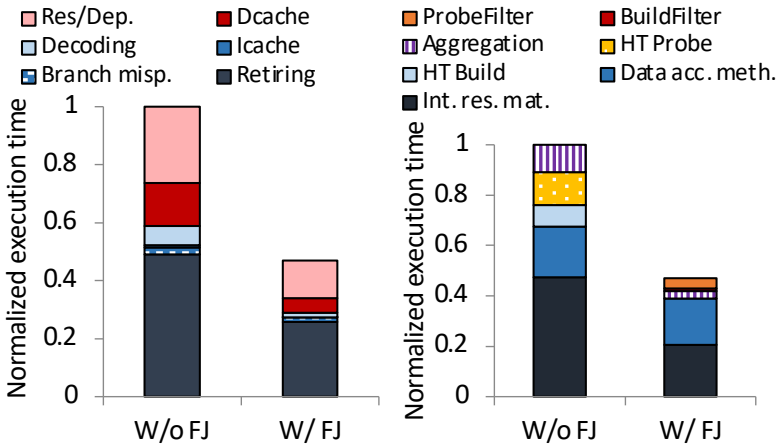


Figure 5.9 – Normalized execution time breakdowns at the hardware- (left) and software-levels (right) for *Quickstep* when it runs the large join micro-benchmark query, as single-threaded, with and without using Filter Join (FJ). The execution times are normalized to without using FJ.

In Figure 5.9, we examine the breakdown of the normalized execution time at the hardware- (left) and software-level (right), with and without using FJ. We make a best effort categorization of *Quickstep*'s methods. To illustrate this, the intermediate-result materialization category (Int. res. mat) includes methods such as `bulkInsertTuplesWithRemappedAttributes()` and `appendUntypedValue()`. Data-access methods (Data acc. meth.) include methods such as `getUntypedValue()`, `getTypedValue()`, and `next()`. The rest of the categories are all single-method categories, and includes the method that does what the category implies, e.g., HT Build method (`BuildHashWorkOrder::execute()`) builds a hash table.

FJ improves the execution time by ~50%. *Quickstep* does not suffer from Dcache stalls, even without using FJ. This is due to *Quickstep*'s materialization overhead. *Quickstep* spends (with and without FJ) half of its time in intermediate-result materialization. *Quickstep* does not implement late materialization, and hence suffers from intermediate result materialization of the join results. Intermediate materialization is known to be a major performance bottleneck

for OLAP systems [2].

DBMS V, Tectorwise & Typer spend most of their time in Dcache stalls for both middle-sized and large-sized joins. This is because the build-side tables do not fit into last-level cache for middle- and large-sized joins. Hence, the build and probe phases of the hash join algorithm are dominated by the last-level cache misses, which makes them Dcache-stalls-bound. We examine *Typer*'s and *Tectorwise*'s function-call traces. They both spend almost 100% of their time inside the join and aggregation functions.

Build vs. Probe Phases: Hash join is composed of two phases: build and probe. Both phases compute a hash value given a key. Then, they make a hash table lookup for an insertion purpose while building, and for a read purpose while probing. We observe that the micro-architectural behavior of both build and probe phases are largely Dcache-stalls dominated. Hence, the random data-accesses dominate both phases.

Probe phase dominates the execution time for single-threaded execution (61% of the time), while build phase dominates the execution time for multi-threaded execution (53% of the time), both for *Typer* and *Tectorwise*. This is due to the scalability bottlenecks of build phase that relies on atomic exchange instructions for concurrent hash-table inserts.

5.6 TPC-H

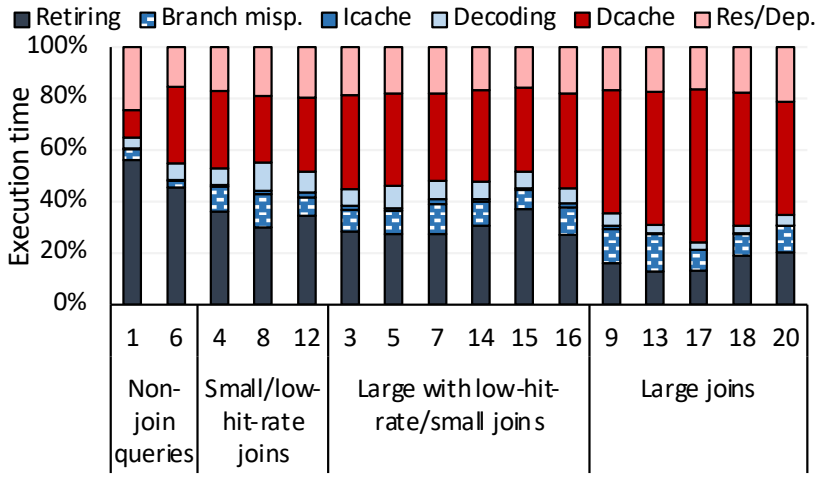


Figure 5.10 – Execution time breakdown for a large subset of TPC-H queries for single-threaded execution.

We present TPC-H benchmark evaluation. We first profile a large subset of TPC-H queries when they are run on *DBMS V*. We then choose six representative queries, and continue with the cross-system comparison.

We identify two main dimensions in the categorization of the TPC-H queries: join size and hit rate. Join size is defined by the size of the probe-side hash table, as it defines how cache-

	Q1	Q6	Q3	Q5	Q9	Q18
R	25.7	16.7	9.3	10.4	4.7	2.7
C	21	4.5	3.4	5	3.4	1.9
Qs	5	6.4	1.8	4.6	1.8	0.3
V	1	1	1	1	1	1
Tw	1.1	0.6	1.1	1	0.7	0.3
Ty	0.7	0.7	1.5	1.4	0.9	0.3

Table 5.8 – Normalized execution times for TPC-H queries for single-threaded execution. The execution times are normalized to *DBMS V*.

resident the hash join is. Hit rate is defined by the number of times that the probe side finds a matching entry at the build side. If this value is less than 10%, we identify the join as a low hit-rate join. Low hit-rate enables us to use bloom filters to reduce the number of hash probes [16]. The smaller the join size is and the lower the hit rate is, the less the Dcache stalls the join suffers from.

Figure 5.10 presents the breakdown of the execution times for single-threaded execution. Based on their micro-architectural behavior, there are four main classes of queries. Q1 and Q6 are the non-join queries and have relatively high retiring time. Q4, 8 and 12 are queries with small-sized joins or large-sized joins, both with low hit-rates. They suffer from Dcache stalls at ~25%. Q3, 5, 7, 14, 15 and 16 are queries with large joins mixed with low hit-rate large joins or small joins. These queries suffer from Dcache stalls at ~35%. Lastly, Q9, 13, 17, 18 and 20 are joins with large sizes with high hit-rates. These queries suffer from Dcache stalls at ~45%.

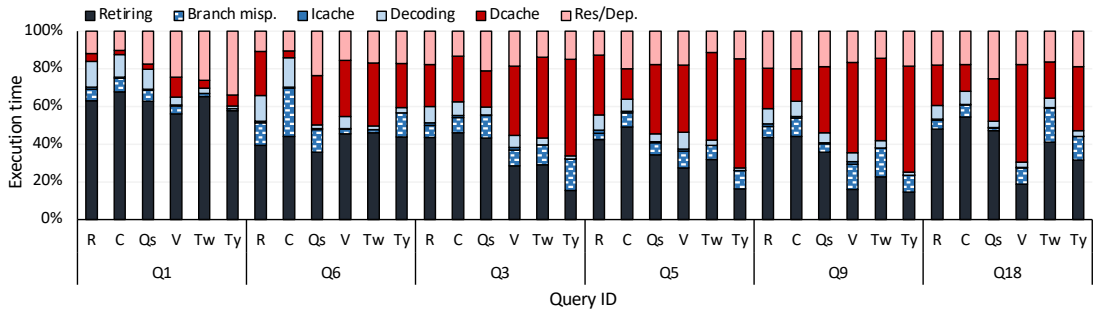
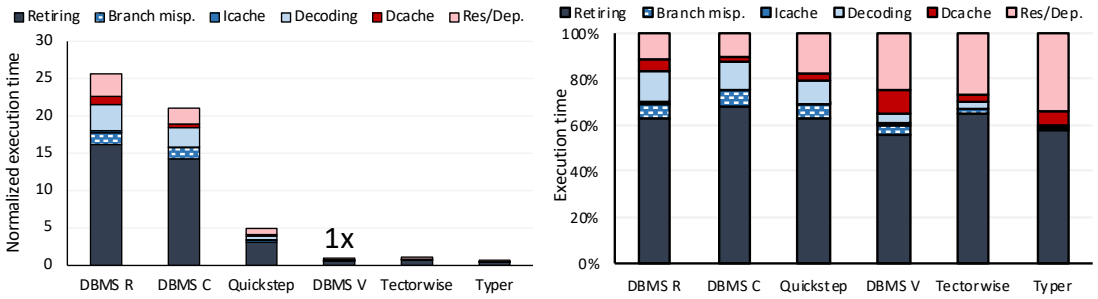


Figure 5.11 – Execution time breakdowns for TPC-H Q1, Q6, Q3, Q5, Q9 and Q18 for single-threaded execution.

We choose Q1, 6, 3, 5, 9 and 18 to continue with the cross-system evaluation. Our selection of the queries corroborates with the used TPC-H queries by [56]. Figure 5.11 shows the breakdown of the execution times for single-threaded execution. Table 5.8 and 5.9 show normalized execution times, and Table 5.10 and 5.11 show consumed bandwidth values for single- and multi-threaded executions. We present the breakdown of the execution times for multi-threaded execution only for Q6 in Figure 5.13, as they are the same as the single-threaded breakdowns for all the other queries and systems.

Q1 is an aggregation-heavy query with high temporal-locality. All the systems have high



(a) Normalized execution time breakdowns. The execution times are normalized to *DBMS V*.

(b) Execution time breakdowns streched to 100%.

Figure 5.12 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for TPC-H, Q1 for single-threaded execution.

	Q1	Q6	Q3	Q5	Q9	Q18
R	23.1	16.7	4.9	11.6	4.8	1.9
C	18.9	4.1	2.3	6.5	6.7	1.2
Qs	4.4	4.9	0.8	4.3	1.4	0.3
V	1	1	1	1	1	1
Tw	1	0.6	0.4	0.9	0.6	0.3
Ty	0.6	0.7	0.5	1.1	0.7	0.3

Table 5.9 – Normalized execution times for TPC-H queries for multi-threaded execution. The execution times are normalized to *DBMS V*.

retiring times. *DBMS R* and *C* are 25.7 and 21 times slower than *DBMS V* is. In Figure 5.12, we present the breakdown of the normalized execution times (left) and the breakdown of the execution times stretched to 100% (right) for Q1. As the figure shows, *DBMS R* and *C* are slower than *DBMS V* is, mainly due to the execution of a significantly larger number of instructions.

Quickstep is 5 times slower than *DBMS V*, due to its aggregation overhead as explained in Section 5.3. *DBMS V* and *Tectorwise* have close performances, which shows that the additional work that *DBMS V* needs to perform as being a real-life system is compensated in an aggregation-heavy query. *Typer* is 30% faster than *DBMS V* and *Tectorwise*, as *Typer* does not suffer from materialization overhead that systems following vector-at-a-time execution model suffer.

Q6 is a scan-intensive query that scans three columns and evaluates five predicates over them. We use *Typer*'s branched and *Tectorwise*'s branch-free versions. As the selectivity of the query is very low (1.9%), most of the time is spent in predicate evaluations.

DBMS C is 4.1 times slower than *DBMS V*. As the selectivity of the first predicate of Q6 is high, *DBMS C* is less likely to use its block-skipping optimization on the evaluation of the first predicate. However, for the subsequent columns, it can largely benefit from it. In addition, *DBMS C* suffers largely from branch mispredictions, suggesting that it does not predicate the conditional expressions. *Quickstep* is 4.9 times slower than *DBMS V*. It suffers from

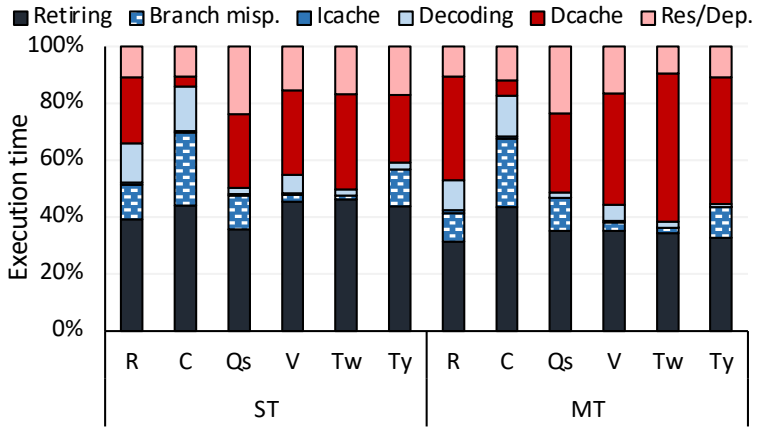
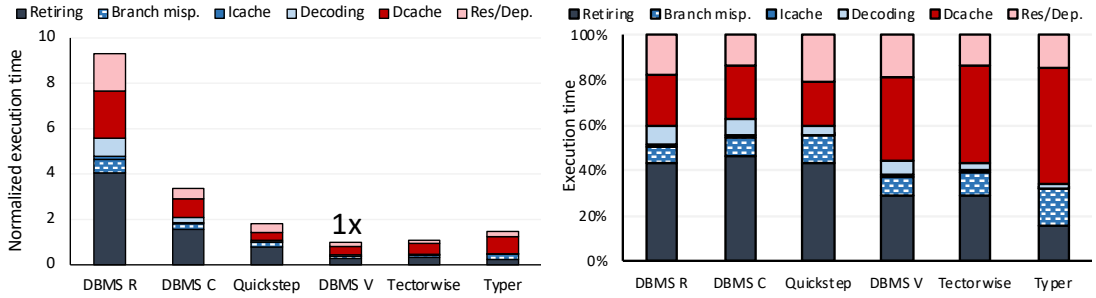


Figure 5.13 – Execution time breakdowns for TPC-H, Q6 for single- (ST) vs. multi-threaded (MT) executions.



(a) Normalized execution time breakdowns. The execution times are normalized to *DBMS V*.

(b) Execution time breakdowns streched to 100%.

Figure 5.14 – Normalized execution time breakdowns (left) and execution time breakdowns streched to 100% (right) for TPC-H, Q3 for single-threaded execution.

its selection operator overhead and Dcache stalls due to 4K Aliasing that are similar to the selection micro-benchmark shown in Section 5.4.

DBMS V, *Tectorwise*, and *Typer* have high retiring time at single-threaded execution, but significantly suffer from Dcache stalls at multi-threaded execution (see Figure 5.13). This is because all the three systems approach the bandwidth limits (see Table 5.11). *DBMS C*'s and *Quickstep*'s execution time breakdowns are similar at single- and multi-threaded executions due to their low bandwidth stresses.

Q3 is a join-intensive query where three large tables of TPC-H, lineitem, orders and customer are joined, with cardinalities of 240M, 51M and 2.1M. The query firstly joins orders and customer tables, and then joins the lineitem table with the result of former join. It finally performs a group by operation over 2.1M tuples into 792K groups.

DBMS R and *C* are significantly slower than *DBMS V* is. In Figure 5.14, we present the breakdown of the normalized execution times (left) and the breakdown of the execution times

	Q1	Q6	Q3	Q5	Q9	Q18
R	0	0	0	0	0	0
C	0	0	0	0	0.1	0.1
Qs	0	0	0	0	0	0.2
V	0.4	2.2	0.4	0.1	0.4	1
Tw	0	5	0.9	0.2	1.1	0.6
Ty	1	5	0.1	0.1	0.3	1.3

Table 5.10 – Consumed bandwidth in GB/s for TPC-H queries for single-threaded execution.

stretched to 100% (right) for Q3. The figure shows that *DBMS R* and *C* execute a number instructions significantly larger than *DBMS V* executes, which makes them slower than *DBMS V* is. *Quickstep* is 1.8 times slower and 20% faster than *DBMS V*, at the single- and multi-threaded executions, respectively. *Quickstep* implements two important join optimizations: Filter Joins (FJ) and Lookahead Information Passing (LIP) filters. FJ replaces the build-side hash table with a bitvector, as explained in Section 5.5. LIP filters are bloom filters that are passed down in the query plan so that a join can drop rows that would satisfy the current join, but not a future join.

We turn on/off FJ and LIP filter, and we measure how useful they are for Q3. FJ improves Q3's execution time by 47%, and LIP filters further improve it by 28%; overall, providing a 65% reduction in the execution time. Despite eliminating the major costs of hash joins in Q3, *Quickstep* still spends a significant amount of time retiring instructions, which shows that it nevertheless suffers from overhead that is identified in Section 5.3, 5.4 and 5.5.

Tectorwise and *Typer* are slower than *DBMS V*, because *DBMS V*, as the hit rate is low (1%), uses bloom filter on its join with lineitem and the result of the orders and customer join. To test our hypothesis, we perform a subquery analysis for Q3 by excluding, and then including, the lineitem join. We see that the Dcache stalls are DRAM-dominated without the lineitem join, whereas L2- and L3-dominated with the lineitem join, which supports our conclusion.

Tectorwise is faster than *Typer*. *Tectorwise* separates hash computing from hash probing by saving computed hashes in an intermediate vector. This enables the overlapping of costly random data-accesses at the hash probing phase. *Typer*, on the other hand, performs hash computation and hash probing one-after-the-other. It also combines the filtering condition and hash table probe operation in a single if condition. This mixes the random data-access further with a sequential scan of a column, which is used to filter the data (such as: `if(o_orderdate[i] < c1 & ht1.contains( o_custkey[i]))`). As a result, *Typer* is not able to overlap the random data-accesses as much as *Tectorwise* does. This shows that materialization overhead of the vectorized engine pays off for the hash join operation unlike the case for projection and selection. *DBMS V*, *Tectorwise* and *Typer* are all Dcache-stalls-dominated in their execution time due to hash join's large number of random data-accesses.

Q5 is a join-intensive query similar to Q3. *DBMS R* and *C* are 10.4 and 5 times slower than

	Q1	Q6	Q3	Q5	Q9	Q18
R	7.4	41.9	30.4	22.8	21.3	24.2
C	0	5.9	12.8	11.2	8.2	7.4
Qs	4.9	6.7	8.5	7.6	7.8	10.4
V	22.9	40.8	12.1	9.7	16.2	21.7
Tw	18.9	56.2	22.4	20.7	27.7	17.8
Ty	29.1	57.9	16.7	14.4	21.9	21.1

Table 5.11 – Consumed bandwidth in GB/s for TPC-H queries for multi-threaded execution.

