ADAPTIVE ARCHITECTURES FOR ROBUST DATABASE MANAGEMENT SYSTEMS

Doctoral thesis by
Tiemo Christian Bang, M.Sc.

submitted in fulfillment of the requirements for the
degree of *Doktor-Ingenieur* (Dr.-Ing.)

at the Computer Science Department
of the Technical University of Darmstadt

Reviewers
Prof. Dr. rer. nat. Carsten Binnig
Prof. Joseph M. Hellerstein, Ph.D.

Darmstadt 2022

Tiemo Christian Bang
Adaptive Architectures for Robust Database Management Systems
Darmstadt, Technical University of Darmstadt, 2022
Viva voce: 26.07.2022

Please cite this work as
URN: urn:nbn:de:tuda-tuprints-213831
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/21383

This document is provided by TUprints,
The Publication Service of the Technical University of Darmstadt
https://tuprints.ulb.tu-darmstadt.de

ERKLÄRUNG LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 2022

Tiemo Christian Bang

"It is the pervading law of all things organic and inorganic, of all things physical and metaphysical, [...] of all true manifestations of the head, of the heart, of the soul, that the life is recognizable in its expression, that form ever follows function."

— Sullivan, 1896 in *The Tall Office Building Artistically Considered*

Mé milující ženě
(To my loving wife)

ABSTRACT

Ever since, the workload and hardware conditions for Database Management Systems (DBMSs) are expanding through new use cases and hardware. Starting from the first transactional DBMSs supporting the moon landing, today's DBMSs process billions of sales transactions for online commerce on a single day and process many further workloads like reporting or fraud detection. Similarly, for the early DBMSs only few hardware platforms were available, while today's DBMSs face a host of diverse hardware platforms. Indeed, today a single DBMS is exposed to changing and load-fluctuating workloads, e.g., depending on the popularity of sales items, and is operated on changing hardware.

However, serving changing workloads and supporting diverse hardware platforms is non-trivial, as for the best performance DBMS designs must be specialized. For flexibly specializing DBMS components to changing workload or hardware conditions, adaptation approaches have been proposed, e.g., adaptive query execution. Whereas the DBMS architecture that deploys the components is manually specialized and statically implemented at design-time. While in essence all DBMS architectures determine which components are executed together on what resource partitions, today there exist only few static architectures that largely predetermine this at design-time for specific conditions, e.g., for multi-processor hardware the NUMA-aware architecture dictates a resource partitioning per processor for all components. Besides high re-implementation effort for adjusting these static architectures, this approach also inadequately simplifies architectures with coarse-grained specialization for many components at once, neglecting the distinct workload and hardware effects on individual DBMS components and their contained functions. Hence, these static DBMS architectures severely degrade DBMS performance, when unfit.

This dissertation pursues the adaptation of DBMS architectures. For high and robust performance under changing workload and hardware, the static specialization at design-time is progressed to the flexible and precise adaptation of the architecture when deploying the DBMS or even at runtime. The approach is an initial evaluation of DBMSs with static architectures. Then, general concepts for the adaptation of DBMS architectures are proposed, based on which adaptive architectures for the classes of single-server and multi-server DBMSs are realized.

The overall idea for the adaptation of DBMS architectures is to flexibly compose fine-grained building blocks of the DBMS to a best-fit architecture, i.e., adapting at the granularity of distinct functions of DBMS components without requiring any re-implementation. Besides the effortless adjustment of the architecture, this dissertation proposes

concepts with an emphasis on fine-granular and separate adaptation for distinct DBMS functions, such that optimizers can derive architectures best-fit for the specific conditions and functions at hand. By constructing a navigable optimization space for architectures of single- and multi-server DBMSs, the proposed concepts not only enable the flexible mimicking of any existing architecture, but importantly enable the creation of entirely new architectures.

The key findings are that both the realized adaptive single-server and the adaptive multi-server architecture prove effective and efficient, for adapting to the conditions considered in this dissertation. Under changing transactional and mixed workloads, the proposed adaptive architectures generally perform at least on par with the individually best state-of-the-art architecture. Indeed, when adopting novel better-fit architectures, all existing architectures are outperformed, e.g., with resource assigned at a granularity unlike any of today's single-server architectures or when separately specializing for distinct queries of mixed workloads rather than compromising as today's multi-server architectures. That is, the proposed flexible and precise adaptation demonstrates higher and more robust performance.

While our findings exhibit novel better-fit architectures only for a subset of possible workload and hardware conditions, this dissertation overall indicates high potential for adapting architectures with the proposed concepts. As the proposed concepts make a vast optimization space generally navigable, optimizers will be able to adapt DBMS architectures flexibly and more precisely to many workloads and hardware. Instead of fragile static architectures, the proposed adaptive architectures thus provide the necessary foundation for DBMSs to achieve high and robust performance under changing workload and hardware.

ZUSAMMENFASSUNG

Seit jeher entwickeln sich die Arbeitslasten und die Hardware für Datenbankmanagementsysteme (DBMS) durch neue Anwendungsfälle und Hardware. Ausgehend von den ersten transaktionalen DBMS, die die Mondlandung ermöglichten, verarbeiten die heutigen DBMS Milliarden von Verkaufstransaktionen für den Online-Handel an einem einzigen Tag und verarbeiten viele weitere Arbeitslasten wie Berichterstattung oder Betrugserkennung. In ähnlicher Weise waren für die frühen DBMS nur wenige Hardware-Plattformen verfügbar, während die heutigen DBMS mit einer Vielzahl unterschiedlicher Hardwareplattformen konfrontiert sind. In der Tat ist ein einzelnes DBMS heute wechselnden und lastschwankenden Arbeitslasten ausgesetzt, z.B. abhängig von der Beliebtheit der Verkaufsartikel, und wird auf wechselnder Hardware betrieben.

Die Bewältigung dieser wechselnder Arbeitslasten und die Unterstützung der verschiedener Hardware-Plattformen ist jedoch nicht trivial, denn um die beste Leistung zu erzielen, müssen DBMS-Designs spezialisiert werden. Für die flexible Spezialisierung von DBMS-Komponenten auf sich ändernde Arbeitslast- oder Hardware-Bedingungen wurden Anpassungsansätze vorgeschlagen, z.B. die adaptive Abfrageausführung. Im Gegensatz dazu wird die DBMS-Architektur, in der die Komponenten eingesetzt werden, zur Entwurfszeit manuell spezialisiert und statisch implementiert. Während im Wesentlichen alle DBMS-Architekturen festlegen, welche Komponenten zusammen auf welchen Ressourcenpartitionen ausgeführt werden, gibt es heute nur einige wenige statische Architekturen, die dies zur Entwurfszeit für bestimmte Bedingungen weitgehend vordefinieren, z.B., bei Multiprozessor-Hardware diktiert die NUMA-aware Architektur eine Ressourcenpartitionierung pro Prozessor für alle Komponenten. Neben dem hohen Re-Implementierungsaufwand für die Anpassung dieser statischen Architekturen vereinfacht dieser Ansatz auch Architekturen inadäquat mit grobkörniger Spezialisierung für viele Komponenten auf einmal, und vernachlässigt somit die unterschiedlichen Auswirkungen von Arbeitslast und Hardware auf einzelne DBMS-Komponenten und die darin enthaltenen Funktionen. Diese statischen DBMS-Architekturen verschlechtern daher die DBMS-Leistung erheblich, wenn sie ungeeignet sind.

Diese Dissertation verfolgt die Adaption von DBMS-Architekturen. Um eine hohe und robuste Leistung bei wechselnder Arbeitslast und Hardware zu erreichen, wird die statische Spezialisierung zur Entwurfszeit zu der flexiblen und präzisen Adaption der Architektur weiterentwickelt. Der Ansatz ist eine erst Evaluation von DBMS mit

statischen Architekturen. Anschließend werden allgemeine Konzepte für die Adaptation von DBMS-Architekturen entworfen, auf deren Basis adaptive Architekturen für die Klassen der Einzel- und Multi-Server-DBMS realisiert werden.

Die generelle Idee für die Adaption von DBMS-Architekturen besteht darin, feinkörnige DBMS-Bausteine flexibel zu einer bestgeeigneten Architektur zusammensetzen, d.h. die Anpassung auf der Granularität einzelner Funktionen von DBMS-Komponenten, ohne eine Neuimplementierung zu erfordern. Neben der leichten Anpassung der Architektur werden in dieser Dissertation Konzepte mit dem Schwerpunkt auf feingranularer und separater Anpassung für verschiedene DBMS-Funktionen vorgeschlagen, sodass Optimierer Architekturen ableiten können, die für die jeweiligen Bedingungen und Funktionen am besten geeignet sind. Durch die Schaffung eines navigierbaren Optimierungsraums für Architekturen von Single- und Multi-Server-DBMS, ermöglichen die vorgeschlagenen Konzepte nicht nur die flexible Nachahmung jedweder bestehender Architektur, sondern ermöglichen vor allem auch die Bildung völlig neuer Architekturen.

Die wichtigsten Ergebnisse sind, dass sowohl die realisierte adaptive Einzel-Server- als auch die adaptive Multi-Server-Architektur sich als effektiv und effizient erweisen, für die Anpassung an die in dieser Dissertation betrachteten Bedingungen. Bei variierenden transaktionalen (OLTP) und gemischten (HTAP) Arbeitslasten schneiden die vorgestellten adaptiven Architekturen im Allgemeinen mindestens genauso gut ab wie die jeweils beste State-of-the-Art-Architektur. In der Tat werden alle bestehenden Architekturen übertroffen, wenn neue, besser geeignete Architekturen angewandt werden, z.B. durch die Ressourcenzuweisung mit einer Granularität, die mit den heutigen Einzel-Server-Architekturen nicht vergleichbar ist, oder bei der separaten Spezialisierung auf verschiedene Abfragen gemischter Arbeitslasten, anstelle von Kompromissen wie in den heutigen Multi-Server-Architekturen. Das heißt, die vorgestellte flexible und präzise Adaption zeigt eine höhere und robustere Leistung.

Obwohl unsere Ergebnisse neuartige, besser geeignete Architekturen nur für eine Teilmenge möglicher Arbeitslast- und Hardware-Bedingungen zeigen, weist diese Dissertation insgesamt auf ein hohes Potenzial für die Anpassung von Architekturen mit den vorgeschlagenen Konzepten hin. Da die vorgeschlagenen Konzepte einen großen Optimierungsraum allgemein navigierbar machen, werden Optimierer in der Lage sein, DBMS-Architekturen flexible und präziser an viele Arbeitslasten und Hardware anzupassen. Anstelle von fragilen statischen Architekturen, bieten die vorgeschlagenen adaptiven Architekturen somit die notwendige Grundlage für eine hohe und robuste Leistung von DBMS bei variierender Arbeitslast und Hardware.

PUBLICATIONS

The following peer-reviewed publications are part of this cumulative dissertation. Their content is printed in Part II, Chapters 7 – 10.

[1] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. ACM, 2020. DOI: 10.1145/3399666.3399910.

[2] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. "Robust Performance of Main Memory Data Structures by Configuration." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, New York, NY, USA: ACM, 2020. DOI: 10.1145/3318464.3389725.

[3] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "AnyDB: An Architecture-less DBMS for Any Workload." In: *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*. 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper10.pdf.

[4] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware." In: *The VLDB Journal*. 2022. DOI: 10.1007/s00778-022-00742-4.

Further co-authored peer-reviewed publications are:

[1] Benjamin Hilprecht et al. "DBMS Fitting: Why should we learn what we already know?" In: *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*. 2020. URL: http://www.cidrdb.org/cidr2020/papers/p34-hilprecht-cidr20.pdf.

ACKNOWLEDGMENTS

First and foremost, I am sincerely grateful to Prof. Carsten Binnig for his dedicated mentoring. His energetic and encouraging guidance has fostered my academic and personal growth, making my journey towards this dissertation alongside Carsten truly a pleasure and an honor.

Just as important are the co-advisors Norman May, Prof. Ilia Petrov, and Ismail Oukid. I am forever thankful to Norman for his kind and diligent advice leading me to this dissertation. I am equally thankful for the invigorating discussions with Ilia as well as Ismail's benign advice on academic matters and more.

I would like to thank Prof. Joseph Hellerstein as second reviewer for kindly taking the time to review this dissertation.

Besides the academic advisory, I am grateful to Arne Schwarz for fostering the co-innovation between academy and SAP, not only with the budget for the joint research project but overall with his dedication to the SAP HANA Database Campus.

It has been a pleasure to work in such innovative groups as the Data Management Lab at TU Darmstadt and the HANA Database Campus at SAP. I am thankful for the academic discussion we have had and greatly enjoyed our refreshing off-work activities. Also, I am grateful for the many more pleasant and inspiring acquaintances within the academic as well as the industrial community.

I would like to thank my parents for their lasting support and for encouraging curiosity throughout my life.

To my wife Alena, you have been on my side through ups and downs. I am looking forward to continuing our journey by your side.

CONTENTS

Part I

SYNOPSIS

1 INTRODUCTION

Today, database management systems (DBMSs) represent a great success story. This success story started in the 1960s with the first commercial DBMS, Information Control System and Data Language/Interface (ICS/DL/I). It was developed to transactionally manage large bills of materials for the Apollo project, providing essential data management to make the moon landing possible [108]. From there on DBMSs have significantly evolved, providing broader features and ever-higher performance. Hence, today DBMSs have found wide adoption by many businesses relying on their features and their performance, e.g., online commerce conducting billions of sales transactions on Singles' Day or Black Friday [62, 89, 176]. Accordingly, the use cases for DBMSs have been expanding ever since and there is a lasting demand for ever-higher performance, which significantly shaped and increasingly challenged DBMS designs.

1.1 THE EVOLUTION OF DBMS DESIGNS

From the early days of DBMSs in the 1960s until today, DBMS designs have been driven by the support for new use cases and the continuous demand for ever-higher performance. Therefore, the design space of DBMSs has significantly expanded to ever more workloads to support and hardware to operate on. As illustrated in Figure 1.1, for the early DBMSs the design space was very narrow, but DBMSs steadily supported more workloads and operated on more hardware, requiring DBMS designs to cover an ever-larger design space.

The early DBMSs until the 1970s were designed for very specific workload and hardware. For example, ICS/DL/I was specifically designed for the management of bills of materials, i.e., for ensuring transactional consistency of hierarchically structured data, and was specifically designed to operate on NASA's mainframe [108]. Following ICS/DL/I, the first relational DBMSs, e.g., System R [14], Ingres [173] or Oracle [51], were designed to support broader workloads, i.e., enabling efficient querying of managed data independent of the physical storage format, using the relational model and related abstractions [44, 45, 133]. Still, although few hardware platforms were available, the early DBMSs only operated on specific ones, e.g., System R on IBM System/360 [14], Ingres on Programmed Data Processor-11 (PDP-11) [173], and Oracle V2 on Virtual Address eXtension (VAX) systems [51]. The hardware of that time required distinct programming, such that the early DBMSs had specialized implementations to

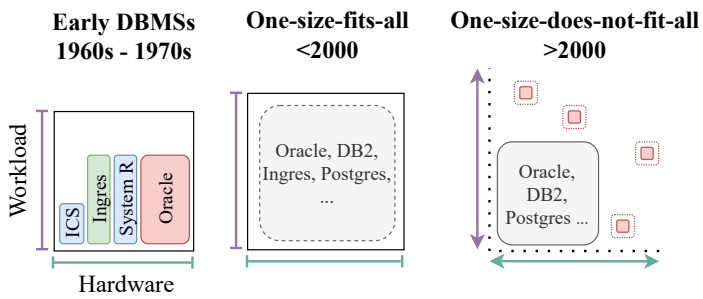


Figure 1.1: Illustration of the expanding design space of workload and hardware in which DBMSs have been designed over the years.

both operate on a specific hardware at all and to take advantage of its distinct features.

As DBMSs found wider adoption, they had to support more diverse workloads, e.g., analytical workloads with complex queries for decision support, transactional workloads with high performance demands for online commerce or graph workloads with complex structured data for fraud detection. Hence, already early DBMSs started specializing in specific workloads, e.g., Teradata's DBC/1012 in analytical processing [167] and Berkeley DB in key-value workloads [134]. Still, in the "one-size-fits-all" design era until the 1990s, major relational DBMSs like DB2 [69], Ingres [173], Oracle [51], and Postgres [171] also attempted to support the entire range of workloads from transactional to graph analytics [170]. Especially, Postgres' extensible data model with abstract data types is a canonical example for the very generic designs of that era. However, it was found that these generic DBMS designs hindered high performance for the individual workloads and also hardware had evolved significantly.

In the following "one-size-does-not-fit-all" design era, many distinct DBMS designs superseded the traditional generic designs to improve DBMS performance for individual use cases, especially for specific workloads and hardware [174]. Most notable, the hardware evolution made large quantities of main memory affordable but also changed hardware to diverse multi-core processors and multi-processor platforms. This hardware evolution thereby offered opportunities for DBMSs to improve performance but also demanded the support of diverse hardware, founding today's many main memory DBMSs. Today there is an abundance of DBMSs specialized for specific workloads or hardware, e.g., relational main memory DBMSs (e.g., HyPer [104], IBM solidDB [121], MonetDB [29, 92], SAP HANA [65]) specialized for hardware with large main memory and diverse transactional and analytical workloads, graph DBMSs (e.g., Neo4J [113]), and a range of simplified key-value DBMSs specialized for main memory and SSDs (e.g., Memcached [68, 130], Redis [124], RocksDB [156]). Consequently, the diversity of workloads and hardware has expanded the design space for DBMSs to a vast scale.

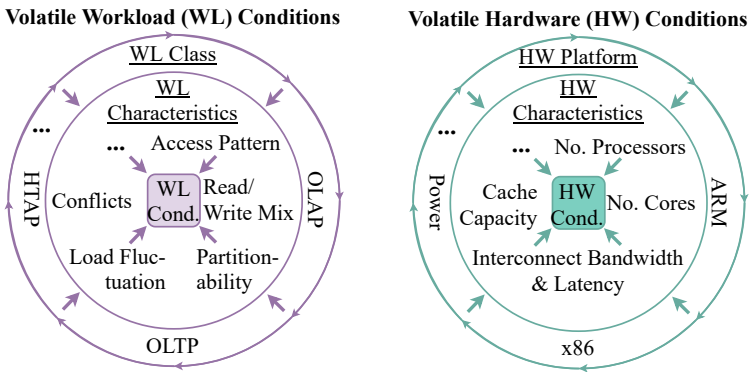


Figure 1.2: Illustration of volatile conditions challenging DBMS design. Changing workloads and hardware platforms influence fundamental characteristics that comprise the volatile conditions.

Besides the vast scale of the design space, DBMS consumption has significantly changed as well, exposing DBMSs to increasingly volatile workload. Today, businesses emphasize agility of their IT infrastructure, flexibly consuming DBMSs as a service and likely in the cloud [129]. Further, many businesses operate online, such that even for the same use case like online commerce the specific workload fluctuates, e.g., due to varying popularity of offered items [177]. Therefore, use cases evolve and demand fluctuates today, exposing DBMSs to changing workload conditions.

Similarly, ever-more hardware is readily available to operate DBMSs, especially in the cloud. The challenge used to lie in distinct DBMS implementations in order to support diverse hardware at all, e.g., DB2 employing four different code bases [32, 38, 77, 99, 174]. Instead, today's DBMSs generally support a range of instruction set architectures (ISAs) and hence the many hardware platforms with processors that implement these ISAs [158], e.g., PowerISA-based [98], x86-based [1, 95] and ARM ISA-based platforms [12]. Today the challenge for DBMSs rather is to utilize the diverse hardware well, despite their diverse characteristics. As we show in our evaluation in Chapter 3, for example, already the varying interconnect bandwidth and latency between processors can impose severe performance degradation on unfit DBMS designs. Overall, there are many distinct characteristics of today's processors (e.g., design of hardware parallelism) and hardware platforms that demand specialization of the DBMS designs. Moreover, the current hardware evolution is seeking new ways to further progress, thus steadily increasing the diversity of hardware characteristics, e.g., as could be observed with the advent of ARM platforms in the cloud [6, 13, 43]. Therefore, today's hardware diversity challenges DBMS design with continuous hardware-specific specialization and DBMSs repeatedly struggle when operating on (new) hardware they are not specialized for.

Consequently, the design space of DBMSs is expanding to vast workload and hardware diversity, and in this space DBMSs must execute changing workloads on changing hardware. As illustrated in Figure 1.2 and will be discussed throughout this dissertation, this exposes DBMS designs to many workload and hardware characteristics that vary over time, making for volatile conditions challenging DBMS design. As follows, despite these volatile conditions, DBMS designs are specialized to maximize performance.

1.2 SPECIALIZATION OF DBMS DESIGNS

Over the years, a general structure for the DBMS design has been established within which DBMSs evolved [78, 149]. That is, full-fledged DBMSs generally follow the design structure shown in Figure 1.3, organizing the DBMS functionality into the depicted components and organizing the deployment of these components with an underlying architecture. Even as DBMS designs were specialized, full-fledged DBMSs continued to follow this general structure, e.g., regardless whether for relational or graph workloads [155]. However, within this general structure, new designs of components and of the architecture were devised for full-fledged DBMSs to better support the growing diversity of workloads and better utilize new hardware. In fact, a breadth of alternative designs specialized to distinct workloads and hardware have emerged, as described below for relational DBMSs.

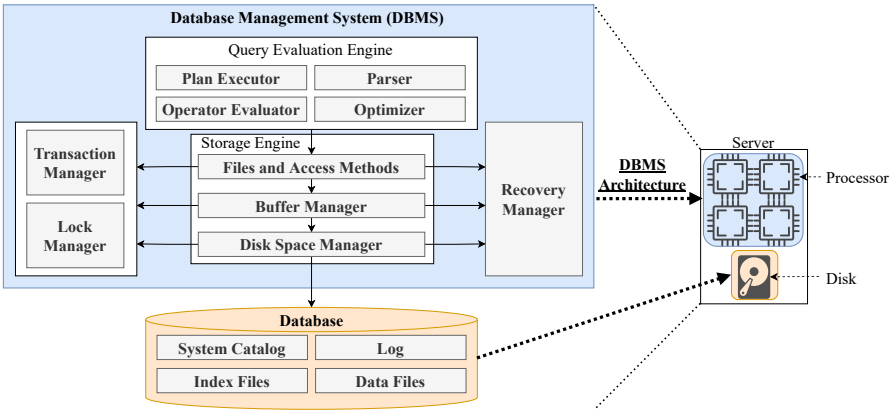


Figure 1.3: General structure of a DBMS design showing the common components and their interaction as well as the underlying DBMS architecture [78, 149].

SOPHISTICATED SPECIALIZATION OF DBMS COMPONENTS For all of the DBMS components shown in Figure 1.3 many designs have emerged in relational DBMSs, advancing and indeed specializing the individual components. Prime examples are the components of the query evaluation engine and the storage engine, whose designs significantly evolved in response to the diversity of workloads and

hardware. For example, the query evaluation engine has reached a broad number of alternative relational operator designs specialized for efficient query execution under specific workloads or hardware, like branch-less or branching selection specialized for filtering under specific selectivities of the workload and for the branching performance of the underlying hardware [87]. Similarly, the storage engine today comprises many specializations of the physical storage design for specific workloads and hardware, e.g., the row-wise record layout for transactional workloads and the column-wise layout for analytical workloads or partitioning of tables regarding the underlying NUMA hardware or multi-server infrastructure [119]. Accordingly, DBMS components have a breadth of alternative designs today, which are geared towards specific conditions.

In the breadth of DBMS component designs, today's DBMSs indeed advanced to automatic specialization with optimizers. Today, optimizers flexibly navigate between alternative specializations of DBMS components, rather than relying on a single fixed implementation of a specific specialization for specific hardware and workload. For example, the query evaluation engine uses an optimizer to automatically specialize the execution of queries for a given workload and hardware, generating an execution plan out of operators considered best-fit [119] and adaptively re-optimizing during the execution to correct unfit execution plans [16, 203]. Similarly, there is a host of advisors that propose specialized designs for DBMS components, e.g., for index selection [114, 193], partitioning [86, 152], and optimizing buffer management [30]. To achieve the best possible performance for the specific conditions at hand, automatic optimization methods choose from alternative designs and comprehensively specialize distinct components. Indeed, optimizers not only specialize components to improve processing speed (e.g., transaction throughput) but even allow specialization for alternative performance metrics like monetary cost per request. Consequently, for many DBMS components sophisticated automatic adaptation approaches have been established which flexibly specialize these components, effectively navigating alternative designs and even supporting alternative optimization objectives.

STATIC SPECIALIZATION OF DBMS ARCHITECTURES Besides evolving designs of the different DBMS components, also a choice of DBMS architectures emerged with the expanding design space. Figure 1.4 displays the architectures commonly implemented in today's relational single-server scale-up DBMSs and multi-server scale-out DBMSs. These architectures determine how DBMS components are instantiated and resources are assigned for executing the DBMS, as follows.

(1) The shared-everything architecture for single-server scale-up DBMSs instantiates a single system partition with all available resources, e.g., implemented in Hekaton [52], MySQL [161] or Post-

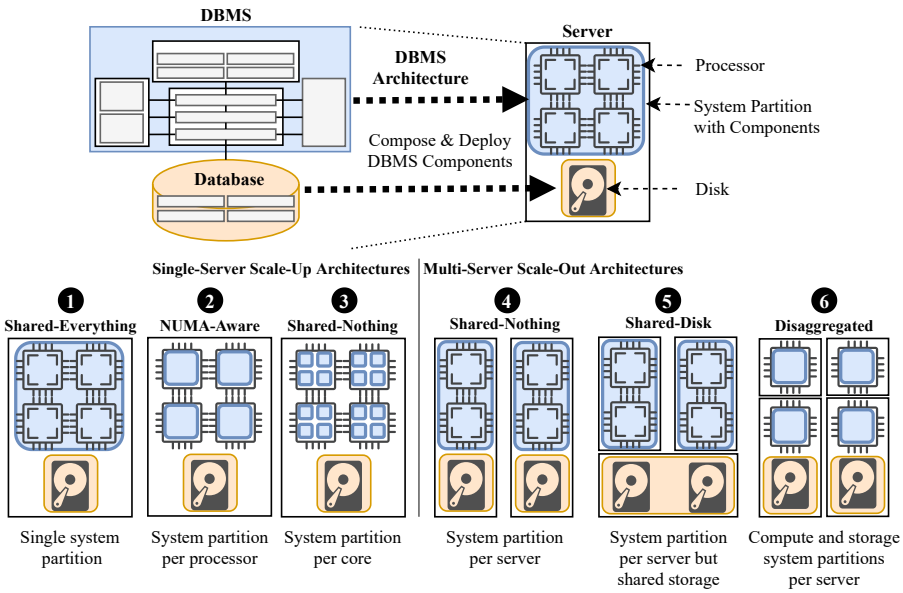


Figure 1.4: Common DBMS architectures for single-server scale-up DBMSs and multi-server scale-out DBMSs.

gres [183]. Instead, **(2)** the NUMA-aware architecture instantiates a system partition containing all DBMS components on each processor, e.g., implemented in SAP HANA [65] or Oracle TimesTen [112]. **(3)** The shared-nothing architecture for scale-up DBMSs instantiates the entire stack of DBMS components on every core on each processor in the hardware platform [172], today found in Teradata Database [182] as well as the main memory DBMSs VoltDB [103, 175]. Among scale-out DBMSs, **(4)** the shared-nothing architecture similarly individually instantiates the entire stack of DBMS components on every single server of the distributed DBMS, e.g., implemented in Postgres or Amazon Redshift [75] While **(5)** the shared-disk architecture also instantiates system partitions on distinct servers, it devises common storage of the entire database, e.g., implemented in Oracle RAC [37] or DB2 Data Sharing [102]. Lately, the shared-disk architecture has been advanced to **(6)** the disaggregated architecture separating storage related DBMS components into distinct system partitions, e.g., implemented in the cloud-native DBMSs Amazon Aurora [194], Azure SQL Database Hyperscale [8, 131], and Alibaba PolarDB [35].

In detail, the above DBMS architectures are all used for a specific purpose. Modern DBMSs implement these architectures, i.e., their distinct instantiation of components and assignment of resources, to specialize for distinct workloads and hardware. While these DBMS architectures broadly target either single-server hardware or multi-server hardware, these are further specialized for distinct workload and hardware characteristics like non-uniformity or large main memory capacity. For example, the modern main memory DBMS engine Hekaton specifically employs the shared-everything architecture for high resource

utilization under non-uniform workloads [52]. Instead, the shared-nothing architecture is employed for partitionable workloads, e.g., for partitionable analytical workloads in Teradata Database [182], but also for partitionable transaction workloads in traditional DBMSs like Postgres [183, 184] or modern main memory DBMSs like Oracle TimesTen and VoltDB [103, 112, 175]. However, these DBMSs only perform well for either workload according to their implemented architecture, but severely degrade for other workloads [141]. Similarly, the NUMA-aware architecture today has significantly improved the performance of scale-up DBMSs like Oracle or SAP HANA, specializing them for non-uniform hardware with many processors [42, 66, 154]. Yet, this is a specific specialization to utilize hardware parallelism of many processors, but today the massive parallelism within a single processor raises similar issues, again degrading performance due to unfit specialization for hardware parallelism [206]. Finally, also the disaggregated architecture specializing cloud-native DBMSs like Amazon Aurora [4, 194] or Azure SQL Database Hyperscale [8, 131] for elastic cloud infrastructure comes at a cost, e.g., benefiting compute-intensive workloads like analytics but inhibiting performance of latency-sensitive transactional workloads [118].

Consequently, today's relational DBMSs only achieve high performance for a specific predetermined purpose, i.e., for the specific workload and hardware characteristics as well as the objective their implemented architecture specializes in. For other purposes, however, their architectures are unfit, severely degrading DBMS performance and hence generally restricting today's DBMSs to a specific predetermined purpose. As explained in the following, these statically implemented architectures have such strong impact on the DBMS design, that the sophisticated specialization of DBMS components has limited effect beyond this predetermined purpose.

1.3 STATIC ARCHITECTURES DESPITE EXPANDING DESIGN SPACE

The specialization of DBMS architectures described above reveals that today's DBMSs implement a specific architecture for a predetermined purpose, with severe consequences when hardware or workload deviate. Precisely, their specialized architectures aim for the best deployment of the DBMS components to operate specific workload on specific hardware, assigning resources to DBMS components in expectation of specific conditions of workload and hardware characteristics. For example, the previously described architectures commonly consider the workload and hardware characteristics depicted in Figure 1.2, e.g., partitionability, access pattern, read/write ratio or the interconnect latency of hardware topology. But, these architectures distinctly specialize for specific conditions of these characteristics, e.g., either for partitionable workloads or non-partitionable workloads. Importantly,

regardless the diverse data models (e.g., hierarchical, relational or graph), the diverse interaction modes (e.g., direct user interaction, batch processing or stored procedures) or the diverse workload classes (e.g., transactional or analytical), it is these common fundamental characteristics impacting all the diverse DBMS designs that architectures specialize for [52, 65, 103, 113, 141].

Hence, the underlying problem is the increasing diversity of conditions that these fundamental workload and hardware characteristics can assume in the expanding design space. Indeed, the previously explained agile DBMS consumption demands today's DBMSs to operate changing load-fluctuating workloads and causes their deployment to varying hardware even at runtime, e.g., for elastic resource scaling, such that these conditions are also increasingly volatile. These fundamental workload and hardware characteristics changing at deploy-time or even at runtime are a significant challenge for today's DBMS architectures. While adaptive DBMS components allow flexible specialization to such volatile conditions even after implementing the DBMS, today's specialization approach for DBMS architectures takes fundamental design decisions for specific conditions, dictating a static specialization not easily revised.

The specialization of the DBMS architecture is manually decided early in the design process and for a specific purpose, e.g., a specific objective like high transaction throughput for specific conditions like non-partitionable workload. Specifically, DBMS designers set out such specialization purpose of the architecture at the start of the design process and decide the architecture, then the DBMS is implemented based on this architecture, and finally the resulting DBMS is deployed for the initially targeted purpose. A simple example is Hekaton's target of non-partitionable transaction workload on main memory, which resulted in the implementation of the shared-everything architecture enabling all resources to operate all components and access all data. However, deploying Hekaton for a lot of main memory capacity on large non-uniform hardware, the performance significantly degrades, as our experiments will show. Instead, Hekaton would need to implement a NUMA-aware architecture. Moreover, while Amazon Aurora and Azure SQL Database Hyperscale undertook significant re-implementation of their ancestors (i.e., MySQL and Microsoft SQL Server), they again implement specific static disaggregated architectures [8, 194]. Notably, under distinct assumptions, for Amazon Aurora the disaggregation of compute and storage was decided, whereas for Azure SQL Database Hyperscale it was decided to additionally disaggregate the log component. Whichever choice turns out better or whenever the conditions change, these modern cloud DBMSs again need re-implementation for adjusting the disaggregation of their components. Accordingly, the choice of the architecture, i.e., the choice which components are operated together on what resources

in a system partition, today means that these components are also implemented together. Moreover, the architecture and especially the intended resource assignment determine implementation details like the process model in the DBMS or coordination and data sharing between components and system partitions [80].

In addition, this manual specialization not only predetermines a fixed purpose but also artificially simplifies DBMS architectures. That is, all the previously described architectures only assign coarse-grained and uniform stacks of components to coarse-grained resources, cf. Figure 1.4. For example, the NUMA-aware architecture assigns an entire stack of components to a processor. Thereby, this NUMA-aware architecture mitigates non-uniform hardware but indistinctly enforces a system partition per processor, demanding according data partitioning and costly cross-partition operation of the DBMS. However, for read-dominated workloads the cross-partition operation indeed is more expensive than direct operation across processors. Inversely, for write-dominated workloads finer-grained partitioning with fewer resources yields better performance, since coordination for direct operation is more expensive than cross-partition operation [141]. Moreover, these workload effects and the load differ across components and instances of components, e.g., between index instances storing popular and unpopular data items, as our experiments will show. That is, the coarse-grained composition and resource assignment in today's architectures inhibits the sensitive balancing of the effects of the fundamental workload and hardware characteristics. Therefore, the artificial simplification in current specialized architectures misses optimization potential and indeed causes unfit specialization.

In conclusion, the current specialization approach manually decides the DBMS architecture at design-time and implements the DBMS accordingly. At best, the resulting specialized architecture enables the DBMS to perform well for the very specific predetermined purpose, e.g., balancing the underlying effects of the distinct workload and hardware conditions. In the expanding design space, however, workload and hardware characteristics are increasingly volatile, such that there are increasing possibilities for static architectures to become unfit, degrading DBMS performance. Even re-implementing a better-fit architecture does not solve the issue, as any static specialization is bound to become unfit under volatile conditions, whether changing workloads, evolving hardware or even alternative optimization objectives like minimal cost. Indeed, the strong coupling of the architecture and the DBMS implementation today limits the benefit of DBMS components flexibly adapting to volatile conditions. Hence, static specialization of DBMS architectures is a significant issue and deciding the DBMS architecture early in the design process to be fixed in the DBMS implementation is fundamentally in conflict with the changing workload and hardware conditions.

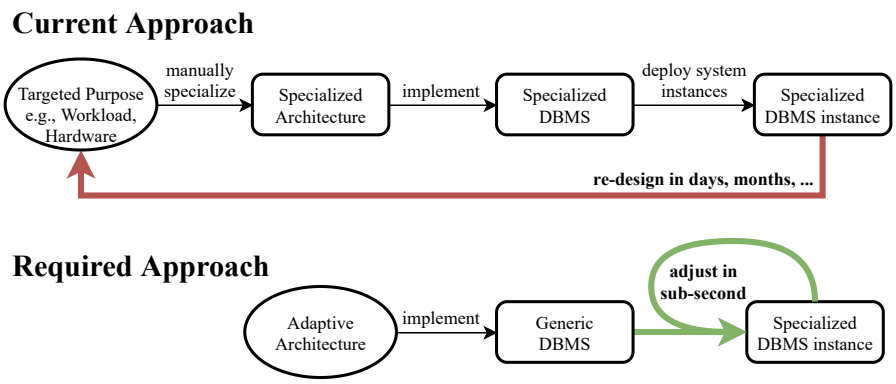


Figure 1.5: Overview of the current approach implementing a statically specialized architecture to be re-designed for changing conditions versus the new approach required for robust specialization allowing to adjust the architecture.

1.4 NEED FOR ROBUST SPECIALIZATION OF ARCHITECTURES

Statically specialized architectures make DBMSs susceptible to degradation under change, thus causing poor DBMS performance under the volatile conditions of the expanding design space. Instead, a new approach is required to enable high performance across volatile conditions, i.e., an approach for robust specialization of DBMS architectures. As outlined in Figure 1.5, the decision for a DBMS architecture must be deferred from the design-time to deploy-time or even the runtime. Hence, the goal is a generic DBMS implementation whose architecture can be adjusted as conditions change. By flexibly and effectively shaping the DBMS architecture, the goal is to robustly specialize for complex, volatile effects of changing workload and hardware conditions that DBMSs are exposed to. Precisely, for the best performance in specific conditions but overall robustness, this dissertation seeks robust specialization of DBMS architectures by deriving best-fit architectures for any given conditions and establishing the flexibility to change between these.

NEED FOR PRECISE SPECIALIZATION OF ARCHITECTURES The first facet required for robust specialization is to derive best-fit architectures for any given conditions. Today the complex effects of volatile workload and hardware conditions are not covered well, since the few predetermined architectures only specialize for a specific purpose, i.e., a specific objective as well as specific workload and hardware. Instead, comprehensive specialization is required allowing to derive any conceivable architecture. Rather than choosing from discrete predetermined architectures, the goal is to derive the best-fit architecture by more contiguously navigating all aspects of the architecture. Specifically, today's manual specialization simplifies DBMS architectures

by assigning only coarse-grained and uniform stacks of components to coarse-grained resource partitions, but the architecture must be precisely specialized for the distinct effects of the workload and hardware conditions on each component to reach best-fit architectures. For example, even the more sophisticated scale-up NUMA-aware or the disaggregated scale-out architecture still assign a stack of most of the components to an entire processor or an entire server, respectively. However, distinct components like the transaction manager and the query execution engine have distinct resource demands that depend on their individual exposure to the workload or hardware, like the partitionability of the workload. Hence, for best-fit architectures precise specialization for the distinct demands of individual components is required, including fine-grained resource assignment and independent partitioning of components. Additionally, to effectively find best-fit architectures despite the increased complexity, a principled method for specializing to any given and even unseen future conditions is required. Ultimately, optimizers should be able to automatically derive best-fit architectures.

NEED FOR FLEXIBLE ADJUSTMENT OF ARCHITECTURES The second facet for robust specialization is flexible and effective adjustment of the architecture, allowing to maintain best-fit architectures when conditions change. Today's fixed implementation of DBMS architectures hinders their adjustment as conditions change, thus resulting in unfit architectures. Instead, a flexible specialization approach is required to change the architecture of the DBMS without the need of re-implementation. Specifically, it must be possible to generically implement the DBMS but change all key aspects of the architecture without re-implementing the DBMS or its components. For this generic implementation, especially today's dependencies of the DBMS component implementations on the architecture must be resolved, allowing to implement the DBMS components without knowing the architecture. However, for effective adjustment of the architecture despite this flexibility, the new specialization approach also must be efficient in order to materialize the benefit. That is, the overhead must be sufficiently small, such that the performance of the flexibly derived architecture is on par when existing architectures are best-fit and indeed facilitates performance benefits over unfit architectures. Consequently, this dissertation strives for a generic DBMS implementation whose architecture can be flexibly and comprehensively adjusted, thereby achieving robust specialization throughout the volatile conditions of the expanding design space.

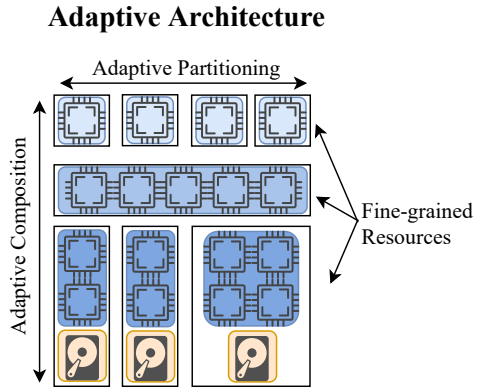


Figure 1.6: Sketch of adaptive architectures.

1.5 ADAPTIVE ARCHITECTURES FOR ROBUST DBMS

This dissertation proposes adaptive DBMS architectures to achieve the robust specialization of architectures and in turn of the overall DBMS. With adaptive architectures, concepts are proposed to depart from brittle static specialization, but instead achieve best-fit architectures regardless the conditions. For such robust specialization, we seek the best-fit yet flexible specialization of DBMS architectures throughout volatile conditions without the necessity of re-implementing the DBMS. This dissertation strives for a general solution to specializing adaptive architectures, enabling automatic optimizers to specialize DBMS architectures to any conditions, including unknown future workloads and hardware or even changing optimization objectives.

CONCEPTS FOR ROBUST SPECIALIZATION In detail, this dissertation proposes three concepts for the precise and flexible adaptation of DBMS architectures, as above identified necessary for robust specialization. The following concepts are proposed for the general robust specialization of both scale-up and scale-out DBMS architectures.

The first concept is the fine-granular assignment of arbitrary resources to arbitrary components. For example, as shown in Figure 1.6, a storage engine partition could be assigned two processors on a large scale-up hardware (rather than strictly one processor as in the NUMA-aware architecture), but also a single index could be assigned a few CPU cores. The purpose of this concept is to enable the precise specialization not only for coarse DBMS components but indeed for any piece of DBMS functionality that requires distinct specialization, i.e., precise specialization of fine-grained building blocks of the DBMS. That is, the concept proposes to assemble best-fit architectures based on fine-grained building blocks representing parts of DBMS components. To precisely specialize for their individual needs but also an overall balanced architecture, these can be independently partitioned across and composed into system partitions with individually suitable

amounts of resources. Thereby, this fine-grained assignment enables navigation to all conceivable architectures, allowing to derive the best-fit architecture for any DBMS in any conditions.

Based on this fine-grained assignment, the second concept is to separate components in the architecture as much as possible. With this concept we aim to separate building blocks (e.g., components) whose specialization competes, enabling competitive specialization of distinct aspects of the architecture. For example, for a scale-out DBMS serving compute-intensive read-only analytical queries and short-running write-mostly transactions, the ideal architectures be a shared-disk and a shared-nothing architecture, respectively. Here, our concept proposes to instantiate a hybrid architecture mimicking both ideal architectures providing the best performance for both kinds of requests rather than sub-optimally deciding for either of the architectures. Thereby, this separation strives to enable a best-fit architecture for all individual components.

Finally, the third concept is a programming model to enable effective adaptation of the architecture. We propose an asynchronous programming model to enable the generic implementation and efficient execution of any DBMS functionality as building blocks to specialize the architecture with.

CONTRIBUTIONS TOWARDS ADAPTIVE ARCHITECTURES This dissertation proposes the above concepts as general approach for adapting DBMS architectures. However, this dissertation distinctly realizes these concepts for the single-server setting of scale-up DBMSs and the distributed multi-server setting of scale-out DBMSs, contributing an adaptive for each of these DBMS classes. Moreover, for a rigid study despite the vast design space of DBMSs, this dissertation investigates adaptive architectures for the widespread relational DBMSs, the prevalent CPU-based hardware, and mainly transactional workloads, as will be explained in Chapter 2. Within this scope, this dissertation employs extensive performance evaluation of the DBMS with static architectures to guide the realization of adaptive architectures. Then adaptive architectures for scale-up and scale-out DBMSs are realized. The specific contributions of this dissertation are as follows:

1. *The first contribution* of this dissertation is a detailed evaluation of transaction processing DBMSs on a range of hardware and workloads, complementing prior evaluation works. This evaluation exhibits the complex interaction of the system design, the workload characteristics, and characteristics of the underlying hardware, which together determine the DBMS performance. It concludes the necessity of adaptive architectures and especially the necessity to precisely adapt for distinct DBMS components to successfully achieve robust specialization.

2. *The second contribution* of this dissertation is the realization of an adaptive architecture for single-server scale-up DBMSs. The adaptive scale-up architecture is realized as a programming model of asynchronous tasks, allowing a configuration of the architecture to flexibly compose individual data structures in heterogeneous system partitions. Based on this configurability, the benefit of adaptive architectures is demonstrated with an optimizer automatically specializing for robust transaction throughput. Hence, the value of robust specialization by adaptive architectures is proven for changing conditions and complex specialization objectives.

3. *The third contribution* of this dissertation is the realization of an adaptive architecture for multi-server scale-out DBMSs. The adaptive scale-out architecture is realized with a reactive programming model, allowing the fine-grained adaptation of execution and data flow at runtime. Based on this fine-grained online adaptation, aggressive optimizations are proposed to address the challenges of scale-out and especially cloud DBMSs, i.e., distinct architectures per query for best performance in mixed workloads. Thereby, adaptive architectures indicate significant opportunities for coming cloud DBMSs.

With the above contributions, this dissertation demonstrates the robust specialization of DBMS architectures within the specified scope. However, the contributions overall suggest that the proposed concepts and the realized adaptive architectures facilitate the robust specialization of DBMS architectures in general. As will be discussed throughout Chapters 4–6, our programming models, on one hand, generally permit the implementation of full-fledged DBMSs and all the DBMS components necessary to support various workloads beyond transaction processing. As will be explained by the accompanying integration approaches, DBMSs can generically implement DBMS components with various designs based on our programming models, which then can be flexibly operated in our adaptive architectures. On the other hand, already for the evaluated workload and hardware the distinct DBMS components expose distinct adaptation demands, such that our fine-grained resource assignment and the separate specialization of these components yield novel hybrid architectures with superior performance over the state-of-the-art architectures. Hence, our evaluation results demonstrate that the many architectures conceivable based on these concepts bare significant potential for optimizers to precisely specialize our adaptive architectures for various conditions. In combination, we are confident that our proposed concepts allow the flexible and precise specialization of our adaptive architectures to diverse DBMS components, workloads, and hardware, facilitating robust performance of full-fledged DBMSs in volatile conditions.

OUTLINE As initially depicted in Figure 1.7, the synopsis of this cumulative dissertation continues in Chapter 2, further detailing the scope and approach leading to above outlined contributions. Afterwards, we summarize the distinct contributions of this dissertation, backed by the peer-reviewed publications attached in Part II. Chapter 3 summarizes the evaluation of the robustness of current DBMSs; Chapter 4 summarizes the contributions towards adaptive architectures for scale-up DBMSs; and Chapter 5 summarizes the contributions towards adaptive architectures for scale-out DBMSs. Finally, Chapter 6 closes this synopsis with conclusions about adaptive architectures for robust DBMSs and an outlook on future research directions towards adaptive architectures for the cloud.

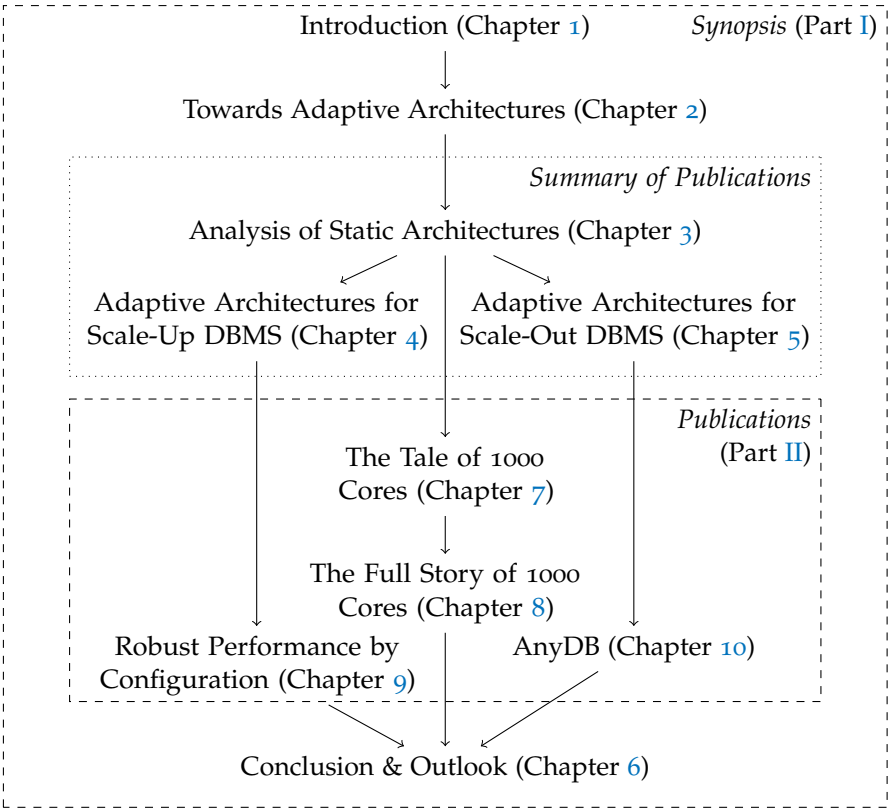


Figure 1.7: Outline of this dissertation.

2 TOWARDS ADAPTIVE ARCHITECTURES

The following chapter describes the approach of this dissertation towards adaptive architectures. It pursues the goal of broadly and flexibly specializing DBMS architectures to enable robust (and high) DBMS performance despite volatile conditions. Overall, the approach is structured according to three contributions previously outlined in Section 1.5, i.e., the extensive performance analysis detailing the limitations of current static architectures and the realizations of the adaptive scale-up and scale-out architectures. The following sections outline the design of the analysis and the distinct challenges and main ideas for the adaptive architectures. Notably, given the vast conditions to which DBMS architectures can be adapted, this dissertation focuses on adaptive architectures for particular classes of workload and hardware. Hence, the scope that makes the research on adaptive architectures tangible in this dissertation is first specified, before describing the specific approach.

SCOPE OF THIS DISSERTATION Taking the first steps from static specialization to adaptation of DBMS architectures, this dissertation focuses on the adaptivity of DBMS architectures, common hardware, and challenging workloads:

1. This dissertation focuses on making the architectures of scale-up and scale-out DBMSs adaptive. The scope includes resolving the undesirable re-implementation of scale-up and scale-out DBMSs when specializing their architectures, i.e., their generic implementation and the independent flexible specialization of their architectures. Departing from static scale-up and scale-out architectures, this dissertation approaches the adaptation of DBMS architectures separately for these two classes of architectures. Future work may integrate these orthogonal approaches for the combined adaptation within and across diverse hardware.

2. As our adaptive architectures aim to facilitate manual and automatic specialization, the scope further includes guidelines for deriving best-fit architectures and support for optimizers. To demonstrate performance benefits of best-fit specialization of our adaptive architectures to volatile conditions, we particularly seek initial use cases for optimizers utilizing the adaptive architectures for automatic specialization. Yet, the development of comprehensive optimizers on top of adaptive architectures is out of scope and left for future work.

3. The scope includes only the predominant general-purpose CPU-based hardware. We focus on the hardware platforms that today's DBMSs generally support and whose broad hardware characteristics currently challenge statically specialized architectures [141, 158]. Out of scope is special-purpose hardware, e.g., GPUs or FPGAs. Indeed, once initial research on special-purpose hardware gains adoption in DBMSs [3, 28], this hardware will open an entire new realm of hardware to specialize the architecture for. An extended implementation of adaptive architectures may incorporate this kind of hardware in future [5, 182].

4. This dissertation studies mainly diverse transactional workloads. These are an important workload class and already challenge DBMS architectures [62, 89, 176], as for example transactional workloads exhibit many short running transactions with varying access patterns of both read and write operations. Additionally, these workloads demand complex coordination across the DBMS to ensure the consistency of individual operations and the overall transaction. These workload characteristics make transactional workloads volatile and sensitive to hardware effects and adaptation overhead. We additionally consider long running compute-intensive analytical queries for the adaptation of scale-out DBMSs, as scale-out DBMSs are commonly used to flexibly provide the extensive compute resources for these queries.

With the above scope, this dissertation pursues the provably effective adaptation of DBMS architectures to a common but challenging subset of the extensive volatile conditions. Towards adaptive architectures within this scope, the following sections outline the specific approach leading to the distinct contributions. Specifically, Section 2.1 starts with the overall design of our initial performance analysis. Section 2.2 describes our high-level approach to adaptive architectures for scale-up DBMSs and Section 2.3 for scale-out DBMSs. These are detailed and their results are summarized in the subsequent chapters, i.e., in Chapter 3, 4, and 5, respectively.

2.1 ANALYZING STATIC DBMS ARCHITECTURES

First, we analyze the limitations of the DBMSs with static architectures with an extensive experimental evaluation. The goal is to identify the extent and reasons of workload and hardware characteristics impacting DBMS performance. While there exist numerous performance evaluations for scale-up and scale-out OLTP DBMSs [10, 56, 79, 100, 141, 142, 164–166, 178, 179, 204, 206], these primarily analyze workload effects and focus on specific hardware. However, OLTP DBMSs lack analysis for the broad spectrum of today's complex multi-processor hardware. Hence, this dissertation contributes a detailed analysis of

the effects of today's predominant complex hardware on transaction processing in current DBMSs. Thereby, this analysis complements the existing performance analyses, together guiding the subsequent realization of adaptive architectures.

We approach our experimental evaluation with scalability experiments guided by concurrent transaction execution as the key determinant of the performance of modern main memory OLTP DBMS. We setup these scalability experiments based on a representative prototype DBMS [206], three common but distinct multi-processor hardware platforms, and the standardized OLTP benchmark TPC-C [185]. Specifically, this prototype DBMS represents OLTP DBMSs through the transaction manager component essential for concurrent transaction execution, including a broad range of alternative designs commonly found in today's OLTP DBMSs [65, 112, 161, 183]. Prior work originally developed this prototype to analyze hardware effects of anticipated many-core processors with up to 1000 cores [206]. Instead, we study the effects of today's predominant multi-processor hardware, which challenges OLTP DBMSs not only with high hardware parallelism but also with many further and diverse hardware characteristics [83, 112]. Specifically, we study the hardware effects of three common x86- and PowerISA-based multi-processor platforms, whose distinct processors especially differ in the designs for parallel execution (i.e., physical cores and Simultaneous Multithreading), and the topologies notably differ in latency and bandwidth between the processors. In conjunction with this hardware setup, we use the TPC-C benchmark to study diverse workload characteristics. For the analysis of distinct workload characteristics, we devise common types of OLTP workloads but also manipulate distinct aspects of the benchmark. Especially, we use the common parameter of "warehouses" to devise high and low conflict workloads significantly influencing transaction execution by the necessary coordination of concurrent conflicting transactions.

In our initial series of experiments, we first analyze how today's multi-processor hardware impacts transaction processing performance compared to the anticipated many-core hardware of prior work [206]. After an initial overview of the performance for the canonical high and low conflict workloads, we analyze the reasons of the strongest discrepancies. Thereby, we detail the impact of the distinct characteristics of the two hardware types but also verify the comparability of the prototype with today's DBMSs.

In a second series of experiments, we then analyze transaction processing across today's hardware spectrum. We first conduct experiments to overview the performance across all our three hardware platforms, again using the high and low conflict workloads. Afterwards, the subsequent experiments, on one hand, detail the effects of the most impactful and distinct aspects of today's hardware platforms, i.e., distinct hardware parallelism within the processor and distinct

characteristics of the topologies connecting processors. On the other hand, the effects of further workload characteristics on these platforms are detailed. Specifically, we extend from analyzing the impact of conflicts in the workload to further significant workload characteristics like the type and number of operations of transactions.

The key finding is that transaction processing performance of today's OLTP DBMSs is subtly impacted by many hardware and workload characteristics. Our analysis exposes varying effects of today's broad hardware on the distinct transaction manager designs and exposes further cross-effects with the workload. For example, we observed nuanced effects of the distinct bandwidth and latency characteristics of the hardware topologies in combination with the footprint of the workload (i.e., number of records accessed by a transaction) on the distinct transaction manager designs. Through numerous such observations our analysis indicates a fragile balance of bottlenecks in the DBMS determined by the complex interaction of the underlying hardware, the specific workload, and indeed the design of DBMS components. Hence, DBMS architectures (as well as the components) must be adapted to stay balanced despite volatile conditions. Indeed, given the complex interacting cross-effects, adaptive architectures must enable optimizers to specialize the DBMS flexibly and effectively (i.e., robustly) under volatile conditions. A summary of our analysis is presented in Chapter 3 and the according publications in Chapters 7-8.

2.2 TOWARDS ADAPTIVE ARCHITECTURES FOR SCALE-UP DBMS

As a second contribution, this dissertation approaches an adaptive architecture for scale-up DBMSs. In contrast to scale-out DBMSs, this class of scale-up DBMSs operates on a single server. Constrained to a single server, scale-up DBMSs hence must make best use of the limited resources but can easily access all data in shared memory. Accordingly, the distinct purpose of scale-up architectures is to organize the DBMS components in a manner that best utilizes the limited resources pool, especially with resource partitioning. In the following, we detail the shortcomings of current static architectures and propose an adaptive scale-up architecture addressing these.

Today's scale-up architectures organize the DBMS by partitioning the resource across system partitions, as previously depicted in Figure 1.4 (1)-(3). Among other factors, these architectures primarily attempt a partitioning that balances partition-local coordination cost and cross-partition coordination cost for their individually targeted workload and hardware, i.e., for the predetermined purpose of these statically specialized architectures. For example, the NUMA-aware architecture attempts to balance these coordination costs by partitioning per processor for partition-local coordination within a processor and cross-partition coordination across processors.

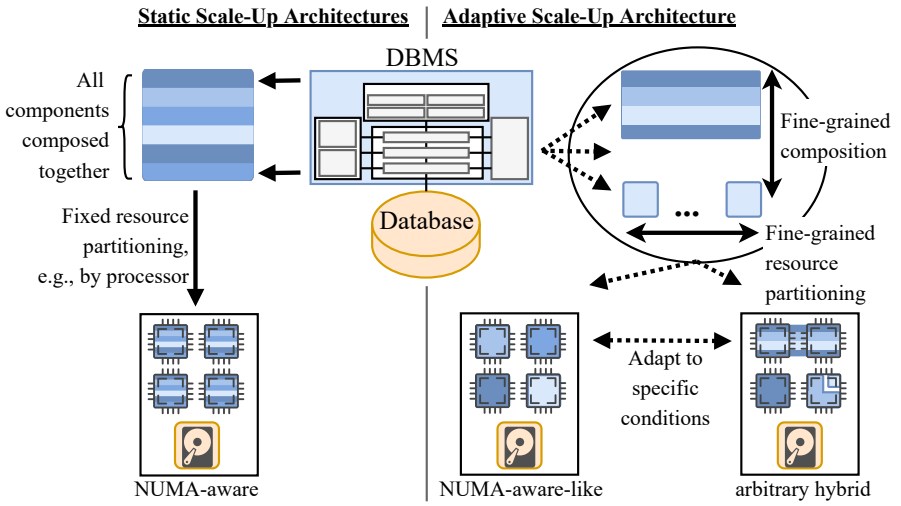


Figure 2.1: Approach of specializing static scale-up architectures (left) versus proposed fine-grained adaptation (right). **Left:** Current architectures statically specialize for all DBMS components together, e.g., the NUMA-aware architecture dictates the resources partitioning by processor for its system partitions which each contain the complete stack of components. **Right:** The proposed adaptive scale-up architecture enables fine-grained arbitrary resource partitioning for arbitrary compositions of DBMS components, facilitating flexible architectures with arbitrary system partitions.

However, in the volatile conditions of the expanding design space these architectures are not only unfit due to their static specialization but indeed due to their coarse-grained specialization, as illustrated on the left of Figure 2.1. Especially the coarse-grained resource partitioning for the entire component stack as a whole does not adequately account for the complex effects on core data structures. Workload and hardware have complex effects on the coordination cost within concurrent data structures e.g., the read/write ratio impacts the amount of coordination or the latency of the memory hierarchy impacts the cost of coordination. Moreover, the coordination cost further differs between types of data structures and indeed between instances of the same data structures when workload is imbalanced [49].

Instead, we propose adaptive architectures organizing scale-up DBMSs via fine-grained, arbitrary partitioning and additionally fine-grained, arbitrary composition, as illustrated on the right of Figure 2.1. Thereby, we aim to adapt scale-up architectures precisely and flexibly to any workload and any hardware. The idea for the adaptive scale-up architecture is to allow the partitioning of the limited resources into system partitions that ideally suit the distinct demands of individual data structure instances from across DBMS components, forming a hybrid architecture best fit for the given data structures instances under the given conditions. For example, for the index data structures of the storage manager such a hybrid architecture could incorporate

a shared-nothing-like architecture for heavily updated indexes and at the same time a NUMA-aware-like architecture for read-mostly indexes.

Accordingly, the key to robust specialization with the adaptive architecture is to choose suitable abstractions for the generic implementation of the DBMS and for the independent, flexible specialization of the architecture at runtime. Our approach is to decompose the DBMS implementation into tiny building blocks centered around operations on concurrent data structures and to enable optimizers to declaratively assemble these into arbitrary architectures at runtime. We hence propose a programming model based on asynchronous data-aware tasks providing the abstraction to implement arbitrary DBMS functionality. These asynchronous data-aware tasks enable the adaptation of this functionality without re-implementation and aid the efficient, controlled execution. In addition, we propose a configuration policy as the abstraction for an optimizer to declare architectures with arbitrary fine-grained system partitions. As will be discussed in Chapter 4, we support these two core concepts with a principled adaptation procedure implemented in an initial optimizer and an efficient runtime executing the data-aware tasks for a given configuration policy.

As our key finding, the results of our adaptive architecture for scale-up DBMSs prove a significant benefit. Across the spectrum of workload and hardware in which we have evaluated our approach against the state-of-the-art statically specialized architectures, our adaptive architecture typically performs better and never worse. Indeed, the superior performance stems from the optimizer declaring superior hybrid architectures based on the introduced fine-grained specialization. Hence, fine-grained and automatic adaptation indicate high potential for robust specialization of scale-up DBMSs to volatile conditions. More details are presented in the summary in Chapter 4 and the according publication in Chapter 9.

2.3 TOWARDS ADAPTIVE ARCHITECTURES FOR SCALE-OUT DBMS

To complement the adaptation of scale-up architectures, this dissertation also strives for the adaptation of scale-out architectures for the second class of multi-server scale-out DBMSs. Especially in the cloud, scale-out DBMSs are used for their ability to operate across a variable number of servers, allowing them to flexibly adjust compute and storage resources to match the load fluctuation of cloud workloads. Accordingly, scale-out architectures have the primary purpose of efficiently orchestrating the DBMS across scattered resources of a runtime-variable (elastic), network-connected resource pool. In the following, we describe why static architectures fail to orchestrate scale-out DBMSs for diverse workloads. We then outline our approach to ideally orchestrate scale-out DBMSs for highly diverse and even mixed

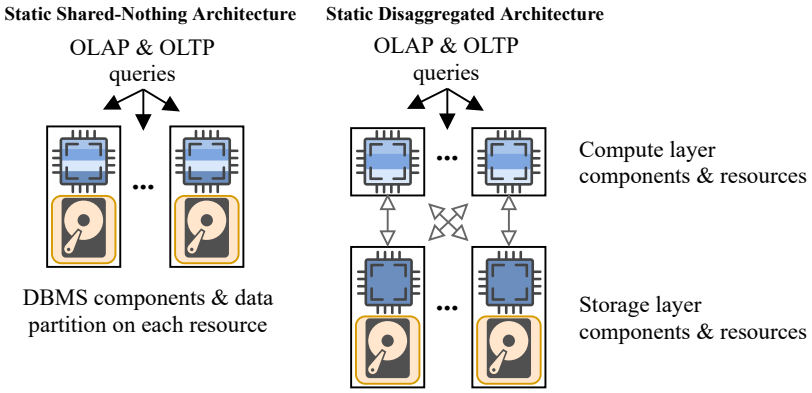


Figure 2.2: Static scale-out architectures orchestrate DBMSs by statically composing system partitions for their targeted workload and deploying these on the scattered elastic resource pool. The static shared-nothing architecture composes all DBMS components and a data partition, e.g., benefiting OLTP queries with fast local data access. The static disaggregated architecture composes distinct compute and storage layers, e.g., benefiting OLAP queries with vast compute resources.

cloud workloads with our novel adaptive scale-out architecture. In contrast to the adaptive scale-up architecture, it is concerned with the efficient distributed execution of the DBMS across network-connected resources and exploits the resource elasticity.

Today's scale-out architectures statically compose system partitions with DBMS components to be operated together. They then attempt deploying these static system partitions in a balanced manner. In particular, these static architectures compose the DBMS components in the attempt to balance the load of the resources versus network communication between those. For example, as shown in Figure 2.2, the shared-nothing architecture co-locates DBMS components and data in system partitions on each server, achieving even resource load and minimal network communication for uniform partitionable workload. But under today's volatile workloads this architecture is prone to resource imbalance and to high data re-partitioning cost. In contrast, the disaggregated architecture composes the DBMS into distinct compute and storage layers, such that it better balances non-uniform workloads and does not demand data partitioning. Yet, it entails cost for frequently transferring data between these layers across the network, especially harming latency-sensitive transaction workloads. Moreover, both these architectures disregard the distinct demands of distinct DBMS components. For example, the query executor would benefit from executing compute-intensive analytical (OLAP) queries with many resources across many system partitions, while the transaction manager is very sensitive to coordination of OLTP queries across system partitions. Consequently, none of the statically specialized

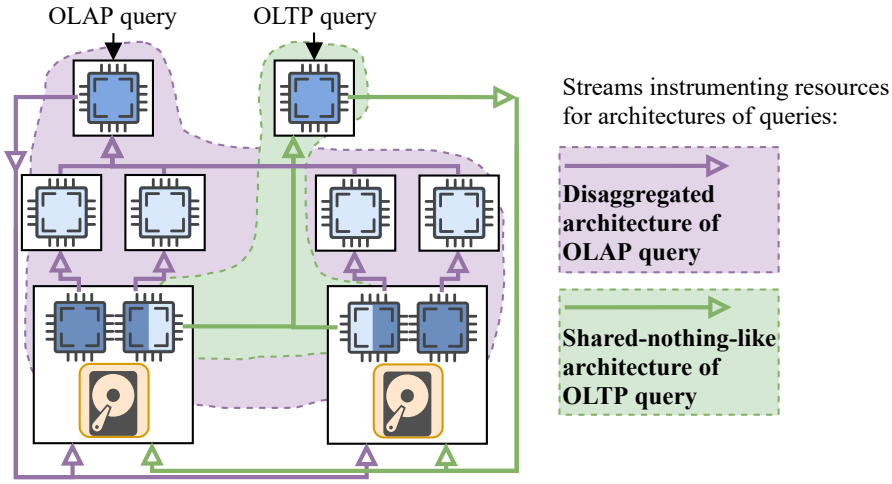


Figure 2.3: *Architecture-less* adaptation proposed for simultaneous specialization of scale-out DBMSs to distinct queries. The adaptive architecture instruments the resources to act as DBMS components via event and data streams, orchestrating distinct architectures for the concurrent OLTP and OLAP queries. The purple streams orchestrate a disaggregated architecture for the OLAP query, while the green streams simultaneously orchestrate a shared-nothing architecture for the OLTP query.

scale-out architectures ideally suit today's breadth of workloads nor the distinct demands of the DBMS components.

Instead, scale-out architectures require fine-grained adaptation to best specialize for the opposing demand of DBMS components. Similar to our adaptive scale-up architecture, the adaptive scale-out architecture must enable fine-grained arbitrary composition and partitioning of DBMS components. However, facing cloud workloads which not only vary over time but also comprise a variable mix of OLTP and OLAP queries, indeed demands opposing specialization for concurrent queries in the DBMS, e.g., as for Hybrid Transactional/Analytical Processing workload (HTAP) [11, 115, 126, 147].

For the adaptation of scale-out architectures, we hence propose a radical new approach called *architecture-less DBMSs*. We propose to derive individually specialized architectures as queries arrive at the scale-out DBMS and to simultaneously orchestrate these architectures within the same DBMS. The core idea of the architecture-less DBMS is to swiftly enact architectures on elastic resources. Instead of the predetermined system partitions of the static architectures, we propose the flexible instrumentation of elastic resources so that these temporarily act as (part of) different DBMS components and contribute to many individually specialized architectures for distinct queries. For example, instrumenting shared and exclusive elastic resources as shown in Figure 2.3 allows us to simultaneously orchestrate a disaggregated architecture for an OLAP query and a shared-nothing-like architec-

ture for an OLTP query. Including fine-grained arbitrary composition and partitioning of the architecture, we thereby approach simultaneous adaptation to hybrid architectures better specializing for the demands of distinct DBMS components under the specific workload characteristics of individual queries.

At the core of the adaptive scale-out architecture, we propose an asynchronous reactive model for the distributed execution of scale-out DBMSs, enabling the flexible and efficient orchestration of multiple architectures across elastic resources. This execution model employs a single generic component on each resource (e.g., server) which reacts to an event and a data stream. Essentially, these event and data streams instrument the generic components on elastic resources to act as (a part of) a DBMS component for a moment and together form an architecture. Specifically, this execution model decomposes DBMS components into their distinct operations and delivers all their required input state, e.g., for the query executor operations are query operators consuming records and for the query optimizer operations are SQL statements (queries) consuming catalog statistics. With the event stream encoding these operations and the data stream delivering the input state, the generic components asynchronously react to incoming events and data, allowing to execute any part of a DBMS component anywhere on some elastic resource. Consequently, the key aspect of this execution model is that architectures can be individually specialized simply by dispatching events/data streams per query and these architectures can be flexibly orchestrated across elastic resources by routing these streams.

The key finding is that the proposed architecture-less concept is promising to adapt scale-out DBMSs to volatile mixed workloads in the cloud, simultaneously orchestrating hybrid architectures on elastic resources. Having proposed a new radical approach to adaptive scale-out architectures, we focused on exploring the opportunities for the distinct demands of queries from different workload classes, e.g., distinct parallel execution per transaction. Indeed, broad degrees of freedom in adapting the overall architecture and the execution strategy at the micro-level appear beneficial for specializing architectures for distinct queries. The additional simultaneous orchestration of such specialized architectures overall indicates promising adaptation for scale-out cloud DBMSs. More details on adaptive scale-out architectures are provided in the summary in Chapter 5 and the according publication in Chapter 10.

3 ANALYSIS OF STATIC ARCHITECTURES

In this chapter, we summarize the first step of this dissertation towards adaptive architectures for robust DBMSs, a detailed performance evaluation. As previously described in Section 2.1, the goal of the following evaluation is to contribute to a sound foundation for robust specialization of DBMSs overall and for realizing adaptive architectures in particular. This evaluation analyses transaction processing performance of statically specialized OLTP DBMSs with static architectures on a broad spectrum of complex multi-processor hardware. Thereby, it complements existing evaluations [10, 56, 79, 100, 141, 142, 164–166, 178, 179, 204, 206] with detailed insight on the effects of the characteristics of today's predominant hardware and the combined cross-effects with diverse workload characteristics.

In the following, in Section 3.1 we first reference the two publications constituting the evaluation and describe the contributions of the author of this dissertation to these publications. Then, we summarize the design of the evaluation as a whole in Section 3.2 and summarize the resulting findings in Section 3.3. Finally, in Section 3.4 we discuss implications for robust specialization of DBMSs, especially for adaptive architectures. For the full details of the evaluation, we refer to the according publications attached in Chapters 7-8.

3.1 PUBLICATIONS

PUBLICATIONS This evaluation work is published in two peer-reviewed publications. The initial evaluation is published as *The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware* in the proceedings of the *International Workshop on Data Management on New Hardware (DAMON'20)* [17], cf. Chapter 7.

Following the invitation as one of the best papers of DAMON'20, we have contributed a significantly extend evaluation in the work *The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware*, that is published in the special issue for *Best Papers DaMoN 2020* in *The International Journal on Very Large Data Bases (The VLDB Journal)* [21], cf. Chapter 8.

Additionally, the supplementary material of the evaluation is published. The detailed data set [19] and the source code [20] are publicly accessible for reproducibility and future research.

CONTRIBUTIONS OF AUTHORS The contributions to the two above publications by Tiemo Bang, the author of this dissertation, are as follows. For the initial evaluation work [17], Tiemo Bang is the leading author. He is responsible for the design and execution of all the experiments as well as the analysis of the experimental results constituting this evaluation work. Tiemo Bang is also the main contributor of the manuscript. The co-authors Norman May, Ilia Petrov, and Carsten Binnig have contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

Similarly, for the second extended evaluation work [21], Tiemo Bang is the lead author, responsible for design, execution, and analysis of the evaluation. He also is the main contributor of the manuscript. Also for this publication, the co-authors have contributed invaluable feedback and agree with the use of the publication for this dissertation.

3.2 DESIGN OF THE EVALUATION

In the following, we summarize the essential aspects of our evaluation design and subsequently provide further details. We use prior evaluation work as a starting point [206], which developed the prototype DBMS DBx1000 representing modern main memory OLTP scale-up DBMSs. Since the transaction manager significantly determines the performance of OLTP DBMSs, this prototype in particular features a range of designs of this DBMS component crucial for transaction processing, i.e., a set of concurrency control schemes. Notably, these concurrency control schemes diversely expose the transaction manager to workload and hardware characteristics, hence well representing the varying exposure of DBMS components to these characteristics which we seek to address with adaptive architectures. While this prior work evaluates hypothetical many-core hardware and other works focus on workload effects, we extensively evaluate the effects of today's diverse multi-processor hardware on transaction processing workloads.

Based on this prototype, we conduct the evaluation in two parts. In the first part, we analyze how today's multi-processor hardware influences the DBMS performance versus the simulation of the hypothetical many-core hardware, revisiting that prior work with real hardware [206]. Notably, as one result we identify shortcomings of the prototype DBMS compared to today's state of the art, such that we apply optimizations common in today's DBMSs before conducting further analyses.

In the second part, we conduct detailed analyses on a wide spectrum of hardware configurations, i.e., different platform types and especially different platform sizes (1 to over 1000 CPU cores). Accordingly, the detailed analyses in this evaluation are mainly based on scalability experiments for a range of platform sizes, in which we control for distinct workload as well as hardware characteristics and

apply performance analysis tools. This allows us to study the distinct effects of these characteristics as well as their complex cross-effects on the alternative transaction manager designs, as follows.

3.2.1 *Setup of the Evaluation*

SYSTEM UNDER TEST For the detailed analyses of workload and hardware effects on today's DBMSs, our version of the prototype DBMS DBx1000 implements an extended set of designs of the transaction manager DBMS component [20]. Table 3.1 displays the concurrency control schemes of the evaluated transaction manager designs, where SILO [192] and TICTOC [208] are the additional schemes.

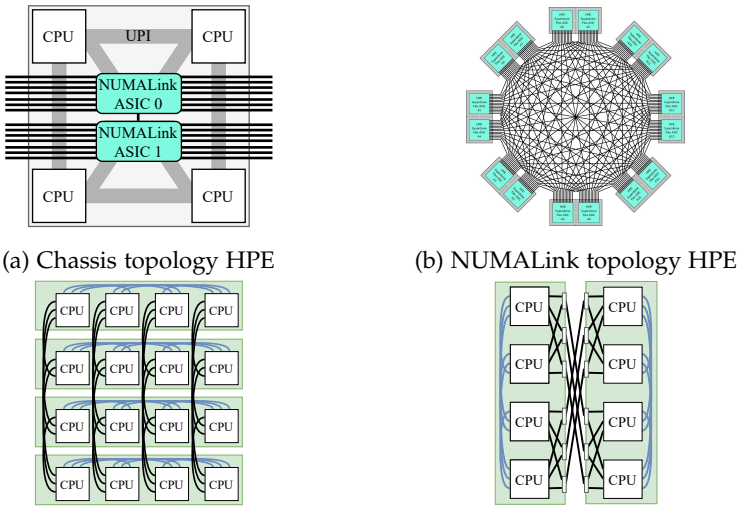
DL DETECT	2PL with deadlock detection [24]
NO WAIT	2PL with non-waiting deadlock prevention [24]
WAIT DIE	2PL with wait-and-die deadlock prevention [24]
MVCC	Multi-version T/O [25]
OCC	Optimistic concurrency control [111]
HSTORE	T/O with partition-level locking [103]
SILO	Epoch-based T/O [192]
TICTOC	Data-driven T/O [208]

Table 3.1: Two-phase locking (2PL) and timestamp ordering (T/O) concurrency control schemes of the evaluated transaction manager designs implemented in our OLTP DBMS prototype DBx1000.

For our scalability experiments, an important implementation detail of this prototype is that each transaction executor fully runs a single transaction from start to end (i.e., begin to commit) and each transaction executor is exclusively and permanently allocated to a logical core (i.e., hardware thread) of the platform. This corresponds to the inter-transaction parallel execution scheme commonly found in today's DBMSs [8, 65, 103, 106, 112, 183]. Also, it implies that the number of concurrent transactions equals the platform size in our experiments.

DBx1000 additionally implements detailed instrumentation measuring where in the DBMS time is spent for executing transactions. We use this instrumentation for time breakdowns, as an analysis tool providing detailed insight in the behavior of the DBMS.

BENCHMARK FOR TRANSACTIONAL WORKLOADS For analyzing transactional workloads, DBx1000 implements the standardized OLTP benchmark TPC-C [185]. With this benchmark we analyze the effects of distinct workload characteristics, using its standardized parameters but also manipulating aspects outside its standard.



(a) Chassis topology HPE

(b) NUMALink topology HPE

(c) 16-processor IBM Power9

(d) 8-processor IBM Power8

Figure 3.1: System topologies of the HPE, Power9, and Power8 platforms [82, 84, 168, 195]. The lines show the interconnects of the hardware platforms forming distinct topologies to connect all their processors. Each platform distinctly connects its processors within a hardware chassis as shown in (a) and the blue interconnects in (c) and (d). Also, the hardware chassis are distinctly connected as shown in (b) and the black interconnects in (c) and (d).

Most importantly, we use the parameter of the number of warehouses to devise high and low conflict workloads. This warehouse parameter determines the amount of data in the database, such that few warehouses likely cause concurrent transactions to access the same data and hence demand the coordination of these conflicting transaction by the transaction manager. Especially when the number of concurrent transactions exceeds the number of warehouses, this conflict potential is high and thus the load on the transaction manager, as in our high conflict workload with four warehouses. Inversely, our low conflict workload uses at least as much warehouses as concurrent transactions, such that the conflict potential is low and hence the load is on other aspects of the DBMS like indexes.

To analyze the effect of further workload characteristics, we manipulate distinct aspects of the TPC-C benchmark. For example, throughout our experiments we manipulate the amount of data in the tables, the mix of transactions, and the operations or access pattern of transactions. While these manipulations are outside the standard of this benchmark, they allow us to broadly analyze effects of workload characteristics and we document these manipulations in the setup of the distinct experiments.

HARDWARE PLATFORMS To analyze the effect of hardware characteristics, we employ three multi-processor hardware platforms representing the prevalent hardware platforms in production today [83,

96, 97, 112]. Specifically, we employ a x86-based hardware platform from HPE with 28 Intel processors [83, 132] and 2 PowerISA-based hardware platforms from IBM with 8 Power8 and 16 Power9 processors [195, 196], all of which distinctly connect their processors and main memory to a single system, as displayed in Figure 3.1. On one hand, these processors represent distinct designs differing in crucial processor characteristics, especially hardware parallelism via distinctly designed physical cores and Simultaneous Multithreading (SMT). On the other hand, these distinct platforms especially represent diverse interconnect characteristics (i.e., bandwidth and latency) between the processors, as found in today's distinct platform types and platform sizes.

3.2.2 *Evaluation of Multi-Processor vs. Simulated Many-Core Hardware*

In the first part of the evaluation, we revisit the prior simulation of then anticipated large many-core hardware [206] with today's real hardware which indeed reaches the anticipated amount of compute resources (i.e., "over 1000 cores") but as multi-processor platform. We hence analyze how today's actual hardware conditions influence transaction processing performance compared to the anticipated conditions in the simulation [206]. Focusing on this comparison, we use only the x86-based hardware platform in this first part. We separately analyze transaction processing performance across today's production hardware in the second part.

Specifically, to gain an initial understanding how the real hardware influences transaction processing compared to the simulated many-core hardware, we observe performance of the transaction manager designs for the two significant high and low conflict TPC-C workloads. That is, we analyze how these designs generally perform by observing their transaction throughput, while the according time breakdowns give initial indications on the underlying reasons for this performance.

As second step, we detail the evaluation for the strongest discrepancies between the two hardware platforms to detail which hardware and workload characteristics cause diverging performance. For example, the availability of a synchronized clock in the processor enables the hardware-accelerated allocation of crucial timestamps, hence likely changing the exposure of the transaction manager designs to these characteristics. We distinctly analyze the effects of this aspect as well as further discrepancies by again observing transaction throughput and time breakdowns.