DBMS V, similar to Q3. *Quickstep* is 4.6 times slower than *DBMS V*. Unlike Q3, *Quickstep* is not able benefit from FJ and only partially benefits from LIP filters when running Q5. This is because the orders and customer join in Q5 requires attributes from both from the probe and build sides. Similarly, it is not able to benefit from LIP filters as much as Q3 does as it can only filter out rows from the customer table while joining with the small nation and region tables, which are already not costly.

DBMS V and *Tectorwise* have a comparable performance, as *DBMS V* uses a bloom filter while joining the lineitem table. The join's hit-rate is 3%. Similarly, *DBMS V* spends less time in Dcache stalls compared to *Tectorwise*. *Tectorwise* is faster than *Typer* as *Tectorwise* can overlap random data-accesses. Nevertheless, all three high-performance systems spend most of their time in Dcache stalls.

Q9 is a join-intensive query with large joins and high hit-rates. *DBMS R* and *C* are 4.7 and 3.4 times slower than *DBMS V*, similar to Q3 and Q5. *Quickstep* is only 1.8 times slower than *DBMS V*. LIP filters improve Q9's time by 55% thanks to filtering out lineitem tuples in its join with the sub-tree of joins of partsupp, part and supplier.

DBMS V is not able to benefit from bloom filters for Q9 as the hit rate is high for all the joins. Hit rate is 20% for the join between the orders and lineitem table, and 50% for the join between the partsupp table and the sub-tree of joins of the rest of the tables. *Tectorwise* is once again faster than *Typer* thanks to its random data-access overlapping ability. *DBMS V*, *Tectorwise* and *Typer* all spend most of their time in Dcache stalls as Q9 is join-intensive.

Q18 contains a large group by on the lineitem table based on `l_orderkey`, without any filter on top of the lineitem table. It creates 105M groups out of the 420M-sized lineitem table (the scaling factor is 70). For each group, it requires doing a simple `sum(l_quantity)` aggregation.

DBMS V is 3.3x times slower than *Tectorwise* and *Typer*. This is because *DBMS V*'s query optimizer produces a sub-optimal plan. The optimal plan does a single group by over the lineitem table, filters it based on the `having` condition, and feeds the result into a series of joins with orders, customer and another lineitem. This way, the amount of data fed to the upper levels of the query is reduced by orders-of-magnitude as the number of groups that satisfy the `having` condition is four orders of magnitude less than the total number of groups. *DBMS V*, however, is not able to push the large group by down the tree. It requires a full join

	Proj.+Q3		Q6+Q3	
	Proj.	Q3	Q6	Q3
Qs	1	1.1	1	1
V	1.1	1.6	1.1	1.5
Ty	1.1	1.9	1.0	1.4

Table 5.12 – Normalized execution times for mixed query workload evaluation. Mixed projection and Q3: Numerator is execution time when concurrently running projection and Q3. Denominator is execution time when running projection or Q3 alone on the server. Mixed Q6 and Q3 follows the same normalization scheme.

among the lineitem, orders and customer tables, which it further joins with the filtered group by. This results in high execution time, together with high Dcache stalls. As a result, it produces 3.3x higher execution time. Similarly, it spends significantly larger portion of its execution time in Dcache stalls compared to the other systems.

Quickstep is as fast as *Tectorwise* and *Typer* thanks to using LIP filters, which reduces *Quickstep*'s execution time by 30%. Furthermore, *Quickstep* does not suffer from the intermediate result materialization, as the intermediate results are small (due to the `having` condition) for Q18.

Tectorwise and *Typer* have relatively high retiring time for Q18 compared to other join-intensive queries. *Typer* does thread-local pre-aggregations for each data block, i.e., morsel, it processes. Then, *Typer* globally combines the local pre-aggregations for the final group by. Similarly, *Tectorwise* creates local groups per vector. Then, it combines the groups globally at the end. Hence, they both work with smaller-sized hash tables that are more cache-resident, which results in high retiring times.

5.7 Mixed Query Workload

We present mixed query workload evaluation. Scan-intensive queries are bandwidth-bound, hence do not scale after a certain number of cores. Join-intensive queries do not create enough memory traffic, hence leave the bandwidth underutilized. In this section, we concurrently run a scan- and join-intensive query, where we create enough memory traffic and also use all the cores on the chip.

We examine the following two scenarios: (i) projection micro-benchmark query with the degree of four running with TPC-H, Q3, and (ii) TPC-H, Q6 running with TPC-H, Q3. We use *Quickstep*, *DBMS V* and *Typer* to evaluate the concurrency scenarios. We choose these three systems, as they stress the memory bandwidth at different levels.

We use eight threads for projection and Q6, and use six threads for Q3 on *Typer*, as projection does not scale after eight cores on *Typer*. We use ten threads for projection and Q6, and use four threads for Q3 on *DBMS V*, as Q3 does not scale after four cores on *DBMS V*[1]. We use the

[1] We micro-benchmarked Q3 on *DBMS V* by varying the selectivity of the filter on the lineitem table from 100%

same configuration for *Quickstep* as for *Typer*.

Table 5.12 presents concurrent execution times normalized to the corresponding non-concurrent execution times. For *DBMS V* and *Typer*, Q3's execution time is significantly higher when it runs with the projection and TPC-H, Q6, whereas, for *Quickstep*, Q3's execution time does not change significantly. Because *DBMS V* and *Typer* sufficiently stress the memory bandwidth to interfere with Q3, while *Quickstep* does not. On the other hand, projection's and Q6's execution times do not change significantly. Hence, concurrent execution enables us to use the underutilized cores left by the scan-intensive query. However, it causes interference in the shared memory bandwidth, hence results in a decreased execution time for the join-intensive query.

We also examine the micro-architectural behavior. The results are as expected. Q3's Dcache stalls are increased substantially when running with projection/Q6 on *Typer* and *DBMS V*, whereas remain same on *Quickstep*. The total consumed bandwidth of a concurrent execution is the sum of the individual bandwidth consumptions unless the sum reaches to the maximum bandwidth.

```
1 // branched version
2 if( (a < v1) & (b < v2) & (c < v3) )
3     result += (d + e);
4 // branch-free, predicated version
5 bool decision = (a < v1) & (b < v2) & (c < v3);
6 result += (decision * (d + e));
```

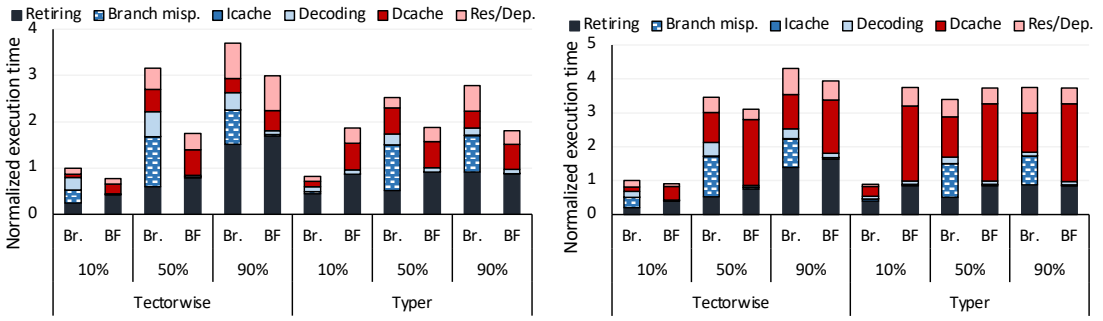
Listing 5.1 – Predication example.

5.8 Predication

We present predication evaluation. Listing 5.1 presents an example of predication. Line 1 to 3 present regular branched execution, whereas Line 4 to 6 present the predicated execution. Predication takes the conditional expression out of the `if()` statement and assigns the expression to the variable `decision`. Then, it uses the `decision` variable to compute the final result. If the `decision` is zero, it will not affect the final result, whereas if the `decision` is one, it will update the final result as in the branched execution. Predication requires more computation but allows for avoiding costly branch mispredictions.

The conditional expression uses bitwise and as opposed to logical and. Compiler generates a branch instruction for each logical and. Hence, a logical and triggers branch predictor even if it is not in an `if()` statement. However, bitwise and translates into a set of bitwise operations followed by a single conditional branch, which can be eliminated by taking the expression out of the `if()` statement.

to 50%. We realized that Q3 starts not scaling when the selectivity drops below 70%. This suggests that the reason for not scaling is the load imbalance issue of the exchange operator that creates uneven loads at lower selectivities.



(a) Single-threaded.

(b) Multi-threaded.

Figure 5.15 – Normalized execution time breakdowns for predication for single- (left) and multi-threaded (right) executions. The execution times are normalized to *Tectorwise* using branched implementation with 10% selectivity (left-most bar in each graph).

Figure 5.15 (left) shows the breakdown of the normalized execution times for single-threaded execution where all values are normalized to the left-most bar. We see that branch mispredictions are a major source of cost. Predication reduces the execution time for all the cases except for *Typer* at 10% selectivity.

Typer improves performance the highest at 90%, although branch misprediction cost is the highest at 50%. This is because the computation overhead that predication brings is less at 90% compared to 50%, as the unpredicated query computes the aggregation for most of the tuples at 90%.

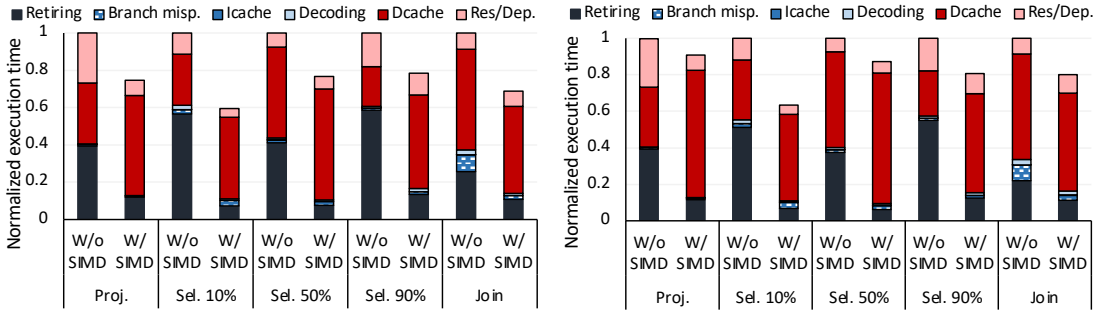
Typer suffers significantly from branch mispredictions at 90% selectivity. Because *Typer* uses bitwise and to implement the conjunction, hence suffers from the overall selectivity of the conjunction rather than the individual predicate selectivities. In this case, it is 90% × 90% × 90% = 73%, which is less predictable than 90%.

Figure 5.15 (right) presents the breakdown of the normalized execution times for multi-threaded execution. It shows that the majority of the performance gains are lost due to bandwidth limitations for all the queries. Hence, while predication can significantly reduce the execution time, its multi-core benefits are limited by the maximum memory bandwidth.

We also profiled predicated TPC-H, Q6 on *Typer* and *Tectorwise*, and we reached similar conclusions.

5.9 SIMD

We present SIMD evaluation. SIMD instructions perform multiple arithmetic/logic operation in a single instruction. We test *Tectorwise* when running the projection, selection and join micro-benchmarks, with and without using SIMD. As our Broadwell server does not support AVX-512 instructions, we do all the SIMD experiments on a Skylake server supporting AVX-512



(a) Single-threaded.

(b) Multi-threaded.

Figure 5.16 – Normalized execution time breakdowns for SIMD for single- (left) and multi-threaded (right) executions. The execution times are normalized to without using SIMD.

	Proj.	Sl.10%	Sl.50%	Sl.90%	Join
W/o SIMD	6	4	8	6	2.8
W/ SIMD	8	8	8	8	4.5

Table 5.13 – Consumed bandwidth in GB/s for SIMD for single-threaded execution.

instructions.

The Skylake server has a different memory hierarchy than that of the Broadwell server. As a result, the reported values that do not use SIMD do not exactly match with the values reported earlier in the chapter (see Section 5.2, Hardware subsection for more details).

5.9.1 Projection & Selection

Figure 5.16 (left) shows the breakdown of the normalized execution times. Table 5.13 presents single-core bandwidth consumption values. We use the predicated, branch-free versions of the selection queries as SIMD is more effective when branch mispredictions are eliminated. The figure shows that there is a 70% to 87% decrease in the amount of retiring time for all four cases. As the retiring time is correlated to the number of retired instructions, SIMD successfully reduces the number of retired instructions.

While the number of retired instructions is reduced, Dcache stalls are increased by 20% to 2.3x. Hence, overall SIMD gains are limited by the increased Dcache stalls. Table 5.13 shows that the projection and selection queries are single-core-bandwidth bound when using SIMD, as the maximum per-core bandwidth is 8GB/s. Hence, per-core SIMD gains are limited by per-core bandwidth.

Figure 5.16 (right) shows breakdown of the normalized execution times, and Table 5.14 presents the bandwidth consumption for multi-threaded execution. SIMD gains are less at the multi-threaded execution due to approaching the bandwidth limits.

We also run projection and predicated selection without SIMD on *Typer* on the Skylake server.

	Proj.	Sl.10%	Sl.50%	Sl.90%	Join
W/o SIMD	71.4	49.9	76.2	63.7	34.8
W/ SIMD	79.4	73.1	81.4	79.2	40.4

Table 5.14 – Consumed bandwidth in GB/s for SIMD for multi-threaded execution.

Typer saturates the per-core and multi-core bandwidth and does not scale after 8 cores. Hence, its SIMD gains would be less than that of *Tectorwise* that, without SIMD, is not able to saturate per-core or per-socket bandwidths.

5.9.2 Join

Figure 5.16 shows that SIMD significantly reduces the number of retired instruction and does not increase the Dcache stalls. Table 5.13 shows that single-core bandwidth consumption is well below the maximum (7GB/s), without and with SIMD. Hence, SIMD is able to exploit the available core bandwidth and reduce the execution time, without increasing the Dcache stalls time. SIMD gains are less pronounced at the multi-threaded execution, due to the stress on the memory bandwidth. As Table 5.14 shows, join consumes 40GB/s of the 60GB/s random access bandwidth.

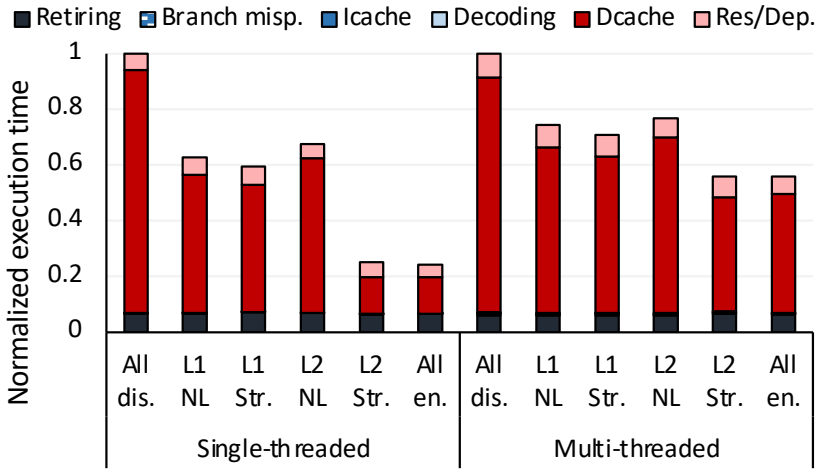


Figure 5.17 – Normalized execution time breakdowns for hardware prefetcher for single- and multi-threaded executions. The execution times are normalized to having all prefetchers disabled.

5.10 Hardware Prefetchers

We present the hardware prefetcher evaluation. We study four hardware prefetchers that today's server processors provide: L1 next line (L1 NL), L1 streamer (L1 Str.), L2 next line (L2 NL) and L2 streamer (L2 Str.) prefetchers. We turn on each prefetcher individually and examine its effect on the micro-architectural behavior.

	All dis.	L1 NL	L1 S	L2 NL	L2 S	All en.
ST	1	3	3	3	8.4	10.1
MT	32.9	42.4	43.2	41.2	62.5	62.8

Table 5.15 – Consumed bandwidth in GB/s for hardware prefetcher for single- (ST) and multi-threaded (MT) executions.

	DBMS V		Twise		Typer	
	2H	28H	2H	28H	2H	28H
Proj.	0.95	1.03	0.82	1.00	0.92	1.00
Join	0.80	0.84	0.78	0.83	0.77	0.82
Q6	0.82	1.03	0.78	0.93	0.76	0.95
Q3	1.03	1.01	0.85	0.90	0.66	0.71
Sel.50%-Br.	-	-	0.66	0.96	0.68	0.94
Sel.50%-BF	-	-	0.83	1.01	0.87	1.01

Table 5.16 – Normalized execution times for hyper-threading. 2H: The throughput of running two hyper-threads on the same physical core is normalized to the throughput of running one hyper-thread on the same physical core. 28H: The throughput of running 28 hyper-threads on 14 physical cores of a same socket is normalized to throughput of running 14 hyper-threads on 14 physical cores of the same socket.

Figure 5.17 shows the breakdown of the normalized execution times, and Table 5.15 shows the consumed bandwidth values when running the projection query of degree four on *Typer* for single- and multi-threaded executions.

Prefetchers reduce the execution time by 75% for single-threaded execution. This shows that hardware prefetchers are highly useful for a sequential-scan-heavy query. For multi-threaded execution, prefetchers reduce the execution time less than single-threaded execution. Hence, the benefits of the prefetchers are limited by the maximum bandwidth.

We also examine the projection on Tectoriwse and the branched and branch-free selection on *Typer* and *Tectorwise*. The results agree with our findings for the projection query on *Typer*. We also examined the join micro-benchmark. Prefetchers reduce the execution time modestly by ~ 20% for the large-sized join both for *Typer* and *Tectorwise*.

5.11 Hyper-threading and Turbo-boost

We present the hyper-threading (HT) and turbo-boost (TB) evaluation. We first compare single-core performance only having TB enabled. We examine *DBMS V, Tectorwise* and *Typer* for the projection (of degree four) query, the large join query, branched and branch-free versions of the selection query (at 50% selectivity), TPC-H, Q1, Q6, and Q3. The maximum speedup we observe is by 27% for TPC-H, Q1 that runs on *Typer*. As Q1 is arithmetic-operation-heavy, and *Typer* does not suffer from the materialization overhead, turbo-boost worked the best. The other improvements were modest and were 10 to 20%. TB's benefits are less as the

	Proj.		Join	
	2H	28H	2H	28H
All dis.	0.66	0.74	0.75	0.80
L1 NL	0.91	0.94	0.78	0.81
L1 Str.	0.92	0.93	0.76	0.81
L2 NL	0.80	0.86	0.75	0.81
L2 Str.	0.93	1.01	0.75	0.81
All en.	0.92	1.00	0.77	0.82

Table 5.17 – Normalized execution times for hyper-threading for different hardware prefetcher configurations on *Typer*. 2H: The throughput of running two hyper-threads on the same physical core is normalized to the throughput of running one hyper-thread on the same physical core. 28H: The throughput of running 28 hyper-threads on 14 physical cores of a same socket is normalized to throughput of running 14 hyper-threads on 14 physical cores of the same socket.