Indeed, we identify significant shortcomings of the original prototype implementation. We hence finally investigate state-of-the-art optimizations as last step in this first part of the evaluation. With our optimized prototype, we detail the effects of the individual optimizations and repeat the comparison to the prior simulation. Importantly,

this final step ensures a sound representation of today's main memory OLTP DBMSs and provides initial insights on complex workload and hardware effects, both supporting our subsequent detailed analyses.

3.2.3 *Detailed Evaluation of Hardware and Workload*

In the second part, we evaluate the effects of hardware and workload characteristics on today's OLTP DBMSs in detail. Importantly, we thus broaden the evaluation to all three described multi-processor platforms commonly used in production today, i.e., the HPE platform with Intel processors as well as the Power platforms with Power9 and Power8 processors.

We first establish an overview of the transaction processing performance across these three platforms, extending our initial insights from the first part. We then use this extended overview to guide our subsequent analysis of hardware and workload characteristics.

For the detailed analysis of hardware effects, we detail those hardware aspects for which the above overview indicates significant impact on transaction processing performance. Specifically, for main memory OLTP DBMSs on multi-processor hardware platforms, these are the aspects concerning the parallel execution of concurrent transactions (1) within a processor and (2) across a topology of many processors. For the first aspect of parallel execution within the processor, we detail the effect of different hardware designs combining physical CPU cores and different Simultaneous Multithreading (SMT) approaches. We again use the high and low conflict TPC-C workloads, as these workloads previously proved to have distinct cross-effects with hardware characteristics. For the second aspect, parallel execution across a topology of processors, we instead analyze an extreme scenario before a more realistic but also more complex scenario. That is, as the topology connecting the processors is tiered, its performance characteristics depend on which processors communicate with each other, i.e., the Non-Uniform Memory Access (NUMA) effect. Hence, we first isolate the basic effect of each tier (i.e., NUMA distance) before analyzing the complex NUMA effect of cross-processor communication imposed by the workload. Notably, for these detailed analyses we not only observe the transaction throughput and time breakdowns but importantly also measure distinct aspects of hardware performance (i.e., profile the memory hierarchy), allowing us to identify complex cross-effects.

For the detailed analysis of workload effects, we extend from the initial insights on the impact of workload characteristics, now analyzing how further workload characteristics affect transaction processing in conjunction with the canonical conflict characteristic. Specifically, we consider different transaction types with varying operation mixes (i.e., read, update, and insert operations) as well as access patterns. These are important workload characteristics by themselves, i.e., requiring

distinct coordination requirements or imposing distinct load on indexes, and influence further workload characteristics like transaction duration or the footprint of records accessed per transaction. With these transaction types we further analyze workload effects, especially cross-effects between workload characteristics and additional cross-effects with hardware characteristics using our three hardware platforms.

3.3 KEY FINDINGS

While the detailed experimental results of our above outlined evaluation are presented in Chapters 7 and 8, we provide our key findings about the hardware and workload effects on OLTP DBMSs in the following.

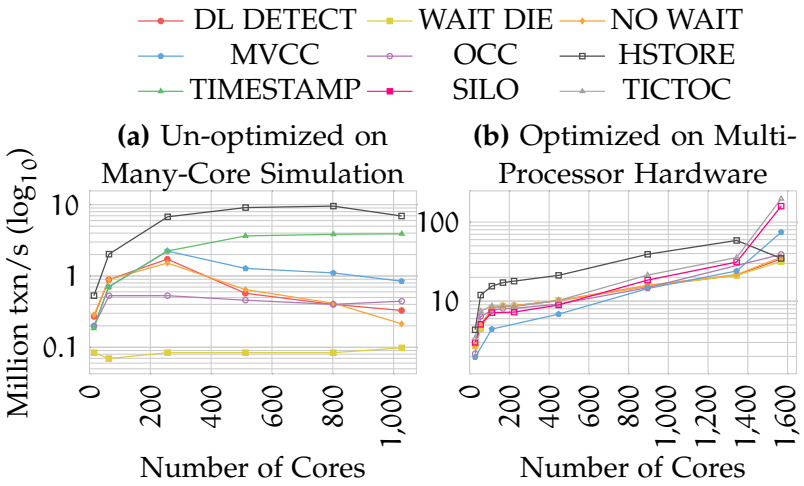


Figure 3.2: Transaction throughput for TPC-C low conflict workload of the transaction manager designs in **(a)** the original un-optimized OLTP DBMS prototype DBx1000 in the prior simulation [206] and **(b)** our optimized prototype on our real hardware Intel-based platform from HPE. State-of-the-art optimizations significantly improve performance and scalability, cf. Section 8.3.1 and 8.3.3.2.

MULTI-PROCESSOR VS. SIMULATED MANY-CORE HARDWARE In the first part, when revisiting OLTP on hardware with many cores, we identify several discrepancies between the prior simulation of anticipated many-core hardware and real multi-processor hardware. Most notably, the state-of-the-art optimizations establish remarkable transaction throughput close to 200 million transactions per second with 1568 cores when using all 28 processors of the HPE hardware platform, as shown in Figure 3.2. Importantly, these results contradict the prior simulation and shine new light on transaction processing on today's large hardware platforms. Transaction processing (i.e., all concurrency control schemes) indeed scales well beyond 1000

cores for low conflict workloads, when employing state-of-the-art optimizations. Notably, we observe that combinations of optimizations are necessary to improve performance, due to shifting bottlenecks. For example, hardware-accelerated timestamp allocation only improves performance with additional optimizations, as shifting contention then overwhelms the synchronization primitives, i.e., causes latch thrashing. Instead, under high conflict workload transaction processing still is overwhelmed by conflicts, degrading even more drastically in our real multi-processor hardware than in the simulation.

DETAILED HARDWARE AND WORKLOAD EFFECTS The results of the second part of our evaluation, in which we broadly analyze the impact of workload and hardware characteristics, present a breadth of detailed effects impacting the performance of OLTP DBMSs. Across the broad range of analyzed hardware and workload characteristics, we observe common outstanding effects and complex nuanced effects on transaction processing performance.

First, under workloads with many conflicts between transactions (i.e., high conflict workloads), we observe severe performance degradation for all concurrency control schemes. Regardless the hardware characteristics or other workload characteristics, the many conflicts only allow for the high transaction processing performance for very small platform sizes (number of cores) but prevent adequate utilization of all our large hardware platforms. A major cause of the severe performance degradation with increasing platform size is the simple inter-transaction parallel execution scheme, indeed commonly found in today's DBMSs [8, 65, 103, 106, 112, 183]. This common but simple scheme executes at least one transaction per processor core, such that it can only utilize high hardware parallelism (i.e., large platforms) with equally high transaction concurrency. Therefore, it necessarily amplifies contention in the DBMS when using higher hardware parallelism and thus increases transaction processing overhead to the extent that the overhead quickly prevails and performance degrades. Hence, conflicts in the workload have significant impact on the transaction processing performance of DBMSs and this simple, rigid execution scheme plays a significant role in that.

Second, we observe more nuanced effects from the interaction of transaction manager designs, the hardware, and the workload. Different hardware characteristics prove significant depending on the detailed design of the transaction manager, i.e., specific design details of the concurrency control scheme. For example, temporary copies of optimistic concurrency control schemes impact transaction processing performance depending on the cache capacity in the processor and the available interconnect bandwidth between processors, while locking of pessimistic concurrency control schemes proves sensitive to latency especially the increased latency of the interconnect between processors.

Importantly, we observe a negative effect of some hardware characteristics only when their capacity is exceeded, e.g., for cache capacity but also for the transmission across the interconnects between processors when reaching their bandwidth limit. That is, these capacity related effects do not appear as long as sufficient resources were available. Moreover, the workload further influences this interaction between the transaction processing performance and hardware characteristics. For example, the transaction footprint (accessed records) amplifies the cache demand, further degrading the performance of optimistic concurrency control when the cache is too small. However, at the same time, this transaction footprint also alleviates contention of synchronization primitives (i.e., latches), instead improving performance of pessimistic concurrency control schemes. Consequently, hardware and workload have complex effects on transaction processing and overall on performance bottlenecks in the DBMS.

CONCLUSION An agglomeration of performance bottlenecks in the DBMS determines the cost of transaction processing and overall DBMS performance. To reason about the performance of DBMSs, it thus is important to understand how the cost of individual bottlenecks scales (with workload and hardware characteristics) and how all the bottlenecks in the system interact. In this regard, this evaluation shows complex effects of workload and hardware as well as complex interaction of bottlenecks adding up, amplifying each other, but also dampening or hiding each other. Hence, to achieve best performance of an OLTP DBMS all its bottlenecks must be well understood and must be balanced for the specific hardware and workload at hand. Consequently, adapting the DBMS design to maintain the ideal balance as workloads change and hardware evolves is the path towards robust DBMS performance.

3.4 DISCUSSION

The above findings lead us to the following three major recommendations to achieve robust DBMS performance, concerning concretely robustness of transaction processing performance and more broadly two general approaches towards robust specialization of DBMSs.

COMPREHENSIVE CONTENTION MANAGEMENT As discussed in the above findings, we observe overall poor performance of OLTP DBMSs across the scalability experiments. Especially, all the transaction manager designs (i.e., concurrency control schemes) surrender to concurrent transaction execution eventually. The transaction manager designs degrade performance significantly under contention in concurrent transaction execution. Moreover, we find further significant performance degradation due to further DBMS components and distinct

implementation details of DBMS components, e.g., synchronization of index accesses. Depending on the workload and hardware characteristics, contention in concurrent transaction execution degrades performance in many aspects of the DBMS design.

We therefore recommend comprehensive system-wide contention management for robust transaction processing performance. First, there is the need to involve all DBMS components in contention management, since many factors across the DBMS affect contention and inversely contention affects many aspects across the DBMS design. For example, we particularly observe the inter-transaction parallel execution scheme to induce contention which the transaction manager cannot resolve. Additionally, there is the need for overall balancing of contention, since we also observe contention to shift between DBMS components, e.g., from latches of the transaction manager to latches of indexes in the storage manager. Lastly, there is the need to account for the complex interaction between the DBMS design, the workload, and the underlying hardware, as we discover complex cross-effects between these. These three requirements lead to our recommendation of system-wide contention management which should manage contention across the entire DBMS by leveraging the most effective options available throughout all the DBMS components.

Further, we recommend using adaptive concurrency control for the adaptation of the transaction manager and recommend integrating it with the proposed system-wide contention management. That is, since we recognize benefits of distinct concurrency control schemes, adaptive concurrency control as proposed by CormCC [180] would improve performance. However, we see the necessity to integrate it with the proposed system-wide contention management, since contention is a strong signal for choosing the concurrency control scheme, which interacts with the surrounding DBMS components.

ADVANCED PERFORMANCE MODELS Observing the complex interaction of many factors impacting DBMS performance, we argue for comprehensive (e.g., learned) performance models to support robust specialization. That is, performance models are an essential building block for automated specialization of DBMSs but will need comprehensive approaches beyond simple handcrafting to incorporate this complex interaction. For example, query optimizers employ models of query operators to decide the execution plan for a given query and workload [119]. However, traditional query optimizers using hand-crafted analytical models struggle to account for the complex interaction of many factors. Hence, comprehensive learned models are proposed as a better alternative to improve query optimization [85, 128, 211]. Indeed, such advanced models are proposed for many DBMS components [58, 86, 125, 163] and drive novel approaches for unprecedented automated specialization of DBMSs, e.g., automatic synthesis

of data structures and key-value DBMSs [39, 91]. These advances indicate great opportunities of comprehensive learned models to model overall DBMS performance under the observed complex interaction between the DBMS design, workload characteristics, and hardware characteristics. Such models would greatly assist in adapting DBMS architectures to complex conditions and in the long term could lead to robust specialization of DBMSs with performance guarantees.

ADAPTIVE ARCHITECTURES Our findings confirm the need for adaptive DBMS architectures capable of re-balancing, i.e., the goal of this dissertation, and overall adaptive DBMS designs. The optimal DBMS performance requires the entire DBMS design to ideally balance bottlenecks. However, this balance differs between distinct workloads and hardware platforms. Moreover, this balances changes over time, due to workload volatility but also progress of hardware development and state-of-the-art DBMS design. Beyond the existing adaptation and synthesis strategies [39, 91, 93, 140, 180], we hence argue for flexible system-wide adaptation, which exceeds adaptation of individual components and opposes rigid instance optimization. Towards effective system-wide adaptation, performance prediction and adaptation overhead are significant challenges. As recommended above, performance predictions will benefit from advanced performance models, whereas we consider flexible DBMS architectures and execution models to drive efficiency. Specifically, we envision the decomposition of system designs into fine-grained building blocks which can be efficiently composed at runtime, e.g., through a generic programming model and an optimizer for DBMS architectures. This will enable adaptive DBMS architectures to broadly transform a DBMS, balancing bottlenecks across all components. Thereby, we envision adaptive DBMS architectures to successfully adjust to volatile workloads, evolving hardware, and even variable optimization objectives, robustly specializing DBMSs for any conditions.

4 ADAPTIVE ARCHITECTURES FOR SCALE-UP DBMS

In this chapter, we summarize the second step of this dissertation towards adaptive DBMS architectures, the realization of the adaptive architecture for scale-up DBMSs. While current static scale-up architectures are coarsely specialized for entire stacks of DBMS components, we argue for fine-grained adaptation of the scale-up architecture for individual DBMS components and indeed their distinct data structures. We hence pursue precise and flexible adaptation, that allows to ideally specialize the scale-up architecture to the distinct demands of distinct data structure instances even under volatile conditions. The approach is to realize fine-grained, arbitrary composition and partitioning for the adaptive scale-up architecture to flexibly and precisely organize the distinct data structure instances and resources of the scale-up DBMS. While we propose a programming model for the generic implementation of the DBMS independent of the architecture, at the center of our adaptive architecture is our re-configuration approach, which allows to simply declare any architecture ideally specialized to the conditions at hand.

In Section 4.1, we first describe the contributions of the author of this dissertation to the backing peer-reviewed publication. In Section 4.2, we detail the challenges for scale-up architectures to achieve robust performance and explain how the configuration of our adaptive architecture addresses these. With the system overview in Section 4.3, we then present the essential aspects to realize the adaptive architecture. Finally, Section 4.4 concludes with the key findings. The full details on the adaptive scale-up architecture are in the according publication in Chapter 9.

4.1 PUBLICATION

PUBLICATION The work on the adaptive architecture for scale-up DBMSs is published in the peer-reviewed publication *Robust Performance of Main Memory Data Structures by Configuration* in the *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)* [22], cf. Chapter 9.

CONTRIBUTIONS OF AUTHORS The contributions to the above publication by Tiemo Bang, the author of this dissertation, are as follows. Tiemo Bang is the leading author. He is responsible for the proposed approach to the adaptive scale-up architecture, the experimental evaluation, as well as the manuscript. The co-authors Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication in this dissertation.

4.2 ADAPTIVE SCALE-UP ARCHITECTURES BY RE-CONFIGURATION

Today's static scale-up architectures predetermine a fixed organization of DBMS components and resources in scale-up DBMSs, specializing for specific workloads and hardware. However, our performance analysis in the previous chapter showed that these static architectures cause unstable performance of scale-up DBMSs, cf. Section 3.4. While changing workloads and hardware clearly contradict the fixed organization of these static architectures, our analysis also confirmed that these architectures neglect the distinct effects of these volatile conditions on the individual data structures from across DBMS components. Crucially, the analysis unveiled complex effects of workload and hardware characteristics within and across DBMS components, e.g., performance bottlenecks differing between transaction manager designs or shifting from the transaction manager to distinct indexes of the storage manager.

We hence propose an adaptive scale-up architecture for the flexible organization of the scale-up DBMS at the granularity of distinct data structure instances, facilitating best-fit specialization of the architecture to volatile conditions. As will be argued next, we utilize search data structures as the core data structures of DBMSs for our explanation and evaluation of the adaptive scale-up architecture.

DATA STRUCTURES AT THE CORE OF DBMS PERFORMANCE The distinct effects of the volatile conditions on individual data structure instances causes imbalance in static architectures DBMS. Their static and coarse-grained specialization to specific workloads and hardware struggles to balance the volatile and diverse partition-local operation cost of the individual data structure instances in DBMS components versus the cross-partition coordination. The core data structures, that especially exhibit the challenges of volatile conditions and underline the demand for fine-grained flexible specialization of the architecture, are the search data structures.

Search data structures like trees and hash maps are central to many DBMS components and thus critical for the overall DBMS performance. Across the DBMS, search data structures enable the buffer manager to efficiently locate data pages loaded from storage into main memory, organize memory allocation in the memory manager, and even constitute central aspects of operators for query execution like the hash-based join operator. Besides, search data structures play a central role for transaction processing, in particular in the transaction manager and the indexes of the storage manager. The transaction manager employs these data structures for the crucial coordination of concurrent transactions, e.g., to organize lock requests. And, the indexes of the storage manager are exactly these tree and hash map data structures. These indexes accelerate the access to distinct records, especially for the primary key accesses central to transaction processing.

Indeed, our performance analysis shows that the concurrent read and write operations on these data structures of the transaction manager and storage manager comprise a considerable fraction of the transaction execution cost. The reason is that transactions typically read, update, or insert only few records, but typically do not involve many compute-intensive operators like joins. As these operations occur concurrently, they require synchronization (e.g., via latches) which related work [10, 49, 106, 154] and our evaluation show to become significant performance bottlenecks. Accordingly, maintaining indexes or coordinating transactions with concurrent read or write operations on search data structures are central to transaction execution and the performance of OLTP DBMSs.

In the following, we hence focus on OLTP databases with workloads mainly characterized by differed mixes of read and write operations ranging from read-heavy to write-heavy mixes that are typically executed on these search data structures, especially on indexes such as modern versions of B-trees or hash maps. On the case of these B-tree and hash map search data structures, we discuss the many different causes for degraded performance of core data structures in main memory OLTP databases and why current static architectures address these unreliably. Afterwards, we elaborate on the key aspects of our approach enabling robust performance by (re-)configuration of our adaptive architecture. Notably, a more comprehensive system overview follows in the next section.

4.2.1 *Pitfalls of Static Architectures*

The sources of performance degradation of core data structures in main memory OLTP DBMSs are manifold. Our experiments in Section 9.7 as well as related work [106, 154] show that already these crucial search data structures can significantly degrade transaction processing performance in unfit static architectures, when exposed to the volatile read/write mixes of OLTP workloads on multi-processor hardware.

One primary source for performance degradation is the overly high contention resulting from too many resources accessing a data structure instance, e.g., executing too many concurrent read and write operations on an index [49, 63, 74]. Other sources of performance degradation include increased memory latency from cross-processor memory accesses or from cache coherence overhead of concurrent accesses to the same memory [106, 202]. Moreover, the impact of these significantly varies, not only between distinct hardware platforms (e.g., by distinct memory latencies) or workloads (e.g., by distinct read/write ratios of transactions). Indeed, within the same DBMS these sources of performance degradation distinctly appear in data structures with different designs. For example, our experiments show

that different index designs like a B-tree and a hash map distinctly react to the same operation mix.

Despite these manifold and volatile sources of performance degradation, today's DBMS architectures follow relatively simple and static strategies. A prevalent strategy is NUMA-aware partitioning for mitigating the impact of increased latencies caused by cross-processor operation and cache coherence as proposed by the NUMA-aware DBMS architecture [116, 139, 146]. However, this static and coarse-grained architecture can still lead to degraded performance since its partitioning at the granularity of a processor can cause too high contention for some data structures and workloads, like heavily updated indexes as shown in our experiments.

Instead, the static shared-nothing architecture partitions the resources by individual hardware threads entirely avoiding contention for the coordination on data structures, e.g., in H-Store [103]. However, this strategy has several other drawbacks like the sensitivity to load imbalance between system partitions under workload skew or the fact that complex workloads cause an increased coordination across partitions. For example, for mostly read indexes this fine-grained partitioning causes excessive cross-partition coordination.

Indeed, the static shared-everything architecture as in Hekaton entirely rejects partitioning to mitigate such negative effects of cross-partition coordination [52]. While this benefits those mostly read indexes for example, yet again other data structure instances like heavily updated indexes can severely degrade. Consequently, the manifold and volatile effects on individual data structure instances are not appropriately addressed, hence eventually degrading performance of all of today's static coarse-grained DBMS architectures.

4.2.2 *Robust Performance by Configuration*

While the partitioning strategies of current static architectures may have their sweet spots, they likely are unfit for individual data structures under volatile conditions. We thus propose a flexible and fine-grained approach to DBMS architectures and suggest framing the architecture as a flexible execution strategy allowing to declare the architecture at runtime. Essentially, given a mix of data structures and workload present in a concrete instance of a DBMS, the idea is to adapt any partitioning strategy ranging from shared-nothing to shared-everything for distinct data structures by simple configuration. We aim for a configuration policy flexibly declaring an architecture for a DBMS instance, using so-called *virtual domains* to partition resources for distinct data structures in an optimal manner.

Specifically, we devise these virtual domains as flexible system partitions to be configured with an arbitrary subset of resources and data structure instances. As shown in Figure 4.1, virtual domains

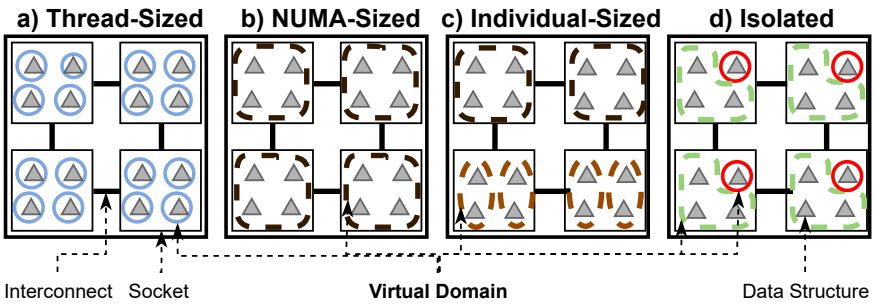


Figure 4.1: Flexible partitioning via configuration of virtual domains for scale-up architectures on a four-processor machine: **(a)** Thread-sized partitioning with virtual domains per CPU core like the shared-nothing architecture; **(b)** NUMA-sized partitioning with virtual domains per processor like the NUMA-aware architecture; **(c)** Individual-sized partitioning with two sizes of virtual domains as one novel hybrid architecture; **(d)** Isolated with separate virtual domains for hot data structures as another hybrid architecture.

thus provide many more configuration options, apart from flexibly mimicking any of the existing partitioning strategies from static architectures. First, when partitioning the resources of a machine into virtual domains, not all virtual domains need to have identical sizes in terms of processor or memory resources, but a configuration can declare virtual domains of different sizes ideally supporting a mix of different data structures and workload within a single architecture. Second, another configuration option provided by virtual domains is the isolation of hot data structures into separate virtual domains providing dedicated resources for more stable performance. These broad configuration options of virtual domains allow optimizers to broadly adapt architectures. By simple configuration, optimizers can declare novel scale-up architectures of many forms in a flexible and fine-grained manner.

At the moment, our approach applies new configurations by offline re-configuration, i.e., all active operations in the system must complete before the reconfiguration and then the DBMS can restart with a new configuration. This offline reconfiguration already enables significant adaptation of the architecture. Besides more precise specialization to the distinct demands of data structures for a given workload, the actual hardware characteristics can be more precisely considered. Rather than anticipating the hardware characteristics at design time, with our configuration approach the architecture can be adapted to the specific hardware when deploying the DBMS. For changing workloads, this offline reconfiguration can be triggered when workload changes are known a priori or by workload prediction for reoccurring workload patterns, e.g., for Black Friday. These techniques are well-known from automated table partitioning [30, 177]. Indeed, we plan to extend our approach to further support online reconfiguration similar to the fine-

grained and gradual techniques for table re-partitioning, and thus also support cases where the workload changes are less predictable.

4.3 SYSTEM OVERVIEW

In the following, we provide an overview of the main building blocks of our adaptive architecture for scale-up DBMSs. The two main building blocks that the DBMS needs to provide are asynchronous tasks and a configuration. DBMSs are to generically implement their components via asynchronous tasks, i.e., the access operations of DBMS components on their data structures, and are to flexibly declare their architecture via the configuration, assigning data structures to optimally sized system partitions (virtual domains). Additionally, the runtime system is an important building block, for efficiently executing the asynchronous tasks for a given configuration. Below, we outline the essential aspects of these main building blocks and discuss how to integrate our adaptive architecture into a full-fledged DBMS.

4.3.1 *Asynchronous Tasks & Configurations*

ASYNCHRONOUS TASKS An asynchronous task is a container for access operations of the DBMS components on their data structures, e.g., an insert or a lookup operation on a B-Tree of the storage manager. On one hand, this task serves as the abstraction for the generic implementation of DBMS functionality. Tasks establish the API to implement DBMS components as tiny building blocks, allowing the adaptive composition of the architecture without any re-implementation. On the other hand, these tasks are execution units, serving as the abstraction for the lightweight and effective execution of a configured architecture. In contrast to operating system threads, tasks in our approach not only are much more lightweight but also are *data-aware*, i.e., a task is only executed inside the virtual domain where the data structure resides. This notion of tasks allows our adaptive architecture to effectively control contention and locality of operations on distinct data structures by simple means of a configuration. Notably, the specific API of our task abstraction is described in Section 9.4.

CONFIGURATIONS In addition to tasks, the DBMS must specify a configuration to control contention and locality within its DBMS components, i.e., on the distinct data structures. The configuration of our adaptive architecture comprises two parts: (1) The first part of a configuration defines which virtual domains are being used to execute the DBMS on a given hardware. Here the important aspect is the definition of how many virtual domains are used and how resources are allocated to each. This allows heterogeneous resource partitioning with virtual domains having distinct amounts of resources in a granularity independent of the hardware topology. That is, virtual domains do not need to be defined uniformly or for example in processor-granularity. (2) The second part of a configuration defines how data

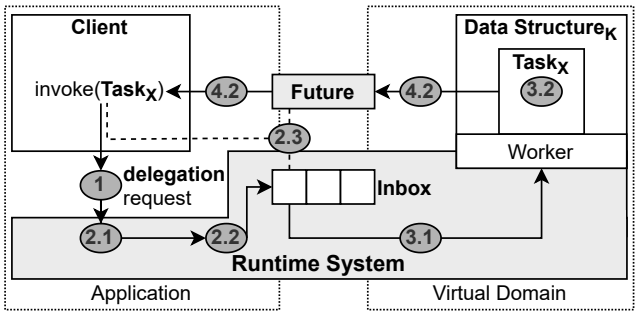


Figure 4.2: Overview of execution flow for delegating tasks under a configured architecture: Client threads delegate asynchronous tasks to workers in a virtual domain which reply using futures. With this execution flow the runtime flexibly executes a DBMS for configured architecture, first delegating the tasks to the virtual domain to which the accessed data structure is mapped and then executing this task with the resources assigned to that domain.

structure instances are mapped to virtual domains. Importantly, this allows the configuration to flexibly compose data structures from across DBMS components into virtual domains ideally suiting their individual demands under the given conditions.

We discuss the procedure for configuring an optimized architecture in Section 9.5 of the publication. There we describe our procedure that guides the configuration of the adaptive architecture for the hardware, workload, and set of data structures of a given DBMS instance. Also, we describe our initial optimizer, implementing this procedure as Integer Linear Program (ILP) to maximize overall system throughput when given these conditions as input.

Notably, a DBMS may split a data structure into several instances to achieve higher throughput. The necessary partitioning and replication strategies are commonly implemented in DBMSs for index or table data structures and can be simply applied on top of our adaptive architecture [15, 136]. Besides, with our adaptive architecture DBMSs in fact become less sensitive to the actual partitioning strategy being used. That is, our experimental evaluation shows that our adaptive architecture allows to better handle severe issues such as contention or lacking locality.

4.3.2 *Runtime System*

The main objective of our runtime system is the efficient execution of tasks given a configuration. For efficient task execution, the runtime system provides a delegation mechanism based on highly optimized in-memory message passing. Noticeably, the aim of the runtime system is not to provide a full-fledged DBMS but to act as a thin *virtualization layer* on top of the hardware providing the foundation for robust performance of a DBMS built on top. Below, we discuss the execution flow when delegating tasks for a give configuration.

Figure 4.2 presents an overview of our runtime system. A client thread of the DBMS submits an asynchronous task to be executed (step 1) and obtains an invocation handle, so-called future, on the submitted task (step 2.3) to consume the result of the task execution. Internally, the runtime system identifies the virtual domain responsible for the referenced data structure upon the invocation of an asynchronous task (step 2.1). It then places the task into the corresponding inbox (step 2.2) returning the future (step 2.3).

The counterpart to the DBMS's client threads are worker threads executed by the resources assigned to the virtual domain. These workers continuously poll the inbox for new tasks. Once a worker detects a new task (step 3.1), it executes this task (step 3.2) within the virtual domain on behalf of a client thread. Upon its completion, the task places its result in the earlier allocated future (step 4.1). The DBMS's client thread then eventually retrieves the result (step 4.2), potentially using it as input for further tasks on other data structures.

This execution flow of asynchronous tasks between client threads of the DBMS and worker threads within virtual domains enables the flexible and efficient execution of any configured architecture. For implementation details of our runtime system and especially the message passing we refer to Section 9.6 of the publication.

4.3.3 *Discussion of DBMS Integration*

As mentioned before, the main contribution is not to provide a full-fledged DBMS, but an adaptive architecture backed by a runtime system on top of which DBMSs can be implemented. We believe that our task-based adaptive architecture indeed can be used for implementing a DBMS. In fact, we show in our experimental evaluation that we are able to execute typical OLTP workloads by implementing a lightweight OLTP engine. In the following, we hence discuss the main design choices involved in building an OLTP engine utilizing our adaptive scale-up architecture.

A first design choice when using our adaptive architecture for OLTP is the mapping of transaction logic (i.e., the sequence of reads and writes) to tasks that can be executed by our runtime. A naïve way for this is to map every individual read/write operation of a transaction to a separate task to be submitted to our runtime system by the OLTP engine. Alternatively, our programming model also allows more sophisticated implementations where transactions are chopped into sub-transactions and then are mapped to tasks as a whole, e.g., as proposed in [64, 162]. While studying the detailed effects of chopping is an interesting research direction, the naïve mapping already proves sufficient for efficient OLTP execution in our experiments, see Section 9.7.3 of the publication.

A second design choice in addition to mapping transactions to tasks, is how tables of a database (and their indexes) are distributed across virtual domains. For this purpose, our configuration procedure takes any set of data structures as input including any tables or indexes of a database and compiles a configuration aiming to maximize the overall throughput for a given workload and hardware. Before applying this configuration procedure, the DBMS can still apply conventional partitioning strategies on tables as mentioned before and input these table partitions (as well as their partitioned indexes) as data structures to our configuration procedure.

In addition to these two main design aspects (i.e., mapping transactions to tasks as well as finding optimal configurations for a set of tables), further DBMS components need to be implemented, such as the transaction manager for concurrency control as well as recovery mechanisms. The design of those components, however, is orthogonal since many different schemes can be implemented in DBMSs on top of our adaptive architecture. Indeed, DBMSs should opt for adaptive designs of their components, e.g., adaptive concurrency control like CormCC [180], whose integration is an interesting research direction but beyond the scope of this dissertation.

For our evaluation, we hence omit these components for our lightweight OLTP engine as well as for all baselines (for a fair comparison), as further explained in Section 9.7 of the publication. In short, for concurrency control we rely on latches to avoid data races but do not prevent other anomalies like lost updates. While this allows no direct comparison with full-fledged DBMSs incorporating those components, it allows us to precisely analyze the effect of the distinct architectures. We leave out the DBMS components to focus on the partitioning and composition strategies of the static architectures versus our adaptive architecture. Thereby, we can compare the benefits of our execution scheme for OLTP workloads versus the classical OLTP engine designs.

Finally, an interesting future aspect when designing an OLTP engine on top of our adaptive architecture is that the asynchronous execution model opens up many new opportunities for optimizations. In a classical design of a main memory OLTP engine, typically a transaction executor sequentially executes the operations a transaction step by step, which is parallelized via concurrent transactions across parallel transaction executors. However, this inter-transaction parallelism has drawbacks, e.g., amplifying conflicts between transactions, as our previous evaluation work has shown, cf. Chapter 3. Instead, an OLTP engine building on our programming model could explore the flexible combination of inter- and intra-transaction parallel execution. For example, to further maximize transaction throughput under volatile workloads, an OLTP engine could execute the tasks of a transaction in sequence (i.e., inter-transaction parallel) under low conflict workload but execute these tasks increasingly intra-transaction parallel with

increasing conflicts. However, analyzing such optimizations is beyond our scope and needs a more thorough investigation in future work.

4.4 KEY FINDINGS

This dissertation proposes the above outlined re-configuration approach for adaptive scale-up architectures, pursuing robust performance for scale-up DBMSs under volatile conditions. The publication in Chapter 9 details the building blocks of this re-configuration approach. Importantly, it also presents an evaluation which exposes our adaptive architecture and the state-of-the-art static architectures to volatile conditions to judge their robustness. Specifically, the evaluation represents the volatile workload and hardware conditions using a range of workloads from standardized benchmarks (i.e., YCSB [47] and TPC-C [185]) as well as small to large multi-processor hardware. Also, for a fair comparison it uses a prototype scale-up DBMS which implements both our adaptive architecture and the state-of-the-art static architectures, i.e., shared-everything [52], NUMA-aware [66], and shared-nothing [103] architectures. Notably, our optimizer re-configures the adaptive architecture to the specific conditions at hand, as in future DBMSs utilizing our adaptation approach. Whereas, the static architectures remain unchanged, as in today's DBMSs statically implementing a state-of-the-art architecture. Based on this setup, we judge the robustness of an architecture by considering how consistently it achieves high performance across the diverse conditions in comparison to the other architectures. In the following, we highlight the key findings of this evaluation and finally draw a conclusion about our proposed re-configuration approach.

ADAPTATION AT THE DATA STRUCTURE GRANULARITY Having claimed best-fit architectures demand adaptation for individual data structures at the core of DBMSs, we now highlight the results for the according experiments. In these experiments, we embed the core search data structures into the different architectures and execute write-heavy to read-heavy YCSB workloads on these data structures. The results, highlighted in Figure 4.3, prove that the static architectures only perform well in distinct sweet spots, i.e., for distinct data structures and workloads. In contrast, for all the data structures and workloads, our adaptive architecture (Opt. Configured) performs on par with the best static architecture or even better. For example, under the read-update workload (WL 1), the shared-nothing architecture is ideal for the hash map (H), whereas the BW-tree (BW) prefers the NUMA-aware architecture. Moreover, for the FP-tree (FP) and the B-tree (B) indeed none of the existing static architectures are ideal. Our optimizer instead configures a novel architecture with resource partitions the size of half a processor, which are in between the smaller

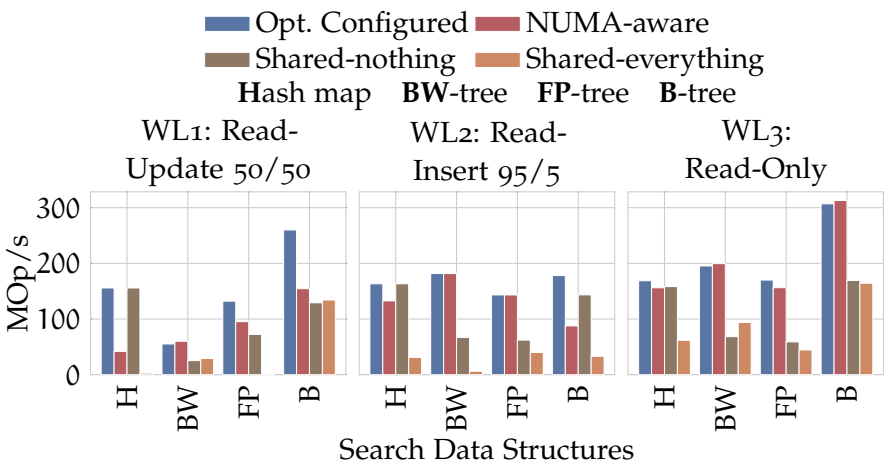


Figure 4.3: Throughput of our adaptive architecture with optimally configured resource partitioning (Opt. Configured) versus the partitioning strategies of current static architectures for a range of search data structures and YCSB workloads [47]. While the static architectures impose a fixed and coarse-grained resource partitioning, our optimizer configures our adaptive architecture with resource partitioning specifically optimized for the distinct data structures and workloads.

resource partitions of the shared-nothing architecture and the larger resource partitions of the NUMA-aware architecture.

Consequently, these results show that no static architecture performs well for all the search data structures commonly found at the core of DBMSs. Our flexible re-configuration instead allows the efficient adaptation of the architecture better fitting the distinct data structures under the specific workload at hand. Besides the flexibility to adjust to volatile conditions, indeed the fine-grained adaptation of our re-configuration approach proves performance benefits, as our optimizer more precisely specializes novel hybrid architectures.

ADAPTATION FOR A COMPREHENSIVE OLTP DBMS Besides the highlighted results for distinct data structures, our further experiments indicate that the observed benefits transfer to adapting the architecture for a comprehensive OLTP DBMS. First, one of the experiments corroborates our above observations for a large number of data structures. That is, also large DBMSs are expected to benefit from the flexible and precise adaptation of their architecture by our re-configuration approach. Second, further experiments confirm these benefits of our adaptive architecture beneficial for executing OLTP workloads in our lightweight OLTP engine. Also for the data structures of this OLTP engine and the OLTP workload, our optimizer configures a novel hybrid architecture that provides higher performance than the typical static NUMA-aware architecture of classical OLTP engine. An additional interesting insight is that compared to the high sensitivity of

classical OLTP engines to cross-partition transaction execution our adaptive architecture and runtime system also significantly reduce this sensitivity for our OLTP engine, e.g., improving the performance for non-partitionable workloads.

CONCLUSION The above evaluation results demonstrate robust performance of our adaptive scale-up architecture, due to the re-configuration of the architecture for the distinct search data structures and the specific workload and hardware conditions. We conclude that our approach successfully realizes the flexible and precise adaptation of scale-up architectures, for the following reasons. On one hand, these results indicate that indeed fine-grained specialization of the architecture is necessary to ideally accommodate the distinct data structures at the core of DBMSs. On the other hand, we observe our abstractions (i.e., tasks and configurations) to enable optimizers precisely and flexibly configure high performing scale-up architectures for the distinct data structures and changing conditions. Considering our discussed integration approach, we believe our re-configuration approach to allow the adaptation of the architecture for all the core data structures of the DBMS components of a full-fledged DBMS. Consequently, we derive that full-fledged DBMSs employing our re-configuration approach will be able to distinctly specialize their architecture for all their core data structures and will be able to adjust such precise architecture to changing workloads and hardware, thereby achieving robust performance under volatile conditions

5 ADAPTIVE ARCHITECTURES FOR SCALE-OUT DBMS

In this chapter, we summarize the realization of the adaptive architecture for scale-out DBMSs. As previously outline in Section 2.3, current static scale-out architectures orchestrate the DBMS components of scale-out DBMSs across an elastic network-connected resource pool, struggling to achieve high performance throughout the volatile and mixed cloud workloads. These static architectures not only struggle due to their fixed and coarse-grained design. Indeed, concurrent analytical and transactional queries of mixed HTAP workloads, for example, require opposing specialization of the architecture [11, 115, 126, 147], such that today's static architectures are sub-optimal for at least one of these simultaneous queries. This dissertation hence proposes a new radical approach, the architecture-less DBMS, striving for high and robust performance under mixed workloads with the simultaneous adaptation to distinct concurrent queries. For this purpose, our reactive execution model, explained in the following, establishes fine-grained flexible instrumentation for elastic resources to temporarily contribute to the specialized architecture of a specific query.

In the following, we first describe the contributions of the author of this dissertation to the backing peer-reviewed publication, in Section 5.1. In Section 5.2, we then outline the reactive execution model at the core of our architecture-less DBMS. Finally, we conclude with our key findings about our architecture-less concept for the adaptation of scale-out DBMSs, in Section 5.3. The full details on the adaptive scale-out architecture are in the according publication in Chapter 10.

5.1 PUBLICATION

PUBLICATION The work on the adaptive architecture for scale-out DBMSs is published in the peer-reviewed publication *AnyDB: An Architecture-less DBMS for Any Workload* in the proceedings of the *11th Annual Conference on Innovative Data Systems Research (CIDR '21)* [18].