	Proj.		Sel.10%-BF		Join	
	2H	28H	2H	28H	2H	28H
W/o SIMD	0.72	0.94	0.71	0.76	0.77	0.83
W/ SIMD	0.82	0.96	0.71	0.93	0.74	0.74

Table 5.18 – Normalized execution times for hyper-threading with and without SIMD on *Tectorwise*. 2H: The throughput of running two hyper-threads on the same physical core is normalized to the throughput of running one hyper-thread on the same physical core. 28H: The throughput of running 28 hyper-threads on 14 physical cores of a same socket is normalized to the throughput of running 14 hyper-threads on the 14 physical cores of the same socket.

number of cores is increased. Hence, the single-core results provide the best case. Hence, TB-alone provides only modest performance improvements.

Next, in addition to TB, we enable HT. We examine how much HT improves the performance. We examine two scenarios: 2H and 28H. 2H shows the execution time of using 2 HTs that share a physical core normalized to the execution time of using one physical core. 28H shows the execution time of using 28 HTs that share 14 physical cores normalized to the execution time of using 14 physical cores. All the queries scale well across 2 physical cores.

Table 5.16 presents the results for a selection of queries we have covered up to now. Proj. is the projection query with the degree of four, and Join is the large-sized join query. 2H reduces the execution time maximum by 34%, for *Typer* when running Q3. As *Typer* is not able to effectively overlap the random data-accesses at the software-level, HT enables overlapping them. Similarly, branched selection query at 50% selectivity benefits 34% on *Tectorwise* and 32% on *Typer* as it can overlap the branch misprediction stalls. For the rest of the cases, 2H improve the performance modestly by 20% on average. 28H improves performance less than that of 2H due to the increased stress on memory bandwidth.

Table 5.17 shows how useful HTs are for different prefetcher configurations, for the projection

(of degree four) and large join query on *Typer*. The less aggressive the prefetchers are, the more benefits HTs provide. Hence, while prefetchers significantly improve the performance, it limits the benefit of HTs. For join, HT benefits do not change much, as prefetchers are not very useful for join.

Table 5.18 presents the case for SIMD, for the projection query (of degree four), selection query at 10% selectivity (branch-free), and large join query on *Tectorwise* (using the Skylake server). It shows that HT is less useful when using SIMD, though the difference usually is not high.

5.12 Conclusion

In this chapter, we have presented micro-architectural analysis of OLAP workloads. The results show that the commercial row-store and its column-store extension efficiently use the CPU cycles; however, they require the execution of a significantly larger number of instructions hence are 2 to 56 times slower than the column-stores that follow the tuple-at-a-time, vector-at-a-time, and compiled execution models. The column-store that follows the tuple-at-a-time execution model also efficiently uses the CPU cycles; however, it requires a significantly larger number of instructions hence is 1.7 to 5 times slower than the column-stores that follow the vector-at-a-time and compiled execution models.

The column-stores following the vector-at-a-time and compiled execution models spend most of their execution time waiting for data-cache misses. The stalls due to the data-cache misses are caused by the high pressure on the memory bandwidth, or by random data-accesses. Concurrently executing scan-intensive and join-intensive queries enable the saturation of both memory bandwidth and the number of cores. However, the concurrent executions create interference in the shared memory-bandwidth and result in a sub-optimal performance.

6 Hybrid Transactional and Analytical Processing Workloads

Application requirements and workload characteristics have evolved in the past five years. Customers today require running real-time analytical queries on the fresh transactional data. This demand gave rise to Hybrid Transactional and Analytical Processing (HTAP) systems and workloads. HTAP systems combine individual OLTP and OLAP systems into a single, unified system and concurrently serve for OLTP and OLAP workloads. An important problem the HTAP systems face is the interference problem. The individual OLTP and OLAP components interfere with each other due to sharing the fresh, transactional data and also sharing the hardware resources.

In this chapter, we present the performance characterization of HTAP systems with a focus on interference. The results show that the OLAP query execution time is exponentially increased if the OLTP component generates fresh tuples faster than the OLAP component can process them. Hence, the OLAP component should be allocated enough resources to process the fresh tuples faster than the OLTP component generates them. The OLTP and OLAP components of the real-life systems do not interfere with each other in the shared hardware resources, whereas the OLTP and OLAP components of the academic prototype interfere with each other in the shared hardware resources. Although the OLAP throughput is not affected by the concurrently running OLTP workload, the OLTP throughput drops by 22 to 40%.

6.1 Introduction

As workload characteristics and requirements evolve, data management system architectures evolved into separate transactional and analytical processing systems. Online Transactional Processing (OLTP) systems run update-intensive point queries with strong consistency and durability guarantees. Online Analytical Processing (OLAP) systems run read-only, arithmetic-operation- and bandwidth-intensive queries with tight, hardware-friendly execution loops processing large amounts of data. Using separate transactional and analytical processing systems enabled system designers to specialize to a particular type of workload, and hence, design systems delivering orders of magnitude higher performance than using a single system

both for transactions and analytical queries [18].

On the other hand, there has been an increasing demand for running real-time, analytical queries on fresh, transactional data. Hybrid Transaction and Analytical Processing (HTAP) systems combine OLTP and OLAP engines into a single, unified system, and allow running OLTP and OLAP systems concurrently on the same data, and on the same server. Thereby, the OLAP system can access to the fresh-most, transactional data, and provide real-time insights on the fast changing transactional data. OLTP and OLAP systems running on the same data creates software-level interference. As the OLTP and OLAP components access the same data, the components interfere with each other at the software-level. OLTP and OLAP systems running on the same server creates the challenge of hardware resource sharing. While an individual OLTP system can scale well on a dedicated server, an OLTP engine embedded in an HTAP system competes with the OLAP engine for the shared hardware resources such as CPU caches and memory bandwidth, and hence faces scalability limitations.

Micro-architectural analysis is important in understanding the main performance bottlenecks, and hence designing high-performant database systems. As we have shown in Chapter 4 and 5, while in-memory OLTP systems mainly suffer from long-latency data-cache misses, column-stores, i.e., OLAP systems using a column-oriented storage, suffer from long-latency data-cache misses as well as saturated memory bandwidth. HTAP systems combine OLTP and OLAP systems into a single, unified engine, and hence create a novel scenario in terms of hardware-software interaction due to the shared data and hardware resource challenges they face.

In this chapter, we present a performance characterization of a breadth of HTAP systems. We use two real-life HTAP systems and one academic prototype that we build based on existing open-source OLTP and OLAP systems. We quantify the level of interference for HTAP workloads both for real-life systems and the academic prototype. In this chapter, we show the following:

- Software-level interference depends on how rapidly the OLTP component generates fresh data and how fast the OLAP component consumes the fresh data. We empirically and theoretically characterize the query execution time in the face of continuous fresh-data ingestion. We show that OLAP query execution time is exponentially increased if the OLTP component generates fresh tuples faster than the OLAP component processes them. Hence, the OLAP component should be allocated enough resources to process the fresh tuples faster than the OLTP component can generate them.
- The OLTP and OLAP components of the real-life HTAP systems do not interfere with each other in the shared hardware resources. Both the OLTP and OLAP components of the HTAP systems only lightly stress the memory bandwidth with a less than 15% memory-bandwidth consumption.
- The OLTP and OLAP components of the academic prototype interfere with each other

in the shared hardware resources. Although the OLAP throughput is not affected by the concurrently running OLTP workload, the OLTP throughput drops by 40% when running with a sequential-scan-heavy projection query, and by 22% when running with a random-access-intensive join query. The OLTP component lightly stresses the memory bandwidth with a less than 15% memory bandwidth consumption, whereas the OLAP component significantly stresses memory bandwidth with up to 90% memory bandwidth consumption.

The rest of the chapter is organized as follows. Section 6.2 presents the setup and methodology. Section 6.3 and 6.4 present the hardware- and software-level interference analysis of the real-life systems. Section 6.5 presents the analysis of the academic prototype that we built based on existing open-source OLTP and OLAP engines, where we examine the interference deeper by providing a theoretical formulation of query execution time for the software-level interference, and study the hardware-level interference for the individual micro-architectural resources. Finally, Section 6.6 concludes.

6.2 Setup and Methodology

In this section, we present the experimental setup and methodology.

Software- and hardware-level interference: We define interference as the amount of throughput drop for the OLTP or OLAP component. We first run OLTP and OLAP concurrently for the given configuration. We, then, run OLTP and OLAP separately alone on the server with the same configuration. We divide the former to the latter and use it to quantify the interference. For software-level interference, running alone means to run OLTP or OLAP on an isolated database without having the other component (OLTP for OLAP, OLAP for OLTP) using the same database. For hardware-level interference, running alone means to run OLTP or OLAP alone on the NUMA node (i.e., CPU socket) without having the other component (OLTP for OLAP and OLAP for OLTP) concurrently running on the same NUMA node (i.e., CPU socket).

Benchmarks: We use micro-benchmarks and the CH benchmark [23]. We use projection and join microbenchmarks as they contain the basic data access patterns that database workloads exhibit: sequential-scan and random-access. All the systems use the hash join algorithm when running the join microbenchmark. For both micro-benchmarks, the OLTP workload is a single transaction with a single update statement that updates the value of a randomly chosen row. There is a single table with key and value columns. Keys are unique.

All the microbenchmarks use the CH benchmark schema. The projection microbenchmark does a single SUM() over a single column from the orderline table. *ol_amount* column. We add the projected column inside the SUM(). The join joins the orderline and orders table, and it does a SUM() over the addition of the column that the projection query uses.

We profile Q3 and Q6 of the CH benchmark, as they represent two main categories of the CH

Processor	Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell)
#sockets	2
#cores per socket	14
Hyper-threading	Off
Turbo-boost	Off
Clock speed	2.40GHz
Per-core bandwidth	12GB/s (sequential) 7GB/s (random)
Per-socket bandwidth	66GB/s (sequential) 60GB/s (random)
L1I / L1D (per core)	32KB / 32KB 16-cycle miss latency
L2 (per core)	256KB 26-cycle miss latency
L3 (shared)	(inclusive) 35MB 160-cycle miss latency
Memory	256GB

Table 6.1 – Server parameters.

benchmark: (i) non-join queries and (ii) join-intensive queries. We profile the New-Order transaction of the CH benchmark as it is the heaviest transaction among the most frequently executed transactions in the CH benchmark.

Hardware: We conduct our experiments on an Intel Broadwell server. Table 6.1 presents the server parameters. We use Intel's Memory Latency Checker (MLC) [47] to measure cache access-latencies and maximum single/multi-core and random/sequential-access bandwidth.

HTAP systems: We examine two real-life HTAP systems, *DBMS A* and *DBMS B*, and one academic prototype that we built based on existing open-source OLTP and OLAP systems, Silo [114] and Typer [56].

OS & Compiler: We use Ubuntu 16.04.6 LTS and gcc 5.4.0.

VTune: We use Intel VTune 2020 on the Broadwell server. We use VTune's built-in uarch-exploration (general-exploration on VTune 2018) analysis for the breakdown of the CPU cycles, which performs Intel's Top-down Micro-architectural Analysis Methodology (TMAM) explained in Chapter 3. We use VTune's built-in memory-access analysis to measure the consumed memory bandwidth. As we numa-localize our experiments on a single socket, we report average bandwidth per-socket values.

We provide an overview of Intel's TMAM explained in Chapter 3. Each instruction issue slot is categorized into one of two components: retiring and stalling. A retiring slot is a slot where the slot is used for retiring an instruction. A stalling slot is a slot where the slot stalls, i.e., has to wait due to a particular issue. Ideally, all issue slots would be used for retiring.

Stalling slots are further decomposed into five components: (i) branch misprediction, (ii) Icache, (iii) decoding, (iv) Dcache and (v) resource/dependency stalls. Branch misprediction stalls are the slots that stall due mispredicted branch instructions. Today's processors use a hardware unit called branch predictor; it predicts the outcome of a branch instruction (i.e., an if() statement) and speculatively executes instructions per the predicted branch direction and/or target. If the processor then realizes the prediction is not correct, it undoes whatever it has been doing and starts executing the correct set of instructions. This cost is defined as the branch misprediction stalls and can be very costly, as it requires canceling a large amount of work. Icache stalls are the slots that stall due to instruction-cache and instruction translation lookaside buffer misses. Decoding stalls are the slots that stall due to sub-optimal micro-architectural implementation of the instruction decoding unit. Dcache stalls are the slots that stall due to data-cache misses. Resource/dependency stalls are the slots that stall due to resource and/or data dependencies. For example, if two instructions require using the same arithmetic-logic unit, one has to wait for the other. This time is identified as the resource/dependency time. Or, if an instruction's operand depends on the result of another instruction, the instruction with the dependent operand has to wait for the other instruction to finish. This time is identified as the resource/dependency time.

Measurements: For every experiment, we first populate the database. We use a three-minute warmup period, followed by a 10-minute performance or VTune profiling period. We disable hyper-threading (HT) and turbo-boost (TB), as they jeopardize VTune counter values [45]. We examine HT separately, in Section 6.5.3 for the open-source system we built. We examine hyper-thread sharing only for the academic prototype, as we do not have the control on which threads run on which cores on the real-life DBMSs.

We numa-localize every experiment by using Linux's numactl command. We do single- and multi-threaded experiments. We choose a scaling factor of 350 (the database of 30GB) for all the experiments. We generate statistics before profiling each database. We disable compression for *DBMS A*, but not for *DBMS B*. *DBMS B* does not allow disabling compression and uses an internally-decided compression scheme whose details are not revealed.

6.3 *DBMS A*

This section presents the evaluation of *DBMS A*.

6.3.1 Micro-benchmark

This section presents the micro-benchmark evaluation.

		1T+13A	7T+7A	13T+1A
Projection	OLTP	0.99	0.98	0.97
	OLAP	0.98	0.99	1.00
Join	OLTP	0.96	0.99	0.97
	OLAP	0.94	0.92	1.00

Table 6.2 – Hardware-level interference results. Normalized OLTP and OLAP throughputs for *DBMS A* when running the micro-benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone. Example: First, OLTP throughput with 1 OLTP thread is measured when concurrently running with 13 OLAP threads. Second, OLTP throughput with 1 OLTP thread is measured when running alone on the server. The former is divided by the latter, and the result is reported as hardware-level interference at the OLTP side, for projection query for 1T+13A configuration (shown by cell in the first row and column).

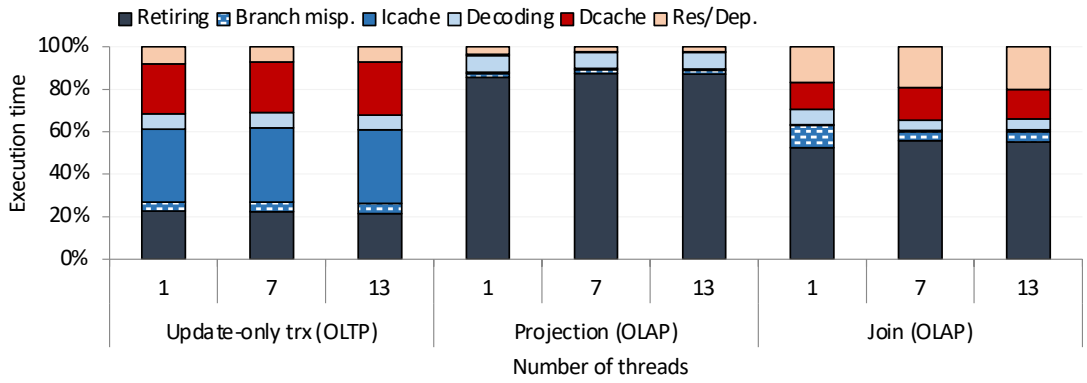


Figure 6.1 – Micro-architectural behavior for OLTP- and OLAP-alone executions for varying number of threads for the micro-benchmark.

Hardware-level Interference

This section presents the hardware-level-interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use different tables.

Table 6.2 presents the normalized throughputs, where the numerator is the throughput when running OLTP and OLAP concurrently, and the denominator is the throughput when running the OLTP and OLAP alone. The table shows that the OLTP and OLAP throughputs are only minimally affected by each other. Hence, the OLTP and OLAP components of *DBMS A* do not interfere with each other in the shared hardware resources.

Figure 6.1 presents the micro-architectural behavior of the OLTP and OLAP components when they run alone on the server. The figure shows that the OLTP component spends most of its execution time in Icache stalls, similarly to *DBMS D* and *M*, as we examined in Chapter 4. The OLAP component spends most of its execution time retiring instructions for both projection and join queries, similarly to *DBMS C*, as we examined in Chapter 5. Table 6.3 presents the memory bandwidth consumptions. The consumed bandwidth is very low for all the cases.

	Number of threads		
	1	7	13
Update trx (OLTP)	0.05	0.90	2.49
Projection (OLAP)	0	1.90	3.37
Join (OLAP)	0.09	2.80	3.30

Table 6.3 – Consumed memory bandwidth in GB/s for OLTP- and OLAP-alone executions for varying number of threads for the micro-benchmark.

		1T+13A	7T+7A	13T+1A
Projection	OLTP	0.63	0.79	0.91
	OLAP	0.62	0.52	0.33
Join	OLTP	0.92	0.96	1.00
	OLAP	0.90	0.80	0.40

Table 6.4 – Software-level interference results. Normalized OLTP and OLAP throughputs for *DBMS A* when running the micro-benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

Despite the fact that maximum available bandwidth is 66GB/s, the consumed bandwidth is always less than 4GB/s. Hence, both the OLTP and OLAP components of *DBMS A* only lightly stress the memory bandwidth.

Software-level Interference

This section presents software-level interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use the same table.

Table 6.4 presents the normalized throughputs. The OLAP throughput drops by 38% to 67% for projection. This is because the OLAP component requires accessing the fresh tuples that usually reside at the OLTP side. Hence, it makes the necessary extra work and suffers from the reduced throughput. The throughput is more and more decreased as the number of OLTP threads is increased. As the number of OLTP threads is increased, the number of fresh tuple that the OLAP side requires accessing is increased. Hence, the OLAP side requires performing more and more extra work and is interfered more and more.

The OLAP throughput is decreased less for the join query than it is for the projection query. This is because the join query is more expensive, and hence its performance is less affected by the extra work that is needed to access the fresh tuples. Nevertheless, for 13 OLTP threads, its throughput is decreased by 40%, which is close to the projection micro-benchmark.

The OLTP throughput significantly drops (by 37%) when running with the projection query, for 1 OLTP and 13 OLAP threads. The throughput is decreased less and less as the number of OLTP threads is increased. The throughput is decreased by 21% for 7 OLTP threads and 9% for 13 OLTP threads. As the previous section has shown, the OLTP and OLAP components do not

interfere with each other at the hardware-level. Hence, this interference is due to the OLTP components being affected by the OLAP component. We explain the reason as follows.

DBMS A uses the two-copy HTAP architecture (see Chapter 2, Section 2.4), where the system maintains an intermediate data structure, usually called *delta*, to keep track of the fresh tuples that are at the OLTP side, but not yet at the OLAP side. Periodically, DBMS needs to propagate the fresh tuples from the OLTP side to the OLAP side to prevent delta from growing too much. One way to propagate the fresh tuples is doing it right before the query execution. When a query is submitted, the OLAP side first propagates the fresh tuples to the OLAP data, and then performs the query execution on the OLAP data that just has had the fresh tuples propagated. BatchDB follows this execution model [73].

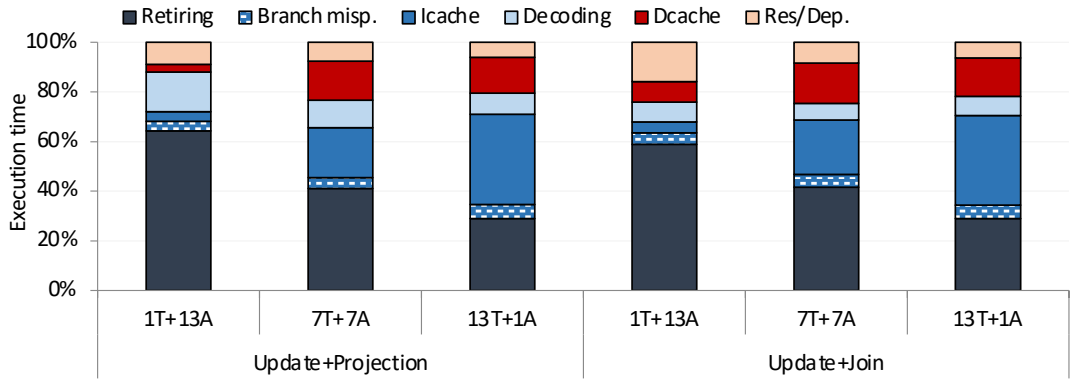


Figure 6.2 – Micro-architectural behavior for concurrently running OLTP and OLAP on the same NUMA node when sharing data for varying number of threads for the micro-benchmark.1T+13A stands for having 1 OLTP and 13 OLAP threads.

An important design choice for fresh-tuple propagation is whether or not to block the OLTP side while propagating the fresh tuples. Blocking the OLTP side can be necessary if the OLTP side is faster than the fresh-tuple propagation. During the fresh-tuple propagation itself, there can be even more fresh tuples generated by the OLTP side. Hence, the number of fresh tuples would be exponentially increased. To prevent this from happening, DBMS may choose the block the OLTP side during the fresh-tuple propagation.

The reason for the OLTP throughput to drop can be that *DBMS A* performs fresh-tuple propagation right before the query execution and also blocks the OLTP side during the propagation. As the number of OLAP threads is decreased, the query execution time is increased. Hence, the frequency of fresh-tuple propagation is decreased. As the frequency of the fresh-tuple propagation is decreased, the amount of time that the OLTP side is blocked is also decreased. As a result, the OLTP side is less and less interfered by the OLAP side, both for projection and join queries.

Furthermore, for the join query, the OLTP side is significantly less interfered than it is for the projection query. This is also consistent with our conclusion. The join query is significantly

		1T+13A	7T+7A	13T+1A
Q6	OLTP	0.99	0.99	0.96
	OLAP	1.04	1.01	1.00
Q3	OLTP	1.05	0.88	0.96
	OLAP	1.01	0.94	0.96

Table 6.5 – Hardware-level interference results. Normalized OLTP and OLAP throughputs for *DBMS A* when running the CH benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

more expensive than the projection query. Hence, the frequency of fresh-tuple propagation is significantly less than it is for the projection query. As a result, the OLTP side is less interfered by the join query compared to the projection query.

We also examine the micro-architectural behavior of *DBMS A* when running the OLTP and OLAP components on the same data. Figure 6.2 shows the results. We observe that the micro-architectural behavior is approximately the average of the micro-architectural behavior when running the OLTP and OLAP components alone (see Figure 6.1). This shows that the micro-architectural behavior of *DBMS A* does not significantly change when the OLTP and OLAP components concurrently run on the same data.

6.3.2 CH benchmark

This section presents the CH benchmark evaluation.

Hardware-level Interference

This section presents the hardware-level-interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use different tables.

Table 6.5 presents the normalized throughputs. As the table shows, the OLTP and OLAP throughputs do not drop significantly. Hence, the OLTP and OLAP components of *DBMS A* do not interfere with each other when running the CH benchmark similar to when running the micro-benchmark.

Figure 6.3 presents the micro-architectural behavior of the individual OLTP and OLAP components when running alone. The results are similar to the micro-benchmark results. The OLTP component spends most of its execution time in Icache stalls, whereas the OLAP component spends most of its execution time retiring instructions. Table 6.6 presents the memory-bandwidth consumptions. Once again, the results are similar to the micro-benchmark results. The consumed bandwidth is very low for all the cases. Despite the fact that maximum available bandwidth is 66GB/s, the consumed bandwidth is less than 4GB/s. Hence, the OLTP and OLAP components of *DBMS A* only lightly stress the memory bandwidth.

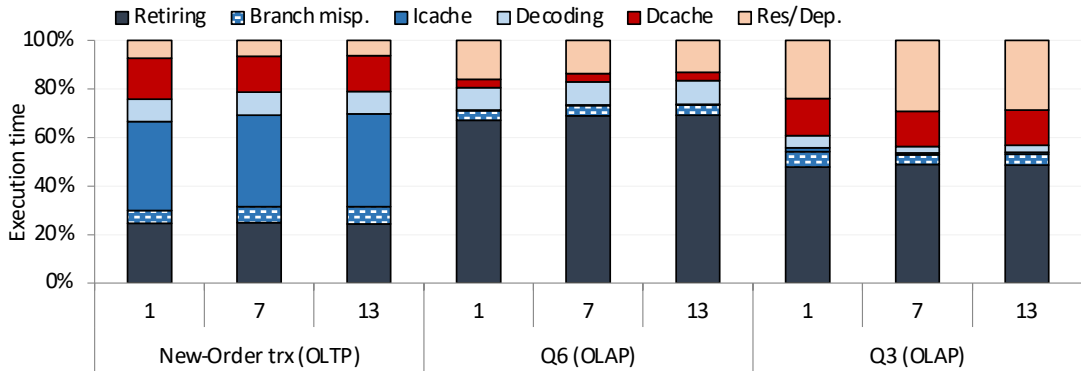


Figure 6.3 – Micro-architectural behavior for OLTP- and OLAP-alone executions for varying number of threads for the CH benchmark.

	Number of threads		
	1	7	13
New-Order trx.(OLTP)	0.08	0.59	1.04
Q6 (OLAP)	0	1.84	1.89
Q3 (OLAP)	0	2.88	3.21

Table 6.6 – Consumed memory bandwidth in GB/s for OLTP- and OLAP-alone executions for varying number of threads for the CH benchmark.

Software-level Interference

This section presents software-level interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use the same table. As the CH benchmark is insert-heavy, the OLAP query execution time is naturally increased in the face of insert-heavy OLTP transactions. Hence, we normalize the OLAP query execution time based on the number of processed tuples.

Table 6.7 presents the normalized throughputs. Neither the OLAP nor the OLTP throughput does not drop significantly. While we are not aware of the internal implementation of the insert operation for *DBMS A*, the reason is of not having software-level interference for the CH benchmark can be that it is an insert-heavy benchm ark. The new-order transaction has 12 inserts with 2 updates and 3 selects. In a two-copy HTAP architecture, insert operation can be implemented in a much more efficient way than update operation. While updates require tracking down the existing tuples usually by means of an index, inserts can simply be appended to the existing data. Hence, propagating inserts can be much simpler. We verified our assumption by doing in insert-only micro-benchmark. We have observed less than 5% interference at both OLTP and OLAP sides when running an insert-only transaction with the projection and join micro-benchmark queries used in Section 6.3.1.

We also examine the micro-architectural behavior of *DBMS A* when running the OLTP and OLAP components on the same data in Figure 6.4. The micro-architectural behavior is approximately the average of that of when running the OLTP and OLAP components alone.

		1T+13A	7T+7A	13T+1A
Q6	OLTP	0.98	0.97	0.96
	OLAP	0.96	0.94	0.93
Q3	OLTP	0.97	0.96	0.95
	OLAP	0.98	0.89	0.79

Table 6.7 – Software-level interference results. Normalized OLTP and OLAP throughputs for *DBMS A* when running the CH benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

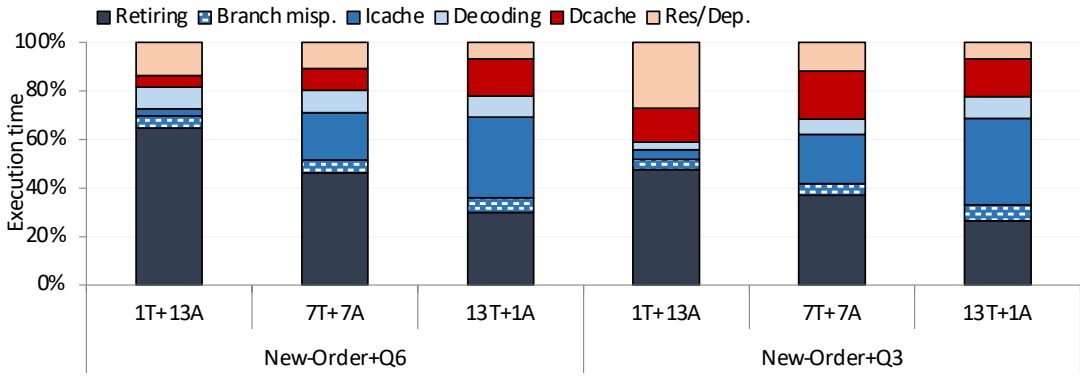


Figure 6.4 – Micro-architectural behavior for concurrently running OLTP and OLAP on the same NUMA node when sharing data for varying number of threads for the CH benchmark.1T+13A stands for having 1 OLTP and 13 OLAP threads.

6.4 *DBMS B*

This section presents the evaluation of *DBMS B*.

6.4.1 Micro-benchmark

This section presents the micro-benchmark evaluation.

Hardware-level Interference

This section presents the hardware-level-interference analysis. We run the OLTP and OLAP workloads on the same NUMA node, but on different tables. Table 6.8 presents the normalized throughputs. The OLTP and OLAP throughputs drop only by 1% to 11%. Hence, OLTP and OLAP components of *DBMS B* do not interfere with each other, similar to *DBMS A*.

Figure 6.5 presents the micro-architectural behavior of the OLTP and OLAP components when they run alone on the server. The figure shows that the OLTP component spends most of its execution time in Icache stalls, whereas the OLAP component spends most of its execution time retiring instructions, similarly to the way *DBMS A* does. Table 6.9 presents memory-

		1T+13A	7T+7A	13T+1A
Projection	OLTP	0.97	0.91	0.97
	OLAP	0.98	0.97	1.00
Join	OLTP	0.94	0.89	0.98
	OLAP	1.02	0.98	0.95

Table 6.8 – Hardware-level interference results. Normalized OLTP and OLAP throughputs for *DBMS B* when running the micro-benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

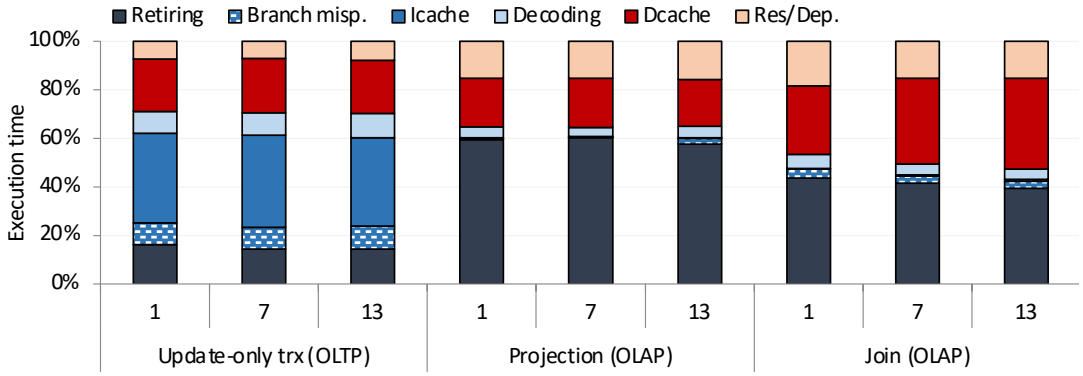


Figure 6.5 – Micro-architectural behavior for OLTP- and OLAP-alone executions for varying number of threads for the micro-benchmark.

bandwidth consumptions. The consumed bandwidth is low for all the cases. Despite the fact that the maximum available bandwidth is 66GB/s, the consumed bandwidth is always less than 8GB/s. Hence, the OLTP and OLAP components of *DBMS B* only lightly stress the memory bandwidth.

Software-level Interference

This section presents software-level interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use the same table.

Table 6.10 presents the normalized throughputs. The OLAP throughput drops by up to 53% for projection, as the OLAP component requires scanning the delta data structure to access the fresh tuples. As the delta is a pointer-intensive index structures, it is costlier than scanning the column-store, where the bulk of the data is kept. Hence, it costs up to 53% for the projection query to scan the fresh tuples. The throughput is more and more decreased as the number of OLTP threads is increased. As the number of OLTP threads is increased, the number of fresh tuple that the OLAP side requires accessing is increased. Hence, the OLAP side requires performing more and more extra work and is interfered more and more.

The OLAP throughput is decreased significantly less for the join query than it is for the projec-

	Number of threads		
	1	7	13
Update trx (OLTP)	0.02	0.23	0.47
Projection (OLAP)	0.04	5	7.18
Join(OLAP)	0.07	2.40	4.36

Table 6.9 – Consumed memory bandwidth in GB/s for OLTP- and OLAP-alone executions for varying number of threads for the micro-benchmark.

		1T+13A	7T+7A	13T+1A
Projection	OLTP	0.98	0.93	0.91
	OLAP	0.99	0.59	0.47
Join	OLTP	0.97	0.94	0.90
	OLAP	0.98	0.94	0.84

Table 6.10 – Software-level interference results. Normalized OLTP and OLAP throughputs for *DBMS B* when running the micro-benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

tion query. This is because the join query is more expensive, and hence its performance is less affected by the delta-scan that is needed to access the fresh tuples.

Table 6.10 shows that the OLTP throughput drops by only 5-10%. This shows that *DBMS B* does not block the OLTP at every query execution, unlike *DBMS A*. This can be due to an efficient implementation of fresh tuple propagation, which does not require blocking the OLTP side as it can propagate the fresh tuples faster than the OLTP side generates.

Both projection and join throughputs are decreased less for *DBMS B* compared to *DBMS A* (see Table 6.10). Furthermore, the OLTP throughput drops by only 5-10%, unlike *DBMS A*. These shows that the new-generation *DBMS B* is more optimized for a more efficient real-time analytical processing than the traditional *DBMS A*. Hence, the level of interference at both OLAP and OLTP sides is significantly less for *DBMS B*, compared to *DBMS A*.

We also examine the micro-architectural behavior of *DBMS B* when running the OLTP and OLAP components on the same data. Figure 6.6 shows the results. We observe that the micro-architectural behavior is approximately the average of the micro-architectural behavior when running the OLTP or OLAP components alone (see Figure 6.5), except when running 7 OLTP and 7 OLAP threads for the projection micro-benchmark. For this case, we observe a larger amount of Dcache stalls than the average of the Dcache stalls of 7 OLTP or OLAP individual executions. Furthermore, Table 6.10 shows that the OLAP throughput drops by 41% when running 7 OLTP and 7 OLAP threads due to the software-level interference. Hence, the increased Dcache stalls shows the affect of the pointer-intensive delta scan operation, which requires large number of random data-accesses.

Note that the OLAP throughput is significantly interfered when running 13 OLTP and 1 OLAP

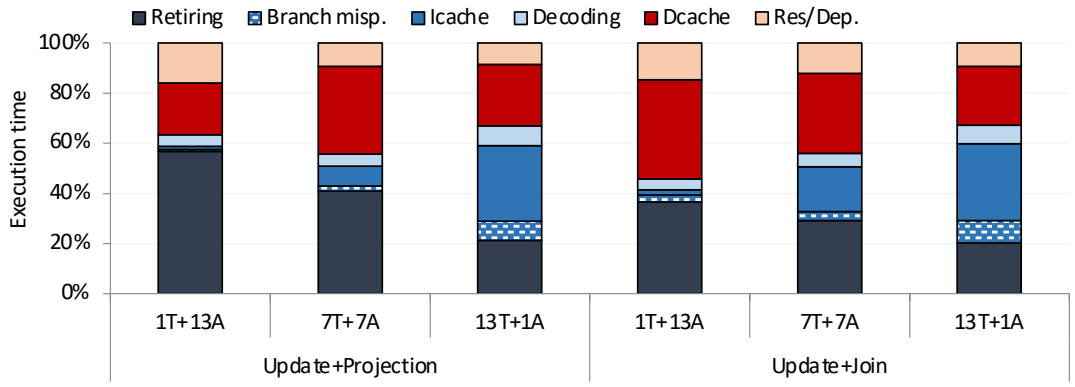


Figure 6.6 – Micro-architectural behavior for concurrently running OLTP and OLAP on the same NUMA node when sharing data for varying number of threads for the micro-benchmarks.1T+13A stands for having 1 OLTP and 13 OLAP threads.

		1T+13A	7T+7A	13T+1A
Q6	OLTP	1.01	0.88	0.97
	OLAP	1.04	0.98	1.01
Q3	OLTP	0.99	0.91	0.98
	OLAP	1.02	0.97	1.01

Table 6.11 – Hardware-level interference results. Normalized OLTP and OLAP throughputs for *DBMS B* when running the CH benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

threads. Hence, one would expect that the interference is reflected in the micro-architectural behavior. However, when 13 cores running the OLTP transactions and 1 core running the OLAP query, the overall micro-architectural behavior is highly dominated by the OLTP behavior. Hence, the individual OLAP behavior is not visible in the overall micro-architectural behavior. As *DBMS B* is a closed-source system, we are not able to track down the individual threads and present the micro-architectural behavior for individual threads. We examine this in our open-source system analysis, where we have the control over threads.