CONTRIBUTIONS OF AUTHORS The contributions to the above publication by Tiemo Bang, the author of this dissertation, are as follows. Tiemo Bang is the leading author. He is responsible for the proposed approach to the adaptive scale-out architecture, the experimental evaluation, as well as the manuscript. The co-authors Norman May, Ilia

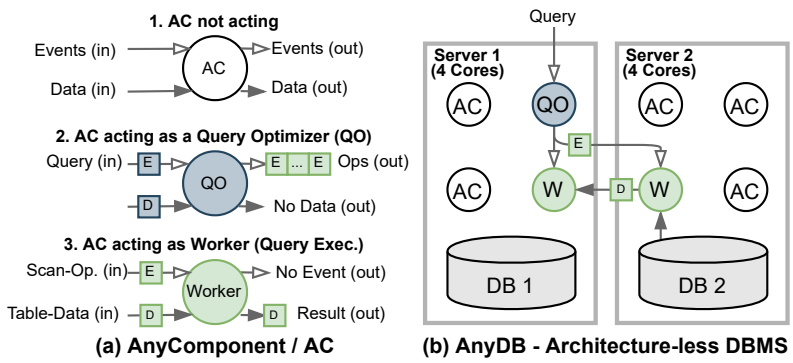


Figure 5.1: Execution model of an architecture-less DBMS. Generic components called AnyComponents (ACs) are instrumented by event and data streams. Depending on the incoming events an AC can act as a Query Optimizer (QO) or a Worker (W) executing a scan or a join operator or any other component, e.g., log writer, etc. An AC can also produce new event and data streams for other ACs. For example, an AC that acts as a scan operator produces a data stream with results of the scan operation.

Petrov, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication in this dissertation.

5.2 ADAPTATION OF ARCHITECTURE-LESS SCALE-OUT DBMS

At the core of the architecture-less DBMS, we propose a reactive execution model to enable the simultaneous adaptation of specialized architectures per query. In contrast to the previously explained adaptive scale-up architecture for a single server, this reactive execution is tailored to the efficient adaptation of scale-out DBMSs, exploiting abundant elastic resources but also accounting for the network communication, as will be explained.

In the following, we provide an overview of this execution model. We then highlight its underlying key principles. Subsequently, we discuss the key challenges in realizing this execution model and full-fledged DBMSs on top. As will be explained, our early prototype relies on simplified implementations of some of the aspects of the following proposal, allowing the early exploration of their opportunities.

5.2.1 *Overview of Execution Model*

Our reactive execution model realizes flexible instrumentation of specialized architectures, the main idea for the simultaneous adaptation for distinct concurrent queries. As illustrated in Figure 5.1 (a), our execution model defines generic components, so-called *AnyComponents* (ACs), which consume and produce event and data streams. The event stream encodes the operations of DBMS components to be

executed by an AC, while the data stream delivers the input state for these operations. When these events and data are sent to an AC, it simply reacts. Depending on the incoming events and data, the AC then temporarily acts as a query optimizer in one moment and in the next moment as a query executor or any other component. In turn, an AC may produce events and data for instrumenting subsequent operations on further ACs, e.g., the query optimizer produces events for executing query operators. Consequently, simply routing these event and data streams between ACs instruments these to temporarily act together as the architecture for a query.

Figure 5.1 (b) illustrates how the adaptation of an architecture generally works based on this instrumentation with streams. A SQL query (or a transaction) arriving at the architecture-less DBMS is the initial event that gets routed to some AC. This AC acts as query optimizer (QO) determining the architecture for this query. The query optimizer AC then produces events for executing the query operators, e.g., scans and joins. According to the determined architecture, these subsequent events are routed through the DBMS from one AC to another, temporarily instrumenting these as workers (i.e., query executors) to execute the distinct operators. In parallel, the query optimizer initiates the accompanying data streams to transfer the required input state, e.g., catalog data for the query optimizer itself but also table data (i.e., records) for the operators. That is, by the routing decisions for the query's event and data the query optimizer flexibly instruments ACs and defines the DBMS architecture perceived by a query.

This flexible temporary instrumentation of ACs allows the architecture-less DBMS to adapt specialized architectures as queries arrive. For example, the architecture-less DBMS can swiftly mimic a shared-nothing or disaggregated architecture, as in Figure 5.2. That is, when a query touches only one data partition and there is moderate load in the system, then the query optimizer can route streams of a query such that the architecture-less DBMS acts as a shared-nothing architecture. In case the query load in the system increases, however, resources for additional ACs can be added and the routing of streams can be adjusted to execute queries like a disaggregated architecture. Consequently, simply the distinct routing of these streams allows to flexibly specialize the architecture as queries arrive, shifting the architecture for queries in the system as in this example or orchestrating distinct architectures. Moreover, this example illustrates flexible resource isolation when the streams of distinct queries are routed to distinct resources, allowing further distinct specialization to individual queries.

5.2.2 *Key Design Principles*

For achieving both high flexibility and high efficiency, two key design principles underlie our reactive execution model. For the instrumen-

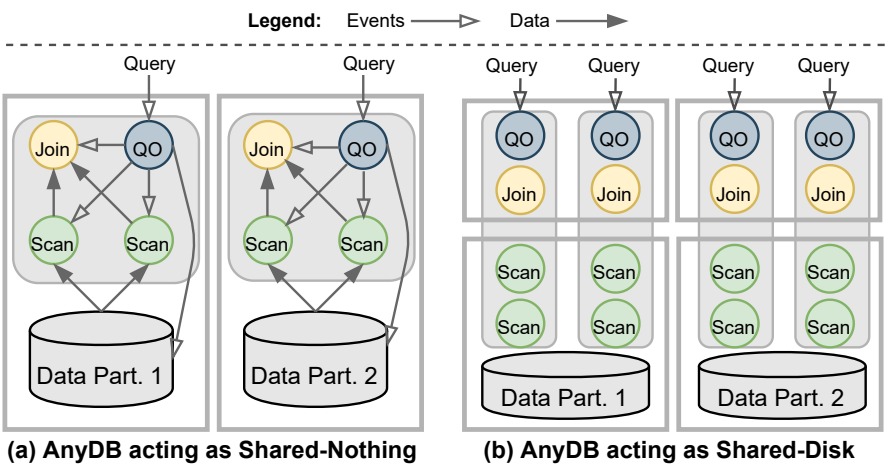


Figure 5.2: The architecture-less DBMS can mimic diverse architectures simply by using different routing schemes for event and data streams. In (a), two servers act as a shared-nothing database while in (b) additional resources (i.e., two servers with additional ACs) are added to act as a disaggregated architecture dealing with a higher query load. For simplicity, we only show the event and data streams for (a). The gray-shaded boxes around the ACs, however, indicate in (b) which ACs execute events of the same query.

tation of ACs across an elastic network-connected resource pool, the first principle concerns state management across ACs and the second principle concerns the execution within ACs. We explain these key principles in the following and detail their according design challenges in the next section. Further details are provided in Sections 10.3-10.4 of the publication.

FULLY STATELESS/ACTIVE DATA ACs are designed to be fully stateless meaning that events can be processed by any AC and all input state required to execute an event is being delivered to the AC via data streams, including table data but also catalog data, statistics, and any other state. By designing ACs fully stateless, we gain a high degree of freedom as any DBMS function can be executed anywhere. Moreover, in architecture-less DBMSs data is active, meaning that data is not pulled after an event is scheduled but it is actively pushed from data sources to the ACs before it is actually needed. This allows ACs to efficiently execute any DBMS function, facilitating the orchestration of distinct architectures across distinct resources but also elasticity for individual DBMS functions.

NON-BLOCKING/ASYNCHRONOUS EXECUTION The second key principle is that ACs are executing events in a non-blocking manner. This means an AC never waits for data of an event if data is not available yet. Instead, another event with available data is being processed.

For example, a filter or a join operator is only processed once its input data, a batch of records, is arriving via the data stream. To provide this non-blocking execution, ACs use queues to buffer input events and data items. In addition, these queues decouple the execution between ACs as much as possible, i.e., ACs can process events asynchronously from each other. This asynchronous execution model, which is only implicitly synchronizing the execution across ACs through events and data streams, opens up many new opportunities, as indicated in the following and discussed in the publication (cf. Chapter 10).

5.2.3 *Key Design Challenges*

There are several key challenges in designing the architecture-less DBMS for efficient query-based adaptation of architectures. One of them is the optimal routing of events and data for the given workload of queries in the system. Another one is to handle concurrency and updates, e.g., for transaction execution. In the following, we briefly discuss the main ideas how we aim to address these design challenges. Some of these ideas are already built into our prototype AnyDB while others represent future research directions.

EVENT AND DATA ROUTING A key challenge of an architecture-less DBMS is the decision how to handle a query, i.e., deciding how to route its events as part of query optimization. Depending on the requirements of an application (e.g., latency guarantees), load in the system, and the workload, the query optimizer must define an optimal event routing. Considering the opportunities identified in our exploration of the architecture-less DBMS (cf. Section 10.3) and the results of our prior evaluation work (cf. Section 3.4), we believe that this is an interesting avenue for learned query optimizers. In our current prototype, however, we do not focus on this problem but use an optimal decision to showcase the potential of our approach.

A second challenging aspect is the data streaming. As mentioned before, this aspect is important for the efficient data access, e.g., hiding latency across the network. We utilize the decoupling of data streams from events in our execution model to solve this challenge. The main observation is that in DBMS execution one often knows which data is accessed way ahead of time before the data is actually being processed. For example, complex OLAP queries need to be optimized and compiled, while the tables contributing to the input of a query are already known before query optimization. For efficient data access in the architecture-less DBMS we make use of knowledge and initiate data streams as early as possible. Once initiated a data stream actively pushes data to the AC where, for example, a filter operator will be executed once query optimization finished. We call this feature *data beaming* as data is often available at an AC before the according event

arrives, entirely hiding latencies of data transfers. We analyze the opportunities of data beaming for OLAP in detail in the publication, cf. Section 10.4.

Additionally, data streams of frequently accessed may be buffered, e.g., aiding transaction processing on hot records. That is, ACs indeed must be designed to solely consume input state from data streams, such that the ACs can execute any DBMS function on any resource when streaming the data there. However, for frequent streaming of the same data, a local buffer will aid performance. Crucially, when updating buffered data its consistency must be maintained across ACs and the storage. Rather than using a traditional buffer manager which would be challenged to coordinate the consistency across ACs, we believe integration with the protocols for consistent transaction processing (i.e., concurrency control) also provides benefits for buffered data streaming in the architecture-less DBMS like the integrated buffering for traditional multi-server DBMSs proposed by [209].

CONCURRENCY AND UPDATES In general, updates are executed by event streams directed towards the storage which ingests these events and produces acknowledgment events when the updates have been processed, as required for transaction coordination in OLTP. A major challenge in handling updates thus is to hide latencies of updates as much as possible. Since updates are represented as events, these are asynchronously executed by ACs like all operations of a transaction. Thereby, an update can be sent to the storage by one AC while other (independent) operations of the transaction can progress, hiding latencies of updates and decreasing the overall latency for executing a transaction. Additionally, for consistency of updates within the same transaction (i.e., read-your-own-writes) dependent operations can also consume updates as stream. Crucially, only the commit operation at the end of a transaction needs to know if the update successfully persisted and thus needs to wait for the acknowledgment event coming from the storage. As we show in Section 10.3 of the publication, this asynchronous model provides many interesting opportunities for OLTP and results in higher performance under various workloads.

Another challenge that is harder to solve is to execute concurrent updates correctly and efficiently. A naïve way would be to implement a lock manager by encoding lock operations as events and data streams providing the state of the lock table [79]. A more clever way, however, is to rethink concurrency protocols and route events and data streams such that their processing order already captures the requirements of a particular isolation level for concurrency control, as we discuss also in Section 10.3.

FAULT-TOLERANCE AND RECOVERY Fault-tolerance and recovery are two major challenges any DBMS needs to address. For an

architecture-less DBMS this is a challenge due to the asynchronous (decoupled) execution of multiple ACs which may fail individually.

Again, a naïve approach would be to implement standard write-ahead logging by sending log events from ACs to durable storage [194]. For recovery the DBMS could be stopped and the log could be used to bring the DBMS into a correct state. Instead, for an architecture-less DBMS we believe that we can learn from the streaming community. For example, as the entire execution of a DBMS is represented as streams, another direction is to make the streams reliable, such that upon AC failure the streams (events and data) can be rerouted to another AC [50]. Applying these ideas is again an interesting avenue of future research.

5.3 KEY FINDINGS

Proposing the new radical architecture-less concept for adapting scale-out architectures as outlined above, we explore its opportunities for the mixed cloud workloads in the publication, in Chapter 10. Specifically, we implement an early prototype of an architecture-less scale-out DBMS, called AnyDB, to explore the opportunities over the static shared-nothing or disaggregated scale-out architectures. Since the architecture-less DBMS provides many degrees of freedom to adapt the architecture, we focus this exploration on the core aspects for processing either workload class appearing in mixed cloud workloads, i.e., transaction processing and analytical processing. Below, we summarize the key findings about our architecture-less concept for the adaptation of scale-out DBMSs and indicate new research directions.

ADAPTATION AT THE MACRO-LEVEL At the macro-level of scale-out DBMSs, our exploration demonstrates high efficiency of the flexible disaggregation of the architecture-less DBMS for transaction processing. As shown in Figure 5.3, AnyDB achieves on par with the static shared-nothing architecture which is ideal for the partitionable OLTP workload (WL 1). The reason is that our execution model, although logically disaggregating the DBMS design into fine-grained events of operations of the DBMS components, still allows to execute these events in a physically aggregated manner if desired. As these result show, this duality of disaggregation facilitates the efficient mimicking of even the fully aggregated shared-nothing architecture. Consequently, we believe an architecture-less DBMS to efficiently enact diverse architecture ranging from the fully aggregated shared-nothing to fine-grained disaggregated architectures.

ADAPTATION AT THE MICRO-LEVEL Representing transactions as event streams in the architecture-less DBMS also provides opportunities at the micro-level of scale-out DBMSs. For example, for the workload shift from the partitionable OLTP workload (WL 1) to the skewed OLTP workload (WL 2), our execution model provides the

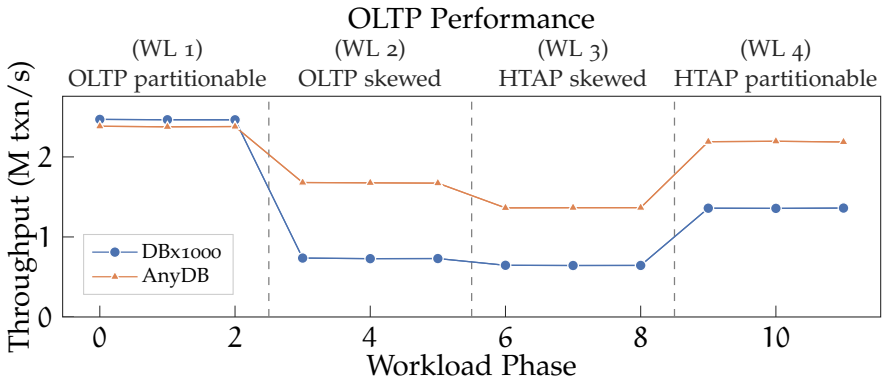


Figure 5.3: Performance of AnyDB across a workload evolving from partitionable OLTP (phase 0-2), over a skewed OLTP (phase 3-5), to skewed HTAP (phase 6-8), and then to partitionable HTAP (phase 9-11). The y-axis only shows the throughput of the OLTP transactions excluding the OLAP queries in the HTAP phases.

significant performance benefit due to (1) adaptation of the execution strategy and (2) the integration of the transaction manager component.

First, the flexible physical aggregation and routing of events also allows to adapt the parallel execution strategy for transactions. For example, it allows to shift from inter-transaction parallel execution of concurrent transaction to intra-transaction parallel execution of the operations within a transaction, thereby contributing to the observed performance benefit. While the intra-transaction parallel execution allows to alleviate contention between concurrent transactions, only a specialized intra-transaction parallel execution strategy provides best throughput, when also balancing resource load and communication overhead. We hence envision that optimizers can utilize the freedom on the micro-level of an architecture-less DBMS to derive execution strategies ranging from pure inter-transaction to fine-grained intra-transaction, ideally suiting the specific workload of individual transactions (as well as the underlying hardware).

Second, we identify significant benefits from integrating DBMS components into the architecture-less DBMS. That is, the performance benefit for the skewed OLTP workload (WL 2) in Figure 5.3 also stems from a novel transaction manager integrated into the routing of transaction events, which utilizes our event-based execution to avoid active synchronization. While we notice general support for traditional protocols, we find that the integration into our architecture-less concept opens new directions for the efficient elastic execution of DBMS components, e.g., alleviating inefficiencies for synchronization but also state management across elastic resources.

ADAPTATION FOR ANALYTICAL & MIXED WORKLOADS Finally, the high transaction processing performance for the mixed HTAP workloads (WL 3 & 4) indicates the benefit of simultaneously orches-

trating distinct architectures for OLAP and OLTP queries. While the flexible disaggregation and parallel execution strategies also benefit the OLAP queries, our detailed results show that the underlying data streams and our data beaming concept provide efficient data transfer between the storage and elastic resources executing our AnyComponents (ACs). Crucially, this efficient data transfer not only aids data-intensive OLAP queries but also facilitates the operation of distinct architectures on distinct elastic resources. Thereby, the architecture-less DBMS realizes independent adaptation and execution of the OLAP and OLTP queries, enabling observed high transaction throughput.

CONCLUSION Our above findings indicate significant benefits of fine-grained and flexible adaptation of our architecture-less DBMS concept, thanks to our reactive stream-based execution model. On one hand, our observations indicate that the broad freedom to adapt at the macro- and micro-level (i.e., overall architecture and execution model) facilitate best-fit specialization of scale-out architectures for distinct queries. On the other hand, the flexible routing and data beaming show the efficient simultaneous orchestration of distinct architectures, facilitating opposing specialization for concurrent queries. We hence conclude that our architecture-less concept will enable scale-out DBMSs to achieve higher performance for distinct queries of volatile mixed workloads, hence establishing overall more robust performance.

6 CONCLUSION & OUTLOOK

In the following, we conclude with the contributions of this dissertation for the specialization of DBMS architectures under volatile conditions. We subsequently provide an outlook how the proposed adaptive architectures will serve the robust specialization of DBMSs and even of novel cloud applications.

6.1 ADAPTIVE ARCHITECTURES FOR ROBUST DBMS

This dissertation proposed adaptive DBMS architectures. It addressed the problem that current DBMS architectures are statically implemented for predetermined workload and hardware conditions. These static architectures become unfit when conditions change and thus severely degrade performance, e.g., due to workload fluctuation or operation on diverse hardware. Instead, this dissertation approached the fine-grained and flexible adaptation of DBMS architectures for high and robust performance under changing conditions, as follows.

As a starting point, our evaluation of OLTP DBMSs with such static architectures identified a breath of complex effects impeding their performance. The results in Chapter 3 showed that the DBMS performance is determined by an agglomeration of performance bottlenecks, which interact with each other but also individually react differently to workload and hardware conditions. Hence, these evaluation results prove flexible and fine-grained adaptation of the DBMS architecture mandatory for best-fit specialization to volatile conditions.

This dissertation thus proposed concepts for the adaptation of DBMS architectures and realized these for the architectures of single-server scale-up and multi-server scale-out OLTP DBMSs, as outlined in Chapters 4–5. This dissertation realized a re-configuration approach to adapt scale-up architectures, targeting the single-server setting that has constrained resources but shared memory. It devises data-aware tasks and flexible resource partitions (virtual domains) for optimizers to declaratively configure architectures at the granularity of data structures. For scale-out DBMSs, this dissertation proposed the architecture-less concept to adapt their architecture in the multi-server setting flexibly and efficiently. To efficiently utilize elastic resources across the network of this multi-server setting, the architecture-less concept entails a reactive stream-based execution model for the flexible instrumentation of generic AnyComponents (ACs) on these resources. Simply by routing event and data streams the architecture-less scale-out DBMSs can instrument these ACs to temporarily enact specialized scale-out architectures for distinct concurrent queries.

The key findings are that both these realized adaptive architectures enable high and robust DBMS performance throughout volatile workloads and changing hardware. Where static architectures failed to maintain high performance, these adaptive architectures provided effective and efficient adaptation of the respective DBMS architectures, under all the conditions considered in this dissertation. In our experiments focusing on transaction processing (i.e., OLTP), the adapted architectures performed on par or better than the individually best static architectures, across diverse workloads and CPU-based hardware. Indeed, we observed superior performance of novel hybrid architectures which optimizers could derive through extensive and fine-grained specialization based on our adaptation approaches. In particular, the fine-grained re-configuration of scale-up architectures proved to better suit the distinct data structures at the core of scale-up DBMSs. Similarly, the fine-grained instrumentation of scale-out architectures better accommodated individual concurrent queries with distinctly specialized architectures, even for mixed transactional and analytical (i.e., HTAP) workloads. That is, under changing OLTP and HTAP workloads, our adaptation approaches provided higher and more robust performance due to flexible adjustment and more precise specialization of the scale-up and scale-out architectures.

Overall, we conclude that our proposed abstractions successfully resolve the limitations of static architectures implemented in today's DBMSs. We are confident that the adaptive architectures generally enable robust performance. While the above findings demonstrated robust performance for specific conditions, our adaptation approaches generally make a vast optimization space of architectures navigable, regardless the workload and hardware a DBMS may be facing. The proposed adaptation approaches hence generally enable optimizers to precisely and flexibly derive novel superior architectures for many workloads and hardware, besides mimicking established architectures.

Rather than necessitating the re-implementation of DBMSs, our adaptation approaches can serve as foundation for DBMSs to maintain suitably specialized architectures. In the expanding design space challenging today's static DBMS architectures, DBMSs instead will be able to adapt to changing workloads as well as new hardware, both enabling a single generic DBMS implementation and preventing performance degradation under changing, even future conditions. In addition, better specialized architectures also will aid the demand for ever-higher performance. We hence conclude our adaptive architectures serve as general platform for DBMSs to achieve high and robust DBMS performance within the expanding design space.

6.2 TOWARDS ADAPTIVE ARCHITECTURES FOR THE CLOUD

While our adaptation approaches prove beneficial for either scale-up or scale-out architectures, today's DBMSs indeed implement both kinds of static architectures for scaling-up and -out [4, 35, 51, 65, 102, 183]. Moreover, heterogeneous hardware is becoming generally available and bares significant potential for DBMS specialization [3, 28], e.g., FPGAs, GPUs, and programmable NICs. Hence, DBMSs soon must not only decide the amount of resources (i.e., the scale) but also the type of resources to assign to the DBMS components. Also, to achieve high performance throughout volatile conditions, optimizers must utilize our provided adaptivity to derive best-fit architectures.

As follows, we therefore see promising research in the directions of (1) an adaptive *multi-scale* architecture to specialize for distinct DBMS functions and queries in the combined optimization space, and (2) sophisticated optimizers capable of quickly specializing the architecture despite the high complexity. Additionally, we see broader promising research ultimately leading to automatically and robustly specialized cloud applications.

ADAPTIVE MULTI-SCALE ARCHITECTURE Observing the superior novel scale-up and scale-out architectures, we expect a future *adaptive multi-scale architecture* to provide even more potential for specializing novel architectures, when jointly adapting within and across elastic heterogeneous resources. Rather than coarse and competitive scaling-up and -out of today's DBMSs, the multi-scale architecture can enable joint and flexible adaptation of novel hybrid architectures within a common optimization space.

On one hand, the best amount and type of resources differ between distinct DBMS functions, workloads, and objectives. For example, scaling-up or scaling-out the transaction manager distinctly benefits objectives like transaction throughput or monetary cost, where an ideal amount of resources must balance the various workload and hardware effects [142]. However, today's DBMSs only coarsely scale-up and scale-out many components at once. On the other hand, so far DBMS components like the query optimizer or the storage engine individually decide to use heterogeneous hardware, and are limited to the available resources of the server the DBMS is deployed on [3, 28]. However, thereby the heterogeneous resources are predetermined at deploy-time, and the distinct DBMS components take local decisions to use these limited resources, failing to optimize the DBMS for the overall objective under the workload at hand.

Instead, the integration of our variable resource partitions with the flexible instrumentation of our reactive stream-based execution model and the extension to heterogeneous resources will result in an adaptive multi-scale architecture that allows to flexibly (dis-)aggregate functions of any DBMS components and elastically scale-up or -out het-

erogeneous resources, for distinct concurrent queries or transactions and indeed various objectives. When extending our AnyComponents (ACs), these ACs will be able to enact any DBMS component (or function) on any amount and type of resource, simply by consuming the event stream that encodes the DBMS function(s) to execute and the data stream that delivers the according input state, cf. Chapter 5. For example, thereby the multi-scale architecture could instrument ACs to scale-up the transaction manager on a single server for maximal throughput of non-partitionable transactions, while simultaneously instrumenting ACs on elastic heterogeneous resources to scale-out the transaction manager for minimal monetary cost of partitionable transactions. Importantly, the adaptive multi-scale architecture will extend the navigable optimization space, enabling optimizers to define many novel architectures for cloud DBMSs to best utilize the broad elastic cloud infrastructure for distinct queries and transactions.

LEARNED ARCHITECTURE OPTIMIZERS For automatically exploiting the large optimization space of adaptive (multi-scale) architectures, sophisticated optimizers appear as second interesting research direction. As starting point, we see the need for learned performance models allowing to accurately reason about the complex hardware and workload effects on the distinct DBMS components and functions, as found in our evaluation (Section 3.4). Given that such learned models are today embedded into promising proposals for learned DBMS components [58, 85, 86, 125, 163], we see great potential in learned optimizers for DBMS architectures. On one hand, we expect high quality architectures as learned optimizers will better model the observed complex effects and thus will better specialize the hardware deployment to the specific needs of distinct DBMS functions. On the other hand, there is also potential for faster optimization when using inference rather than traditional methods, as today is promising for learned query optimization compared to methods like dynamic programming and indeed even for core data structures like learned indexes [7, 109, 127]. We hence envision learned optimizers to quickly specialize high quality multi-scale architectures, especially enabling cloud DBMSs to fully facilitate their potential as queries or transactions arrive.

ADAPTIVE ARCHITECTURES FOR CLOUD APPLICATIONS Beyond these two research directions immediately enhancing our adaptive architectures, we see promising research in extending the adaptation to the entire DBMS design and indeed the application on top. This dissertation shows that the performance of the DBMS is determined by many performance bottlenecks, which must be balance together for maximizing performance. However, adaptation of the architecture cannot ideally address all of these and extensive information about the workload and hardware effects are required for ideal adaption. Hence, the combined flexible specialization of the entire DBMS design and indeed the application using the DBMS appears promising.

To achieve an overall balanced DBMS design, we envision a common DBMS design optimizer. Besides optimizing across the DBMS architecture and components, it could utilize application knowledge to better derive the specific workload on distinct DBMS functions. In particular, such DBMS design optimizer could trace the specific workload on a DBMS function across its preceding functions. On one hand, this would detail the distinct workload effects on specific DBMS functions, allowing to better optimize these. On the other hand, it would also provide detailed information about the cross-effects between DBMS functions, for better balancing the overall design. For example, such optimizer could maximize throughput of a cloud DBMS when declaring an adaptive architecture to execute write-heavy but coordination-free DBMS functionality close to the storage and read-only DBMS functionality on elastic resources.

Indeed, we see broader opportunities for robust specialization based on our adaptive architecture. Instead of adapting cloud DBMSs, we believe that our approach can generally serve as adaptive architecture for future cloud programming [40], which aims to compile applications and data management functionality into new cloud applications. While our programming model proves to serve the generic implementation of DBMS functions, it can generally serve as abstraction to generically implement arbitrary functions to be adapted, enabling the proposed cloud compilers [40] to integrate with our adaptive architecture. We hence expect the work of this dissertation and the outlined research to allow the flexible specialization for distinct functions of future cloud applications, e.g., enabling individual elastic scaling on heterogeneous resources for maximal throughput or minimal cost. Consequently, we see this research to lead to novel cloud applications automatically and robustly specializing for volatile conditions and flexible objectives.

Part II

PEER-REVIEWED PUBLICATIONS

7

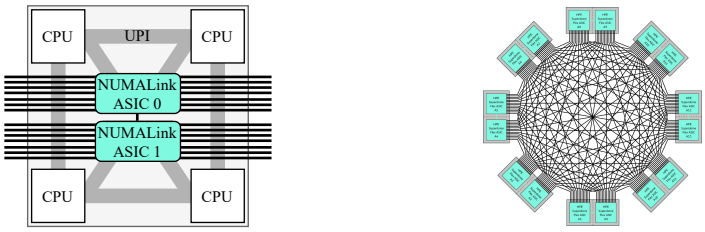
THE TALE OF 1000 CORES: AN EVALUATION OF CONCURRENCY CONTROL ON REAL(LY) LARGE MULTI-SOCKET HARDWARE

ABSTRACT

In this paper, we set out the goal to revisit the results of "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [206] and analyse in-memory DBMSs on today's large hardware. Despite the original assumption of the authors, today we do not see single-socket CPUs with 1000 cores. Instead multi-socket hardware made its way into production data centres. Hence, we follow up on this prior work with an evaluation of the characteristics of concurrency control schemes on real production multi-socket hardware with 1568 cores. To our surprise, we made several interesting findings which we report on in this paper.

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the peer-reviewed work *The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware* by Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig in the *International Workshop on Data Management on New Hardware (DAMON'20)*. The contributions of the author of this dissertation are summarized in Part I Chaper 3 Section 3.1.



(a) Chassis topology (b) NUMALink topology

Figure 7.1: System topology of the HPE SuperdomeFlex [82, 84].

7.1 INTRODUCTION

We are now six years after "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [206], which presented an evaluation of concurrency schemes for in-memory databases on simulated hardware. The speculation of the authors at that time was that today we would see hardware providing single-chip CPUs with 1000 cores. However, so far reality is different [83, 112]. Instead of single-chip CPUs with 1000s of cores, multi-socket machines are prevalent and made their way into production data centres, indeed offering 1000s of cores. Accordingly, in-memory DBMS are facing not only challenges of massive thread-level parallelism, such as coordination of hundreds of concurrent transactions as predicted by [206], but large multi-socket systems also expose in-memory DBMS to further challenges, such as deep NUMA topologies connecting all CPUs and their memory as in Figure 7.1.

In this paper, we set out the goal to bring in-memory DBMS to a 1000 cores on today's multi-socket hardware, revisiting the results of the simulation of [206] based on the original code, which the authors generously provide as open source. That is, we follow up on [206] with an evaluation of the characteristics of concurrency control schemes on real production multi-socket hardware with 1568 cores. To our surprise, we made several interesting findings: (1) First, the results of running the open-source prototype of [206] on today's production hardware revealed a completely different picture regarding the analysed concurrency schemes compared to the original results on simulated hardware. (2) Afterwards, in a second "deeper look" we analysed the factors leading to the surprising behaviour of the concurrency control schemes observed in our initial analysis, where we then find further surprises such as unexpected bottlenecks for workloads with a low conflict rate. (3) Based on these findings, we finally revisited the open-source prototype of [206] and reran the evaluation with our optimised version of DBx1000, which we think helps to establish a clear view on the characteristics of concurrency control schemes on real large multi-socket hardware.

In the following, we first report the concrete setup used in this paper (Section 7.2) and then discuss our findings (Section 7.3-7.5).

DL_DETECT	2PL with deadlock detection [24]
NO_WAIT	2PL with non-waiting deadlock prevention [24]
WAIT_DIE	2PL with wait-and-die deadlock prevention [24]
MVCC	Multi-version T/O [25]
OCC	Optimistic concurrency control [111]
H-STORE	T/O with partition-level locking [103]
TIMESTAMP	Basic T/O algorithm [24]
SILO	Epoch-based T/O [192]
TICTOC	Data-driven T/O [208]

Table 7.1: Bouquet of concurrency control schemes.

7.2 SETUP FOR OUR EXPERIMENTAL STUDY

In the following, we provide a brief overview of the concurrency control (CC) schemes, the hardware as well as the benchmarking environment used in our evaluation.

BOUQUET OF CONCURRENCY CONTROL: Table 7.1 summarises the evaluated CC schemes. They range from lock-based CC, with diverse mechanisms against deadlocks, to timestamp-ordering-based CC, including multi-versioning, 2-versioning, coarse locking, and advanced ordering. For details on these CC schemes, we to their original publications [24, 25, 103, 111, 192, 208] and to [206]. The first seven CC schemes in Table 7.1 correspond to the prior evaluation in [206]. We further include the more recent schemes *SILO* [192] and *TICTOC* [208], not included in the original study. Unfortunately, *TIMESTAMP* from [206] has a fatal bug in the latest version of the prototype, so we excluded this scheme from our experiments.

REAL HARDWARE WITH A 1000 CORES: The prevalent hardware in production today offering 1000 cores are large multi-socket machines [83, 112]. As shown in Figure 7.1, such hardware connects many CPUs to a single system rather than hosting many cores on a single CPU. Our *HPE SuperdomeFlex* system [83] contains 28 *Intel Xeon 8180* CPUs. It groups four CPUs into hardware partitions (chassis), which are then joined together, forming a single cache coherent system with a total of 1568 logical cores and 20 TB of DRAM. As shown in Figure 7.1a, within the chassis each CPU connects to two neighbouring CPUs and to a *NUMALink* controller via UPI links. Then the *NUMALink* controllers couple all chassis in a fully connected topology (Figure 7.1b), yielding four levels of NUMA with performance properties summarised in Table 7.2.

NUMA level	Latency	Bandwidth
local	102 ns	95.1 GB/s
1 hop UPI	150 ns	17.2 GB/s
2 hop UPI	208 ns	16.6 GB/s
NUMALink	380 ns	11.2 GB/s

Table 7.2: Memory access latency and bandwidth by NUMA level as measured via Intel MLC [198].

Comparing this hardware to potential many-core hardware as simulated in [206] reveals that this multi-socket setup for 1000 cores differs in many aspects. Importantly, one similarity of today's hardware to the simulated architecture of [206] is that both communicate and share cache in a non-uniform manner via a 2D-mesh on the chip [67] (and UPI beyond), such that the cores use the aggregated capacity to cache data but need to coordinate for coherence. This non-uniform communication is an important hardware characteristic, as it can amplify the impact of contention points in the CC schemes on any large hardware (multi-socket and many-core). Otherwise, the simulation differs from today's hardware, since it assumed low-power and in-order processing cores clocked at 1GHz, cache hierarchies with only two levels, and cache capacities larger than today's caches. Notably, it simulates the DBMS in isolation without an OS, disregarding overheads and potential side effects of OS memory management, scheduling etc., omitting essential aspects of real systems [59, 145] like ours.

BENCHMARKING ENVIRONMENT: As [206], we evaluate the CC schemes mentioned before on our multi-socket hardware with the TPC-C benchmark [185] as implemented in the latest version of DBx1000[1]. This version of DBx1000 includes the extended set of CC schemes as mentioned before and bug fixes, beyond the version used in the original paper [206]. For running the benchmarks, we use the given default configuration of DBx1000. This configuration defines the TPC-C workload as equal mix of *New-Order* and *Payment* transactions covering 88% of TPC-C with standard remote warehouse probabilities (1%[2] and 15%). This configuration partitions the TPC-C database by warehouse (WH) ID for all CC schemes. Based on this configuration, we specify four warehouses for the high conflict TPC-C workload and for the low conflict workload we specify 1024 or 1568 warehouses, maintaining the ratio of at most one core per warehouse as [206].

An interesting first observation was, that the TPC-C implementation of DBx1000 does not include insert operations, presumably due to the mentioned limitations of the simulator, e.g., memory capacity and no

1 https://github.com/yxymit/DBx1000/tree/b40c09a27d9ab7a4c2222e0ed0736a0cb67b7040
2 Based on a typo, the original paper [206] states 10% instead of 1%

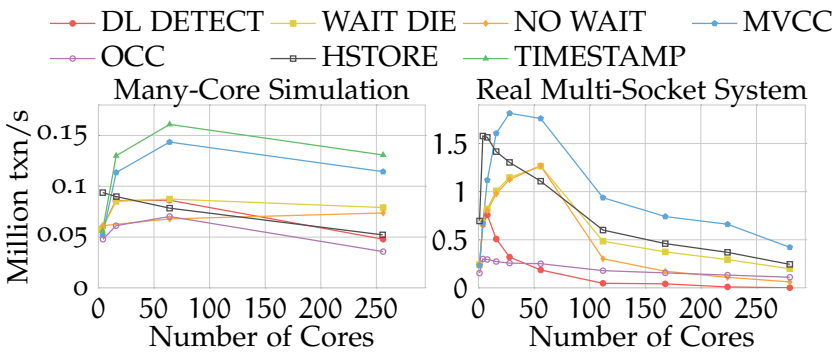


Figure 7.2: Throughput of TPC-C high conflict workload (4 WH) in original simulation [206] and on real multi-socket hardware.

OS. In this paper, we first start with the very same setup, but later we also enable insert operations in the evaluation after taking a first look at the CC schemes. As minor extension, we added a locality-aware thread placement strategy to DBx1000 for all experiments in this paper, which exclusively pins DBMS threads to a specific core. For scaling the DBMS threads in our experiments, we use the minimal number of sockets to accommodate the desired resources, e.g., 2 sockets for 112 threads. Otherwise OS and NUMA effects would dominate the overall results. Note that as consequence of this thread placement strategy, *cores* and *threads* equally refer to a single execution stream (i.e., a worker) of the DBMS. In our initial experiments (Sections 7.3-7.4), we use up to 1024 cores like the simulation in [206]. Only after our optimisations, we leverage the full 1568 cores of our hardware, showing the scalability of our optimised DBMS in Section 7.5.

7.3 A FIRST LOOK: SIMULATION VS. REALITY

We now report the results of running the DBx1000 prototype directly on the multi-socket hardware as opposed to a simulation.

7.3.1 *The Plain Results*

Figures 7.2 and 7.3 display the throughput of TPC-C transactions for *4 warehouses* and *1024 warehouses*, i.e., high and low conflict OLTP workloads. *On the left* of each figure are the original simulation results [206] and *on the right* are our results on a real multi-socket hardware. We first report the plain results. Then we break down where time is spent in the DBMS to better understand our observations.

We first look at the results for 4 warehouses as shown in Figure 7.2. Overall, it is obvious that the absolute throughput differs due to the characteristics of the CPUs in the simulation and our hardware, e.g., low-power 1 GHz cores versus high-power 2.5 GHz cores, which can

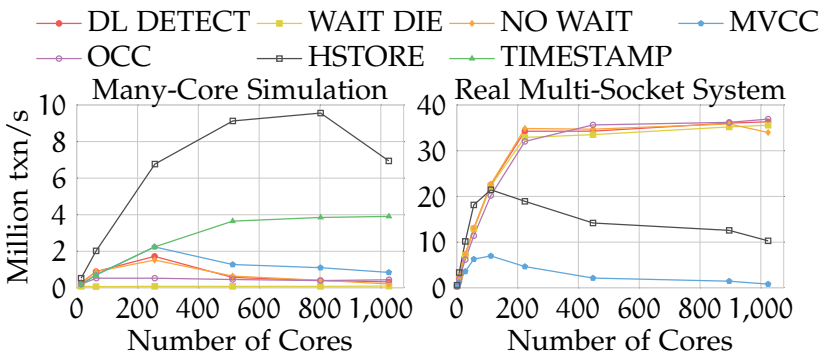


Figure 7.3: Throughput of TPC-C low conflict workload (1024 WH) in original simulation [206] and on real multi-socket hardware.

be expected and therefore only the relative performance of the CC schemes matters. In the following, we now discuss some similarities but also significant differences.

First, comparing the simulation and the real hardware in Figure 7.2, we see that the CC schemes *HSTORE*, *MVCC*, and *NO WAIT* show similar trends. That is, these CC schemes have a similar thrashing point in the simulation and the real hardware, i.e., *HSTORE* at 4 to 8 cores and *MVCC* as well as *NO WAIT* at 56 to 64 cores. After the respective thrashing point, these CC schemes degrade steeper on the multi-socket hardware, which can be linked to the additional NUMA effect of the multi-socket hardware appearing beyond 56 cores. For the other CC schemes the results for the simulation and real hardware differ more widely, especially the diverging behaviour of the pessimistic CC schemes sticks out. Considering these pessimistic CC schemes, *DL DETECT* behaves broadly different already degrading at 8 cores rather than 64 cores and *WAIT DIE* performs surprisingly close to *NO WAIT*. In the Section 7.3.2, we analyse the time breakdown of this experiment to explain these results. It reveals characteristic behaviour of the individual CC schemes, despite the diverging throughput in the simulation and the multi-socket hardware.