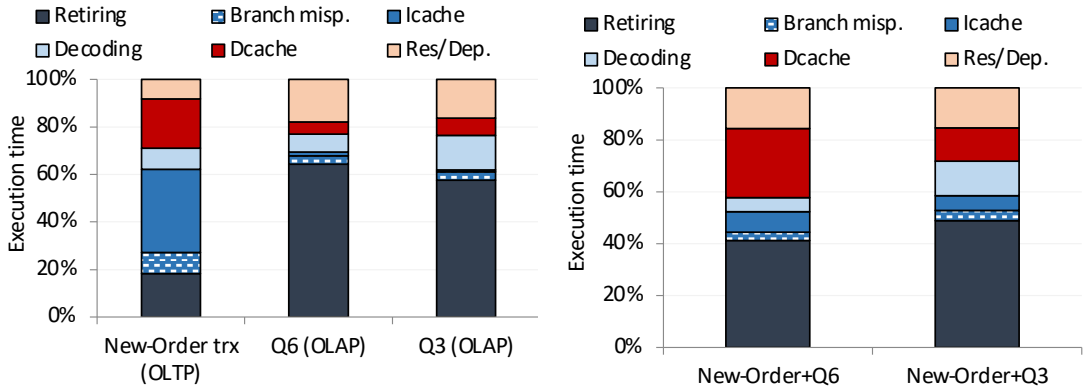
6.4.2 CH benchmark

This section presents the CH benchmark evaluation.

Hardware-level Interference

This section presents the hardware-level-interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use different tables.

Table 6.11 presents the normalized throughputs. Similar to the micro-benchmark results, the OLTP and OLAP throughputs do not drop significantly. Hence, the OLTP and OLAP



(a) OLTP/OLAP-alone.

(b) OLTP+OLAP.

Figure 6.7 – Left: Micro-architectural behavior for OLTP- and OLAP-alone executions for varying number of threads for the CH benchmark. Right: Micro-architectural behavior for concurrently running OLTP and OLAP on the same NUMA node when sharing data for varying number of threads for the CH benchmark. Only for 7 threads for OLTP and OLAP.

	Number of threads		
	1	7	13
New-Order trx. (OLTP)	0.01	0.10	0.50
Q6 (OLAP)	0.01	1.59	1.95
Q3 (OLAP)	0.06	0.27	1.08

Table 6.12 – Consumed memory bandwidth in GB/s for OLTP- and OLAP-alone executions for varying number of threads for the CH benchmark.

components do not interfere with each other when running the CH benchmark.

Figure 6.7 (left) presents the micro-architectural behavior of the OLTP and OLAP components when they run alone on the server. We present the results for seven OLTP and OLAP threads. Similar to the micro-benchmark results, the OLTP component spends most of its execution time in Icache stalls, whereas the OLAP component spends most of its execution time retiring instructions. Table 6.12 presents memory bandwidth consumptions. Similar to the micro-benchmark results, the consumed bandwidth is low for all the cases. Despite the fact that maximum available bandwidth is 66GB/s, the consumed bandwidth is always less than 2GB/s. Hence, the OLTP and OLAP components of *DBMS B* do not significantly stress the memory bandwidth when running the CH benchmark.

Software-level Interference

This section presents software-level interference analysis. We run the OLTP and OLAP workloads on the same NUMA node when the OLTP and OLAP workloads use the same table.

Table 6.13 presents the normalized throughputs. The OLAP throughput drops 12% to 70% for

		1T+13A	7T+7A	13T+1A
Q6	OLTP	0.99	0.95	0.97
	OLAP	0.88	0.47	0.30
Q3	OLTP	0.99	0.98	0.99
	OLAP	0.97	0.91	0.86

Table 6.13 – Software-level interference results. Normalized OLTP and OLAP throughputs for *DBMS B* when running the CH benchmark. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

the projection micro-benchmark. This is due to the costly delta scan operation that the OLAP side requires doing to provide real-time query processing over the fresh data. The cost of delta scan is higher for the projection-like Q6 than it is for the projection micro-benchmark (see Table 6.10). This is because Q6 is a less expensive query than the projection micro-benchmark query. Q6 is ~2x faster than projection. Hence, Q6 is affected more than the projection query. On the other hand, similar to the micro-benchmark results, the join query is not interfered significantly, and the OLTP throughput does not drop significantly.

We also examine the micro-architectural behavior of *DBMS B* when running the OLTP and OLAP components on the same data. Figure 6.7 (right) shows the results for 7 OLTP and 7 OLAP threads. We observe that the micro-architectural behavior is approximately the average of the micro-architectural behavior when running the OLTP or OLAP components alone for Q3 (see Figure 6.7 (left)). For Q6, however, there is a larger amount of Dcache stalls than the average of the Dcache stalls of 7 OLTP or OLAP individual executions. This shows the affect of delta scan operation that requires making random data-accesses to scan the pointer-intensive delta data structure.

6.5 Academic Prototype

This section examines the analysis of the academic prototype that we built to analyze the software- and hardware-level interference for HTAP workloads. We first describe the HTAP architecture we use. Then, we present the software- and hardware-level interference analyses.

6.5.1 HTAP Architecture

We use two-copy mixed-format architecture, which is a popular architecture also used by Oracle and SQL Server [59, 62]. We combine the open-source in-memory OLTP engine, Silo [114], with the open-source column-store, Typer [56] into a single HTAP system. The HTAP system launches Silo and Typer over the same data loaded to both. Then, the system keeps track of the data that Silo modifies by using a hash table, called *delta*. If the number of tuples that the delta keeps track of is more than a certain threshold, the HTAP system propagates the fresh data from the OLTP side to the OLAP side.

		Number of OLTP threads				
		1	7	14	21	28
Number of OLAP threads	1	1.0	1.2	4.2	11.2	19.6
	10	1.0	1.1	4.0	8.9	13.9
	26	1.0	1.1	2.4	4.9	10.8

Table 6.14 – Normalized query execution time for the projection micro-benchmark with varying number of OLTP (rows) and OLAP (columns) sides. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

Delta propagation is a background procedure. During delta propagation neither OLTP nor OLAP components stop executing their workloads. To achieve this, we keep two copies for each OLAP column that is modified by OLTP: (i) one for OLAP to perform the query execution, and (ii) one for the delta propagation thread(s) to propagate the delta. While the query is being executed, delta propagation thread performs the delta propagation, and waits for the next iteration of the query execution to perform the next delta propagation.

The delta is a hash table that keeps track of the pointers to the OLTP tuples that are modified, but not yet propagated to the OLAP side. Silo provides consistent snapshots of the data at 1ms granularity. For each tuple, there is a chain of versions. We use Silo's internal snapshotting mechanism to provide the correct version of the tuples for each submitted query. The OLAP queries access the version chain of the tuples based on the pointer that the delta keeps and perform version traversal to find the correct version.

6.5.2 Software-level Interference

As we have the control over the core and data affinity on the academic prototype, we first examine the software-level interference, where we place OLTP and OLAP sides to the two separate sockets of the machine we use. We place the delta and delta propagation threads on the OLAP-socket. We first present the empirical characterization of the interference. Then, we provide a theoretical characterization of the query execution time in the face of continuous fresh data ingestion. We examine only the OLAP side performance as the OLTP side is not affected by the software-level interference. As shown in the previous section (Section 6.5.1), the HTAP architecture we use do not block the OLTP side, similar to the real-life *DBMS B* that we examined in Section 6.4.

Empirical characterization: We first empirically analyze the average query execution throughput under continuous execution of the OLTP side, with varying number of OLTP and OLAP threads. We fix the number of delta propagation threads to 1. We pin OLTP, OLAP and delta propagation threads to particular individual cores, without sharing any hyper-threading resources among any of the three. We use the projection micro-benchmark. Table 6.14 presents the normalized query execution time for the projection micro-benchmark. The query execution time is normalized to OLAP-alone query execution throughput, i.e., the query execution

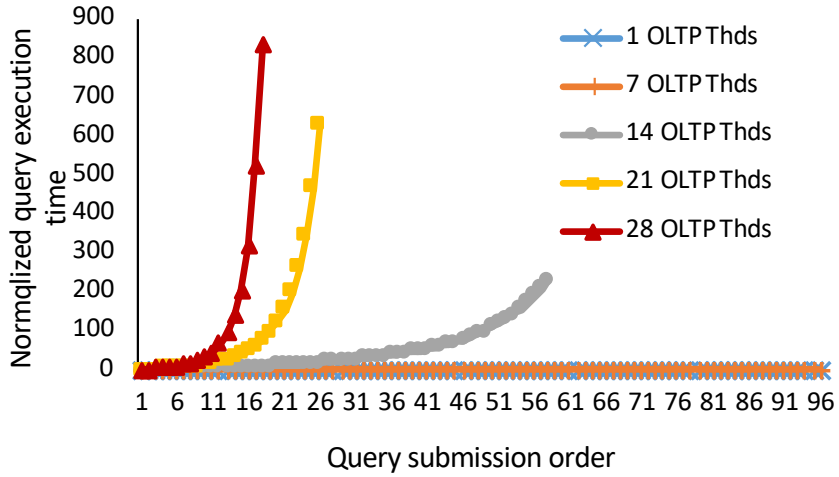


Figure 6.8 – Normalized query execution times for individual number of OLTP threads.

time without having OLTP running with the OLAP.

The table shows that query execution time is increased by 1.1 to 19.6x. For low number of OLTP threads, the fresh data ingestion rate is relatively low. Hence, the cost of processing the fresh tuples by the OLAP threads, and the cost of propagating the fresh data by the delta propagation thread is low compared the the OLAP-only query execution time. Hence, the interference at the OLAP side is low. When the number of threads is 14 or more, the query execution time is heavily increased, by 2.4x to 19.6x for every number of OLAP threads examined.

Next, we examine the query execution time one-by-one for 10 OLAP threads and varying number of OLTP threads. We submit the queries one-after-the-other, and measure the execution time of every individual query. Our goal is to examine whether there is any particular pattern in the query execution times that leads to the high average query execution time in the face of fresh data ingestion.

Figure 6.8 presents the results. We plot the normalized query execution times. Each line is for a particular number of OLTP threads, with a normalized starting value of 1. The figure shows that for low number of OLTP threads, i.e., for 1 and 7 OLTP threads, the OLAP query execution time is relatively constant. Similarly, the average normalized query execution time is only 1.0 and 1.1 for 1 and 7 OLTP threads, as shown by Table 6.14. However, for 14 or more threads, the query execution time is increased exponentially. We theoretically examine the exponentially increasing behavior in the next section.

Theoretical characterization: In this section, we theoretically characterize the query execution time of the HTAP architecture we study. We define two variables: λ, μ, and μ'. λ is fresh data ingestion rate, μ is delta propagation throughput, and μ' is delta processing throughput. All variables are in tuples per second metric. Fresh data ingestion rate is the OLTP throughput in our micro-benchmark. Delta propagation throughput is how many fresh tuples per second the delta propagation thread(s) can propagate. Delta processing throughput is how many

fresh tuples per second the OLAP thread(s) can process. We further define a query execution time that exponentially increases as an unstable query execution time, and a query execution time that converges to a constant value as a stable query execution time.

Delta propagation and processing scans the delta and accesses the set of newly generated, fresh OLTP tuples. The delta propagation task is carried out by the delta propagation thread(s). It reads the fresh tuples from the OLTP side and writes them to the OLAP side. Delta processing task is carried out by the OLAP threads. Each OLAP thread scans a portion of the delta, accesses the fresh tuples and processes the fresh tuples as the query requires, e.g., aggregates them on an accumulator, or starts building a hash table out of them. The OLAP threads, then, continue with regular OLAP processing over the OLAP columns.

Query execution time: As explained in Section 6.5.1, delta propagation is a background task, performed by the delta propagation thread(s) at the background. Hence, its time is ideally not on the critical path. Hence, we first characterize the query execution time. Query execution time is composed of delta processing time and the OLAP-columns-processing time. The OLAP-columns-processing time is constant (assuming that the workload does not have any inserts). At time $t = 0$, the query execution starts without any fresh data. Let us denote this time as T_{Q_0}. In the next iteration, at time $t = 1$, the query execution requires scanning the delta, accessing the fresh OLTP tuples, and processing them as the query requires, in addition to processing the OLAP columns as done at time $t = 0$. Hence, the query execution time will be:

$$T_{Q_1} = \frac{1}{\mu'}|\Delta_0| + T_{Q_0} \tag{6.1}$$

where $|\Delta_0|$ is the size of the delta right before the query execution at time $t = 1$. In other words, it is the number of fresh tuples generated during the query execution at time $t = 0$. Observe that $|\Delta_0|$ is λT_{Q_0}, i.e., multiplication of fresh data ingestion rate and the query execution time at the previous time slot. If we substitute $|\Delta_0|$ in Equation 6.1, we obtain:

$$\begin{aligned} T_{Q_1} &= \frac{1}{\mu'}\lambda T_{Q_0} + T_{Q_0} \\ &= T_{Q_0}(1 + \frac{\lambda}{\mu'}) \end{aligned} \tag{6.2}$$

If we follow the same logic for T_{Q_2}, T_{Q_3}, ..., T_{Q_n}, we obtain the following formula for T_{Q_n}:

$$\begin{aligned} T_{Q_n} &= T_{Q_0}(1 + (\frac{\lambda}{\mu'})^1 + (\frac{\lambda}{\mu'})^2 + ... + (\frac{\lambda}{\mu'})^n) \\ &= T_{Q_0}\Sigma_{i=0}^{n}(\frac{\lambda}{\mu'})^i \end{aligned} \tag{6.3}$$

Equation 6.3 is a geometric series sum with the recurrence term of $\frac{\lambda}{\mu'}$. Hence, it is equal to:

$$T_{Q_n} = \frac{1 - (\frac{\lambda}{\mu'})^{n+1}}{1 - \frac{\lambda}{\mu'}} \tag{6.4}$$

The term in Equation 6.4 is diverging, i.e., unstable, i.e., exponentially increased if $\frac{\lambda}{\mu'} > 1$, whereas it converges to:

$$\frac{1 - (\frac{\lambda}{\mu'})^{n}}{1 - \frac{\lambda}{\mu'}} \tag{6.5}$$

for $\frac{\lambda}{\mu'} < 1$. Hence, for a converging, i.e., stable query execution time, $\frac{\lambda}{\mu'} < 1$ should hold. In other words, an HTAP system must make sure that the delta processing throughput is higher than the fresh data ingestion rate. Otherwise, the amount of ingested fresh data is beyond the processing capacity and the query execution time is exponentially increasing, i.e., unstable. Intuitively, this means that the HTAP system should be able to process the fresh tuples faster than the fresh tuples are generated. As delta processing is carried out by the OLAP threads, the HTAP systems should allocate enough number of cores and threads to the OLAP side to make sure that the delta processing throughput is higher than the fresh data ingestion rate.

Delta propagation time: While delta propagation is ideally a background task, it can take longer time than query execution (e.g., due to having more cores allocated to OLAP than to delta propagation). In this case, the system would be bottlenecked by the delta propagation time.

We characterize the delta propagation time. Let us call B as the delta propagation time at the first iteration that the delta propagation time exceeds the query execution time. Furthermore, delta propagation performs two constant-time-taking activities: delta clearing and delete map clearing. Let us call the amount of time needed for the constant-time-taking activities as C. Let us call the delta propagation time at time t as T_{DP_t}. Then, we can construct T_{DP_t} for

	Number of OLTP threads				
	1	7	14	21	28
λ/μ	0.1	0.62	0.96	1.26	1.7
λ/μ'	0.005	0.02	0.04	0.05	0.06

Table 6.15 – Justification of the theoretical model for query execution time for HTAP systems for 10 OLAP and 1 delta propagation threads.

$t = 0, 1, ..., n$ as follows:

$$\begin{aligned} T_{DP_0} &= B \\ T_{DP_1} &= \frac{\lambda}{\mu}B + C \\ T_{DP_2} &= (\frac{\lambda}{\mu})^2 B + \frac{\lambda}{\mu}C + C \\ &... \\ T_{DP_n} &= (\frac{\lambda}{\mu})^n B + C\Sigma_{i=0}^{n-1}(\frac{\lambda}{\mu})^i \end{aligned} \tag{6.6}$$

As $\Sigma_{i=0}^{n-1}(\frac{\lambda}{\mu})^i$ term is a geometric series sum with the recurrence term of $\frac{\lambda}{\mu}$, the delta propagation time is:

$$T_{DP_n} = (\frac{\lambda}{\mu})^n B + C\frac{1-(\frac{\lambda}{\mu})^n}{1-\frac{\lambda}{\mu}} \tag{6.7}$$

Hence, the delta propagation time is exponentially increased, i.e., diverging if $\frac{\lambda}{\mu} > 1$. For a stable, converging delta propagation time, the delta propagation throughput should be higher than the fresh data ingestion rate. Otherwise, the amount of ingested fresh data *during* delta propagation itself is larger than the amount of fresh data that is being propagated. Hence, the number of fresh tuples is exponentially increased. As a result, the query execution time, which is bottlenecked by delta propagation time, is also exponentially increased. This means that HTAP systems should make sure the delta propagation task is allocated enough cores and threads to perform delta propagation faster than fresh data ingestion rate.

Theoretical model justification: We justify the validity of our theoretical query execution time modelling described in the previous section. To do so, we will examine the λ/μ and λ/μ' terms for the experiments shown in Figure 6.8. Furthermore, we increase the number of delta propagation threads from one to two, and re-examine the query execution time for 14 and 28 OLTP threads to see whether the exponential, diverging behavior converts into a constant, converging behavior.

Table 6.15 presents λ/μ and λ/μ' for 10 OLAP and 1 delta propagation threads. As expected,

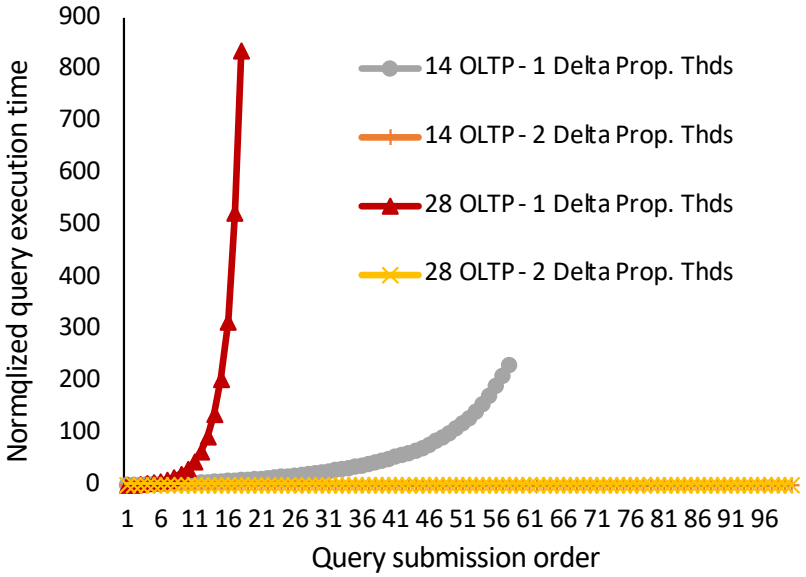


Figure 6.9 – Query execution time for increased number of delta propagation threads.

λ/μ value is close to 1 (0.96) for 14 OLTP threads, where the query execution time behavior is exponential. For 21 and 28 OLTP threads, the λ/μ values are well above 1, hence the execution times are more and more exponentially increased. λ/μ' is much lower than λ/μ. This is because the number of OLAP threads is 10, whereas the number of delta propagation threads is 1. Hence μ', i.e., delta processing throughput, is much higher than μ, i.e., delta propagation throughput. As a result, λ/μ' is much lower than λ/μ.

We also examine the query execution behavior for increased number of delta propagation threads. Figure 6.9 presents the query execution behavior for 10 OLAP threads, and 14, 28 OLTP threads with 1 and 2 delta propagation threads. When using 1 delta propagation thread, the query execution time behavior is exponential for 14 and 28 OLTP threads. Whereas, when using 2 delta propagation threads, the query execution time becomes converging, i.e., stable.

We further examine λ/μ and λ/μ' for increased number of delta propagation threads. Table 6.16 shows the values. As the table shows, λ/μ is decreased approximately by half when the number of delta propagation threads is increased from one to two. As a result, λ/μ drops below 1 and the query execution time behavior becomes converging, i.e., stable.

This justifies our theoretical modelling of the query execution and delta propagation time in the face of continuous fresh data generation for the HTAP architecture that we uses. Our query execution time behavior analysis shows that delta processing and propagation tasks should be allocated enough number of cores and threads such that their speeds are higher than the speed that the fresh tuples are generated. During delta processing and propagation, the OLTP component keeps producing new fresh tuples. At the end of each delta processing or delta propagation task, the size of the delta (i.e., the number of fresh tuples in delta) should be less than the size of the delta that was at the beginning of the delta processing or propagation task. Otherwise, the delta size would grow exponentially, and hence, the query execution time

	14 OLTP - 1 DP	14 OLTP - 2 DP	28 OLTP - 1 DP	28 OLTP - 2 DP
λ/μ	0.96	0.64	1.7	0.77
λ/μ'	0.04	0.04	0.06	0.06

Table 6.16 – λ/μ and λ/μ' for increased number of delta propagation threads.

would be increased exponentially.

While different HTAP architectures would have a different exact formulation of the query execution time, the main conclusion would apply to all HTAP architectures. For all HTAP architectures, the HTAP system should be able to process the fresh tuples faster than they are generated.

Our query execution time formulation in the face of continuous data generation is similar to response time formulation of queuing theory [63]. However, building the concrete relationship between the queuing theory and our response time formulation requires more work. In particular, it is not clear how the two parameters, inter-arrival time distribution and service time distribution, that queuing theory uses in its response time formulation will be used for the query execution time formulation. We leave building the concrete relationship between queuing theory and query execution time formulation as future work.

6.5.3 Hardware-level Interference

Next, we study hardware-level interference. We place the OLTP, OLAP and delta propagation threads and data on the same socket. In this case, these three separate tasks share: last-level cache (LLC), memory bandwidth and hyper-threads. We study LLC & and memory bandwidth, and hyper-threads separately.

LLC and memory bandwidth

We first study the hardware-level interference among the delta propagation and OLAP threads. Then, we examine the hardware-level interference among all the three components: OLTP, OLAP, and delta propagation.

Delta propagation and OLAP: We put the OLTP threads and data to a different socket than the delta propagation and OLAP threads. Then, for a fixed number of delta propagation threads, we gradually increase the number of OLAP threads and measure the normalized delta propagation throughput. Figure 6.17 presents the results for the projection and join micro-benchmarks. As the number of OLAP threads is increased, the normalized delta propagation throughput is decreased for the projection micro-benchmark. The decrease is negligible for 1 OLAP thread, whereas 28% and 33% for 7 and 13 OLAP threads. The reason is that the projection micro-benchmark stresses the memory bandwidth more and more as the number of OLAP threads is increased. As a result, it causes the delta propagation throughput to drop.

		1DP+13A	7DP+7A	13DP+1A
Projection	Delta prop.	0.67	0.71	1.0
	OLAP	0.98	0.99	0.98
Join	Delta prop.	0.89	0.96	1.0
	OLAP	0.99	0.99	0.98

Table 6.17 – Hardware-level interference for last-level cache and memory bandwidth sharing between the delta propagation and OLAP components when running the micro-benchmarks. Normalized delta propagation and OLAP throughputs. Numerator: Throughput measured when concurrently running delta propagation and OLAP. Denominator: Throughput measured when running delta propagation or OLAP alone.

		1T+12A	7T+6A	12T+1A
Projection	OLTP	0.58	0.73	0.80
	OLAP	0.98	0.99	0.95
Join	OLTP	0.78	0.82	0.89
	OLAP	0.99	0.98	0.98

Table 6.18 – Hardware-level interference for last-level cache and memory bandwidth sharing between the OLTP and OLAP components for the micro-benchmarks. The number of delta propagation threads is fixed to 1. Normalized OLTP and OLAP throughputs. Numerator: Throughput measured when concurrently running OLTP and OLAP. Denominator: Throughput measured when running OLTP or OLAP alone.

The delta propagation throughput drops significantly less when running with the join query. This is because the join query is random-data-access-intensive. Hence, it does not stress the memory bandwidth as high as the projection query. As a result, the delta propagation throughput does not drop as significantly. The OLAP query execution time remains the same for all the cases regardless the number of delta propagation threads running with it. Hence, the delta propagation threads do not cause interference for the OLAP threads.

OLAP and OLTP: We place all the three tasks on the same socket. For 1 delta propagation thread, we vary the number of OLTP and OLAP threads to examine the interference among them. Table 6.18 presents the results. OLTP throughput is significantly interfered by the projection query. The OLTP throughput drops by 42% when running 1 OLTP and 12 OLAP threads. As the number of OLAP threads is decreased, the OLTP throughput is interfered less and less: by 27% and 20% for 6 and 1 OLAP threads. The reason is that the sequential-scan-heavy projection query highly stresses the memory bandwidth. As a result, the OLTP threads are interfered at the last-level cache and memory bandwidth.

The OLTP throughput is decreased significantly less when running with the join query compared to when running the projection query. This is because the join query is random-data-access-heavy, and hence stresses the memory bandwidth less than the projection query. As a result, the OLTP throughput is affected less. The OLAP throughput is not affected by the OLTP component. This is because the OLTP component stresses the memory bandwidth significantly less than the OLAP components, both for the projection and join queries.

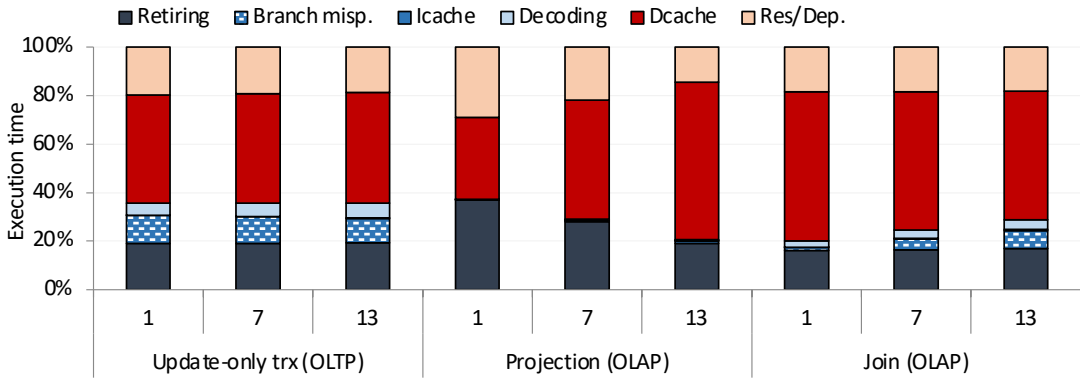


Figure 6.10 – Micro-architectural behavior for OLTP- and OLAP-alone executions for varying number of threads for the academic prototype.

	Number of threads		
	1	7	13
Update trx (OLTP)	0.10	4.10	8.30
Projection (OLAP)	5.40	31.65	58.55
Join(OLAP)	1.19	12.10	21.71

Table 6.19 – Consumed memory bandwidth in GB/s for OLTP- and OLAP-alone executions for varying number of threads for the academic prototype.

Figure 6.10 presents the micro-architectural behavior of the OLTP and OLAP components when they run alone on the server. The figure shows that both the OLTP and OLAP components spend most of their execution time in Dcache stalls. Table 6.19 presents memory-bandwidth consumptions. The OLTP component consumes only a small fraction of the memory bandwidth. Although the maximum memory bandwidth is 66GB/s, the OLTP component consumes a maximum of 8.3GB/s. Hence, the OLTP component only lightly stresses the memory bandwidth. The OLAP component consumes a significant fraction of the memory bandwidth. The projection query consumes about half of the bandwidth for seven threads, and approaches saturation of the bandwidth for 13 threads. The join query also consumes a significant fraction of the bandwidth for 13 OLAP threads. Hence, the OLAP component significantly stresses the memory bandwidth.

Hyper-threads

Next, we examine how much OLTP, OLAP and delta propagation threads interfere with each other when they share hyper-threads. We first examine OLAP and delta propagation, and then we examine OLTP and OLAP sharing hyper-threads.

Delta propagation and OLAP: We run 2 delta propagation on two separate physical cores. We, then, add 2 OLAP threads such that each delta propagation thread shares its core with an OLAP thread and examine how much the newly added OLAP threads decrease the throughputs of the already running delta propagation threads. We do the same by starting with 2 OLAP

		2DP+2A
Projection	Delta prop.	0.5
	OLAP	0.9
Join	Delta prop.	0.8
	OLAP	0.9

Table 6.20 – Hardware-level interference for hyper-thread sharing between the delta propagation and OLAP components when running the micro-benchmarks. Normalized delta propagation and OLAP throughputs. Numerator: Throughput measured when concurrently running delta propagation and OLAP. Denominator: Throughput measured when running delta propagation and OLAP alone.

threads, adding 2 delta propagation threads, and measuring how much OLAP throughput is decreased. Table 6.20 presents the results.

We first examine the projection micro-benchmark results. The OLAP threads cause 50% throughput drop for the delta propagation threads, whereas the delta propagation threads cause only 10% throughput drop for the OLAP threads. Hence, while the OLAP threads significantly affect the delta propagation threads, delta propagation threads modestly affect the OLAP threads. This is because the sequential-scan-heavy projection query uses the core resources more aggressively than the lightweight delta propagation thread.

We also examined how much OLAP and delta propagation threads would interfere with each itself. We run 2 delta propagation threads on two separate physical cores. We, then, add 2 more delta propagation threads such that each delta propagation thread shares its core with another delta propagation thread. We examine how much the newly added threads improve the delta propagation throughput. We do the same for OLAP threads. We observe that delta propagation throughput is increased by 70%, whereas OLAP throughput is increased by only 10%. This is because the lightweight delta propagation threads allow more room for hyper-thread sharing whereas the aggressive OLAP threads do not leave as much hyper-thread sharing room.

Table 6.20 shows that the join query modestly affects the delta propagation throughput, unlike the projection query. This is because the join query is random-data-access-intensive and hence lightly uses the core resources. Similarly, when we place OLAP threads executing the join queries on the same physical cores sharing hyper-threads, the OLAP throughput is increased by 60%. Similar to the delta propagation execution which is also random-data-access-intensive, OLAP threads leave a significant amount of room for sharing hyper-thread resources.

OLAP and OLTP: We run 2 OLTP threads on two separate physical cores. We, then, add 2 OLAP threads such that each OLTP thread shares its core with an OLAP thread. We examine how much the newly added OLAP threads decrease the throughputs of the already running OLTP threads. We do the same by starting with 2 OLAP threads, and then adding 2 OLTP threads, and measuring how much the OLAP throughput is decreased. Table 6.21 presents the results.

We first examine the projection micro-benchmark results. The OLAP threads cause 27% throughput drop for the OLTP threads, whereas the OLTP threads cause 45% drop for the

		2T+2A
Projection	OLTP	0.73
	OLAP	0.55
Join	OLTP	0.77
	OLAP	0.69

Table 6.21 – Hardware-level interference for hyper-thread sharing between the OLTP and OLAP components when running the micro-benchmarks. Normalized OLTP and OLAP throughputs. Numerator: Throughput when concurrently running OLTP and OLAP. Denominator: Throughput when running OLTP and OLAP alone.

OLAP threads. Hence, unlike the delta propagation threads, OLTP threads significantly affect the OLAP threads when sharing hyper-thread resources. This is because OLTP workload is more aggressive than the delta propagation workload. OLTP workload does successive index search operations which include several levels of B+tree search with linear in-node searches. Whereas, the delta propagation threads merely read a value and write the value to a random position in the OLAP columns, without doing any computation. As a result, OLTP threads block the OLAP computation and result in 45% throughput drop.

This shows that the sequential-scan-heavy OLAP workload is sensitive to hyper-thread sharing. The performance of the projection query highly depends on successfully streaming the read requests. If the stream of the read requests are intervened, e.g., due to a micro-architectural resource being occupied by the sibling hyper-thread, its performance significantly deteriorates.

The join micro-benchmark is affected moderately by the sibling OLTP threads. Similarly, the OLTP threads are affected moderately by the sibling projection/join OLAP threads. These show that the random-access-heavy OLTP and join query workloads are more robust to hyper-thread sharing. We also tested how much the OLTP threads would interfere with itself. The OLTP throughput is improved by 57%, showing a similar level of interference between OLTP and join micro-benchmark. Observe that 57% throughput improvement over a hyper-thread sharing means that each of the hyper-threads deliver ~80% of the single-physical-core throughput. Hence, the interference is about 20%.

6.6 Conclusions

In this chapter, we present the performance characterization of HTAP workloads. We examine two real-life HTAP systems and one academic prototype that we built based on existing open-source OLTP and OLAP systems.

The results show that the OLTP and OLAP components of the real-life systems do not interfere with each other in the shared hardware resources. Both the OLTP and OLAP components only lightly stress the memory bandwidth with a less than 15% memory-bandwidth consumption.

The OLTP and OLAP components of the academic prototype interfere with each other in the shared hardware resources. Although the OLAP throughput is not affected by the concurrently

running OLTP component, the OLTP throughput drops by 22 to 40% when the OLTP and OLAP components share hardware resources. The OLTP component lightly stresses the memory bandwidth, whereas the OLAP component significantly stresses the memory bandwidth with up to 90% memory-bandwidth consumption.

Software-level interference depends on how fast the OLTP component generates fresh tuples and how fast the OLAP component can process the fresh tuples. We have shown that OLAP query execution time is exponentially increased if the OLTP component generates fresh tuples faster than the OLAP component processes them. Hence, the OLAP component should be allocated enough resources to able to process the fresh tuples faster than the OLTP component generates them.

7 Lessons Learned, Conclusions, and Future Outlook

This chapter presents the learned lessons for each category of state-of-the-art database workloads. Then, it concludes with a future outlook.

7.1 Lessons Learned

This section presents the lessons learned for each category of state-of-the-art database workloads.

7.1.1 Online Transactional Processing

This section summarizes the highlights of our work on Online Transactional Processing (OLTP) workloads and discusses its implications.

Instruction stalls. *DBMS D* and *M* suffer from instruction-cache misses. Hence, OLTP systems should aim to mitigate instruction-cache misses. Transaction compilation has been shown to be a promising way to reduce the number of instruction-cache misses for OLTP systems as shown by [79] and [98].

Shore-MT's execution time has long shown to be dominated by the stalls due to instruction-cache misses [98, 99, 108, 111]. We show that the improvements in the instruction fetch-unit of Intel's successive micro-architecture generations eliminates the instruction-cache misses that Shore-MT suffers from, thus making Shore-MT's execution time data-stalls-bound. Therefore, further improvements in the micro-architecture can also make the *DBMS D* and *M* data-stalls-bound.

Data stalls. *DBMS N* and *Silo* mainly suffer from data-cache stalls. *DBMS D* and *M* suffer from data stalls only when the instruction-cache misses are mitigated by an increased instruction locality in their instruction stream. A closer look into the data-cache misses has revealed that the data-cache misses are due to the random data-accesses made during the index traversal.

Hence, ground-up designed in-memory systems should firstly optimize the efficiency of their index structures. We have seen that Masstree [75] used by *Silo* significantly outperforms the red-black tree used by *DBMS N*.

While using an efficient index structure improves the performance by making fewer random data-accesses, an efficient index structure's execution time is still dominated by data-cache misses caused by the random data-accesses during the index traversal. Hence, ground-up designed in-memory OLTP systems should adopt techniques that can mitigate the negative performance effect of random data-accesses.

One promising way to improve performance in the presence of random data-accesses is using co-routines. Co-routines is a cheap thread interleaving mechanism that allows interleaving long-latency data stalls with computation. Psaropoulos et al. [86, 87, 88] and Jonathan et al. [49] have shown that co-routines can successfully be used to improve index join and index traversal operations.

Another promising technique to improve the performance in the presence of random data-accesses is using machine learning to learn the distribution of the keys and directly jump to the index location that the key belongs to without actually performing the index traversal. Kraska et al. [58], Llaveshi et al. [69] and Ding et al. [26] have shown that machine-learned indexes can successfully replace/accelerate the index search operation.

Most OLTP indexes, e.g., the indexes used by TPC-B and TPC-C benchmarks, are over primary keys, whose distributions are highly uniform. Hence, simple machine learning techniques, such as linear regression, can successfully predict the location of a given key. For more complex distributions, such as skewed distribution, the prediction is harder. However, using features such as the natural logarithm of the key can reduce the skew in the distribution hence increase the accuracy of the prediction. Llaveshi et al. [69] have shown that the ensemble of linear regression models can predict the location of a given key with ~80% accuracy for uniform and skewed distributions, whereas the ~20% of mispredictions are only of distance one to the node that the key actually resides in.

Hardware. In this study, we conclude that software-level optimizations do not directly translate into more efficient utilization of micro-architectural resources on modern processors. One needs to optimize the hardware and software together as the next step putting micro-architectural utilization as a high priority goal.

OLTP workloads are unable to utilize the wide-issue aggressive out-of-order cores that implement complex hardware mechanisms. Most of the execution time go to memory stalls for bringing either instructions or data blocks from the memory hierarchy to the processor. instruction-cache sizes have been unchanged for the last decade due to the strict latency limitations and we cannot expect them to increase. On the other hand, we have seen that improved instruction fetch units can make a significant change in the micro-architectural behavior of the OLTP systems. Further advancements at the micro-architectural level, espe-

cially at the instruction fetch unit, can still have further potential impact to improve the OLTP system performance, as popular OLTP systems such as *DBMS D* and *M* still highly suffer from Icache stalls.

As ground-up designed OLTP systems mainly suffer from long-latency data-cache misses, hardware designers can invest more on hardware mechanisms that can overlap long-latency data stalls. We have seen that hyper-threading improves performance up to 70% in a carefully designed and implemented in-memory OLTP system. Other hardware features such as turbo-boost and hardware prefetchers are modestly useful for OLTP workloads as OLTP workloads are memory-latency-bound. As the processor's power budget is limited, hardware designers can invest more power budget on the features that would overlap long-latency data stalls such as larger number of hyper-threads per physical core.