Next, we look at the low contention TPC-C workload (1024 warehouses) in Figure 7.3. The results here present fewer similarities of the many-core simulation and the multi-socket hardware, i.e., only the slope of the *MVCC* scheme is similar. Additionally, *DL DETECT* and *NO WAIT* stagnate at high core counts (>224) in the simulation and on the multi-socket hardware. In contrast, *HSTORE* performs worse on the multi-socket hardware than in the simulation. It is slower than the pessimistic CC schemes and *OCC* from >112 cores. Also, *OCC* and *WAIT DIE* achieve higher throughput on the multi-socket hardware, now similar to *DL DETECT* and *NO WAIT*. Moreover, unexpectedly in this low conflict workload, *MVCC* is significantly slower than *OCC* and the pessimistic CC schemes, which is caused by high overheads

of this scheme as we discuss next.

Insight: The initial comparison of concurrency control schemes on 1000 cores presents only minor similarities between the simulation and our multi-socket hardware with surprising differences in the behaviour of the CC schemes mandating further analysis.

Useful	Time usefully executing application logic and operations on tuples.
Abort	Time rolling back and time of wasted useful work due to abort.
Backoff	Time waiting as backoff after abort (and requesting next transaction to execute).
Ts. Alloc.	Time allocating timestamps.
Index	Time operating on hash index of tables including latching.
Wait	Time waiting on locks for concurrency control.
Commit	Time committing transaction and cleaning up.
CC Mgmt.	Time managing concurrency control other than prior categories, e.g., constructing read set.

Table 7.3: Time breakdown categories.

7.3.2 *A First Time Breakdown*

For deeper understanding of the observed behaviour of the CC schemes, we now break down where time is spent in processing the TPC-C transactions on the multi-socket hardware. For this purpose, we apply the breakdown of [206] categorising time as outlined in Table 7.3. For each CC scheme, Figures 7.4 and 7.5 break down the time spent relative to the total execution time of the TPC-C benchmark with a bar for each core count.

The time breakdown of 4 warehouses in Figure 7.4 neatly shows the expected effect of conflicting transactions and aborts for increasing core counts under high conflict workload. That is, most CC schemes result in high proportions of *wait*, *abort*, and *backoff* as soon as the number of cores exceeds the number of warehouses (>4 cores), yielding nearly no *useful work* at higher core counts. Only the *wait* time of *HSTORE* immediately grows at 4 cores concurrently executing transactions, such that *HSTORE* appears more sensitive to conflicts for this workload.

Remarkably, textbook behaviour of the specific schemes becomes visible in the breakdown: Starting with *DL DETECT*, its *wait* time increases with increasing number of concurrent transactions as expected, following the increasing potential of conflicts between concurrent

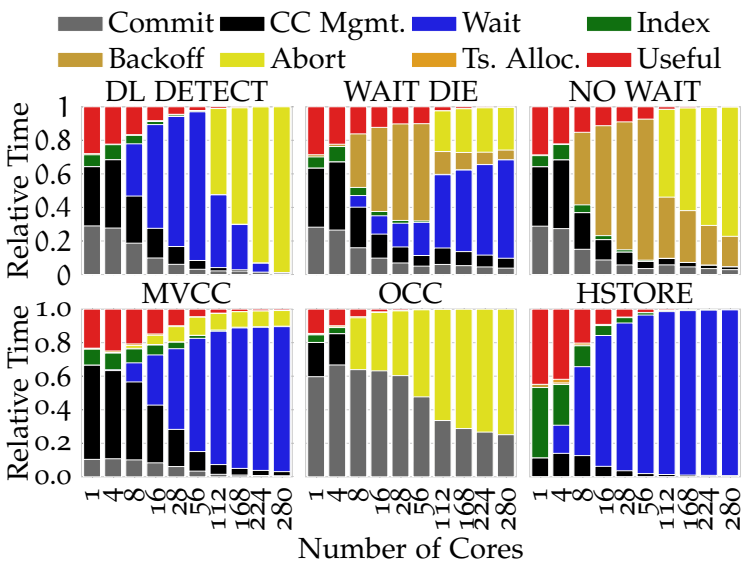


Figure 7.4: Breakdown of relative time spent for high conflict (4 WH) TPC-C transactions on multi-socket hardware.

transactions. Different from *DL DETECT*, *WAIT DIE* spends more time *backing off* and *aborting* due to its characteristic aborts after a short wait time (small wait proportion). Instead *NO WAIT* solely *backs off* without waiting, spending even more time on *aborted* transactions. The optimistic *MVCC* waits on locks during validation, such that its break down shows similar *wait* times like *DL DETECT*. Finally, for *OCC* we can see that the high *abort* portion reflects its sensitivity to conflicts while the high *commit* portion stems from high costs for cleaning up temporary versions at commit time.

Having observed this "expected" behaviour of the CC schemes under high conflict, we now analyse the unexpected behaviour under low conflict (1024 warehouses) as shown in Figure 7.5. Against the expectation, most CC schemes spend considerable amount of time to manage concurrency (black and grey area) such as lock acquisition (except *HSTORE* which we discuss later). For these schemes this results in at most 50% of useful work (red area). Staggeringly, *MVCC* which actually should perform well under low conflicting workloads, spends almost no time with *useful work* despite the low conflict in the workload, i.e., <10% *useful work* from 224 cores. In fact, the low conflict is visible in the overall little time spent *waiting* or *aborting*. Consequently, the slowdown compared to pessimistic CC schemes does not stem from wasted work but from pure internal overhead in execution of this CC scheme under high core counts.

In contrast, we observe for *HSTORE* an increasing impact of *timestamp allocation* and *waiting time*. While *timestamp allocation* is used by the other schemes as well, the relative overhead for *HSTORE* is the highest since lock acquisition in *HSTORE* is cheap. In fact, the au-

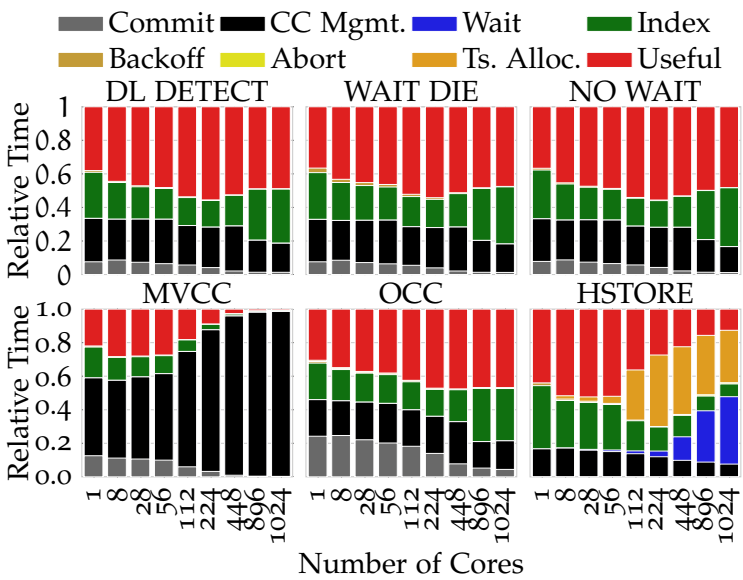


Figure 7.5: Breakdown of relative time spent for low conflict (1024 WH) TPC-C transactions on multi-socket hardware.

thors of [206] did analyse different timestamp allocation methods in their paper but chose *atomic increment* as a sufficiently well performing method that is a generally applicable option when there is no specialised hardware available. However, as we can see this choice is not optimal for multi-socket hardware. Moreover, we attribute the increasing *waiting time* of *HSTORE* to its coarse-grained partition locking to sequentially execute transactions on each partition. This partition-level locking causes a higher overhead if more cores are used since this leads to more conflicts between transactions as shown in prior work [106, 141].

Insight: The analysed CC schemes behave differently on the real multi-socket hardware than in the simulation of [206]. For the high conflict workload (4 warehouses), the behaviour on real hardware and the simulation appears more similar, for which the time breakdown confirms the expected characteristics for each CC scheme. However, low conflict workload (1024 warehouses) causes an unexpectedly high CC management overhead in most CC schemes and transactions execute only a limited amount of useful work, except for *HSTORE* where waiting and timestamp allocation dominate.

7.4 A SECOND LOOK: HIDDEN SECRETS

In this section, we now take a "second look" on the factors leading to the surprising behaviour of the CC schemes observed in our initial analysis and discover equally surprising insights.

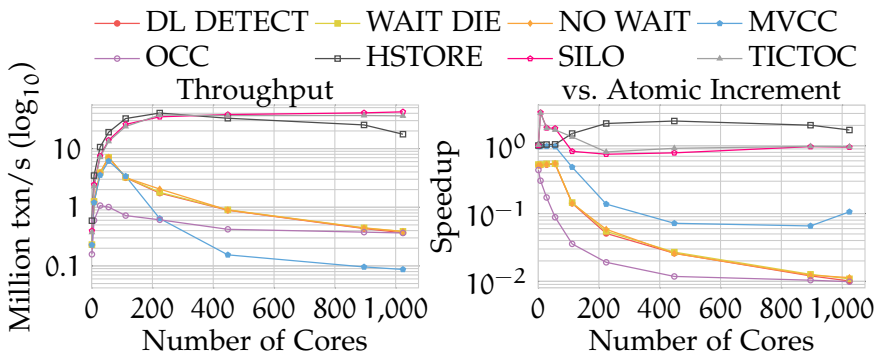


Figure 7.6: Throughput of TPC-C with 1024 warehouses for timestamp allocation with hardware clock.

7.4.1 *Hardware Assistance: The Good?*

In a first step, we analyse the benefit of hardware-assisted timestamp allocation over using atomic counters for the real multi-socket hardware. As explained earlier, the *atomic increment* is generally applicable but may cause contention, which efficient and specialised hardware may prevent if available. Fortunately, as already mentioned in [206], timestamp allocation can also be implemented using a synchronised hardware clock as supported by the Intel CPUs [95] in our hardware (*rdtsc* instruction with *invariant tsc* CPU feature). Therefore, we can replace the default timestamp allocation via atomic increment with this hardware clock.

In the following experiment, we analyse the benefit of this hardware assistance for timestamp allocation. Figure 7.6 shows the throughput of the CC schemes for 1024 warehouses with timestamp allocation based on the hardware clock.

On one hand, *HSTORE* greatly benefits from the hardware clock (as expected) achieving peak throughput of ~ 40 M txn/s with an overall speedup over atomic increment of up to 3x. We now also include *SILO* and *TICTOC* in our results which perform like *HSTORE* except for high core counts as we discuss in the time breakdown analysis below. On the other hand, the remaining CC schemes (*DL DETECT, WAIT DIE, NO WAIT, MVCC*, and *OCC*) degrade drastically when using the hardware clock instead of atomic counters. That is, the pessimistic CC schemes *DL DETECT, WAIT DIE*, and *NO WAIT* perform ~ 50% slower within a socket (0.51 - 0.55x speedup for ⩽56 cores), after which they degrade to 0.01x speedup at 1024 cores (0.37 - 0.39 M txn/s). Likewise, *MVCC* is stable (~1x speedup) up to 56 cores and its speedup drops to 0.1x when exceeding the single socket. Finally, *OCC* does not benefit from the hardware clock at all (0.44 - 0.01x speedup).

Overall, timestamp allocation based on the hardware clock drastically changes the perspective on the performance of the CC schemes. Now *HSTORE* performs best, meeting the initial observations of [206]

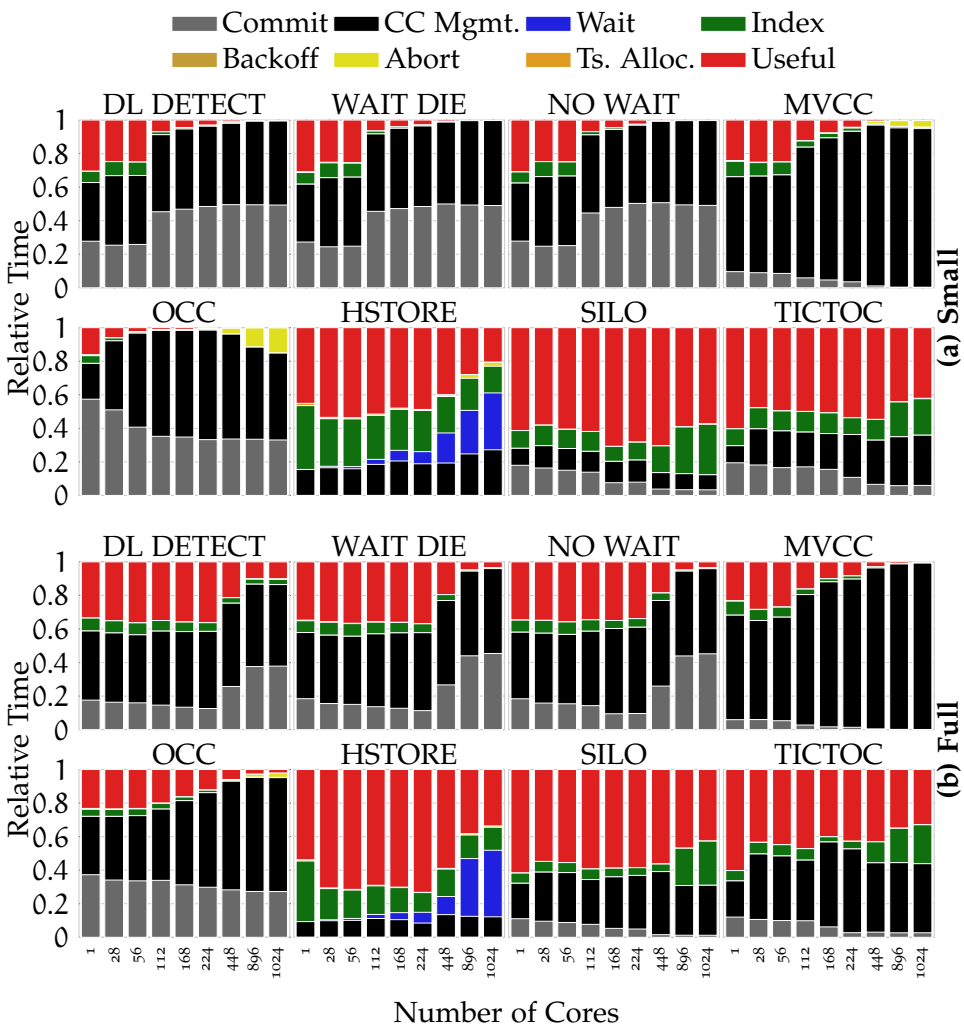


Figure 7.7: Breakdown of relative time spent processing TPC-C transactions on *small* and *full* schema with 1024 warehouses using timestamp allocation via hardware clock.

(joined by *SILO* and *TICTOC*), whereas the pessimistic schemes, *OCC*, and *MVCC* degrade severely.

For better understanding of these diverse effects of the hardware clock, we again look at the time breakdown shown in Figure 7.7a (top row). As expected, *HSTORE* now spends no significant time for timestamp allocation anymore (like *SILO* and *TICTOC*). Its *waiting time* still significantly increases as before. This explains the slowdown for >448 cores and corroborates earlier descriptions of *HSTORE*'s sensitivity to conflicts on the partition level as discussed in Section 7.3.2. An interesting observation is the significant change in the time break down of the other CC schemes. For example, *DL DETECT, WAIT DIE*, and *NO WAIT* show at least double the time spent for *CC Mgmt.* and *committing/cleaning up* (black & grey, bottom two bars) with a sudden increase after 56 cores. *OCC*'s increase of time spent in these categories increases even more drastically with less than 20% of *useful work* at any core count. Only *MVCC* does not show a significant change in this breakdown, since its *useful* time spent was low already.

Profiling these CC schemes reveals contention, that previously was on atomic counters, now results in higher thrashing of latches, despite a latch per row and low conflicts in the workload with 1024 warehouses. Notably, our profiling further reveals more interesting details of the individual CC schemes: The *pthread_mutex* employed in *DL DETECT, WAIT DIE, NO WAIT*, and *OCC* sharply degrades due to NUMA sensitivity of hardware transactional memory [31] used for lock elision and its fallback to robust but costly queuing synchronisation [70] as well as costly interaction with the scheduler of the OS.[3] In contrast, *MVCC* uses an embedded flag as spin latch which is not as sensitive to NUMA but also not robust [49]. Hence, this type of latch shows a slower but also continuous degrading of performance.

Insights: Hardware-assisted timestamp allocation via specialised clocks alleviates contention and leads to better scalability for *HSTORE* (as well as *SILO* and *TICTOC*). However, while introducing hardware-assisted clocks also shifts the overhead in the other schemes, it does not necessarily improve their overall performance as contention moves and puts pressure on other components (e.g., latches), even leading to further degradation of the overall performance.

7.4.2 *Data Size: The Bad?*

In the context of this surprisingly high overhead, our second look at the paper [206] brings the following statement to our attention: "Due to memory constraints [...], we reduced the size of [the] database" [206]. Consequently, we are wondering if the staggering overhead is potentially caused by absence of useful work to execute rather than the

3 pthread_mutex is specific to *libc* and OS as well as configurable.

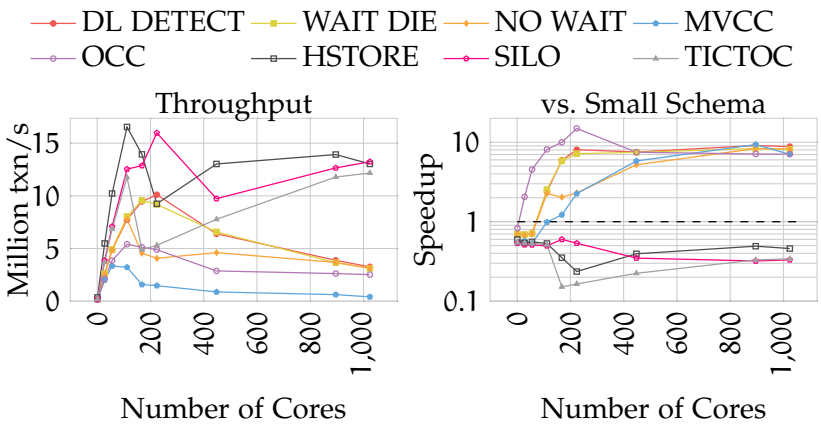


Figure 7.8: Throughput of TPC-C with 1024 warehouses for small schema size like in the simulation versus full schema size both executed on multi-socket hardware.

abundance of overhead in the CC schemes, due to the reduced data size imposed by limited memory capacity of the simulator in [206]. Therefore, we revert the benchmark to the full TPC-C database in the following experiment and report on the surprising effect of the larger data volume.

Figure 7.8 shows the throughput for the full schema with 1024 warehouses and speedup in comparison to the small schema based on the previous experiment (cf. Figure 7.6). We measure quite diverse throughput of the CC schemes. Yet, the speedup indicates that two major effects of the increased data volume appear in the same clusters as in the previous experiment but with inverse outcome. The first cluster of *HSTORE*, *SILO*, and *TICTOC* is slower with the full schema, i.e., 0.2-0.6x, 0.3-0.5x, and 0.2-0.5x, respectively. The second cluster, consisting of the previously "slower" CC schemes, improves inversely to the previously described thrashing points. That is, *DL DETECT*, *WAIT DIE*, and *NO WAIT* have a speedup of 0.7x up to 56 cores, after which they benefit from the full schema with speedups of 2.4-9.1x, 2.5-8.3x, and 2.0-9.3x, respectively. *MVCC* has a speedup of 0.5-0.6x until 56 cores, breaks even (1x) at 112 cores, and then improves with a speedup of 1.2-9.3x. *OCC* has a speedup of 0.8 at 1 core and broadly improves with the full schema with 2.1-14.9x speedup.

The time breakdown in Figure 7.7b (lower row), presents insights on the causes. As for the CC schemes in the first cluster, *HSTORE* has increased *useful* work, whereas for *SILO* and *TICTOC* *CC Mgmt.* increases, both indicate increased cost of data movement, as *HSTORE* directly accesses tuples and the other two create local temporary copies in the CC manager. The second cluster also has an increase of *useful* work to some extent, presenting less staggering overhead of CC management at low core counts. Importantly, the sudden increase of *commit* for *DL DETECT*, *WAIT DIE*, and *NO WAIT* is delayed,

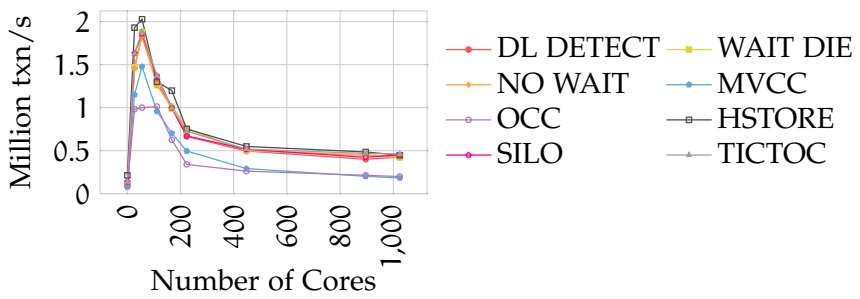


Figure 7.9: Throughput of TPC-C including inserts with full schema size on multi-socket hardware.

indicating that latches thrash only from 448 cores (while previously already from 112 cores). For *OCC* the time spent on *commit* also decreases with the larger data volume, but the increase of *CC Mgmt.* due to larger temporary copies still diminishes *useful work*. Only for *MVCC* the time break down does not change significantly.

We attribute these observations to two effects of the larger data volume: The heavier data movement slows down data-centric operations (e.g., tuple accesses or temporary copies), but in turn alleviates pressure on latches preventing thrashing.

Insight: The effect of larger data volumes in the full schema changes the perspective on the CC schemes again, most notably on the differences between the individual CC schemes. Moreover, also the relation of *useful work* and overhead within each scheme changes. Both are caused by larger data volume reducing performance of data movement, but also alleviating pressure on latches.

7.4.3 *Inserts: Facing Reality!*

Since the simulator of [206] had limited memory capacity and excluded the simulation of important OS features such as memory management, the TPC-C implementation of DBx1000 did not include insert operations and for comparability we initially excluded these as well. For the last experiment in this section, we now complete the picture of concurrency control on real hardware.

Accordingly, Figure 7.9 shows the throughput of TPC-C transactions including inserts (as well as all before-mentioned changes) for 1024 warehouses. As we can see, the inserts drastically reduce throughput of all CC schemes and introduce heavy degradation at the socket boundary (56 cores). Even more interesting, all CC schemes perform similarly with inserts included in the transactions. Indeed, profiling indicates execution of insert operations are the hotspot of the TPC-C transactions now, but the causes are orthogonal to concurrency control.

The two major hotspots are (1) catalogue lookups to locate tuple fields and (2) memory allocation for new tuples during insert operations.

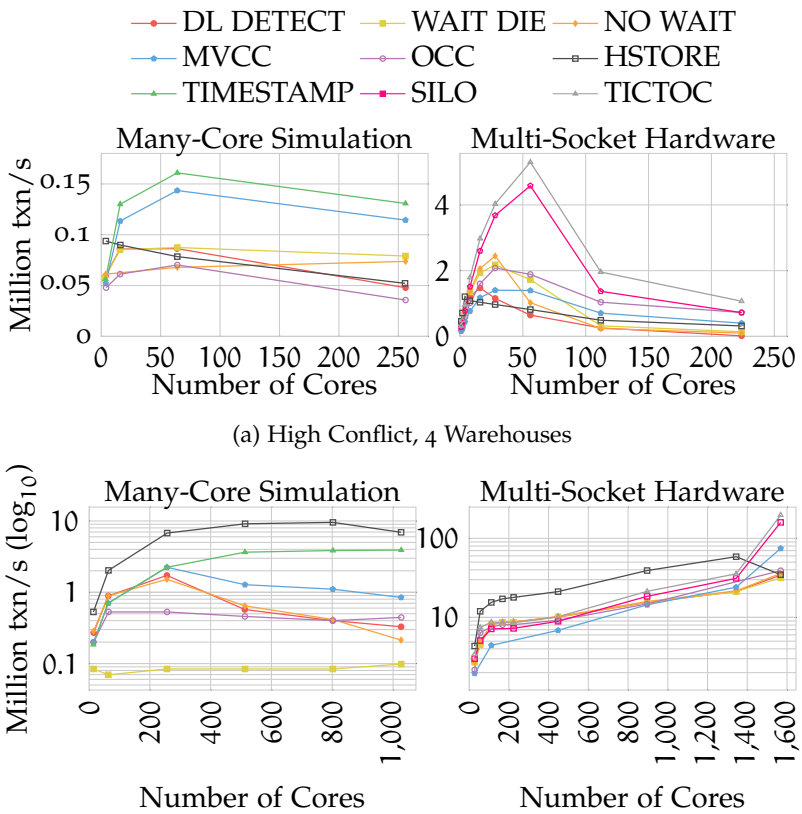
Profiling details show that catalogue lookups cause frequent accesses to L1 and L3 caches. For tuple allocation, profiling details indicate significant time spent in the memory allocator and for OS memory management including page faults. These hotspots are amplified by NUMA in our multi-socket system, since the catalogue is centrally allocated and memory management in Linux is contention- and NUMA-sensitive as well [41]. Consequently, such impact on performance only becomes visible in its full extent on large systems like ours.

Insight: Inserts themselves do significantly affect the performance of the CC schemes in this benchmark. Yet, in all schemes performance is now greatly overshadowed by orthogonal hotspots most notably cache misses of the catalogue and memory allocation during inserts.

7.5 A FINAL LOOK: CLEARING SKIES

Finally, we take a last step to provide a more optimal handling of inserts in DBx1000 to get a clear view on the characteristics of the CC schemes on large multi-socket hardware. To clear this view, we remove previously identified obstacles and further optimise DBx1000 as well as the implementation of the TPC-C transaction based on state-of-the-art in-memory DBMS for large multi-socket hardware [90, 105, 106].

Our optimisations address thrashing (cf. Section 7.4.1) with a queuing latch and exponential backoff [90]. We optimise data movement (cf. Section 7.4.2) with reordering and prefetching of tuple and index accesses, a flat perfect hash index, and NUMA-aware replication of the read-only relations [106]. Additionally, for the hotspots identified in Section 7.4.3, we transform expensive query interpretation (especially catalogue lookups) into efficient query compilation as done by state-of-the-art in-memory DBMS [105] and introduce a thread-local memory allocator that pre-allocates memory, as done in commercial in-memory databases today [83]. Memory alignment is also an interesting trade-off for memory management. On one hand, alignment to cache line boundaries prevents false sharing and may generally be required by some CPUs. On the other hand, this alignment may amplify memory consumption, as records are allocated as multiples of cache lines, e.g., 64 bytes. DBx1000 aligns to 64 bytes and our allocator does so as well now, because false sharing obliterates performance without alignment. Additionally, we reduce CC overhead for read-only relations and update the deadlock prevention mechanisms to state-of-the-art as recommended by [90].



(a) High Conflict, 4 Warehouses

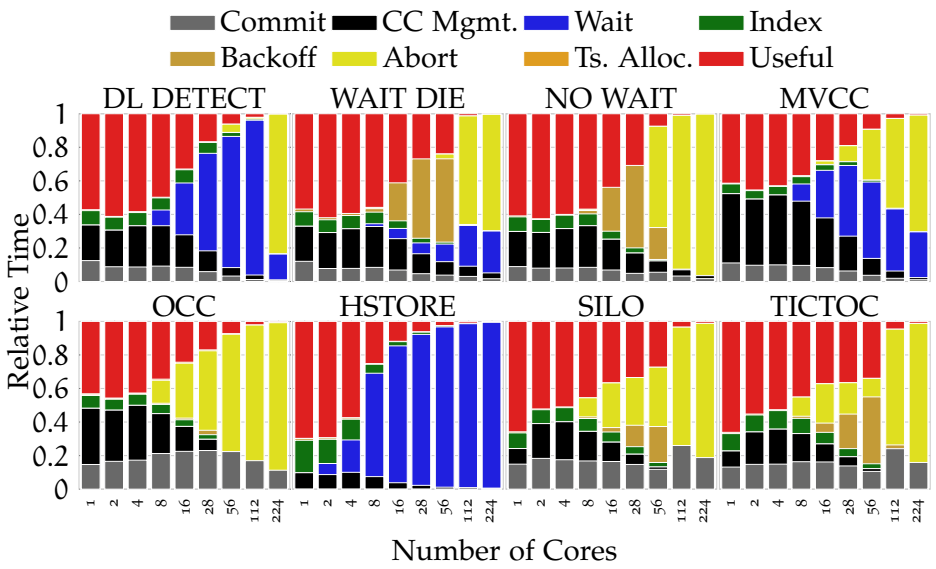
(b) Low Conflict, 1024 Warehouses for Simulation and 1568 for Multi-Socket Hardware

Figure 7.10: Throughput of TPC-C in original many-core simulation [206] without full schema & inserts and our optimised implementation with full schema & inserts on multi-socket hardware.

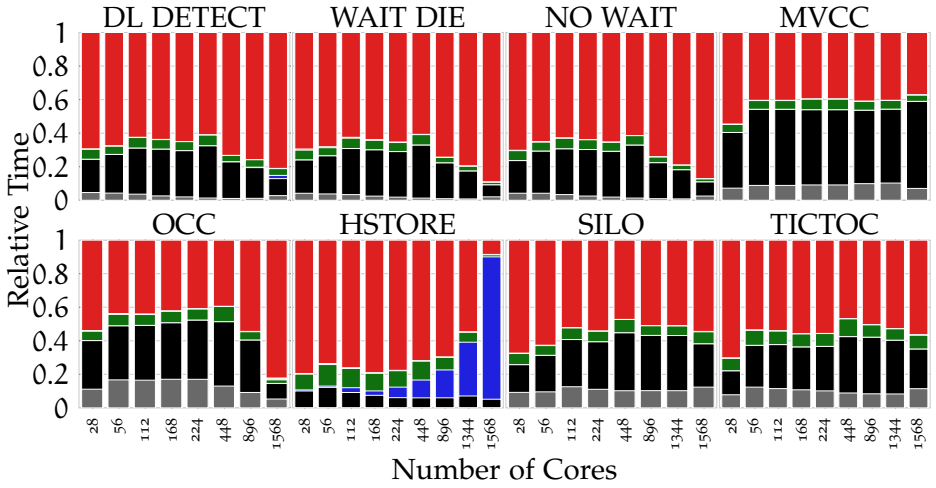
7.5.1 *The Final Results*

With the above optimisations in place, we are finally able to take a clear look at concurrency control for OLTP on large multi-socket hardware. In the following experiment, we repeat our first assessment (cf. Section 7.3) of the concurrency control schemes under OLTP workload with high and low conflict. Though, this time we utilise our optimised implementation and take the full TPC-C schema as well as inserts in the transactions. Notably, we exercise the whole 1568 cores in this experiment, for which we keep the one-to-one relation of cores to warehouses for the low conflict workload (1568 warehouses), as the TPC-C workload induces significant conflict when concurrent transactions exceed the number of warehouses (cf. Section 7.3). Accordingly, Figure 7.10 presents the final throughput for the high conflict and low conflict TPC-C workload. In addition, Figure 7.11 again details the performance of the CC schemes on the multi-socket hardware with time breakdowns for both workloads.

Starting with throughput of the high conflict workload in Figure 7.10a (top row), we again observe similar results as reported in our first



(a) High Conflict, 4 Warehouses



(b) Low Conflict, 1568 Warehouses

Figure 7.11: Breakdown of relative time spent processing TPC-C transactions with optimised DBx1000 using full schema and inserts on multi-socket hardware.

assessment. The many-core simulation and the multi-socket hardware results show different but reasonable behaviour due to the respective hardware characteristics. The only difference is that now our optimisations further offset throughput on the multi-socket hardware. Additionally, we now include the advanced CC schemes *SILO* and *TICTOC* whose peak throughput remarkably outperform the originally covered CC schemes with 4.6 and 5.3 M txn/s, respectively. Yet, those two CC schemes similarly degrade at high core counts converging to the performance of the other CC schemes from 56 cores (>1 socket).

For the other CC schemes, there are minor similarities of the individual throughput curves of the CC schemes between the many-core simulation and the multi-socket hardware. Focusing on the relative performance of the CC schemes other than *SILO* and *TICTOC*, reveals significant improvement of *OCC* and decrease of *MVCC*. Additionally, the pessimistic schemes converge at high core counts only degrading at different points and rates. Finally, *H-Store* still only performs well for small core counts ($\leqslant 4$) and remains slow beyond. Moreover, considering the time breakdown for the high conflict TPC-C workload in Figure 7.11a, we again observe textbook behaviour as in the early time breakdown in Section 7.3.2 with fractions of *wait*, *backoff*, and *abort* characteristic for the individual CC schemes, though the amount of *useful* generally improves and *commit* as well as *CC Mgmt.* decrease through our optimisations.

Next, we analyse the low conflict workload using our optimised implementation. Figure 7.10b reveals that under this workload all CC schemes broadly provide scalable performance with fewer differences as the schemes show in the many-core simulation. That is, up to two sockets the throughput of all CC schemes steeply grows. Then the throughput continues to grow linear up to 1344 cores at a lower growth rate. At the full scale of 1568 cores, the behaviour of the CC schemes differs. *TICTOC*, *SILO*, and *MVCC* make a steep jump reaching 197 M txn/s, 159 M txn/s, and 75 M txn/s, respectively. Also, the growth rate of the pessimistic locking schemes increases but not as much yielding 34 M txn/s for *DL DETECT*, 32 tnx/s for *WAIT DIE*, and 36 M txn/s for *NO WAIT*. *OCC* stays linear achieving 39 M txn/s. In contrast, *HSTORE* degrades from 59 M txn/s at 1344 cores to 35 M txn/s at 1568 cores.

Now with this clear view, we can make out different characteristics of the CC schemes on the large multi-socket hardware, that are visible in their throughput as well as in their time breakdown as shown in Figure 7.11b. Under high conflict, the schemes *SILO* and *TICTOC* are the clear winners, although they do also not scale to high core counts (similar to the other schemes). Under low conflict, *HSTORE* performs the best before the number of concurrent transactions (cores) equals the number of partitions (warehouses). *HSTORE* degrades beyond this point, due to its simple but coarse partition locking, which is identical

to the behaviour in the simulation. In detail, HSTORE's sensitivity to conflicts becomes obvious in the steep increase of *wait* time in the time breakdown.

Under low conflict, *TICTOC* follows as second fastest with *SILO* close by. Both provide significantly lower throughput than *HSTORE* until the tipping point at 1344 cores from which they outperform *HSTORE* by a large margin due to efficient fine-grained coordination, as indicated by their stable amount of *Commit* and *CC Mgmt*. For the other CC schemes, the view is diverse as their relation changes with the NUMA distance between the participating cores. After exceeding 8 sockets (448 cores/2 chassis) the pessimistic schemes fall behind the advanced optimistic CC schemes (*TICTOC* & *SILO*) and eventually also behind *OCC* and *MVCC*. This degrading is unrelated to conflicts (no *wait* time) but correlates with increasing NUMA distances. Consequently, for the low conflict OLTP workload, it appears that pessimistic locking is beneficial when access latencies (NUMA effects) are low, whereas the temporary copies of optimistic CC can hide these latencies, but these temporary copies come at the cost of additional data movement, slowing down throughput at close NUMA distance. To this end, *HSTORE* and *TICTOC* implement these two approaches as well, but they are more efficient, e.g., as *HSTORE* locks less frequently. Notably, there is no difference among the pessimistic CC schemes with different mechanisms against deadlocks, as there is low conflict in the workload, and thus few deadlocks.

Insight: After spending considerable engineering effort bringing state-of-the-art in-memory design to DBx1000, we shed new light on concurrency control on 1000 cores. First, we unveil remarkable peak throughput of the newer CC schemes, *TICTOC* and *SILO*, on high conflict workload, while also presenting textbook behaviour of all CC schemes in the time breakdown. Second, we brighten the grim forecast of concurrency control on 1000 cores for low conflict workload from the simulation of [206]. In fact, under low conflict all CC schemes scale nearly linearly to 1568 cores with a maximum of 200 million TPC-C transactions per second.

7.6 DISCUSSION AND CONCLUSION

In this paper, we analysed in-memory DBMS on today's large multi-socket hardware with 1568 cores, revisiting the results of the simulation in [206], which led us to several surprising findings: (1) A first attempt of running their prototype on today's multi-socket hardware presented broadly different behaviour of the CC schemes. To our surprise, the low contention TPC-C workload with at most one warehouse per thread revealed most concurrency schemes not only stopped scaling after 200 cores but also were very inefficient spending

not even half of their time on useful work. (2) Based on these results, we decided to take a second deeper look into the underlying causes and made several additional discoveries. First, DBx1000 uses atomic increments to create timestamps in the default setting. This was a major bottleneck on the multi-socket hardware. Second, the default benchmark settings of DBx1000 used a TPC-C database which was significantly reduced in size and did not implement any inserts in the transactions. Changing these default setting shifted the picture of our initial assessment completely: while replacing the atomic counter with a hardware clock removed the timestamp creation bottleneck, enabling the original database size and insert operations, however, led to an even darker picture as in our first look. In this second look, we saw that all CC schemes completely collapsed when scaling to more than 200 cores, resulting in devastating 0.5 million txn/s when all cores were used. (3) Finally, we took this challenge and spent significant engineering efforts on the DBx1000 code base to optimise all components from memory management over transaction scheduling to locking. This cleared up the dark skies we faced before and allowed most CC schemes to perfectly scale, providing up to 200 million txn/s on 1568 cores. Even more surprisingly, now, the CC schemes behave very similar with no clear winner.

We speculate that this is due to the fact that most schemes are now memory-bound, emphasising the need to invest in latency hiding techniques such as interleaving with coroutines [101, 144] as an intriguing direction for future work on scalable concurrency control. Having cleared the view on concurrency with this re-evaluation on large hardware, fundamental optimisations like hardware-awareness of OLTP architecture [141] or even adaptive architectures [22] appear exciting for further evaluation of DBMS on such hardware. Also, now that hardware is available, evaluating not only broad concurrency but also utilisation of the full memory capacities is an interesting avenue towards hundred-thousands of TPC-C warehouses on in-memory DBMS.

So, stay tuned for "Part 2 on The Tale of 1000 Cores" :).

We would like to express our great gratitude to the authors of [206] for providing DBx1000 as open source making this work possible.

8

THE FULL STORY OF 1000 CORES: AN EXAMINATION OF CONCURRENCY CONTROL ON REAL(LY) LARGE MULTI-SOCKET HARDWARE

ABSTRACT

In our initial DaMoN paper [17], we set out the goal to revisit the results of "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [206]. Against their assumption, today we do not see single-socket CPUs with 1000 cores. Instead, multi-socket hardware is prevalent today and in fact offers over 1000 cores. Hence, we evaluated concurrency control (CC) schemes on a real (Intel-based) multi-socket platform. To our surprise, we made interesting findings opposing results of the original analysis that we discussed in our initial DaMoN paper [17]. In this paper, we further broaden our analysis, detailing the effect of hardware and workload characteristics via additional real hardware platforms (IBM Power8 and 9) and the full TPC-C transaction mix. Among others, we identified clear connections between the performance of the CC schemes and hardware characteristics, especially concerning NUMA and CPU cache. Overall, we conclude that no CC scheme can efficiently make use of large multi-socket hardware in a robust manner and suggest several directions on how CC schemes and overall OLTP DBMS should evolve in future.

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the peer-reviewed work *The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware* by Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig in *The VLDB Journal*. The contributions of the author of this dissertation are summarized in Part I Chaper 3 Section 3.1.

This version of the article has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s00778-022-00742-4

8.1 INTRODUCTION

We are now 8 years after "Starring into the Abyss [...] of Concurrency Control with [1000] Cores" [206], which presented an evaluation of concurrency schemes for in-memory databases on simulated hardware. The speculation of the authors at that time was that today we would see single CPUs with 1000s of cores. However, so far reality is different [83, 96, 97, 112]. Instead, multi-socket hardware indeed offering 1000s of cores made their way into production data centres. Accordingly, in-memory DBMS are facing not only challenges of massive thread-level parallelism, such as coordination of hundreds of concurrent transactions as predicted by [206], but multi-socket systems also expose in-memory DBMS to further challenges, such as deep NUMA topologies connecting all CPUs [82, 84, 96, 97].

In this paper, we thus set out the goal to bring in-memory DBMS to 1000 cores on today's multi-socket hardware, revisiting the results of the simulation of [206] based on the original code, which the authors generously provide as open source. That is, we follow up on [206] with an evaluation of the characteristics of concurrency control (CC) schemes on real production hardware using their DBx1000 as a starting point. As the main contribution, we provide an extensive analysis of CC schemes on real large hardware, which beyond related evaluation works [10, 79, 165, 179, 204, 206] provides a breadth of insights for OLTP on modern multi-socket hardware platforms, as discussed below. Moreover, as another contribution of this paper we have released all artefacts [19, 20] (code and measurements) for further analysis by the database community. While we already provide an extensive analysis of our data, we believe that the data itself is an interesting source for future findings simply by analysing the data even further.

Part One: This part is based on the results of our recent DaMoN paper [17] where we analysed concurrency control schemes on an Intel multi-socket hardware with 1568 cores. To our surprise, we made several interesting findings: (1) Overall, running the DBx1000 open source prototype of [206] on today's production hardware revealed a very different picture compared to prior observations on simulated hardware with 1000 cores. While simulations indeed are valuable for early path finding, today's real hardware and progress of state-of-the-art have changed the prospect for OLTP on 1000 cores. (2) In a "deeper look", we additionally revisited the limitations and assumptions of the simulation for our real hardware. For example, we found that hardware-assisted timestamp allocation indeed available today has ambiguous benefits, as physical contention shifts rather than disappears. Moreover, aspects of real systems (e.g., memory management) have proven significant performance impact apart from concurrency control. (3) Based on these findings, we revised the original prototype for large multi-socket hardware and finally the experimental results

gave a clear view on concurrency control with 1000 cores, i.e., good scaling of all CC schemes under low conflict and textbook behaviour under high conflict though with thrashing.

Part Two: This new part extends our DaMoN paper [17] significantly and broadens the evaluation in two dimensions: hardware and workload. First, we additionally include two IBM Power-based platforms (Power8 and Power9), which come with different hardware characteristics on the macro-level (e.g., their overall topology) as well as the micro-level (e.g., their simultaneous multithreading implementation). The focus of this part is on singling out the effects of hardware characteristics on the CC schemes (e.g., of different NUMA topologies, cache capacities). Second, we also extended our evaluation in terms of the workload. While in the first part, we only used the common limited transactions mix of the TPC-C benchmark that was available in DBx1000, in part two we also analyse how the full TPC-C transaction mix effects concurrency control on our large multi-socket hardware. The most compelling findings of our deep dive into hardware and workload characteristics are: (1) We could identify clear connections between the performance of the individual CC schemes and specific hardware characteristics. For example, NUMA had an outstanding but nuanced effect on the CC schemes. (2) The significance of hardware characteristics like NUMA effects further depends on the workload. For example, a larger footprint of the workload (accessed tuples) increases the bandwidth demand of the optimistic concurrency control (OCC) scheme and thus OCC scales as long as the underlying NUMA architecture offers sufficient bandwidth. (3) Under high conflict no CC scheme achieves high concurrency on any hardware platform. Our analysis here surfaced inherent issues of today's transaction execution, i.e., increasing inter-transaction parallelism under high conflict will not help without changing the CC schemes and execution schemes.

Overall, our evaluation exhibits the complex interaction of the system design, the workload, and the underlying hardware that determines DBMS performance. Therefore, we recommend reflecting on concurrency in OLTP DBMSs, to put available hardware resources to effective use. Especially with hundreds to thousands of cores, we need broader options for utilising those, apart from executing more concurrent transactions and comprehensive contention management is imperative. Finally, we advocate for performance models to aid exploration of system performance and adaptive DBMS designs towards robust performance in any condition.

Outline: We first provide the background and present the different hardware platforms used in this paper (Section 8.2). In Section 8.3 we then present the results of part one which is based on [17], as mentioned before. Afterwards, in Section 8.4 we present the results of part two which includes the findings of our broadened evaluation with additional hardware platforms and the full TPC-C benchmark.

Finally, we conclude with a summary and discussion of the overall findings in Section 8.5.

8.2 BACKGROUND AND SETUP

In the following, we provide a brief overview of the concurrency control (CC) schemes, the hardware as well as the benchmarking environment used in our evaluation.

8.2.1 *Concurrency Control Schemes*

Table 8.1 summarises the evaluated CC schemes. They range from lock-based CC, with diverse mechanisms against deadlocks, to timestamp-ordering-based CC, including multi-versioning, 2-versioning, coarse locking, and advanced ordering. For details on these CC schemes, we refer to their original publications [24, 25, 103, 111, 192, 208] and to [206]. The first 7 CC schemes in Table 8.1 correspond to the prior evaluation in [206]. We further include the more recent schemes SILO [192] and TICTOC [208], originally not included. Unfortunately, TIMESTAMP [24] from [206] has a fatal bug in the latest version of the prototype, so we excluded it.

DL DETECT	2PL with deadlock detection [24]
NO WAIT	2PL with non-waiting deadlock prevention [24]
WAIT DIE	2PL with wait-and-die deadlock prevention [24]
MVCC	Multi-version T/O [25]
OCC	Optimistic concurrency control [111]
HSTORE	T/O with partition-level locking [103]
SILO	Epoch-based T/O [192]
TICTOC	Data-driven T/O [208]

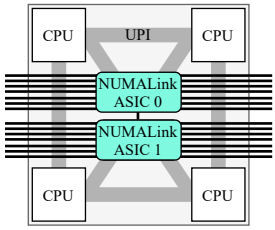
Table 8.1: Evaluated concurrency control schemes.

8.2.2 *Today's Real Hardware with 1000 Cores*

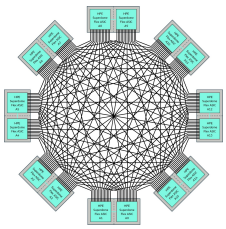
The prevalent hardware in production today offering 1000 cores are large multi-socket hardware platforms [83, 96, 97, 112]. Rather than hosting many cores on a single CPU, these multi-socket platforms connect many CPUs to a single system. In the following, we first introduce the Intel-based HPE platform used in the comparison to the simulation and the later comparison of different real multi-socket platforms. Then we introduce two further IBM platforms. Notably, we choose these three specific platforms for comparing large multi-socket hardware with diverse characteristics, especially different NUMA topologies. In

general, we expect systems with similar NUMA topologies to have similar NUMA effects.

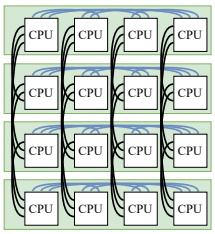
INTEL-BASED HPE PLATFORM Our *HPE SuperdomeFlex* system [83], used for part one of our evaluation (Section 8.3), contains 28 *Intel Xeon 8180* CPUs each having 28 physical cores with SMT-2 [132]. This makes a total of 1568 logical cores (hardware threads), as shown in Table 8.2a. Figures 8.1a-b show how this system groups 4 CPUs into hardware partitions (chassis) [82] and then joins these [84], forming a single cache coherent system with the total of 1568 logical cores and 20 TB of DRAM. As shown in Figure 8.1a, within the chassis each CPU connects to two neighbouring CPUs and to a *NUMALink* controller via UPI links. In turn, the *NUMALink* controllers couple all chassis in a fully connected topology (Figure 8.1b), yielding 4 levels of NUMA with performance properties summarised in Table 8.2c[1].



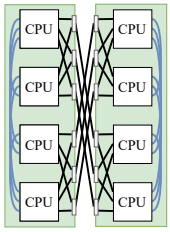
(a) Chassis topology HPE



(b) NUMALink topology HPE



(c) 16-socket IBM Power9



(d) 8-socket IBM Power8

Figure 8.1: System topologies of the HPE, Power9, and Power8 platforms. [82, 84, 168, 195]

Comparing this hardware to potential many-core hardware as simulated in [206] reveals that this multi-socket setup for 1000 cores differs in many aspects. Importantly, one similarity of today's hardware to the simulated architecture of [206] is that both communicate and share cache in a non-uniform manner via a 2D-mesh on the chip [67] (and UPI beyond), such that the cores use the aggregated capacity to cache data but need to coordinate for coherence. This non-uniform communication is an important hardware characteristic, as it can amplify the

1 NUMAPerf is a cross-platform tool in HCMT [159] that performs similar tests like Intel MLC [198]. Different from MLC it provides comparable results across platforms. While it clearly does not implement platform-specific optimisations and thus observed performance can be lower, NUMAPerf allow us to compare the performance across platforms.

Platform	CPUs	∗ Phy. Cores	∗ SMT	= Logical Cores
HPE	28	∗ 28	∗ 2	= **1568**
Power9	16	∗ 12	∗ 8	= **1536** (1504)
Power8	8	∗ 12	∗ 8	= **768** (752)

(a) Number of CPUs, physical cores, and logical cores. In parentheses is the number of logical cores available to the application on Power8 and Power9.

Platform	L1 Inst., Data	L2	L3	Agg.
HPE	32, 32 (16, 16)	1024 (512)	1408 (687.5)	2432 (1216)
Power9	64, 64 (4, 4)	512 (64)	10240 (1280)	10752 (1344)
Power8	32, 64 (2, 4)	512 (64)	8192 (1024)	8704 (1088)

(b) Cache capacity in KB per physical core and in parentheses per logical core. Agg. denotes the aggregated cache capacity of the L2 and the non-inclusive L3 cache. [132, 168, 169]

Platform	NUMA Distance	Latency	Bandwidth
HPE	0: Local	97 ns	101 GB/s
	1: 1 Hop	226 ns	16 GB/s
	2: 2 Hop	260 ns	16 GB/s
	3: Remote	380 ns	12 GB/s
Power8	0: Local	117 ns	193 GB/s
	1: 1 Hop	142 ns	28 GB/s
	2: *n/a*		
	3: Remote	260 ns	43 GB/s
Power9	0: Local	118 ns	148 GB/s
	1: 1 Hop	214 ns	39 GB/s
	2: *n/a*		
	3: Remote	361 ns	90 GB/s

(c) Memory access latency and bandwidth by NUMA distance measured with NUMAPerf[1] [159]. HPE has a deeper topology than the Power platforms. Distance 0 (*Local*) refers to memory directly located the CPU, 1-2 (*1 Hop*, *2 Hop*) refer to memory of neighbouring CPUs within the same chassis, and 3 (*Remote*) refers to accesses across chassis.

Table 8.2: Properties of evaluated hardware platforms.

impact of contention points in the CC schemes on any large hardware (multi-socket and many-core). Otherwise, the simulation differs from today's hardware, since it assumed low-power and in-order processing cores clocked at 1GHz, cache hierarchies with only two levels, and cache capacities larger than today's caches. Notably, it simulates the DBMS in isolation without an OS, disregarding overheads and potential side effects of OS memory management, scheduling etc., omitting essential aspects of real systems [59, 145].

POWER-BASED IBM PLATFORMS In the second part of our evaluation, we consider the IBM Power platforms as prominent hardware platforms for scale-up systems, in addition to the Intel-based platform. There are several distinctive features, making these IBM Power platforms an interesting alternative for our analysis [96, 97, 168].

In particular, we use an IBM Power system E880 (Power8) [195] configured with 8 Power8 CPUs and an IBM Power system E980 (Power9) [196] configured with 16 Power9 CPUs. As outlined in Table 8.2a, these platforms offer a total of 768 and 1536 logical cores (hardware threads), of which 752 and 1504 are available to applications. Both systems provide 16 TB of memory.

The IBM Power CPUs use a RISC-based instruction set architecture (ISA) unlike Intel CPUs, though both types of CPUs share many features such as vector instructions, support of hardware transactional memory, and simultaneous multithreading (SMT). Notably, IBM Power CPUs realise configurable levels for SMT exposing 1 to 8 logical cores (hardware threads) per physical CPU core. In our experiments, we use SMT-8 if not noted otherwise. Therefore, the 12 physical cores in either Power processor provide up to 96 logical cores (hardware threads). We remark that both IBM Power platforms reserve one physical core on some CPUs (2 on our Power8 and 4 on our Power9) to manage logical hardware partitions (LPARs), e.g., governing storage and other periphery. All other cores and memory resources are available for our evaluation without restrictions and any resource sharing.

In the memory hierarchy caches and the NUMA hierarchy differ in important aspects between the two Power platforms as well as the Intel-based platform. The cache capacities in Table 8.2b reveal the large L3 caches per physical core for Power9 and Power8. The Power9 processor contains the largest L3 cache per physical core. Additionally, higher overall cache performance is claimed on Power9 due to increased associativity (20-way in Power9 vs. 8-way on Power8) [168]. Notably, the IBM systems have a L4 cache on their custom DRAM DIMMs. However, this is a memory buffer outside of the processor rather than a CPU cache, thus unlike CPU caches this L4 cache does not hide NUMA effects.

Like the Intel-based platform, the IBM Power systems follow a NUMA architecture to scale to more than 1000 logical cores. In Ta-

ble 8.2c, we summarise the respective latency and bandwidth of memory accesses alongside the Intel-based platform. As shown in Figure 8.1c, our 16-socket Power9 system features a NUMA topology with one hop to sockets in the same chassis and two hops to sockets in its other three chassis, whereas our smaller 8-socket Power8 system can afford a fully connected NUMA topology directly connecting every socket between its two chassis. Additionally, both systems have faster interconnects within the chassis than between the chassis, therefore still establishing three NUMA level on both Power systems.

Notably, a larger Power8 system with 16 sockets would have a similar topology to our Power9 and inversely smaller versions of the HPE and Power9 systems would similarly benefit from stronger connections.

8.2.3 *Benchmarking Environment*

In this paper, we evaluate the CC schemes mentioned before on our multi-socket hardware with the TPC-C benchmark [185] as implemented in the latest version of DBx1000 [207]. This version of DBx1000 includes the extended set of CC schemes as mentioned before and bug fixes, beyond the version used in the original paper [206]. Additionally, we rely on the provided embedded instrumentation to measure the time spent in the system.

For running the benchmarks, we use the given default configuration of DBx1000. This configuration defines the TPC-C workload as equal mix of *New-Order* and *Payment* transactions covering 88% of TPC-C with standard remote warehouse probabilities (1% and 15%). This configuration partitions the TPC-C database by warehouse (WH) ID for all CC schemes. Based on this configuration, we specify 4 warehouses for the high conflict TPC-C workload and 1024 or 1568 warehouses for the low conflict workload, as in our initial DaMoN paper [17]. Similar to the original evaluation, each benchmark runs until the first transaction executor has committed 100 K transactions, and we measure the throughput as the number of transactions committed by all transaction executors in that time. We observed this method to provide reliable measurements despite NUMA effects in our large system or other effects influencing the execution of the benchmark.

An interesting first observation was, that DBx1000's TPC-C did not implement insert statements, presumably due to the mentioned limitations of the simulator, e.g., memory capacity and no OS. In part one of the paper, we thus first start with the very same setup, but later we enable insert statements in the evaluation after taking a first look at the CC schemes. As minor extension, we added a locality-aware thread placement strategy to DBx1000 for all experiments in this paper. It exclusively pins DBMS threads to a specific core. For scaling the DBMS threads in our experiments, we use the minimal number of

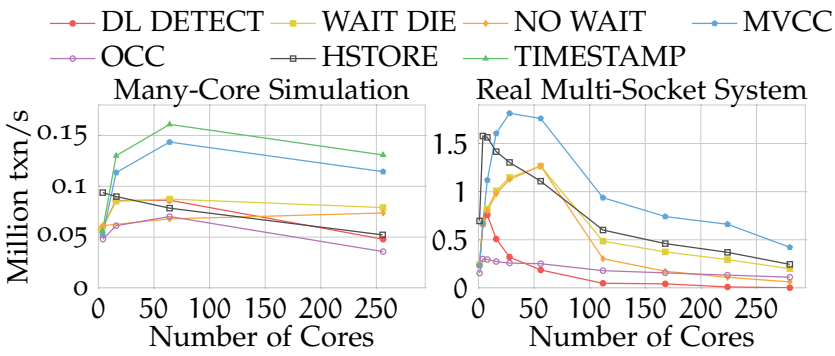


Figure 8.2: Throughput of TPC-C high conflict workload (4 WH) in original simulation [206] and on real multi-socket hardware (Intel, HPE).

sockets to accommodate the desired resources, e.g., 2 sockets for 112 threads, otherwise OS and NUMA effects would dominate the overall results. Note that as consequence of this thread placement strategy, *cores* and *threads* equally refer to a single execution stream (i.e., a worker) of the DBMS.

For the broader evaluation in part two, we use a setup similar to the optimised DBx1000 from part one as discussed before. Notably, we aim to match the thread placement on the additional hardware platforms with the placement we used on HPE. Moreover, throughout part two, TPC-C is configured with the full TPC-C schema and transactions always execute their insert statements. Finally, for the last experiments in part two, we extend DBx1000 to support the remaining TPC-C transactions for the full TPC-C benchmark. For all other experiments in part two, we use the more narrow mix of *New-Order* and *Payment* transactions only, as was the case for the original simulation.

8.3 PART ONE: SIMULATION VS. REAL HARDWARE

8.3.1 *A First Look: Simulation vs. Reality*

We now report the results of running the DBx1000 prototype directly on the multi-socket Intel-based hardware (HPE platform) as opposed to a simulation.

8.3.1.1 *The Plain Results*

Figures 8.2 and 8.3 display the throughput of TPC-C transactions for 4 warehouses and 1024 warehouses, i.e., high and low conflict OLTP workloads. On the left of each figure are the original simulation results [206] and on the right are our results on real multi-socket hardware (Intel-based HPE). We first compare the overall throughput. Then we break down where time is spent in the DBMS to better understand our observations.

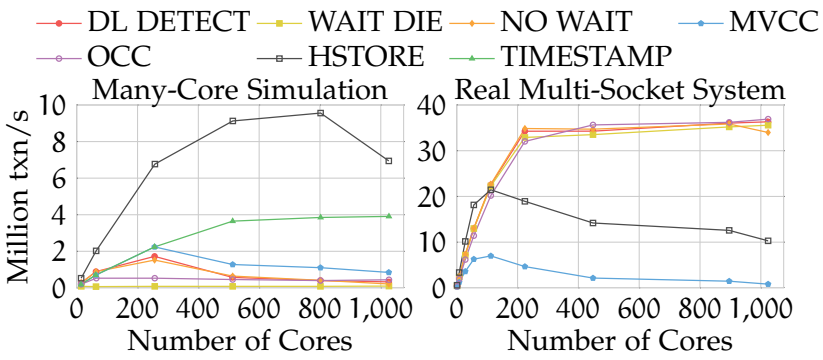


Figure 8.3: Throughput of TPC-C low conflict workload (1024 WH) in original simulation [206] and on real multi-socket hardware.

We first look at the results for 4 warehouses as shown in Figure 8.2. Overall, it is obvious that the absolute throughput differs due to the characteristics of the CPUs in the simulation and our hardware, e.g., low-power 1 GHz cores versus high-power 2.5 GHz cores. This is expected and therefore only the relative performance of the CC schemes matters. In the following, we discuss similarities and significant differences.

First, comparing the simulation and the real hardware (Intel-based HPE platform) in Figure 8.2, we see that the CC schemes HSTORE, MVCC, and NO WAIT show similar trends. That is, these CC schemes have a similar thrashing point in the simulation and the real hardware, i.e., HSTORE at 4 to 8 cores and MVCC as well as NO WAIT at 56 to 64 cores. After the respective thrashing point, these CC schemes degrade steeper on the multi-socket hardware, which can be linked to the additional NUMA effect of the multi-socket hardware appearing beyond 56 cores. For the other CC schemes the results for the simulation and real hardware differ more widely, especially the diverging behaviour of the pessimistic CC schemes sticks out. Considering these pessimistic CC schemes, DL DETECT behaves broadly different already degrading at 8 cores rather than 64 cores and WAIT DIE performs surprisingly close to NO WAIT. In Section 8.3.1.2, we analyse the time breakdown of this experiment to explain these results. It reveals characteristic behaviour of the individual CC schemes, despite the diverging throughput in the simulation and the multi-socket hardware.

Next, we look at the low conflict TPC-C workload (1024 warehouses) in Figure 8.3. The results here present fewer similarities of the many-core simulation and the multi-socket hardware (Intel-based HPE), i.e., only the slope of MVCC is similar. Additionally, DL DETECT and NO WAIT stagnate at high core counts (>224) in the simulation and on the multi-socket hardware. In contrast, HSTORE performs worse on the multi-socket hardware than in the simulation. It is slower than the pessimistic CC schemes and OCC from >112 cores. Also, OCC and WAIT DIE achieve higher throughput on the multi-socket hardware,

Useful	Time usefully executing application logic and operations on tuples.
Abort	Time rolling back and time of wasted useful work due to abort.
Backoff	Time waiting as backoff after abort (and requesting next transaction to execute).
Ts. Alloc.	Time allocating timestamps.
Index	Time operating on hash index of tables including latching.
Wait	Time waiting on locks for concurrency control.
Commit	Time committing transaction and cleaning up.
CC Mgmt.	Time managing concurrency control other than prior categories, e.g., constructing read set.

Table 8.3: Time breakdown categories.

now similar to DL DETECT and NO WAIT. Moreover, MVCC is significantly slower than OCC and the pessimistic CC schemes, due to high overheads of this scheme as we discuss next.

Insight: The initial comparison of concurrency control schemes on 1000 cores presents only minor similarities between the simulation and our multi-socket hardware with surprising differences in the behaviour of the CC schemes mandating further analysis.

8.3.1.2 *First Time Breakdown on Intel-Based Hardware*

For deeper understanding of the observed behaviour of the CC schemes, we now break down where time is spent in processing the TPC-C transactions on the multi-socket hardware. Therefore, we apply the breakdown of [206] categorising time as outlined in Table 8.3. For each CC scheme, Figures 8.4 and 8.5 break down the time spent relative to the total execution time of the TPC-C benchmark with a bar for each core count.

The time breakdown of 4 warehouses in Figure 8.4 neatly shows the expected effect of conflicting transactions and aborts for increasing core counts under high conflict workload. That is, most CC schemes result in high proportions of *wait*, *abort*, and *backoff* as soon as the number of cores exceeds the number of warehouses (>4 cores), yielding nearly no *useful work* at higher core counts. Only the *wait* time of HSTORE grows at 4 cores concurrently executing transactions, such that HSTORE appears more sensitive to conflicts.

Remarkably, textbook behaviour of the specific schemes becomes visible in the breakdown: Starting with DL DETECT, its *wait* time increases with the number of concurrent transactions as expected, following the increasing potential of conflicts between concurrent

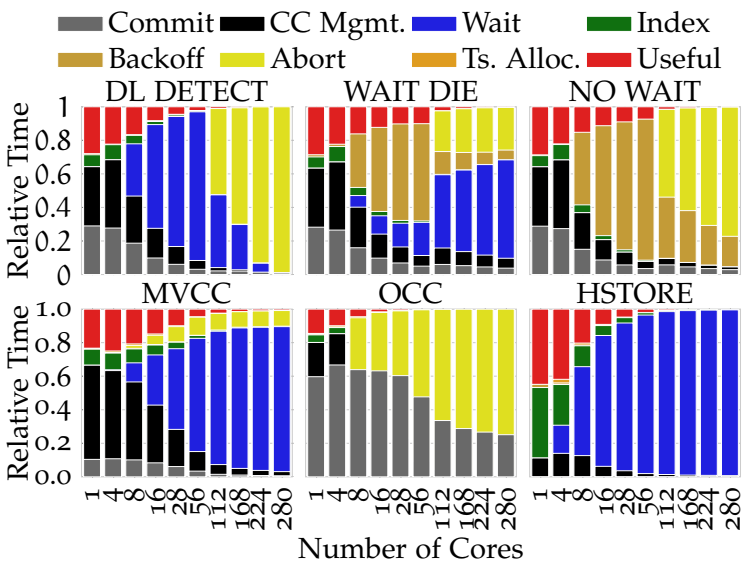


Figure 8.4: Breakdown of relative time spent for high conflict (4 WH) TPC-C transactions on multi-socket hardware (Intel, HPE).

transactions. Different from DL DETECT, WAIT DIE spends more time *backing off* and *aborting* due to its characteristic aborts after a short wait time (small wait proportion). Instead, NO WAIT solely *backs off* without waiting, spending even more time on *aborted* transactions. The optimistic MVCC waits on locks during validation, such that its breakdown shows similar *wait* times like DL DETECT. Finally, for OCC we can see that the high *abort* portion reflects its sensitivity to conflicts while the high *commit* portion stems from high costs for cleaning up temporary versions.

Having observed this "expected" behaviour of the CC schemes under high conflict, we now analyse the unexpected behaviour under low conflict as shown in Figure 8.5. Against the expectation, most CC schemes spend considerable amount of time to manage concurrency (black and grey area) such as lock acquisition (except HSTORE which we discuss later). For these schemes this results in at most 50% of useful work (red area). Staggeringly, MVCC which actually should perform well under low conflicting workloads, spends almost no time with *useful work* despite the low conflict in the workload, i.e., <10% *useful work* from 224 cores. In fact, the low conflict is visible in the overall little time spent *waiting* or *aborting*. Consequently, the slowdown compared to pessimistic CC schemes does not stem from wasted work but from internal overhead in execution of this CC scheme under high core counts.

In contrast, we observe for HSTORE an increasing impact of *timestamp allocation* and *waiting time*. While *timestamp allocation* is used by the other schemes as well, the relative overhead for HSTORE is the highest due to its cheaper lock acquisition. In fact, the authors of [206]

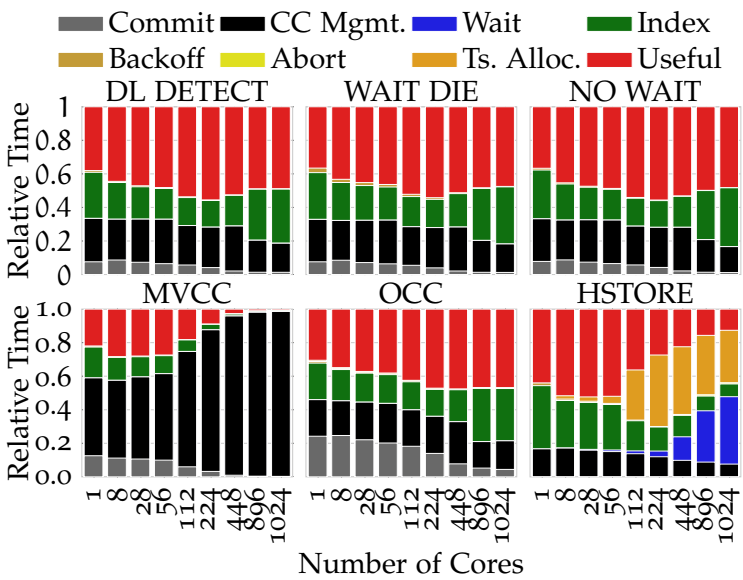


Figure 8.5: Breakdown of relative time spent for low conflict (1024 WH) TPC-C transactions on multi-socket hardware (Intel, HPE).

did analyse different timestamp allocation methods in their paper but chose *atomic increment* as a sufficiently well performing method that is a generally applicable option when there is no specialised hardware available. However, as we can see this choice is not optimal for multi-socket hardware. Moreover, we attribute the increasing *waiting time* of HSTORE to its coarse-grained partition locking to sequentially execute transactions on each partition. This partition-level locking causes a higher overhead if more cores are used since this leads to more conflicts between transactions as shown in prior work [106, 141].

Insight: The analysed CC schemes behave differently on the real multi-socket hardware than in the simulation of [206]. For the high conflict workload (4 warehouses), the behaviour on real hardware and the simulation appears more similar, for which the time breakdown confirms the expected characteristics for each CC scheme. However, low conflict workload causes an unexpectedly high CC management overhead in most CC schemes and transactions execute only a limited amount of useful work, except for HSTORE where waiting and timestamp allocation dominate.

8.3.2 *A Second Look: Hidden Secrets*

In this section, we now take a "second look" at the factors leading to the surprising behaviour of the CC schemes observed in our initial analysis and discover equally surprising insights.

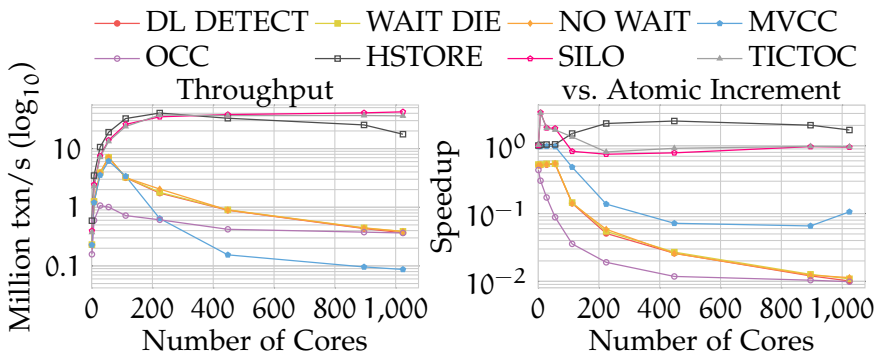


Figure 8.6: Throughput of low conflict TPC-C for timestamp allocation with hardware clock.

8.3.2.1 *Hardware Assistance: The Good?*

In a first step, we analyse the benefit of hardware-assisted timestamp allocation over using atomic counters for the real multi-socket hardware. As explained earlier, the *atomic increment* is generally applicable but may cause contention, which efficient and specialised hardware may prevent if available. Our hardware provides a synchronised hardware clock indeed offering a new option for efficient timestamp allocation, as projected by [206]. Specifically, our Intel processors provide efficient access to a hardware clock on each core [95] (*rdtsc* instruction), which ticks at the same rate on all cores in the entire system (*invariant tsc* feature) and is synchronised across all cores and hardware chassis by the platform firmware, verified by the OS [160, 189–191].

In the following experiment, we analyse the benefit of this hardware assistance for timestamp allocation. Figure 8.6 shows the throughput of the CC schemes for 1024 warehouses with timestamp allocation based on the hardware clock.

On one hand, HSTORE greatly benefits from the hardware clock (as expected) achieving peak throughput of ~40 M txn/s with an overall speedup over atomic increment of up to 3x. We now also include SILO and TICTOC in our results which perform like HSTORE except for high core counts as we discuss in the time breakdown analysis below. On the other hand, the remaining CC schemes (DL DETECT, WAIT DIE, NO WAIT, MVCC, and OCC) degrade drastically when using the hardware clock instead of atomic counters. That is, the pessimistic CC schemes DL DETECT, WAIT DIE, and NO WAIT perform ~50% slower within a socket (0.51 - 0.55x speedup for ⩽56 cores), after which they degrade to 0.01x speedup at 1024 cores (0.37 - 0.39 M txn/s). Likewise, MVCC is stable (~1x speedup) up to 56 cores and its speedup drops to 0.1x when exceeding the single socket. Finally, OCC does not benefit from the hardware clock at all (0.44 - 0.01x speedup).

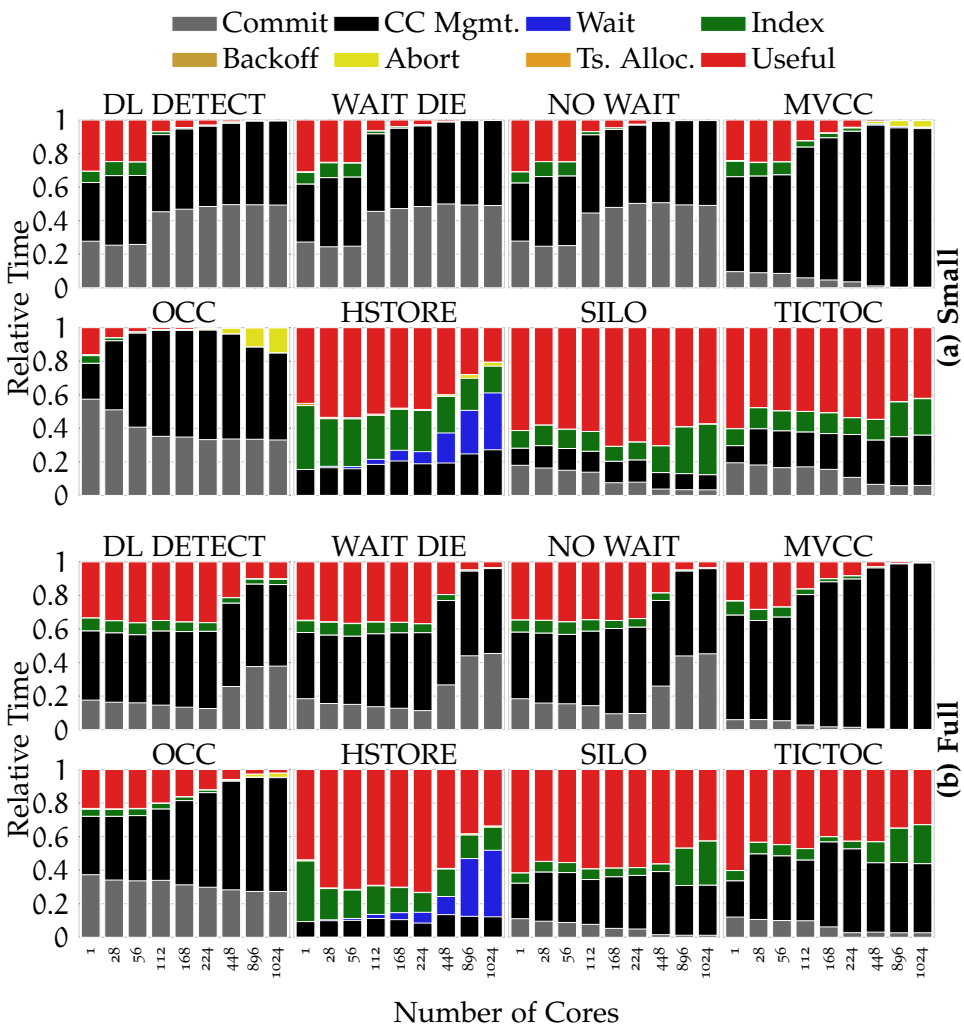


Figure 8.7: Breakdown of relative time spent processing TPC-C transactions on *small* and *full* schema with 1024 warehouses using timestamp allocation via hardware clock.

Overall, timestamp allocation based on the hardware clock drastically changes the perspective on the performance of the CC schemes. Now HSTORE performs best, meeting the initial observations of [206] (joined by SILO and TICTOC), whereas the pessimistic schemes, OCC, and MVCC degrade severely.

For better understanding of these diverse effects of the hardware clock, we again look at the time breakdown shown in Figure 8.7a (top row). As expected, HSTORE now spends little time for timestamp allocation (like SILO and TICTOC). Otherwise HSTORE spends time similarly as with atomic increment, especially with a similar increase of the *waiting time*. Importantly, we do not observe any bias introduced by the hardware clock, since HSTORE (as well as the other CC schemes) spends no significant time aborting and our detailed logs show no outliers for the number of aborted transactions per transaction executor. Consequently, the hardware clock is reliable for timestamp allocation.

An interesting observation is the significant change in the time breakdown of the other CC schemes. For example, DL DETECT, WAIT DIE, and NO WAIT show at least 2x the time spent for *CC Mgmt.* and *committing/cleaning up* (black & grey) with a sudden increase after 56 cores. OCC's increase of time spent in these categories is even more drastic with less than 20% of *useful work* at any core count. Only MVCC changes insignificantly, as *useful* time spent was low already.

Profiling these CC schemes reveals physical contention, that previously was on the atomic counter, now results in thrashing of latches. Previously, the physical contention on the atomic counter has throttled transaction execution including latching, e.g., for lock acquisition. Now, that the hardware clock has removed the physical contention from timestamp allocation, transactions access latches more frequently, indeed reaching their thrashing point despite a latch per row and low conflicts in the workload with 1024 warehouses. Notably, our profiling reveals further details of the individual CC schemes: The *pthread_mutex* employed in DL DETECT, WAIT DIE, NO WAIT, and OCC sharply degrades due to NUMA sensitivity of hardware transactional memory [31] used for lock elision and its fallback to robust but costly queuing synchronisation [70] as well as costly interaction with the scheduler of the OS.[2] In contrast, MVCC uses an embedded flag as spin latch which is not as sensitive to NUMA but also not robust [49]. Hence, this type of latch shows a slower but also continuous degrading of performance.

Insight: Hardware-assisted timestamp allocation via specialised clocks alleviates contention and leads to better scalability for HSTORE (as well as SILO and TICTOC). However, while hardware-assisted clocks also lift the overhead in the other schemes, it does not necessarily improve their overall performance as contention moves and

2 pthread_mutex is specific to *libc* and the OS.

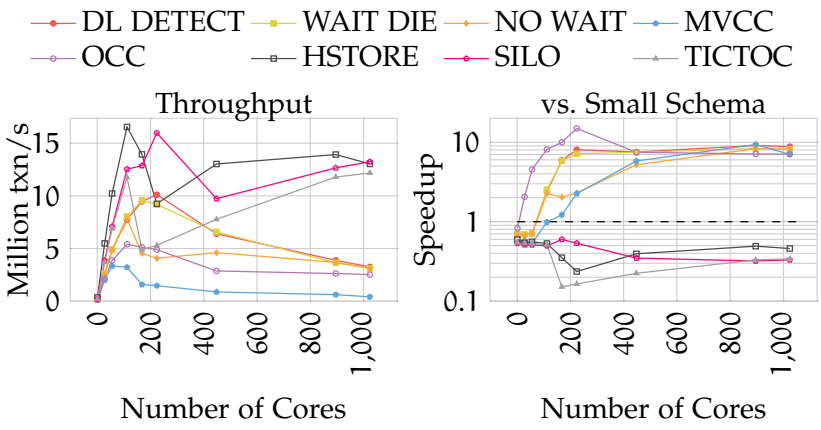


Figure 8.8: Throughput of low conflict TPC-C for small schema size like in the simulation versus full schema size both executed on multi-socket hardware (Intel, HPE).

puts pressure on other components (e.g., latches), even leading to performance degradation.

8.3.2.2 *Data Size: The Bad?*

In the context of this surprisingly high overhead, our second look at the paper [206] brings the following statement to our attention: "Due to memory constraints [...], we reduced the size of [the] database" [206]. Consequently, we are wondering if the staggering overhead is potentially caused by the absence of useful work to execute rather than the abundance of overhead in the CC schemes, due to the reduced data size imposed by limited memory capacity of the simulator in [206].

We revert the benchmark to the full TPC-C database in the following experiment and report on the surprising effect of the larger data volume. In detail, to return to the officially specified database, we increase (1) the cardinality of the item relation from 10 K to 100 K, (2) the factor of customers per warehouse from 20 K to 30 K determining the cardinalities of the customer, order, order-line, and history relations, and (3) we include all attributes rather than only those accessed.

Figure 8.8 shows the throughput for the full schema with 1024 warehouses and speedup in comparison to the small schema based on the previous experiment (cf. Figure 8.6). We measure quite diverse throughput of the CC schemes. Yet, the speedup indicates that two major effects of the increased data volume appear in the same clusters as in the previous experiment but with inverse outcome. The first cluster of HSTORE, SILO, and TICTOC is slower with the full schema, i.e., 0.2-0.6x, 0.3-0.5x, and 0.2-0.5x, respectively. The second cluster, consisting of the previously "slower" CC schemes, improves inversely to the previously described thrashing points. That is, DL DETECT, WAIT DIE, and NO WAIT have a speedup of 0.7x until 56 cores, after which they benefit from the full schema with speedups of 2.4-9.1x,

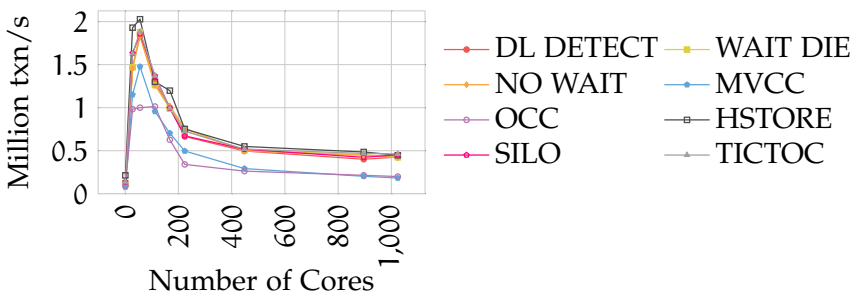


Figure 8.9: Throughput of TPC-C including inserts with full schema size on multi-socket hardware (Intel, HPE).

2.5-8.3x, and 2.0-9.3x, respectively. MVCC has a speedup of 0.5-0.6x until 56 cores, breaks even (1x) at 112 cores, and then improves with a speedup of 1.2-9.3x. OCC has a speedup of 0.8 at 1 core and broadly improves with the full schema with 2.1-14.9x speedup.