Whatever the size of the last-level cache (LLC) is, megabytes of LLC will not be enough to keep the working set of most standard OLTP benchmarks, which are in the orders of gigabytes. Hence, instead of using beefy and complex out-of-order cores consuming large amount of power, using simpler cores with intelligent hyper-threading mechanisms can improve the throughput of OLTP applications with a smaller power budget [28, 30, 72]. Sirin et al. [97] have shown that low-power ARM processor can provide 1.7 to 3 times lower throughput with 3 to 15 times less power consumption than a state-of-the-art Intel Xeon processor, achieving up to 9 times higher energy efficiency.

With that said, the adoption of low-power cores require further research. Kanev et al. [50] have shown that server workloads partially benefit from the wide-issue out-of-order execution. Hence, the use of wimpy cores with narrow-issue execution engines might produce a suboptimal performance and may not satisfy some application requirements. Similarly, Sirin et al. [97] have shown that ARM processors' quantified latency can be up to 11 times higher than Intel Xeon towards the tail of the latency distribution, which makes Intel Xeon more suitable for tail-latency-critical applications.

GPU/FPGA-based acceleration of database systems is another line of research that allows improving database performance by using alternative computing devices to beefy and power-hungry processors. [20, 92, 96] present techniques on using GPUs for accelerating analytical processing queries such as hash join. Kim et al. [57] have proposed a transaction processing engine architecture that exploits the wide-parallelism. Alonso et al. [5] present an open-source hardware-software co-design platform for database systems, that uses CPU and FPGA as the main building blocks. [51, 94] present techniques on integrating FPGAs into common database operations such as data partitioning and regular expression. These studies highlight the opportunities to enrich the traditional computing space of database systems by alternative computing devices such as FPGAs and GPUs. As these computing devices provide massive parallelism and/or low-power consumption, they allow investigating the energy-efficiency space, and potentially serve as the processors of the future database systems.

7.1.2 Online Analytical Processing

This section summarizes the highlights of our work on Online Analytical Processing (OLAP) workloads and discusses its implications.

OLAP systems that follow the tuple-at-a-time execution model efficiently use the CPU cycles; however, they require the execution of a significantly larger number of instructions hence are significantly slower than the systems that use vector-at-a-time and compiled execution models. Therefore, to deliver a high performance, OLAP systems should first adopt an efficient execution model.

Projection and selection times are sensitive to overhead, as they are usually rapid. Having an inefficient data access method inside the projection operator or using a simple bitvector operation inside the selection operator loop can significantly increase the execution time. Intermediate result materialization is a major source of overhead and should be avoided. Using selection vectors is a good alternative to using bitvectors and intermediate result materialization.

Filtering-based techniques, such as Filter Join, Lookahead Information Passing, block-skipping by meta-data processing and bloom filters on low hit-rate joins, are promising techniques for reducing the work being done [85]. They do not eliminate, however, the overhead that would come from the inefficiencies in the execution model. Combining filtering-based techniques with efficient execution models can both reduce the work being done and eliminate the overhead that would come from the inefficiencies in the execution model.

Vectorized engines are faster than compiled engines, only if their materialization cost pays off. The materialization costs pay off for hash join, as vectors of computed hashes enable the overlapping of costly random hash table accesses, but do not pay off for projection and selection. Compiled engines suffer from mixing random data-accesses with hash computation and/or condition checks, preventing them from overlapping the random data-accesses [56].

Scan-intensive queries saturate the memory bandwidth before saturating the number of cores. Hence, scan-intensive queries would benefit from wider main memory bandwidths. Join-intensive queries saturate the number of cores before saturating the main memory bandwidth. Furthermore, join-intensive queries can significantly benefit from hyper-threads as hyper-threads allow overlapping long-latency data-cache misses. Hence, join-intensive queries can benefit from an increased number of hyper-threads and/or physical cores. Similarly, join-intensive queries can benefit from cheap thread interleaving mechanisms, such as co-routines, that allow overlapping long-latency data-cache stalls [86, 87, 88, 49].

Concurrently executing scan- and join-intensive queries provide a scenario where both core and memory resources are fully utilized. However, concurrently executing queries interfere with each other in the shared memory bandwidth, which results in an increased response time for the join-intensive query. Therefore, OLAP systems should carefully schedule their

concurrent queries and be aware of potential interference. Isolation mechanisms, such as Intel's Cache Allocation Technology, can be useful to mitigate the interference [70].

SIMD and predication are useful for improving single-threaded performance but, due to bandwidth limitations, they fall short on multi-threaded performance. Hardware prefetchers are essential for high-performance scans but are not so useful for joins. Hyper-threading is useful for random-access-heavy queries as it allows overlapping the long-latency data stalls, but falls short on sequential-access-heavy queries. Turbo-boost is effective for a few particular scenarios but mostly provide modest speedups. Therefore, hardware (software) developers should design hardware (software) based on software (hardware) characteristics for optimal performance.

7.1.3 Hybrid Transactional and Analytical Processing

This section summarizes the highlights of our work on Hybrid Transactional and Analytical Processing (HTAP) workloads and discusses its implications. HTAP systems significantly suffer from the interference among the OLTP and OLAP components that share the data and hardware. Sharing the data cause software-level interference.

The software-level interference depends on how fast the OLTP component generates fresh data vs. how fast the OLAP component processes it. Intuitively, the OLAP component should be faster in processing the fresh tuples than the OLTP component generates the fresh tuples. Otherwise, the number of fresh tuples generated *during* the processing of the fresh tuples will be even larger than the number of fresh tuples that are being processed. As a result, the number of fresh tuples will be exponentially increased, and the query execution time will also be exponentially increased.

Blocking the OLTP side during fresh tuple propagation can solve the problem of exponentially increasing number of fresh tuples. However, this would result in a significantly reduced OLTP throughput, especially if the fresh tuple propagation happens frequently. Hence, systems should adopt techniques to efficiently propagate the fresh tuples such that the OLAP component is fast enough to consume the fresh tuples, rather than blocking the OLTP component. We have shown that it is possible to have fast enough fresh tuple propagation by using just enough number of cores allocated for fresh tuple propagation.

Hardware-level interference is a problem for the academic prototype that we built. In particular, the OLTP throughput drops by 22 to 40% due to the interference caused by the OLAP component. Hence, HTAP systems should adopt techniques to isolate OLTP and OLAP performance at hardware level by using isolation mechanisms such as Intel's Cache Allocation Technology that allows isolating the last-level cache accesses by partitioning the last-level cache among the cores that share it [70].

7.2 Conclusions and Future Outlook

This thesis presents micro-architectural analysis of modern database workloads. Database workloads have significantly evolved in the past twenty years. Traditional database systems that serve for all types of database operations have evolved into fast, specialized database systems optimized for a particular type of workload. While the initial database systems mostly served for Online Transactional Processing (OLTP) workloads, the data warehousing applications have led to Online Analytical Processing (OLAP) workloads and the specialized OLAP systems used to serve for OLAP workloads. The recent real-time analytical processing applications have led to Hybrid Transactional and Analytical Processing (HTAP) workloads and the specialized HTAP systems used to serve for HTAP workloads.

Similarly, modern processors have significantly evolved in the past twenty years. Unicore, simple processors with megabytes of main memory have evolved into power-hungry, multi-core processors with hundreds of gigabytes or terabytes of main memories. Furthermore, the processors are equipped with complex micro-architectural features such as highly-accurate branch predictor, Single Instruction Multiple Data (SIMD) units and deep cache hierarchies. As a result, the database system architectures further evolved to efficiently use the complex modern hardware features, leading to novel database system architectures and query processing paradigms.

This thesis bridges the gap between the state-of-the-art database workloads and modern processors by presenting of the hardware-software interaction among the state-of-the-art database workloads and modern processors. We study OLTP, OLAP and HTAP workloads separately. We focus on different generations of database systems for each workload to identify the patterns in the hardware-software interaction for modern database workloads.

We show that OLTP workloads spend most of their execution time in instruction-cache or data-cache misses, where the data-cache misses are due to the large number of random data-accesses that OLTP workloads do. Therefore, OLTP workloads would benefit from techniques/mechanisms at the software and hardware levels that would aim to mitigate the instruction-cache and data-cache misses, such as transaction compilation and hyper-threads overlapping long-latency data-cache misses.

The OLAP workloads spend most of their execution time in data-cache misses, if the OLAP system follows vector-at-a-time or compiled execution model. The data-cache misses are due to high pressure in the memory bandwidth or to random data-accesses. OLAP systems that follow tuple-at-a-time execution model efficiently use the CPU cycles; however, they require the execution of a significantly larger number of instructions hence are significantly slower than the systems that follow vector-at-a-time and compiled execution models. Hence, OLAP workloads benefit from efficient execution models and would benefit from techniques/mechanisms at the software and hardware levels that would aim to mitigate the data-cache misses.

HTAP workloads suffer from interference at the hardware level. We show that the OLTP

throughput drops by 22 to 40% when the OLTP and OLAP components share hardware resources. Therefore, to isolate the performance of the OLTP and OLAP workloads that share hardware resources, HTAP systems should use resource isolation techniques and mechanisms. Furthermore, HTAP workloads suffer from an increased OLAP query execution time when the OLTP and OLAP sides share the data. We show that OLAP query execution time is exponentially increased if the OLTP side generates the fresh tuples faster than the OLAP side processes them. Therefore, HTAP systems should make sure that the OLAP component is allocated enough resources to process the fresh tuples faster than the OLTP component generates them.

A Appendix

A.1 CPU Cycles Categorization

In this section, we present how we map each CPU cycles category that VTune provides to the individual categories that we use. Table A.1 presents the mapping.

A.2 Read-write Micro-benchmark

In this appendix, we extend our sensitivity analysis with the results of experiments while running the read-write version of the micro-benchmark.

A.2.1 Sensitivity to Data Size

In this section, we examine the read-write micro-benchmark when we run the senstivity to data size experiments. Figure A.1 shows the execution time breakdowns for all the systems as the database size is increased. All the systems follow similar trends to the trends that we observed for the read-only micro-benchmark, except *Shore-MT*. *Shore-MT* suffers significantly more from Dcache stalls for the read-write micro-benchmark than it is for the read-only micro-benchmark. We examined the function call trace of *Shore-MT* and saw that the increased

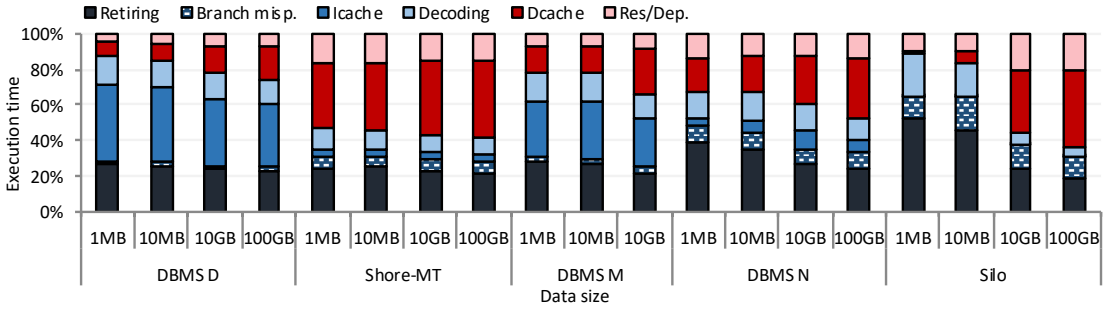


Figure A.1 – Breakdowns of the CPU cycles as we increase the database size when running the read-write microbenchmark.

Table A.1 – The mapping between VTune's original and our CPU cycles categorization.

VTune's original category	Mapped category
Back-End, Memory	Dcache
Back-End, Core	Resource/dependency
Front-End, Front-End Latency, ICache Misses	Icache
Front-End, Front-End Latency, ITLB Overhead	Icache
Front-End, Front-End Latency, Branch Resteer	Branch misprediction
Front-End, Front-End Latency, DSB Switches	Decoding
Front-End, Front-End Latency, Length Changing Prefixes	Decoding
Front-End, Front-End Latency, MS Switches	Decoding
Front-End, Front-End Bandwidth	Decoding
Bad Speculation	Branch misprediction
Retiring	Retiring

Table A.2 – Normalized throughput for the read-write micro-benchmark. Throughput is normalized to the throughput values of the read-only micro-benchmark.

	1MB	10MB	10GB	100GB
DBMS D	0.7	0.7	0.8	0.8
Shore-MT	0.5	0.5	0.6	0.6
DBMS M	0.7	0.7	0.7	-
DBMS N	0.7	0.7	0.7	0.8
Silo	0.8	0.8	0.8	0.9

Dcache stalls are due to the logging meta-data processing that *Shore-MT* does for update queries, but not read-only queries. Being a disk-based system, *Shore-MT* uses a heavy data structure to keep a large amount of log information. *DBMS N*, on the other hand, uses command logging where only the invoked transaction and its parameters are logged, which then replayed in the recovery time if needed. Hence, *DBMS N* moves the expensive logging operation out of the critical path and suffers less from the logging operations.

DBMS M has higher Dcache stalls compared to the read-only micro-benchmark. This is likely due to that *DBMS M* uses a multi-version concurrency concurrency control mechanism, which, in the case of updates, creates chains of versions. Chain traversal requires more random data accesses and hence more Dcache stalls.

DBMS N and *Silo* has less Dcache stalls compared to the read-only micro-benchmark. This is due to that update operation has a higher data locality than the read, as update requires reading from and writing to the same data block. As a result, it results in less Dcache stalls.

Table A.2 shows the normalized throughputs for the read-write micro-benchmark. We normalize the values with respect to the read-only micro-benchmark, as the normalization across the systems (as done in Table 4.2) provides similar results to Table 4.2. We observe that read-write micro-benchmark is always slower for all the systems and data sizes. This is because the update operation requires more work than the read-only operation. It requires more work for concurrency control as well as logging. As a result, its throughput is lower. *Shore-MT*'s relative throughput is the lowest among the systems. This highlights *Shore-MT*'s inefficient locking and logging mechanisms.

As the data size is increased, read-write micro-benchmark's throughput is getting closer to the read-only micro-benchmark. This is because of the data locality of the read-write micro-benchmark. As the data size is increased, data locality matters more for the micro-benchmark throughput. Hence, read-write micro-benchmark's throughput gets closer to the read-only micro-benchmark. Nevertheless, read-only micro-benchmark is 20% to 50% faster than the read-write micro-benchmark.

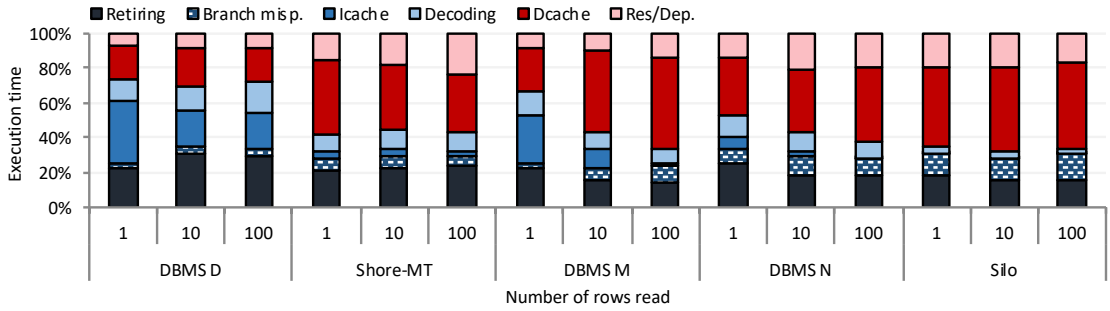


Figure A.2 – Breakdowns of the execution time as we increase the amount of work per transaction when running the read-write micro-benchmark.

Table A.3 – Normalized throughput for the read-write micro-benchmark. Throughput is normalized to the throughput values of the read-only micro-benchmark.

	1 row	10 rows	100 rows
DBMS D	0.8	0.8	0.7
Shore-MT	0.6	0.8	0.8
DBMS M	0.7	0.5	0.5
DBMS N	0.8	0.8	0.8
Silo	0.9	0.9	0.9

A.2.2 Sensitivity to Work per Transaction

In this section, we examine the read-write micro-benchmark when we run the sensitivity to work per transaction experiments. Figure A.2 shows the execution time breakdowns for all the systems as the database size is increased. We observe similar trends to the read-only micro-benchmark As the number of rows updated per transaction is increased *DBMS D* and *M*'s Icache stalls are decreased, and they become more and more Dcache-stalls-bound. *Shore-MT*, *DBMS N* and *Silo*, being already Dcache-stalls-bound systems, have similar micro-architectural behavior across the varied amount of work per transaction.

Table A.3 shows the normalized throughputs for the read-write micro-benchmark. We normalize the values with respect to the read-only micro-benchmark, as the normalization across the systems (as done in Table 4.3) provides similar results to Table 4.3. We observe that the read-write micro-benchmark's throughput is decreased as the amount of work per transaction is increased for *DBMS D* and *M*. This is because the read-only micro-benchmark benefits more from the increased amount of work per transaction compared to the read-write micro-benchmark. The read-write micro-benchmark requires more work, and hence more instructions than the read-only micro-benchmark. As a result, the increased instruction locality is less useful to the read-write micro-benchmark than the read-only micro-benchmark.

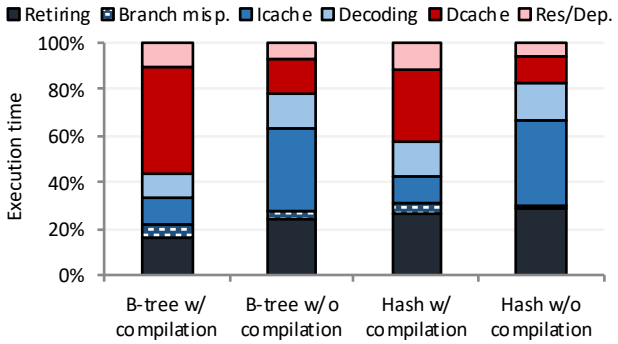


Figure A.3 – Breakdowns of the execution time for different index structures with and without compilation optimizations while running the read-write micro-benchmark.

Table A.4 – Normalized throughput for different index structures with and without compilation when running the read-write micro-benchmark.

	Micro-bench.
B-tree w/ comp.	1
B-tree w/o comp.	0.2
Hash w/ comp.	2.3
Hash w/o comp.	0.3

A.2.3 Index, Compilation and Data Type

In this section, we evaluate the index, compilation and data type for the read-write micro-benchmark. Figure A.3 presents the execution time breakdowns for different index types having compilation turned on and off. We observe that compilation reduces the Icache stalls significantly similar to the read-only micro-benchmark. Table A.4 presents the normalized throughput values. Compilation improves the throughput by 5-7.7x similar to the read-only micro-benchmark.

Figure A.4 shows the execution time breakdowns for *DBMS N* and *Silo* for Long and String data types. *DBMS N* and *Silo* present similar results to the read-only micro-benchmark. *DBMS N* has less Dcache stalls for the String data type, whereas *Silo* has similar amount of Dcache

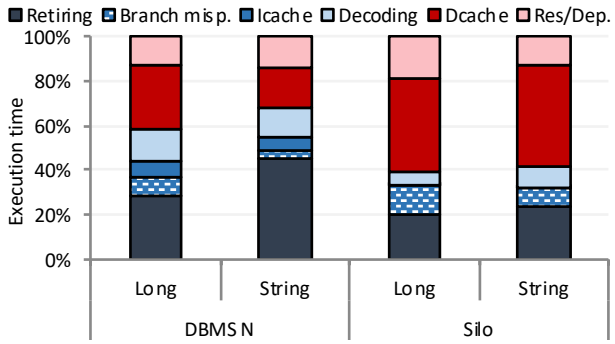


Figure A.4 – Breakdowns of the execution time for String and Long data types while running the read-write micro-benchmark.

Table A.5 – Normalized throughput for String and Long data types while running the read-write micro-benchmark.

	Long	String
DBMS N	1	0.9
Silo	1	0.6

stalls for the String and Long data types.

Table A.5 shows the normalized throughput values for String and Long data types. The results are similar to the read-only micro-benchmark. While *Silo* has lower throughput for String due to the increased amount of work for String, *DBMS N* has similar throughput thanks to utilization of the workload locality for the String data type.

Bibliography

[1] 2012. The Data Deluge. *Nat. Cell Biol.* 14, 8 (2012), 775–775.

[2] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems.* Now Publishers Inc.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go? *(VLDB)*. 266–277.

[4] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. 2017. *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores.* Morgan & Claypool Publishers.

[5] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *CIDR*.

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*. 1–14.

[7] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *CIDR*.

[8] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (1976), 97–137.

[9] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. 2015. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server *(BDCloud)*. 1–8.

[10] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. 2016. Micro-Architectural Characterization of Apache Spark on Batch and Stream Processing Workloads *(BDCloud)*. 59–66.

[11] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. 1998. Memory System Characterization of Commercial Workloads. In *ISCA*. 3–14.

[12] S. Beamer, K. Asanovic, and D. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *IISWC*. 56–65.

[13] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM TODS* 8, 4 (1983), 465–483.

[14] Alexander Böhm, Jens Dittrich, Niloy Mukherjee, Ippokrantis Pandis, and Rajkumar Sen. 2016. Operational Analytics Data Management Systems. *Proc. VLDB Endow.* 9, 13 (2016), 1601–1604.

[15] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. 2006. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine *(SIGMOD)*. 479–490.

[16] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*. 61–76.

[17] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[18] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51 (2008). Issue 12.

[19] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.* 26, 1 (1997), 65–74.

[20] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556.

[21] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[22] E. F. Codd. 1982. Relational Database: A Practical Foundation for Productivity. *Commun. ACM* 25, 2 (1982), 109–117.

[23] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *DBTest*. Article 8, 6 pages.

[24] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*. 1243–1254.

[26] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. *ALEX: An Updatable Adaptive Learned Index*. Technical Report.

[27] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark Silicon and The End of Multicore Scaling. In *ISCA*. 365–376.

[28] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*. 37–48.

[29] G. Graefe and W. J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[30] Jawad Haj-Yihia, Ahmad the, Yosi Ben Asher, and Avi Mendelson. 2016. Fine-Grain Power Breakdown of Modern Out-of-Order Cores and Its Implications on Skylake-Based Systems. *ACM Trans. Archit. Code Optim.* 13, 4 (2016).

[31] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik G. Hallnor, Hong Jiang, Martin G. Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stéphan Jourdan, Steve Gunther, Thomas Piazza, and Ted Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (2014), 6–20.

[32] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. 2007. Database Servers on Chip Multiprocessors: Limitations and Opportunities *(CIDR)*. 79–87.

[33] Stavros Harizopoulos, Daniel J. Abadi, Sam Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*. 981–992.

[34] Gerald Held, Michael Stonebraker, and Eugene Wong. 1975. INGRES: A Relational Data Base System. In *American Federation of Information Processing Societies: National Computer Conference*. 409–416.

[35] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends (R) in Databases* 1, 2 (2007).

[36] Tony Hey, Stewart Tansley, and Kristin Tolle. 2009. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research.

[37] Judith Hurwitz, Alan Nugent, Fern Halper, and Marcia Kaufman. 2013. *Big Data For Dummies* (1st ed.). For Dummies.

[38] HyPer. 2012. Computingat the European Organization for Nuclear Research (CERN). http://cds.cern.ch/record/1997391/.

[39] HyPer. 2015. HyPer. http://hyper-db.de/.

[40] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[41] Intel. 2014. Disclosure of Hardware Prefetcher Control on Some Intel Processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[42] Intel. 2016. Intel(R) 64 and IA-32 Architectures Optimization Reference Manual.

[43] Intel. 2018. Hardware Event-based Sampling Collection. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/hardware-event-based-sampling-collection.

[44] Intel. 2018. Intel VTune Amplifier XE Performance Profiler. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[45] Intel. 2018. Understanding How General Exploration Works in Intel VTune Amplifier. https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe.

[46] Intel. 2019. Intel(R) 64 and IA-32 Architectures Optimization Reference Manual.

[47] Intel. 2020. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[48] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.* 21 (2012), 239–263. Issue 2.

[49] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714.

[50] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *ISCA*. 158–169.

[51] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based Data Partitioning. In *SIGMOD*. 433–445.

[52] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries over Heterogeneous Data Through Engine Customization. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 972–983.

[53] Kimberly Keeton, David A. Patterson, Yong Qian He, Raphael C. Raphael, and Walter E. Baker. 1998. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*. 15–26.

[54] A. Kemper and T. Neumann. 2011. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*. 195–206.

[55] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. 2013. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE DEBull* 36, 2 (2013), 41–47.

[56] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222.

[57] Kangnyeon Kim, Ryan Johnson, and Ippokratis Pandis. 2019. BionicDB: Fast and Power-Efficient OLTP on FPGA. In *EDBT*. 301–312.

[58] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[59] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. 2015. Oracle Database In-Memory: A Dual Format In-memory Database. In *ICDE*. 1253–1258.

[60] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *SIGMOD*. 1177–1184.

[61] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton Memory-Optimized OLTP Engine. *IEEE DEBull* 36, 2 (2013), 34–40.

[62] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. 8, 12 (2015), 1740–1751.

[63] Jean-Yves Le Boudec. 2010. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland.

[64] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE DEBull* 36, 2 (2013), 28–33.

[65] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*. 743–754.

[66] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*. 38–49.

[67] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. 302–313.

[68] Jan Lindstrom, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. 2013. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE DEBull* 36, 2 (2013).

[69] Anisa Llaveshi, Utku Sirin, Anastasia Ailamaki, and Robert West. 2019. Accelerating B+tree Search by Using Simple Machine Learning Techniques. In *AIDB*. 1–10.

[70] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving Resource Efficiency at Scale with Heracles. *ACM Trans. Comput. Syst.* 34, 2, Article 6 (2016), 33 pages.

[71] David Lomet and Paul Larson (Eds.). 2013. Special Issue on Main-Memory Database Systems. *IEEE Data Engineering Bulletin* 36, 2 (2013).

[72] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. 2012. Scale-out Processors. In *ISCA*. 500–511.

[73] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*. 37–50.

[74] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730.

[75] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*. 183–196.

[76] MemSQL 2020. MemSQL. http://www.memsql.com/.

[77] G. E. Moore. 2006. Cramming More Components onto Integrated Circuits. *IEEE Solid-State Circuits Society Newsletter* 11, 3 (2006), 33–35.

[78] Trevor N. Mudge and Urs Hölzle. 2010. Challenges and Opportunities for Extremely Energy-Efficient Processors. *IEEE Micro* 30, 4 (2010), 20–24.

[79] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (June 2011), 539–550.

[80] Thomas Neumann and Viktor Leis. 2014. Compiling Database Queries into Machine Code. *IEEE DEBull* 37, 1 (2014), 3–11.

[81] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. 677–689.

[82] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Kenneth G. Wilson, and Kunyung Chang. 1996. The Case for a Single-Chip Multiprocessor. In *ASPLOS*. 2–11.

[83] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*. 1771–1775.

[84] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *PVLDB* 3, 1 (2010), 928–939.

[85] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB* 11, 6 (2018), 663–676.

[86] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB* 11, 2 (2017), 230–242.

[87] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *VLDB J.* 28, 4 (2019), 451–471.

[88] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations *(DaMoN)*. 13:1–13:8.

[89] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091.

[90] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. Performance of Database Workloads on Shared-memory Systems with Out-of-Order Processors. In *ASPLOS-VIII*. 307–318.

[91] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *SIGMOD*. 2043–2054.

[92] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR*.

[93] Shore-MT. 2017. Shore-MT Official Website. http://diaswww.epfl.ch/shore-mt/.

[94] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. doppioDB: A Hardware Accelerated Database. In *SIGMOD*. 1659–1662.

[95] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of A Column Store Myth. In *SIGMOD*. 731–742.

[96] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE*. 698–709.

[97] Utku Sirin, Raja Appuswamy, and Anastasia Ailamaki. 2016. OLTP on a server-grade ARM: Power, Throughput and Latency Comparison. In *DaMoN*. ACM, 10:1–10:7.

[98] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP *(SIGMOD)*. 387–402.

[99] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. 2017. A Methodology for OLTP Micro-architectural Analysis. In *DaMoN*. 1:1–1:10.

[100] A. Skidanov, A. J. Papito, and A. Prout. 2016. A Column Store Engine for Real-time Streaming Analytics. In *ICDE*. 1287–1297.

[101] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. 2011. Vectorization vs. Compilation in Query Execution *(Damon)*. 33–40.

[102] Shriram Sridharan and Jignesh M. Patel. 2014. Profiling R on a Contemporary Processor. *Proc. VLDB Endow.* 8, 2 (Oct. 2014), 173–184.

[103] R. Stets, K. Gharachorloo, and L.A. Barroso. 2002. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*. 37–48.

[104] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of An Architectural Era: (It's Time for A Complete Rewrite). In *VLDB*. 1150–1160.

[105] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. 340–355.

[106] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[107] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE DEBull* 36, 2 (2013), 21–27.

[108] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in Wonderland – Where Do Cache Misses Come From in Major OLTP Components?. In *DaMoN*. 8:1–8:6.

[109] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in Wonderland: Where Do Cache Misses Come From in Major OLTP Components? *(Damon)*. 8.

[110] Pınar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. 2013. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. *VLDB J.* 22, 2 (2013), 151–175.

[111] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: Analyzing TPC's OLTP Benchmarks – The Obsolete, The Ubiquitous, The Unexplored. In *EDBT*. 17–28.

[112] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, The Ubiquitous, The Unexplored *(EDBT)*. 17–28.

[113] TPC. 2017. TPC Transcation Processing Performance Council. http://www.tpc.org/.

[114] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *SOSP*. 18–32.

[115] VoltDB. 2015. VoltDB. http://www.voltdb.com/.

[116] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal Streams in Commercial Server Applications. In *IISWC*. 99–108.

[117] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture *(ISPASS)*. 35–44.

[118] A. Yasin, Y. Ben-Asher, and A. Mendelson. 2014. Deep-dive Analysis of the Data Analytics Workload in CloudSuite. In *IISWC*. 202–211.

[119] Ahmad Yasin, Jawad Haj-Yahya, Yosi Ben-Asher, and Avi Mendelson. 2019. A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors. *ACM Trans. Archit. Code Optim.* 16, 4, Article 46 (2019), 25 pages.

[120] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. *Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores.* Technical Report. MIT, CMU, Department of Computer Science.

Utku Sirin

Ph.D. candidate at Data-Intensive Applications and Systems (DIAS) Laboratory
School of Computer and Communication Sciences (IC),
École Polytechnique Fédérale de Lausanne (EPFL)
utku.sirin@epfl.ch, utkusirin@gmail.com
www.utkusirin.com

RESEARCH INTERESTS

High-performance database management systems, modern micro-architectures, hardware-software co-design

PROFESSIONAL EXPERIENCE

2014 – 2021 École Polytechnique Fédérale de Lausanne (EPFL) Lausanne, Switzerland
Research and Teaching Assistant, advised by Prof. Anastasia Ailamaki

06 – 08/2017 Microsoft Research (MSR) Redmond, WA, USA
Research Intern at Data Management, Exploration and Mining (DMX) Group

2011 – 2013 Middle East Technical University (METU) Ankara, Turkey
Research and Teaching Assistant, advised by Prof. Faruk Polat

HONORS & AWARDS:

- Swiss National Science Foundation (SNSF) Postdoc Fellowship, 2020
- Teaching assistant award by IC, EPFL, 2019
- Best paper award at DaMoN@SIGMOD, 2017
- ACM SIGMOD Student Research Competition (SRC), Winner, 2017
- Microsoft Research PhD scholarship, 2017
- Doctoral degree fellowship by IC, EPFL, 2014
- Ranked 1st in M.Sc. program, GPA: 4.00/4.00, 2013
- Ranked 1st in B.Sc. program, GPA: 3.94/4.00, 2011
- Master's degree fellowship by Sci. and Tech. Research Council (TUBITAK), 2012 – 2013
- Bachelor's degree fellowship by The Credits and Housing Board (KYK), 2006 – 2011
- Bachelor's degree fellowship by METU Development Foundation, 2006 – 2011
- Ranked 29th out of ~1.5 million students in nation-wide university entrance exam, 2006

EDUCATION

2014 – 2021: Doctoral student **EPFL, Switzerland**
Thesis: Micro-architectural Analysis of Database Workloads

2011 – 2013: M.Sc. in Computer Engineering **METU, Turkey**
Graduation rank: 1, Grade: 4.00 / 4.00
Thesis: Batch-mode Reinforcement Learning for Controlling Gene Regulatory Networks and Multi-model Gene Expression Data Enrichment Framework

2006 – 2011: B.Sc. in Computer Engineering **METU, Turkey**
Graduation rank: 1 / 103, Grade: 3.94 / 4.00

PUBLICATIONS

- Utku Sirin, Sandhya Dwarkadas, Anastasia Ailamaki: Performance Characterization of HTAP Workloads. In **ICDE**, 2021 (6-page short paper).
- Utku Sirin, Sandhya Dwarkadas, Anastasia Ailamaki: Workload Interference Analysis for HTAP. In **CIDR**, 2021 (1-page abstract).
- Utku Sirin, Anastasia Ailamaki: Micro-architectural Analysis of OLAP: Limitations and Opportunities. **PVLDB** 13(6): 840-853, 2020.
- Anisa LLaveshi, Utku Sirin, Robert West, Anastasia Ailamaki: Accelerating B+tree Search by Using Simple Machine Learning Techniques. In **AIDB@VLDB**, 2019.
- Utku Sirin: Energy-efficient Database Machines. In **SIGMOD SRC**, 2017 (2-page abstract).
- Utku Sirin, Ahmad Yasin, Anastasia Ailamaki: A Methodology for OLTP Micro-architectural Analysis. In **DaMoN@SIGMOD**, 2017 (Best paper award).
- Utku Sirin, Raja Appuswamy, Anastasia Ailamaki: OLTP On A Server-grade ARM: Power, Throughput and Latency Comparison. In **DaMoN@SIGMOD**, 2016.
- Utku Sirin, Pinar Tözün, Danica Porobic, Anastasia Ailamaki: Micro-architectural Analysis of In-memory OLTP. In **SIGMOD**, 2016.
- Utku Sirin: OLTP's Core Cycles: What is "Performance Optimization"? In **HPTS**, 2015 (5-min talk).
- Utku Sirin, Faruk Polat, Reda Alhajj: Batch Mode TD(λ) for Controlling Partially Observable Gene Regulatory Networks. In IIEEE/ACM Transactions on Computational Biology and Bioinformatics (**TCBB**), vol. 14, no. 6, pp. 1214-1227 (2015)
- Utku Sirin, Utku Erdogdu, Faruk Polat, Mehmet Tan, Reda Alhajj: Effective Gene Expression Data Generation Framework Based on Multi-model Approach. Artificial Intelligence in Medicine (**AIIM**), 70:41-61 (2014)
- Utku Sirin, Faruk Polat, Reda Alhajj: Employing Batch Reinforcement Learning to Control Gene Regulation Without Explicitly Constructing Gene Regulatory Networks. In **IJCAI**, 2013.
- Utku Sirin, Utku Erdogdu, Mehmet Tan, Faruk Polat, Reda Alhajj: Effective Enrichment of Gene Expression Data Sets. In **ICMLA**, 2012.
- Utku Sirin, Ruket Çakici, Deniz Zeyrek: METU Turkish Discourse Bank Browser. In **LREC**, 2012.

TEACHING ASSISTANTSHIPS

- Spring 2016 – 2020: Introduction to Database Systems by Prof. Anastasia Ailamaki
- Fall 2019: Machine Learning for Database Systems by Prof. Anastasia Ailamaki, and Prof. Christoph Koch
- Fall 2018 – 2020: Computer Networks by Prof. Katerina Argyraki
- Fall 2017: Applied Data Analysis by Prof. Robert West
- Fall 2016: Introduction to Multiprocessor Architecture by Prof. Dionisios Pnevmatikatos

LANGUAGES

Turkish – Native, English – Fluent, French – Beginner

VOLUNTEERING

- Mentor at GirlsCoding organization: http://girlscoding.org/
- LauzHack 2020 against COVID-19: http://tiny.cc/zk9nmz

HOBBIES

- Acting. At the Village Players of Lausanne, an English-speaking theater club in Lausanne, Switzerland: https://www.villageplayers.ch/ (ongoing). At Theater Portrait, a theater club in Ankara, Turkey: http://www.portresanat.com/ (past).
- Playing guitar and singing.

REFERENCES

Anastasia Ailamaki, Professor
EPFL IC IINFCOM DIAS
BC 226 (Bâtiment BC)
Station 14
CH-1015 Lausanne
anastasia.ailamaki@epfl.ch

Surajit Chaudhuri, Distinguished Scientist
Microsoft Building 99
14820 NE 36th Street
Redmond, Washington
98052 USA
surajitc@microsoft.com

Sandhya Dwarkadas, Professor
University of Rochester,
Rochester, NY 14627-0226
sandhya@cs.rochester.edu

Nick Koudas, Professor
University of Toronto,
40 St. George Street Rm BA5240
koudas@cs.toronto.edu

Christoph Koch, Professor
EPFL IC IINFCOM DATA
BC 260 (Bâtiment BC)
Station 14
CH-1015 Lausanne
christoph.koch@epfl.ch