The time breakdown in Figure 8.7b (lower row), details the causes. As for the CC schemes in the first cluster, HSTORE has increased *useful* work, while for SILO and TICTOC *CC Mgmt.* increases. Both indicate increased cost of data movement, as HSTORE directly accesses tuples and the other two create temporary copies in the CC manager. The second cluster shows an increase of *useful* work, presenting less staggering overhead of CC management at low core counts. Importantly, the sudden increase of *commit* for DL DETECT, WAIT DIE, and NO WAIT is delayed, indicating that latches thrash only from 448 cores (while previously already from 112 cores). For OCC the time spent on *commit* also decreases with the larger data volume, but the increase of *CC Mgmt.* due to larger temporary copies still diminishes *useful work*. Only for MVCC the time breakdown does not change significantly.

Insight: The effect of larger data volumes in the full schema changes the perspective on the CC schemes again. We attribute our observations to the effects, that heavier data movement slows down data-centric operations (e.g., tuple accesses or copies), which in turn alleviates pressure on latches preventing thrashing.

8.3.2.3 *Inserts: Facing Reality!*

Since the simulator of [206] had limited memory capacity and excluded the simulation of important OS features such as memory management, the TPC-C implementation of DBx1000 did not include insert statements and for comparability we initially excluded these as well. For the last experiment in this section, we now complete the picture of concurrency control on real hardware (Intel).

Accordingly, Figure 8.9 shows the throughput of TPC-C transactions including inserts (as well as all before-mentioned changes) for 1024 warehouses. The inserts drastically reduce throughput of all CC schemes with heavy degradation at the socket boundary (56 cores).

Even more interesting, all CC schemes perform similarly with inserts in the transactions. Indeed, profiling indicates execution of insert statements are the hotspot of the TPC-C transactions now, but the causes are orthogonal to concurrency control. The two major hotspots are (1) catalogue lookups to locate tuple fields and (2) memory allocation for new tuples.

Profiling details show that catalogue lookups cause frequent accesses to L1 and L3 caches. For tuple allocation, profiling details indicate significant time spent in the memory allocator and for OS memory management including page faults. These hotspots are amplified by NUMA in our multi-socket system, since the catalogue is centrally allocated and memory management in Linux is also contention- and NUMA-sensitive [41]. Hence, such impact on performance only becomes visible in its full extent on large systems like ours.

Insight: Inserts significantly affect the performance in this benchmark, though due to hotspots orthogonal to the CC schemes, most notably cache misses of the catalogue and memory allocation.

8.3.3 *Effect of State-of-the-Art-Optimisations*

Finally, we take a last step to provide a clear view on the characteristics of concurrency control on large multi-socket hardware. First, we elaborate on our optimisations to reach this clear view and provide an overview of their individual speedups. Then, we repeat our assessment of the CC schemes using all optimisations.

Notably, in the following experiments, we use the full TPC-C schema as well as inserts in the transactions, and we exercise the whole 1568 cores for the low conflict workload. We maintain the one-to-one relation of cores to warehouses for the low conflict workload, as the TPC-C workload induces significant conflict when concurrent transactions exceed the number of warehouses.

8.3.3.1 *Overview of Optimisations*

To clear the view, we remove previously identified obstacles and optimise the overall system based on state-of-the-art in-memory DBMS for large multi-socket hardware: **(Opt. 1)** We introduce a thread-local memory allocator that pre-allocates memory, as in today's commercial in-memory databases [65]. Importantly, it aligns allocations to cache line boundaries, otherwise false sharing obliterates performance. **(Opt. 2)** We add a flat perfect hash index [106], e.g., reducing pointer chasing and cache misses. **(Opt. 3)** We address latch thrashing with a queuing latch [49, 106, 181] and exponential backoff [90]. **(Opt. 4)** We replicate read-only relations to each socket, utilising faster local memory instead of slow remote memory [106]. **(Opt. 5)** We reorder and prefetch tuple and index accesses to optimise data movement. **(Opt. 6)** We lift expensive query interpretation (e.g., catalogue lookups) to efficient

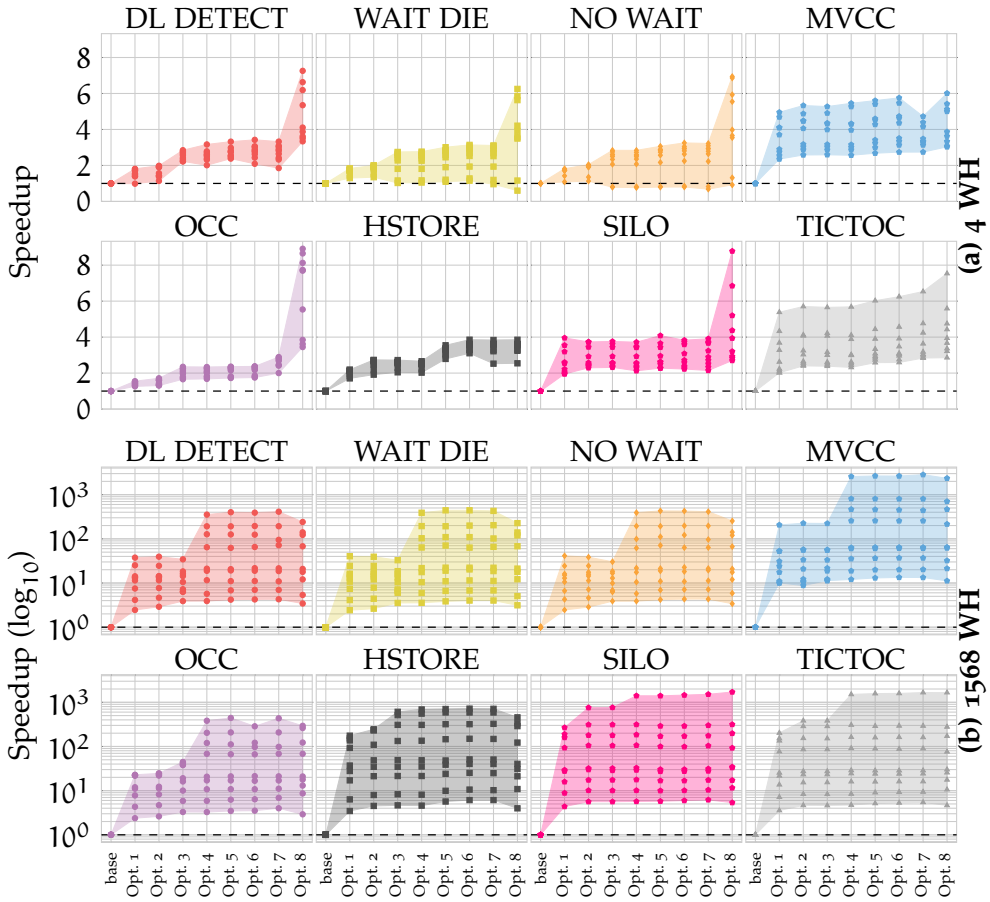


Figure 8.10: Summary of speedup of the CC schemes provided our optimisations Opt. 1-8 described in Section 8.3.3.1 for (a) the high and (b) the low conflict workloads (i.e., 4 and 1568 warehouses). Optimisations are applied one after the other and speedup is reported as the factor of throughput increase over the base implementation.

query compilation as in state-of-the-art in-memory DBMS [65, 105, 183, 187]. **(Opt. 7)** We update the deadlock prevention mechanisms to state-of-the-art [90]. **(Opt. 8)** We eliminate CC overhead for read-only relations.

In Figure 8.10 we report the speedup provided by each optimisation when consecutively adding the optimisations, as the factor of throughput increase over the unoptimized *base* implementation. Note that, we discuss the detailed throughput with all optimisations in place in the next section and the detailed throughput of the individual optimisations is available in [19].

For the high conflict workload, Figure 8.10a shows that the thread-local memory allocator (Opt. 1) and the eliminated CC overhead for read-only relations (Opt. 8) provide significant speedup for all CC schemes with each up to 5.38x and 4.33x. Additionally, the optimised latching (Opt. 3) indeed notably benefits the CC schemes involving heavy latching (pessimistic CC schemes and OCC) with further speedup of up to 2.12x.

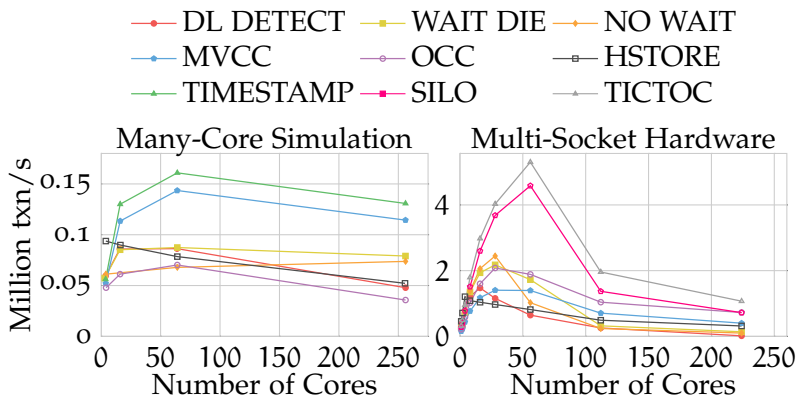
For the low conflict workload, Figure 8.10b shows an even greater speedup for the thread-local memory allocator (Opt. 1), by up to 268x. Additionally, in this low conflict workload the NUMA-aware replication (Opt. 4) proves beneficial with up to 227x speedup, as actual work including record accesses dominates the low conflict workload (rather than concurrency control). Further, there are notable speedups of the other optimisations for distinct CC schemes, e.g., the optimised index (Opt. 2) notably benefits HSTORE, SILO, and TICTOC with up to 2.83x, whereas the optimised latching (Opt. 3) again benefits the pessimistic CC schemes, OCC, and now also HSTORE (up to 2.49x). Finally, the updated deadlock prevention (Opt. 7) significantly benefits OCC with up to 1.71x. The other optimisations show less significant speedups (<1.5x).

Notably, the speedup of the optimisations varies in detail, across the CC schemes, workloads, and number of cores. There are many factors influencing the specific speedup. Their detailed study is beyond the scope of our evaluation of concurrency control.

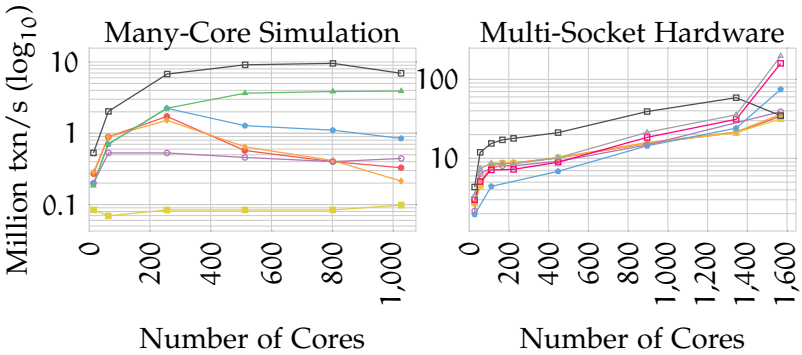
8.3.3.2 *Results after Optimisations*

With the above optimisations in place, we now repeat the detailed assessment of the CC schemes under high and low conflict OLTP workload (as initially in Section 8.3.1). Accordingly, Figure 8.11 presents the throughput of the fully optimised DBx1000 for the high conflict and low conflict TPC-C workloads. In addition, Figure 8.12 again details the performance of the CC schemes on the multi-socket hardware with time breakdowns.

Starting with throughput of the high conflict workload in Figure 8.11a (top row), we again observe similar results as reported in our first assessment. The many-core simulation and the multi-socket hardware



(a) High Conflict, 4 Warehouses



(b) Low Conflict, 1024 Warehouses for Simulation and 1568 for Multi-Socket Hardware

Figure 8.11: Throughput of TPC-C in original many-core simulation [206] without full schema & inserts and our optimised implementation with full schema & inserts on multi-socket hardware (Intel, HPE).

results show different but reasonable behaviour due to the respective hardware characteristics. The only difference is that now our optimisations further offset throughput on the multi-socket hardware. Additionally, we now include the advanced CC schemes SILO and TICTOC whose peak throughput remarkably outperform the originally covered CC schemes with 4.6 and 5.3 M txn/s, respectively. Yet, those two CC schemes similarly degrade at high core counts converging to the performance of the other CC schemes from 56 cores (>1 socket).

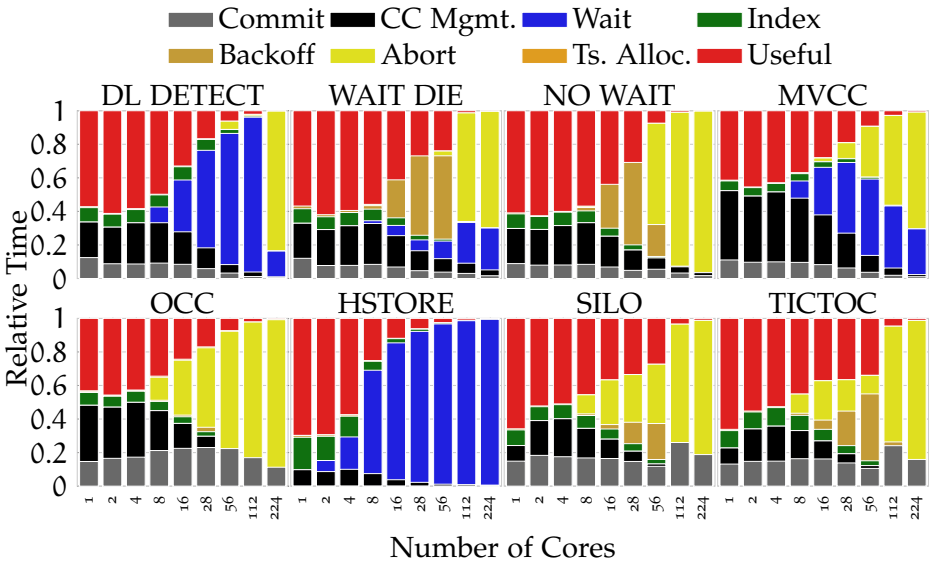
For the other CC schemes, there are minor similarities of the individual throughput curves of the CC schemes between the many-core simulation and the multi-socket hardware. Focusing on the relative performance of the CC schemes other than SILO and TICTOC, reveals significant improvement of OCC and decrease of MVCC. The pessimistic schemes converge at high core counts, only degrading at different points and rates. Finally, HSTORE still only performs well for small core counts ($\leqslant 4$) and remains slow beyond. Moreover, considering the time breakdown for the high conflict TPC-C workload in Figure 8.12a, we again observe textbook behaviour as in the early time breakdown in Section 8.3.1.2 with fractions of *wait*, *backoff*, and *abort*

characteristic for the individual CC schemes, though the amount of *useful* generally improves and *commit* as well as *CC Mgmt.* decrease through our optimisations.

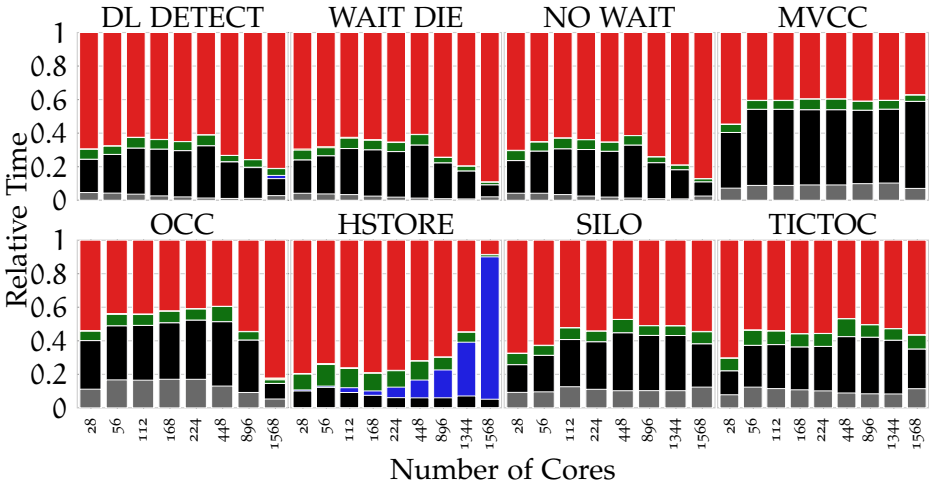
Next, we analyse the low conflict workload using our optimised implementation. Figure 8.11b reveals that under this workload all CC schemes broadly provide scalable performance with fewer differences as the schemes show in the many-core simulation. That is, up to two sockets the throughput of all CC schemes steeply grows. Then the throughput continues to grow linearly up to 1344 cores at a lower growth rate. At the full scale of 1568 cores, the behaviour of the CC schemes differs. TICTOC, SILO, and MVCC make a steep jump reaching 197 M txn/s, 159 M txn/s, and 75 M txn/s, respectively. Also, the growth rate of the pessimistic locking schemes increases but not as much, yielding 34 M txn/s for DL DETECT, 32 txn/s for WAIT DIE, and 36 M txn/s for NO WAIT. OCC stays linear achieving 39 M txn/s. In contrast, HSTORE degrades from 59 M txn/s at 1344 cores to 35 M txn/s at 1568 cores.

Now with this clear view, we can make out different characteristics of the CC schemes on the large multi-socket hardware, visible in their throughput and time breakdown (Figure 8.12b). Under high conflict, the schemes SILO and TICTOC clearly excel, although they neither scale to high core counts (similar to the other schemes). Under low conflict, HSTORE performs the best until the number of concurrent transactions (cores) equals the number of partitions (warehouses). Beyond this point it degrades, due to its coarse partition locking, similarly observed in the simulation. HSTORE's sensitivity to conflicts becomes obvious in the steep increase of *wait* time in the time breakdown.

Under low conflict, TICTOC follows as second fastest with SILO close by. Both provide significantly lower throughput than HSTORE until the tipping point at 1344 cores from which they outperform HSTORE by a large margin due to efficient fine-grained coordination, as indicated by their stable amount of *Commit* and *CC Mgmt.* For the other CC schemes, the view is diverse as their relation changes with the NUMA distance between the participating cores. After exceeding 8 sockets (448 cores/2 chassis) the pessimistic schemes fall behind the advanced optimistic CC schemes (TICTOC & SILO) and eventually also behind OCC and MVCC. This degrading is unrelated to conflicts (no *wait* time) but correlates with increasing NUMA distances. Consequently, for the low conflict OLTP workload, it appears that pessimistic locking is beneficial when access latencies (NUMA effects) are low. The temporary copies of optimistic CC can hide these latencies, but at the cost of additional data movement, slowing down throughput at close NUMA distance. To this end, HSTORE and TICTOC implement these two approaches as well, but they are more efficient, e.g., as HSTORE locks less frequently. Notably, there is no difference among the pes-



(a) High Conflict, 4 Warehouses



(b) Low Conflict, 1568 Warehouses

Figure 8.12: Breakdown of relative time spent processing TPC-C transactions with optimised DBx1000 using full schema and inserts on multi-socket hardware (Intel).

simistic CC schemes with different mechanisms against deadlocks, as the low conflict has few deadlocks.

Insight: After spending considerable engineering effort bringing state-of-the-art in-memory optimisations to DBx1000, we shed new light on concurrency control on 1000 cores. First, we unveil remarkable peak throughput of the newer CC schemes, TICTOC and SILO, on high conflict workload, while also presenting textbook behaviour of all CC schemes in the time breakdown. Second, we brighten the grim forecast of concurrency control on 1000 cores for low conflict workload from the simulation of [206]. In fact, under low conflict all CC schemes scale nearly linearly to 1568 cores reaching 200 million TPC-C transactions per second.

8.3.4 *Summary of Part One*

In this part, we analysed in-memory DBMS on an Intel-based platform with 1568 cores, revisiting the results of the simulation in [206], using their original prototype DBMS DBx1000. This led to surprising findings:

(1) A first attempt of running their prototype on today's multi-socket hardware presented broadly different behaviour of the CC schemes. To our surprise, the low conflict TPC-C workload with at most one warehouse per core (and transaction executor) revealed most concurrency control schemes not only stopped scaling beyond 200 cores but also were very inefficient spending not even half of their time on useful work.

(2) Based on these results, we decided to take a second deeper look into the underlying causes and made several discoveries. First, the default timestamp allocation via atomic increment was a major bottleneck on the multi-socket hardware. Second, the default benchmark settings of DBx1000 used a TPC-C database significantly reduced in size and disregarded inserts in the transactions. Changing these default setting shifted the picture of our initial assessment completely: while replacing the atomic counter with a hardware clock removed the timestamp creation bottleneck, enabling the original database size and insert statements, however, led to an even darker picture than in our first look. In this second look, we saw that all CC schemes completely collapsed when scaling to more than 200 cores, despite absent conflicts in the workload.

(3) Finally, we spent significant engineering efforts on our optimised DBx1000 [20] across all components from memory management over transaction scheduling to locking. This cleared the dark skies we faced before and allowed most CC schemes to scale very well, providing up to 200 million txn/s on 1568 cores. Even more surprisingly, now, the CC schemes behave very similar with no clear winner. Having cleared the view on concurrency control with this evaluation on real

hardware, an interesting question is now how these findings generalise across different scale-up hardware platforms and more demanding workloads.

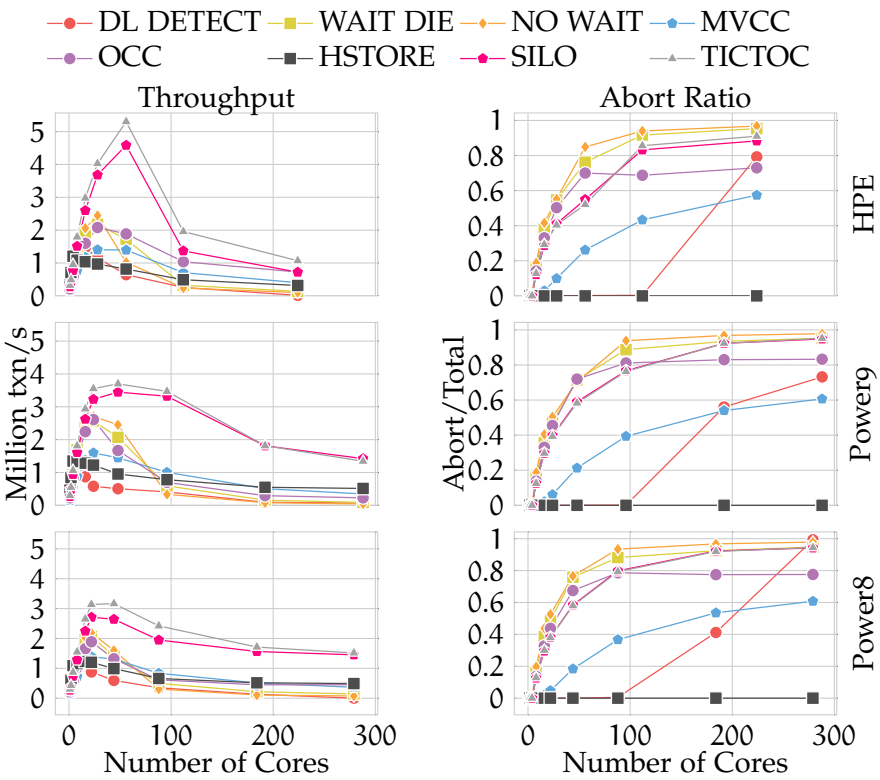
8.4 PART TWO: BROADENING THE EVALUATION

In this second part, we broaden the evaluation of in-memory OLTP DBMS on large hardware. Previously, in the first part, we evaluated how the insights of in-memory DBMS running on a simulated many-core hardware transfer to today's hardware. Indeed, we observed significantly different behaviour of the in-memory DBMS DBx1000 on real hardware compared to the original simulation [206]. For the second part, we now widen the evaluation in the two dimensions hardware and workload, as discussed in Section 8.1. First, we study the CC schemes on a broader set of hardware platforms, before we then look at the full TPC-C transaction mix.

8.4.1 *Intel-Based vs. IBM Power 8/9 Platforms*

We begin with an overview how the different approaches to "1000 cores" of today's hardware affect concurrency control. Initially, we focus on identifying diverging behaviour of CC schemes on the different hardware platforms, performing scalability experiments. Later sections cover detailed root cause analyses. The following scalability experiments determine how the CC schemes respond to increasing number of cores provided by the three different hardware platforms (HPE, Power9, and Power8), when the CC schemes pressure different aspects of the hardware depending on the scale (number of cores). For example, compute resources, caches, and interconnects between the processors are utilised differently depending on the hardware scale (number of cores). As before, we separately evaluate high and low conflict workload, due to their significant effect on the CC schemes. For example, high conflict generally requires more coordination (e.g., latching), while low conflict allows for high concurrency, influencing the behaviour of the CC schemes on the different platforms.

SCALING ON DIFFERENT REAL HARDWARE — HIGH CONFLICT
Figure 8.13 presents the performance of the CC schemes for the high conflict workload on HPE, Power9, and Power8. Overall, for the throughput in Figure 8.13a, we observe vaguely similar scaling behaviour on Power9 and Power8 as previously on HPE, i.e., the CC schemes briefly scale well but eventually thrash. This thrashing is caused by the high conflict in the workload. As indicated by according abort rates in Figure 8.13b, the CC schemes respond to these conflicts similarly on all three hardware platforms. Notably, not only the general behaviour is similar on the three platforms, but also the actual



(a) Throughput (b) Abort Ratio

Figure 8.13: Performance for TPC-C under high conflict on HPE, Power9, and Power8.

throughput of the CC schemes is of the same magnitude as opposed to the simulation, allowing for comparison of absolute performance.

Figure 8.14 details the scaling behaviour of the individual CC schemes on the three hardware platforms side by side (i.e., 1. HPE, 2. Power9, and 3. Power8). As discussed next, their diverse behaviours indicate no clear benefit of either hardware platform, but rather highlight the benefit of individual hardware properties taking effect at specific core counts.

Starting with the pessimistic locking scheme DL DETECT, we find its peak performance on HPE and at only 16 cores (1.5 M txn/s, 5.3x). On Power9, DL DETECT sharply degrades already at 16 cores, falling behind the performance on HPE and Power8. Beyond 16 cores, DL DETECT degrades on all three hardware platforms. Instead, the other two pessimistic locking schemes WAIT DIE and NO WAIT achieve their peak performance at 24 cores on Power9 (2.6/2.7 M txn/s, 9.5/9.8x). Then, these CC schemes gradually degrade similarly on all three hardware platforms until thrashing at 88 cores. Notably, this thrashing occurs when using two sockets on HPE but only one socket on Power9 and Power8, i.e., across NUMA distance 1 on HPE but NUMA-local on the Power platforms. This fact and similar abort ratios on all three

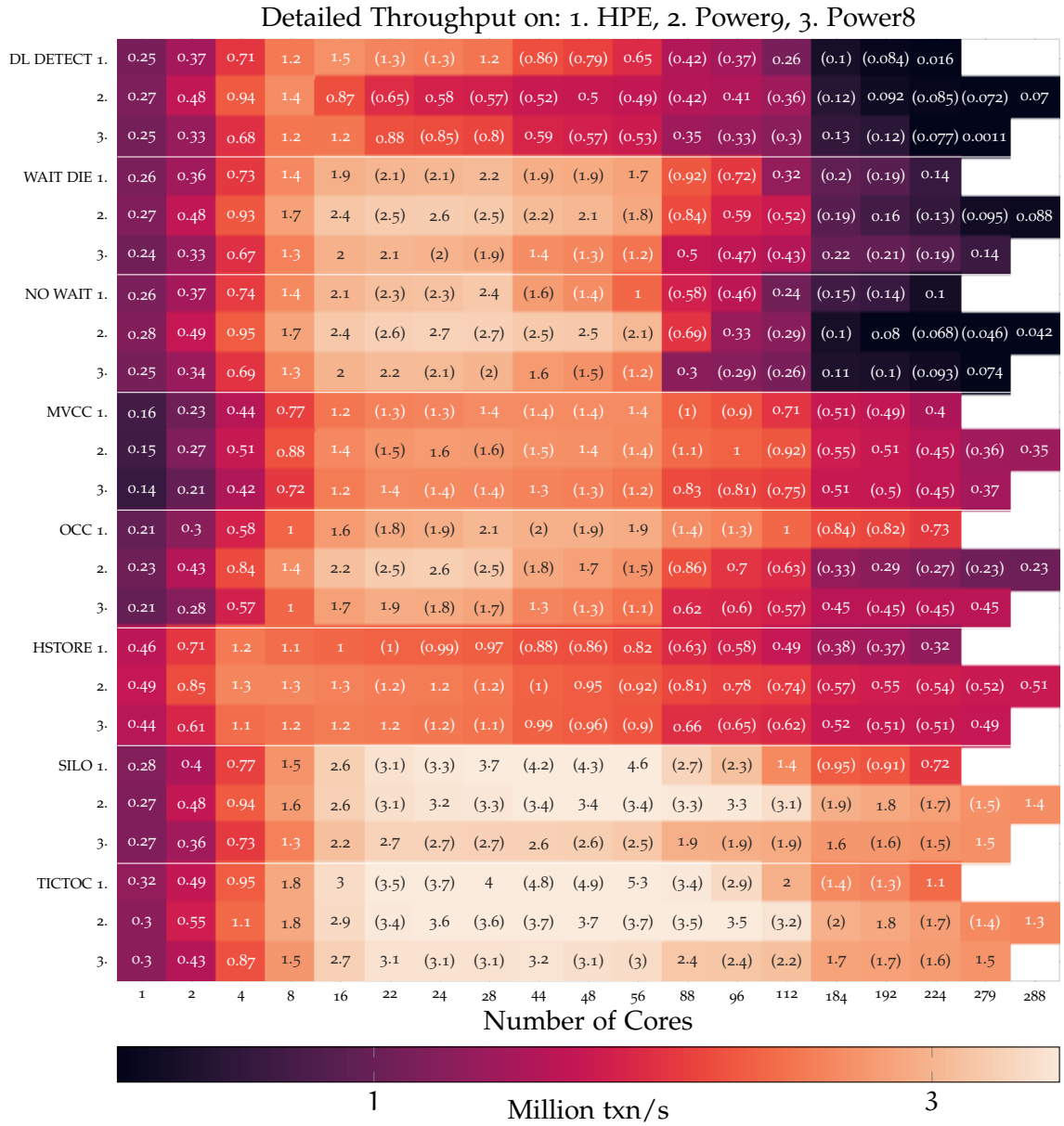


Figure 8.14: Detailed throughput for TPC-C (Million txn/s) under high conflict on HPE, Power9, and Power8. The figure shows the speedup of different CC schemes (on y-axis) when scaling the number of cores (on x-axis). Per CC scheme, we have 3 rows — one for each platform (1. HPE, 2. Power9, 3. Power8).

platforms beyond the thrashing point indicate overwhelming conflicts as cause for this thrashing of WAIT DIE and NO WAIT (rather than NUMA or other hardware properties). At high core counts NUMA additionally takes effect. Then, WAIT DIE and NO WAIT benefit from lower NUMA latency on Power8, though only by less degrading.

Moving forward to the other schemes, we see further interesting behaviours: (1) MVCC and OCC again scale differently than the pessimistic locking schemes on larger core counts, peaking at 24 cores (on Power 9 with 1.6/2.6 M txn/s). Afterwards OCC degrades less on HPE than on Power9 and Power8, resulting in significantly higher throughput on HPE at high core counts despite the stronger NUMA effect on this hardware platform, as we will see later. (2) HSTORE also reaches peak performance on Power9 with 1.35 M txn/s at 4 cores. Afterwards its performance converges between Power9 and Power8. In contrast, HSTORE gradually falls behind on HPE between 4 and 56 cores (one full socket), then worse NUMA effects on HPE further slow down HSTORE. (3) Finally, SILO and TICTOC initially perform best on HPE, peaking at 56 cores with 4.6/5.3 M txn/s. Beyond this peak, SILO and TICTOC degrade steeply on HPE. Instead, on Power9 and Power8 their throughput scales worse with a lower peak but also less degrading than on HPE. Notably, the performance of both CC schemes drops at 88 cores on Power8 within a socket. Since Power9 does not exhibit such performance drop within a single socket, fewer hardware resources of the Power8 processor (especially L3 cache) and subsequent resource contention within SILO and TICTOC seem to cause the earlier performance drop.

Comparing the performance of the CC schemes for this high conflict workload reveals an influence of the hardware properties, e.g., some CC schemes react stronger to NUMA and cache contention than others. Overall, the pessimistic CC schemes degrade strongest at high core counts on all three hardware platforms. Notably, among the pessimistic CC schemes NO WAIT stays ahead until utilising all cores of a socket on the individual platforms, at which point WAIT DIE overtakes. This indicates cache contention and NUMA as factors strongly influencing the pessimistic CC schemes besides conflicts, i.e., the simpler NO WAIT is not only sensitive to conflicts in the workload but also to contention inside the hardware, whereas WAIT DIE copes better with higher conflicts and contention at the cost of overhead. A similar influence of hardware effects versus overhead can be observed between MVCC, OCC, and HSTORE. On Power9 and Power8 at higher numbers of cores with high conflict, HSTORE despite its coarse partition locking catches up with MVCC and OCC circumventing NUMA and resource contention effects due to the lower overhead, whereas the medium overhead OCC performs the best on HPE. Finally, SILO and TICTOC perform the best on all three platforms. Their peak performance is further ahead of the other CC

schemes on HPE than on Power9 and Power8, but as NUMA takes effect SILO and TICTOC degrade less on the Power platforms.

Further analysis of the detailed time breakdowns[3] confirms that the hardware characteristics effect how the individual CC schemes spent time. The time breakdowns on Power8 reveal increased proportions of time spent for index accesses and concurrency control compared to HPE, on which more time is spent for actual work. Profiling confirms that latching within the CC schemes and index traversal are the hotspots on Power8, both of which are sensitive to memory latency. Notably, latching occurs to different extends in the CC schemes and in different phases, i.e., during transaction execution categorised as CC Mgmt. or when committing transactions categorised as *Commit*. The observations for the individual CC schemes are accordingly. For example, the pessimistic locking schemes spend more time acquiring locks and committing, whereas MVCC and OCC spend more time committing. On Power9, the time breakdowns exhibit similar increases in time spent for index accesses and concurrency control, yet lower than on Power8. That is, the time spent for index accesses and concurrency control is related to the cache sizes of the hardware platforms, resulting in the least time spent on HPE with the largest L1 and L2 caches per logical core followed by Power9 with a larger L3 cache than Power8 (cf. Table 8.2b). With increasing core counts and accordingly more conflicts these differences vanish as time spent for waiting or aborting dominates.

Insight: Under high conflict, regardless the hardware platform no CC scheme utilises high core counts effectively, i.e., the CC schemes only initially scale well with increasing core counts, but quickly thrash under the overwhelming conflicts. Yet, their specific scaling behaviour indeed depends on the hardware, especially on processor characteristics (e.g., caches capacity) and NUMA, further analysed in the following sections.

SCALING ON DIFFERENT REAL HARDWARE — LOW CONFLICT In this second experiment, we determine how the different CC schemes scale on the different hardware platforms providing a high number of cores, when low conflict workload permits high concurrency. Accordingly, Figure 8.15 shows the throughput of the CC schemes on HPE, Power9, and Power8. Briefly summarised, all CC schemes present positive scaling behaviour on all three hardware platforms, HSTORE initially performs the best, but most CC schemes follow HSTORE in a pack, and MVCC is behind at least for lower core counts.

However, a closer comparison between the three hardware platforms indicates again two interesting trends for this low conflict workload. First, the throughput of all CC schemes increases in distinctly different slopes on the three platforms, i.e., the hardware platforms seem to

3 Figures omitted for brevity are available online [19].

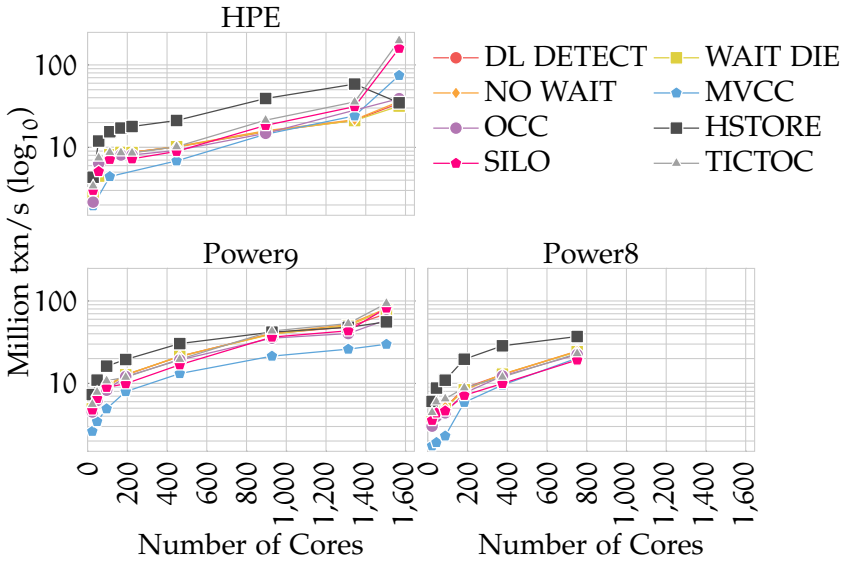


Figure 8.15: Throughput for TPC-C under low conflict on HPE, Power9, and Power8.

have a distinct effect on the scaling behaviour. Second, as the number of cores increases, the relative performance of the CC schemes distinctly differs between HPE and the two Power platforms. For detailed analysis, Figure 8.16 shows comparisons, indicating for each CC scheme the speedup of one platform over another (e.g., Power8 vs. Power9) at the same number of cores.

The comparison of Power9 and HPE in Figure 8.16 (Power9 vs. HPE) indicates that all CC schemes are faster on Power9. However, the speedup on Power9 compared to HPE varies in distinct pattern, corresponding to the increasing NUMA distance. For the pessimistic locking schemes, the throughput difference between Power9 and HPE shrinks until using 2 sockets (112 cores) on HPE, then throughput on HPE falls behind and with more than 224 cores across 4 sockets on HPE (across the farthest NUMA distance 3) throughput drops even further. The other CC schemes react similarly to these on HPE, except for HSTORE, which instead is affected by the closest boundary beyond one socket and farthest boundary above 4 sockets (with NUMA distances 1 and 3). Further, the closest boundary after 56 cores on HPE has a diverse effect on the CC schemes. This NUMA boundary only has a negative effect on the fastest two CC schemes (i.e., HSTORE and TICTOC) as well as OCC. Instead, the other CC schemes scale well past one socket (56 cores) on HPE and in fact close in onto the throughput on Power9.

Comparing Power8 and HPE in Figure 8.16 (Power8 vs. HPE), indicates the same effect as observed in comparison to Power9. Also on Power8, the performance of all CC schemes initially is ahead, then their performance on HPE catches up around 56 - 112 cores (across two sockets with NUMA distance 1). In fact, HPE overtakes Power8, on

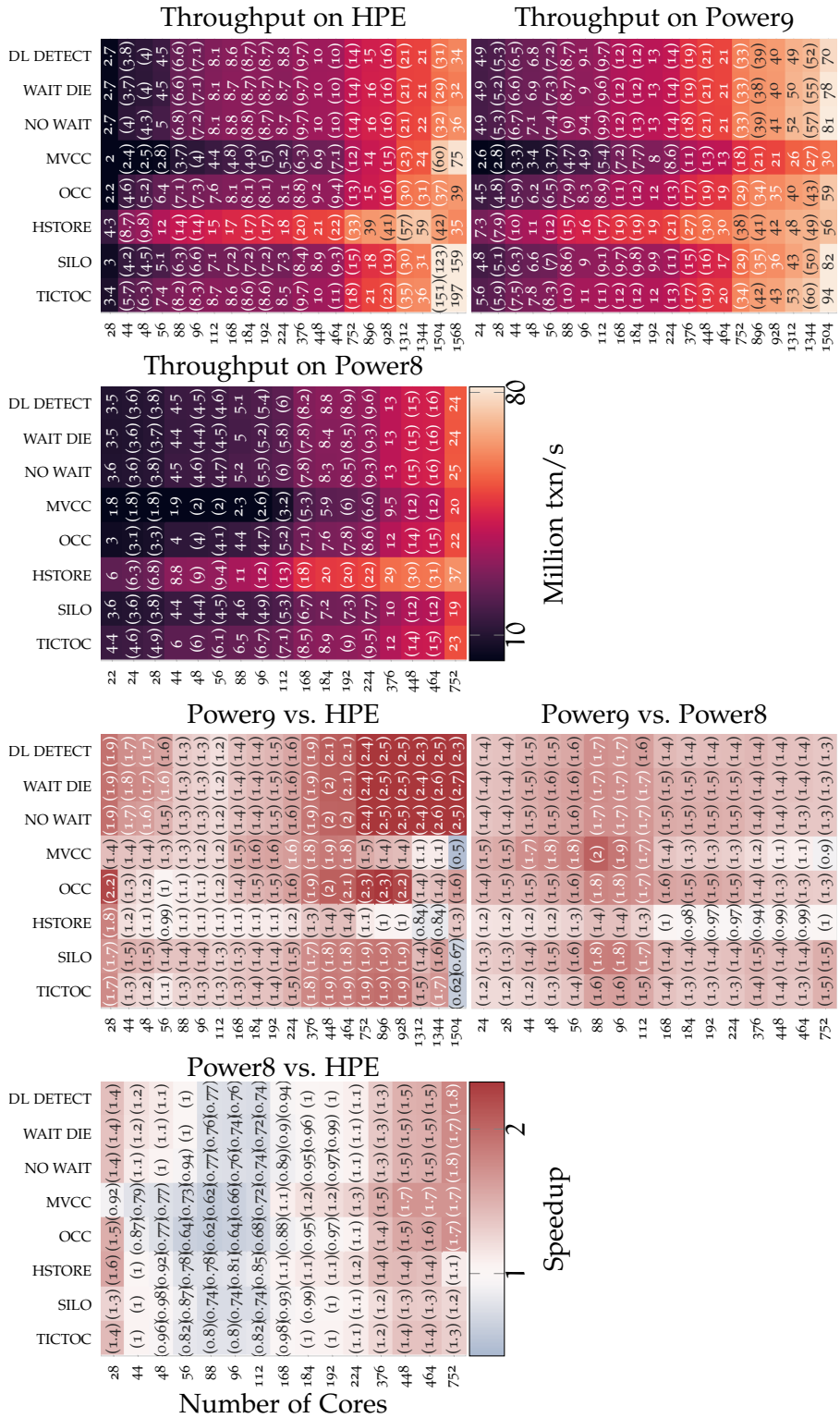


Figure 8.16: Detailed throughput for TPC-C under low conflict on HPE, Power9, and Power8 (1st row) and comparison between the hardware platforms (2nd). The comparisons indicate by which speedup ratio the throughput differs for a CC scheme (on y-axis) at the same number of cores (x-axis) on one platform versus another platform.

which the CC schemes struggle due to resource contention (to be discussed in Section 8.4.2.1). Remarkably, the larger processor resources on HPE compensate for its worse NUMA properties (lower bandwidth and higher latency), when operating across 2 sockets. Across more than 2 sockets (112 cores) the performance of most CC schemes on HPE is even to Power8. Only HSTORE and MVCC straggle on HPE, due to their higher load on the memory subsystem (i.e., sheer performance of HSTORE and overhead of MVCC).

The comparison of Power9 and Power8 in Figure 8.16 (Power 9 vs. Power 8 or vice versa) indicates improved performance of the Power9 processor over Power8, as throughput on one socket is 1.2 - 2x higher. Notably, beyond one socket the performance benefit of Power9 stagnates or even decreases. Consequently, the strong NUMA topologies of both Power platforms similarly boost concurrency control at large scale. This confirms a general advantage of Power's stronger NUMA topology and importantly indicates the relevance of NUMA properties for the performance of concurrency control.

Furthermore, the time breakdowns[4] indicate diverging internal behaviour of the CC schemes on the hardware platforms. Similar to the high conflict workload, on Power8 the CC schemes spend significant time for concurrency control and index accesses, while on HPE for useful work (e.g., accessing records). Also Power9 shows increased time spent for concurrency control and index accesses, but again overall lower than on Power8 and biased towards index accesses (less for concurrency control but more for index accesses). Profiling on Power8 confirms this continued trend, again identifying latching as bottleneck related to memory latency. Conversely, for HPE, these observations hint at memory bandwidth as bottleneck for this low conflict workload.

Regarding the second trend about the relative performance of the CC schemes, on Power9 and Power8 the pessimistic locking schemes perform better than on HPE. Notably, these perform better than SILO and TICTOC for 96 - 1504 and 192 - 928 cores, respectively. Also, OCC improves but only at larger core counts and not as much as the pessimistic schemes. Consequently, on Power, OCC overtakes SILO, but falls behind the pessimistic schemes. In contrast, MVCC provides the worst throughout on Power with a growing gap to the other CC schemes, whereas on HPE MVCC does overtake the other CC schemes at large scale. These differences of the relative performance of the CC schemes indicate two underlying causes for this second trend: (1) Especially the latency sensitive pessimistic locking schemes benefit from the lower latency in Power's NUMA topology; (2) Resource intense CC schemes (e.g., MVCC) benefit from the larger hardware resources of the processors in HPE.

4 Figures omitted for brevity are available online [19].

Insight: Under low conflict the NUMA characteristics of the specific hardware platforms clearly affect the performance of the CC schemes, i.e., the scaling slopes of the CC schemes closely match the NUMA topology. The CC schemes generally benefit from lower latency and higher bandwidth in the NUMA topology. Yet the individual CC schemes benefit differently from either better latency or bandwidth and resource contention within the processor influences their scaling behaviour.

8.4.2 *Zooming into Hardware Aspects*

Having identified diverging behaviour on the different hardware platforms, we now zoom into those aspects that realise the large number of cores: (1) Hardware parallelism within the processors and (2) the topology connecting processors in a single system.

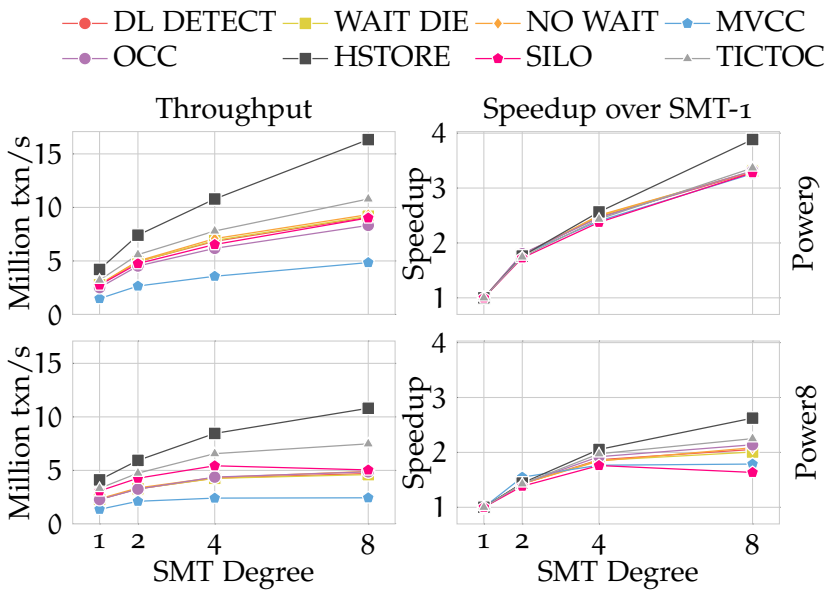
8.4.2.1 *Simultaneous Multithreading*

The superscalar processors of today's hardware employ several techniques to implement hardware parallelism. Besides a high number of physical cores, the processors also employ (superscalar) instruction-level parallelism (ILP) [81] and Simultaneous Multithreading (SMT) [36]. SMT establishes multiple parallel execution streams as logical cores to better utilise the resources of their underlying superscalar physical core, especially to facilitate thread-parallel software such as OLTP DBMSs.

While many of today's superscalar processors employ these general techniques, the specific implementations differ [2, 95–97]. Especially the Power processors utilise sophisticated SMT with a high degree of parallel execution streams on a smaller number of physical cores, up to 8 such streams (i.e., SMT-8) [96, 97]. Notably, from Power8 to Power9 IBM's hardware designers have enhanced the SMT implementation, e.g., with advanced scheduling of the execution streams. In contrast, Intel processors mainly drive hardware parallelism by the number of physical cores and use simpler SMT with two parallel execution streams (SMT-2) [95].

These elaborate techniques of hardware parallelism depend on processor resources and the software as well as the workload running on top. Therefore, our particular questions are how big this benefit can be as OLTP workloads typically strain the memory subsystem more than other processor resources and if the CC schemes allow for sufficient concurrency to utilise the parallel hardware execution streams of SMT.

In the following, we analyse the benefit of SMT for OLTP workloads, focusing on the sophisticated and high-degree SMT (up to SMT-8) of the Power processors. In the experiments, we use all physical cores of a single processor and observe the throughput for increasing SMT



(a) Throughput (b) Speedup over SMT-1

Figure 8.17: Effect of broad SMT in Power9 and Power8 processors on throughput for TPC-C under low conflict.

degree. We first analyse the best-case benefit using the low conflict TPC-C workload, before also considering high conflict scenarios.

HIGH SMT DEGREE FOR LOW CONFLICT OLTP Figure 8.17 shows the throughput for the low conflict TPC-C workload of all CC schemes under increasing SMT degree and the speedup relative to SMT-1. On the Power9 processor, most CC schemes speedup equally with increasing SMT degree, despite differing throughput; 1.7 - 1.8x for SMT-2, 2.4 - 2.5x for SMT-4, and 3.3 - 3.4x for SMT-8. Only HSTORE utilises SMT better with a speedup of 2.6x for SMT-4 and 3.9x for SMT-8, relating to low overhead and exceptional performance for low conflict workloads. Notably, despite a significant speedup of all CC schemes, the speedup of SMT on the Power9 processor is sublinear for this low conflict OLTP workload.

On the Power8 processor, in contrast, the CC schemes achieve overall lower throughput and speedup than on Power9 (SMT-2: 1.4 - 1.5x, SMT-4: 1.8 - 2.1x, SMT-8: 1.6 - 2.6x). That is, SMT of the Power8 processor provides less benefit and the speedup of the CC schemes also diverges, in three distinct groups. (1) HSTORE utilises SMT best with the highest speedup, as on Power9. (2) TICTOC, OCC, and the pessimistic locking schemes follow with still positive speedup for SMT-8, but progressively less in according order. (3) The speedup of MVCC stagnates from SMT-4 and for SILO even decreases from 1.8x for SMT-4 to 1.6x for SMT-8.

Notably, the three groups with distinct benefit of SMT comprise CC schemes with similar memory footprints and the speedup of

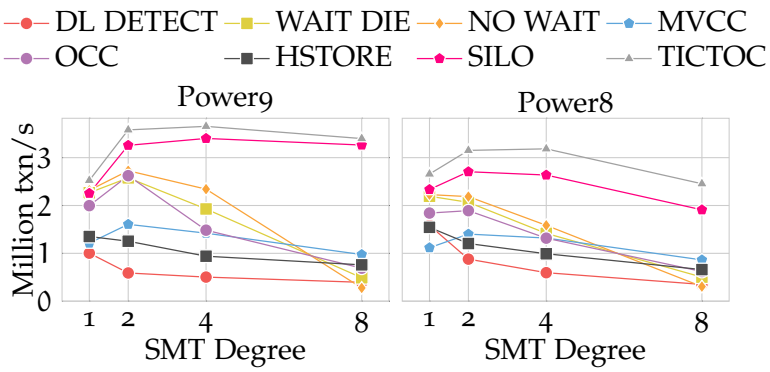


Figure 8.18: Throughput of broad SMT in Power9 & Power8 processors for TPC-C under high conflict.

these groups correlates with these footprints, i.e., the group of CC schemes with the smallest footprint gains most speedup from SMT and inversely the group with the largest footprint gains least. This correlation to the memory footprint and the increasing gap to Power9, indeed indicates increasing resource contention for SMT on Power8. Comparing their cache capacity highlights the larger L3 cache per logical core on Power9 (cf. Table 8.2b) [96, 97]. Evidently, sufficient L3 cache capacity for all the execution streams is an important factor to effectively utilise SMT.

On the Intel processor with only SMT-2, we make similar observations, omitted from Figure 8.17 due to the small SMT degree. For example, SMT-2 of that Intel processor provides a speedup of 1.5x for TICTOC from a throughput of 5.25 M txn/s with SMT-1 to 7.92 M txn/s with SMT-2.

Insight: Overall, SMT indeed benefits our favourable (i.e., low conflict) OLTP workload, yet with sublinear speedup in relation to the SMT degree. The sophisticated SMT of the Power9 processor provides broad benefit for all CC schemes up to the highest SMT degree (SMT-8). In contrast, resource contention limits the benefit of SMT on the Power8 processor, indicating a dependency between the benefit of SMT and the resource footprint of the CC schemes.

HIGH SMT DEGREE FOR HIGH CONFLICT OLTP For the second workload with high conflict, throughput of the CC schemes under increasing SMT degree and speedup relative to SMT-1 is shown in Figure 8.18. Overall, the CC schemes barely benefit from SMT on neither Power9 nor Power8. In detail, on Power9, SILO and TICTOC utilise SMT best. These speed up by 1.4x with SMT-2 and maintain this speedup for SMT-4 and SMT-8. In contrast, the remaining CC schemes speed up with SMT-2 by a smaller factor — if at all. Latest with SMT-4 their speedup declines to a slowdown (<1x speedup). DL DETECT and HSTORE immediately slow down with SMT-2 (0.59x and 0.93x,

respectively). On Power8, the CC schemes benefit even less from SMT, i.e., the speedup for SMT-2 is lower and for higher SMT degrees the slowdown is stronger. Notably, MVCC has the same speedup on both Power9 and Power8, throughput is higher on Power9 by stable 10%. In conclusion, conflicts are the determining factor for the performance of all CC schemes and prohibit general benefit of SMT. Yet, the improved SMT of Power9 is still noticeable, albeit more limited than under low conflict.

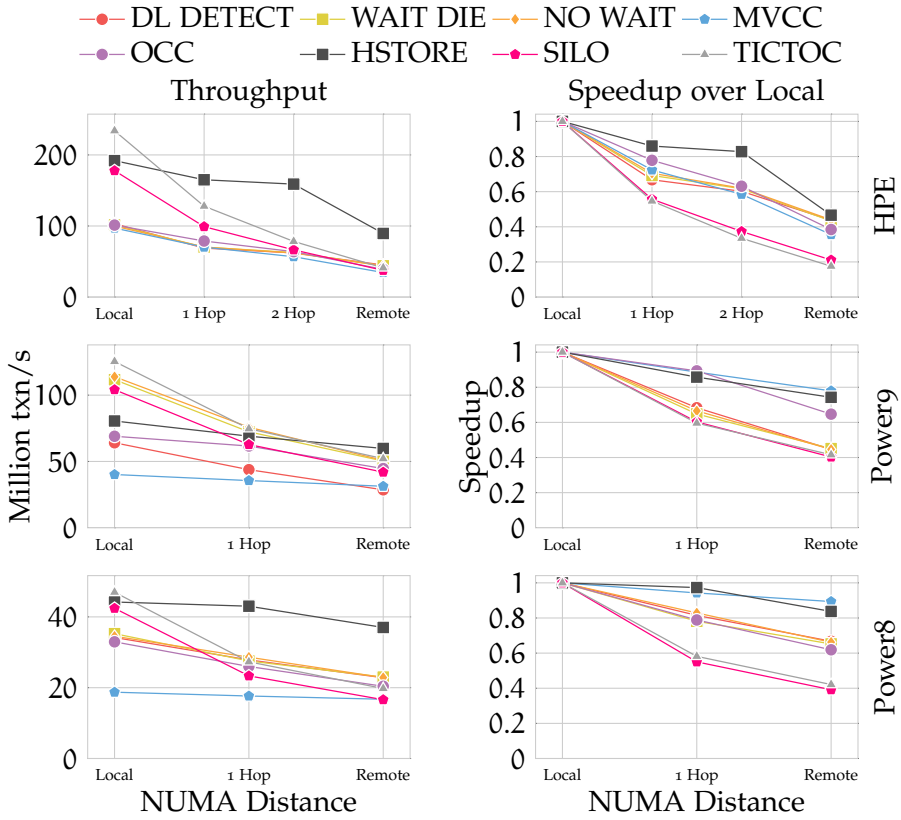
Insight: For high conflict OLTP workload, the performance of all CC schemes is widely determined by the conflicts rather than SMT, yet some benefit of SMT appears, especially from the Power9 processor.

8.4.2.2 *Non-Uniform Memory Access*

Today, thousands of cores are only available via multi-socket hardware imposing the Non-Uniform Memory Access (NUMA) effect for memory accesses. Such multi-socket hardware connects its processors (and memory) in a tiered non-uniform topology, through which the processors communicate and mutually access memory. As the topology connecting the processors is tiered and non-uniform, so are the performance characteristics for processors when communicating or accessing memory, i.e., bandwidth and latency between processors in the topology differ. These diverging performance characteristics of the underlying hardware (the NUMA effect) impact the performance of a DBMS depending on its communication and memory access pattern.

In the following, we analyse the NUMA effect of our three hardware platforms on the CC schemes, which all employ different technologies and topologies to connect their processors (see Section 8.2 for more details). For this analysis, we start with an extreme scenario isolating the NUMA characteristic of the three hardware platforms and their effect on the CC schemes when all memory accesses have a predefined NUMA distance. In a second experiment, we compare the NUMA effect on the CC schemes using a more realistic and complex scenario with NUMA effects imposed by the workload.

ISOLATED NUMA EFFECTS (FIXED DISTANCE) First, we analyse the NUMA effect on the CC schemes in an extreme scenario, where transactions strictly access memory at a fixed (specified) NUMA distance. This extreme scenario overall exposes the differing NUMA characteristics of our hardware platforms and subsequently reveals their influence on the CC schemes. For this scenario, we restrict the TPC-C transactions to only access their home warehouse and allocate this warehouse on memory with the specified NUMA distance. Further, we use the low conflict workload and the maximum cores, isolating the effect of operating across a specified NUMA distance from other effects (e.g., conflicts).



(a) Throughput (b) Speedup over Local

Figure 8.19: Performance for TPC-C under low conflict with strict access to home warehouse at specified NUMA distance (on x-axis) on HPE, Power9, and Power8. Speedup is reported as ratio of throughput at the specified NUMA distance over *Local*. For Power, the distance *2 Hop* is missing since this hardware has a shallower topology than HPE (cf. Section 8.2.2).

Figure 8.19 shows the throughput and speedup of the CC schemes under increasing NUMA distance on HPE, Power9, and Power8. Since the maximum number of cores (where the NUMA effect is strongest) differs on the hardware platforms and thus the throughput, we rather focus on the speedup of the NUMA distances 1 - 3 (*1 Hop*, *2 Hop*, *Remote*) over the local NUMA distance 0 (i.e., when all data is accessed on the local NUMA region/processor), as shown in Figure 8.19b. Overall, as expected the NUMA effect (deteriorating bandwidth and latency) indeed degrades performance as the NUMA distance increases. Yet, throughput and speedup of the CC schemes show several trends on the different hardware platforms.

On HPE when accessing only local memory, the CC schemes HSTORE, SILO, and TICTOC achieve remarkable throughput of 178 - 234 M txn/s. However, when accessing farther memory, SILO and TICTOC immediately slow down sharply by 0.56x and 0.55x at NUMA distance 1, respectively. Then, SILO and TICTOC slow down at a lower rate, to 0.21x (38 M txn/s) and 0.18x (41 M txn/s) for NUMA distance 3 (Remote). On Power9 and Power8, SILO and TICTOC slow down similarly for the NUMA distance 1 (0.55 - 0.6x), but for the farthest NUMA distance (Remote) their slowdown is more graceful (0.39 - 0.42x). In contrast, HSTORE slows down much less on all three hardware platforms. For NUMA distances 1 - 2, HSTORE slows down least on Power8 (0.97x), followed by HPE and Power9 on par (0.86x). Afterwards, the farthest NUMA distance affects HSTORE again the least on Power8 (0.84x), followed by Power9 (0.74x), but HPE falls behind (0.47x).

The remaining CC schemes generally slow down more gracefully under increasing NUMA distance. Notably, on HPE, the pessimistic locking schemes (DL DETECT, WAIT DIE, and NO WAIT initially slow down stronger for NUMA distance 1 (0.67 - 0.7x vs. 0.72 - 0.78x) but then slow down at a lower rate, while MVCC and OCC slow down stronger at the farthest NUMA distance 3 (Remote). On Power9, the pessimistic locking schemes slow down similarly. On Power8, however, these CC schemes slow down less, i.e., by 0.78 - 0.83x for distance 1 and 0.65 - 0.67x for distance 3. That is, the lower latency in the topology of Power8 significantly benefits the pessimistic locking schemes.

In contrast, MVCC slows down most on HPE, with Power9 and Power8 similarly ahead for NUMA distance 1 (HPE: 0.72x, Power9: 0.89x, Power8: 0.94x), but at the farthest NUMA distance Power9 falls behind and Power8 leads again (HPE: 0.36x, Power9: 0.74x, Power8: 0.89). From the higher bandwidth of the Power platforms and in turn higher bandwidth in Power8 than Power9, we conclude that MVCC benefits from higher bandwidth in the topology (and lower latency).

Finally, OCC also presents diverse slowdown across the three hardware platforms. Initially, at NUMA distance 1 OCC slows down least on Power9, followed by HPE and Power8 on par (Power9: 0.89x, HPE: 0.78x, Power8: 0.79x), while at the farthest NUMA distance 3, Power9

and Power8 are equally ahead of HPE (Power9: 0.65x, Power8: 0.62x, HPE: 0.38x).

Insight: Overall, two notable trends appear relating to the NUMA characteristics of the three hardware platforms. First, the latency sensitive pessimistic locking schemes do best on Power8 providing the lowest latency in its topology. On HPE and Power9 instead, which have similarly higher latencies than Power8 for *NUMA Remote* accesses, these schemes perform similarly worse. Second, CC schemes that require more bandwidth, either due to sheer performance as for HSTORE or due to memory overhead as for MVCC, perform better on Power9 and Power8, both of which provide higher-bandwidth interconnects in their topologies. Both these trends confirm our early observations of NUMA effects on the scaling behaviour of the CC schemes on the three hardware platforms.

WORKLOAD-IMPOSED NUMA EFFECT The previous experiment highlights effects on the CC schemes relating to the NUMA characteristics in an extreme scenario. However, realistic operating conditions of OLTP DBMSs are more complex. On one hand, DBMSs commonly attempt to mitigate extreme NUMA effects by strategies like NUMA-aware database partitioning. On the other hand, realistic workload dictates the access pattern, still imposing NUMA effects (and other non-NUMA effects). We now analyse these more realistic workload-imposed NUMA effects using TPC-C's remote transactions. That is, TPC-C is commonly partitioned by warehouses (also in our experiments) mitigating NUMA effects. Yet, TPC-C specifies so-called remote transactions that apart from their home warehouse span further warehouses (remote warehouses), thus these remote transactions are not partitionable and cause workload-imposed NUMA effects.

In the following experiment, we use the combination of following two setups to isolate the NUMA-related from the other non-NUMA effects in this more complex scenario: (1) In the first setup, we analyse the non-NUMA related effects of remote transactions. For this, we vary the amount of remote transactions across warehouses but use only local memory for all warehouses (i.e., no NUMA effects occur); (2) In the second setup, we then distribute warehouses across NUMA regions and thus observe the combined (NUMA and non-NUMA) effects imposed by remote transactions. Consequently, we can thus better isolate the NUMA-related from the non-NUMA-related effects on the CC schemes by comparing their performance in these two settings.

In detail, for setup *(1) NUMA Local*, we allocate the remote warehouses on local memory (alongside the home warehouse and transaction executor). In contrast, for setup *(2) NUMA Remote*, we allocate the remote warehouses the farthest away from the transaction executors, i.e., remote warehouses are at remote NUMA distance 3 but the home

warehouses remain at local NUMA distance 0. The remaining setup is identical to the prior experiment (cf. *Isolated NUMA Effects*).

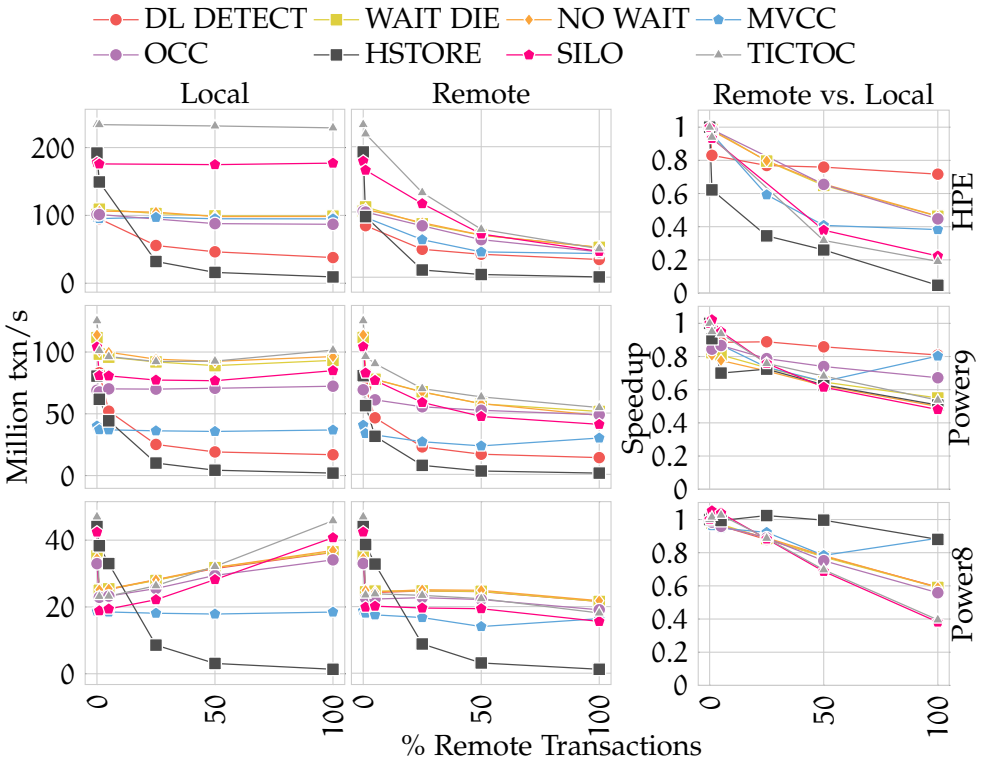
Figures 8.20a shows the performance in the two described settings (*(1) Local* and *(2) Remote*), when transactions increasingly access remote warehouses (% remote transactions) either on (1) local memory or (2) remote memory. In addition, Figure 8.20b compares the performance in these two settings (*Remote vs. Local*) for the same ratio of remote transactions. We first analyse how the different CC schemes are affected by the aforementioned effects focusing first on the HPE platform. In a second step, we then compare the effects across the different hardware platforms to identify the effect of their NUMA characteristics.

Starting with the CC schemes on HPE (top row of Figure 8.20): while most CC schemes provide stable throughput for the *Local* setting on HPE, they all degrade in the *Remote* setting due to the NUMA effect (also on the Power platforms, though with further effects which we discuss later). Notably, DL DETECT and HSTORE significantly degrade already in the *Local* setting without the NUMA effects, i.e., non-NUMA effects impact these CC schemes as well.

Figure 8.20b shows the performance ratio when increasing the NUMA distance for remote warehouses in setup (2) compared to the *(1) Local* setup. This provides a more detailed insight into the effects of remote transactions. Overall, the resulting effects on CC schemes can be grouped into three categories:

1. DL DETECT drops to 0.83x at 1% remote transactions but then degrades only to 0.72x, which indeed is the least effect across all CC schemes. Consequently, conflicts (and other non-NUMA effects) affect DL DETECT more than NUMA.
2. Conversely, the other pessimistic CC schemes (WAIT DIE and NO WAIT) as well as OCC, SILO, and TICTOC suffer more from the NUMA effects. These significantly slow down with *NUMA Remote* compared to *NUMA Local*, while their throughput for NUMA Local is mostly stable.
3. HSTORE and MVCC suffer from the combination of NUMA effects and non-NUMA-related conflicts. While for HSTORE a combined effect relates to its high sensitivity to conflicts (as observed previously), for MVCC an additional effect of conflicts only appears by comparison to the prior experiment on NUMA effects (cf. Figure 8.19). Previously MVCC suffered less NUMA effects, when there were no conflicts in the workload. Consequently, the conflicts indeed amplify the NUMA effect for MVCC.

Comparing the hardware platforms (2nd and 3rd row of Figure 8.20), we see that the CC schemes on the Power platforms behave similar to HPE. However, looking into the detailed behaviour, we see that the workload-imposed NUMA effects depend on the individual



(a) Throughput (b) Speedup

Figure 8.20: Effect of workload-imposed NUMA by increasing ratio of remote transactions (on x-axis) on TPC-C under low conflict in the two different setups (*Local* and *Remote* as described in the setup of this experiment) and the three platforms: HPE, Power9, and Power8. Figure (a) shows the throughput for increasing ratio of remote transactions. Figure (b) shows the ratio of throughput of the *Remote* over the *Local* setup.

NUMA characteristics of the hardware platforms. For example, in our following analysis we confirm the advantage of the better NUMA characteristics of the Power platforms compared to HPE, providing more stable behaviour (as already observed in the previous experiment). This can be seen by the fact that the CC schemes on the Power platforms for the *Remote* setup in Figure 8.20 (right column) show a shallower drop when compared to HPE. In the following, we now discuss the details that lead to this behaviour.

In Figure 8.20a, the throughput of the CC schemes for *NUMA Remote* degrades depending on three factors: the sensitivity of the CC schemes to NUMA, the NUMA characteristics of the specific hardware platform, and non-NUMA effects such as cache pollution. These effects appear as follows on the three hardware platforms for the CC schemes previously categorised as significantly affected by NUMA (and insignificantly by non-NUMA effects): (1) On HPE, as already determined, the NUMA effect strongly and continuously degrades the CC schemes; (2) on Power9, the better NUMA characteristics degrade the CC schemes less, but the smaller cache causes a small drop for 1% remote transactions; (3) on Power8, the small cache causes a significant non-NUMA-related drop for 1% remote transactions for both *NUMA Remote* and *NUMA Local*, afterwards the CC schemes also degrade due to the NUMA effect similar to Power9, as detailed in Figure 8.20b.

Finally, the CC schemes of the other categories (not mentioned above) also diverge between the three platforms. We summarise the most important findings for those schemes in the following. Figure 8.20b indicates that the cache pollution on Power8 exposes DL DETECT to NUMA effects, as there is no NUMA effect on HPE or Power9. Furthermore, observing the speedup of *Remote vs. Local* in Figure 8.20b confirms for the CC schemes of the third category (e.g., HSTORE) that the NUMA effects are amplified by non-NUMA effects. As the NUMA characteristics improve from HPE to Power9, and further from Power9 to Power8, we observe that the speedup of *Remote vs. Local* converges towards 1x, i.e., the performance of these CC schemes indeed becomes independent of NUMA effects and depended on the other non-NUMA effects.

Insight: In the more realistic scenario of workload-imposed NUMA effects (by TPC-C remote transactions), the CC schemes not only face NUMA effects but also other effects. To summarise, we have seen that they are affected in three groups: (1) one group is mainly affected by non-NUMA effects (e.g., conflicts), such as DL DETECT, (2) another group is primarily affected by NUMA effects, e.g., WAIT DIE or TICTOC, and (3) the last group is affected by the combination of NUMA effects and conflicts, e.g., MVCC and HSTORE. These findings apply to all three hardware platforms, in variations according to the specific hardware characteristics as previously observed for the isolated NUMA effect.

8.4.3 *The Full TPC-C Benchmark*

In the previous experiments, we observed a significant impact of conflicts and data locality on the behaviour of the CC schemes. However, besides conflicts and data locality, the type of workload and operations is a major aspect. Therefore, in this final evaluation step, we analyse the effect of the workload on the CC schemes in more detail. In particular, we evaluate the contrast between a more comprehensive workload covering the full TPC-C transaction mix (all 5) versus the often used more narrow transaction mix comprising just the NewOrder and Payment transactions, which was used in the simulation of prior work [206]. Notably, the full transaction mix includes read-heavy and additionally more expensive (i.e., longer-running) transactions, such as StockLevel aggregating records from many districts. In addition, the full mix requires additional indexes increasing the cost of the NewOrder and Payment transactions (used in the more narrow mix) as well.

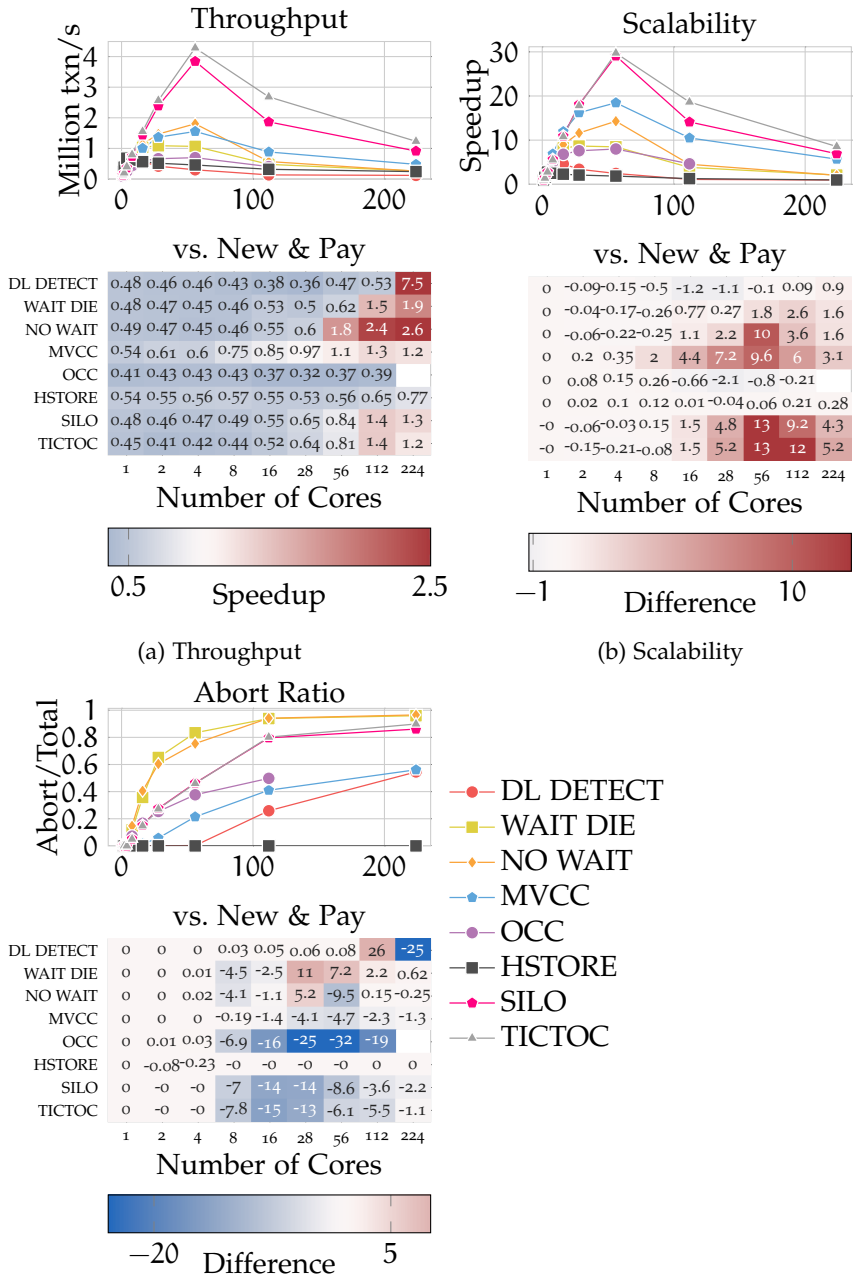
In the following, we again start with an analysis of the high conflict workload and then discuss the results for the low conflict workload.

8.4.3.1 *Full TPC-C under High Conflict*

In this experiment, we analyse how the behaviour of the CC schemes differs between the full and the narrow transaction mixes for high conflict workload. Most notable, the read-heavy transactions of the full mix are expected to affect the CC schemes depending on their ability to handle read-write conflicts. In a first step, we thus focus on diverging behaviour of the CC schemes between these two transaction mixes on the same hardware platform. Then, we assess whether their behaviour further differs across the hardware platforms. As in our previous experiments, we first evaluate the CC schemes on HPE and then compare the Power platforms.

FULL VS. NARROW MIX ON HPE Figure 8.21 displays the performance of the individual CC schemes for the full TPC-C transaction and a comparison to the narrow mix (only NewOrder and Payment transactions), cf. Figure 8.15. The top row provides an overview over the performance of the individual CC schemes. Overall, it shows that the CC schemes scale well initially, but eventually all thrash due to overwhelming conflicts — a similar behaviour as with the narrow transaction mix.

However, the comparison to the throughput of the narrow mix (Figure 8.21a, 2nd row) indicates broadly worse throughput with the full mix until about 56 cores. At higher core counts, though, most CC schemes indeed provide better throughput (e.g., NO WAIT at 224 cores 2.6x over the narrow mix). Remarkably, the CC schemes better handle increasing conflicts and NUMA effects with the more



(a) Throughput

(b) Scalability

(c) Abort Rate

Figure 8.21: Throughput, scalability, and abort ratio for full TPC-C transaction mix under high conflict on HPE. In the 2nd row, throughput and scalability, and abort rate are compared between the full and the narrow TPC-C mix (only NewOrder & Payment). For throughput (a, 2nd row), we compare the speedup of the full over the narrow mix. Since scalability as such is reported as speedup over 1 core (b, 1st row), we rather compare the scalability as difference (b, 2nd row) between the speedup of the full mix minus the narrow mix. Likewise, the abort ratio (c, 2nd row) is compared by the difference, i.e., abort ratio of the full mix minus the abort ratio of the narrow mix. In all plots, one data point for OCC (at 224 cores) is missing, since here the OCC scheme "froze" due to high conflicts.

(read)-intense transactions in this full mix. Only HSTORE does not quite close the performance gap between the full transaction mix and the narrow mix (0.53 - 0.77x the performance of the narrow mix) and OCC's performance for the full mix remains low at 0.4x, not improving at higher core counts.

The detailed scaling behaviour in Figure 8.21b, indeed indicates that this positive effect of the heavier transactions in the full transaction mix already starts at lower core counts. The comparison between the full and the narrow mix (Figure 8.21b, 2nd row), shows, that already from 8 cores the CC schemes exhibit better scaling for the heavier transactions, though beyond 56 cores (more than one socket) the lead decreases. Notably, SILO and TICTOC benefit the most (peak improvement), while MVCC benefits across the widest number of cores.

Having identified diverging impact of the full TPC-C transaction mix on the CC schemes, we now analyse the causes in further details. Specifically, we search for (1) reasons reducing the performance at lower core counts as well as improving the performance at higher core counts and (2) reasons for higher impact on some CC schemes than others.

For the first case, as the full transaction mix introduces additional read-write conflicts and longer transactions, there are two major differences between the full and the narrow transaction mix potentially causing the observed general divergence: Conflict handling and amount of actual work. If conflict handling has a major influence on the observed throughput and scaling behaviour, then the CC schemes should exhibit similarly diverging abort rates. However, in Figure 8.21c the abort rates for the full mix and the comparison to the narrow mix are ambiguous without a clear effect of the heavier transactions. For example, MVCC has similar abort rates for both transaction mixes while throughput significantly differs. Similarly, the improved throughput for the full mix of SILO and TICTOC does not relate to their abort rate. Consequently, the abort rates of the CC schemes surprisingly do not relate to their diverging throughput for the two transaction mixes.

As further step in analysing the impact of read-write conflicts and longer transactions, we analyse the time breakdowns [19] detailing how the CC schemes spend their time processing transactions of the full and the narrow mix (e.g., useful work, aborting, etc., cf. Table 8.3 in Section 8.3). The time breakdowns reveal that lower throughput for the full transactions mix relates to an increase of relative time spent for concurrency control in all CC schemes (i.e., CC Mgmt. or Commit), either in addition to increased waiting/aborting (for DL DETECT, WAIT DIE, NO WAIT, and OCC) or exceeding a reduction of waiting/aborting (for MVCC, SILO, and TICTOC). As the number of cores increases and throughput improves for the full mix, the time spent for concurrency control converges between the full and the narrow mixes. Instead, the time breakdowns of the full mix indicate a slight reduction

of time spent waiting or aborting in conjunction with a slight lead in useful work. Consequently, the higher transaction throughput in the full mix relates to lower conflict at higher core counts. These two trends in the time breakdown imply that, first, the lower throughput for the full mix is not only associated with conflicts, but also with the higher load of the heavier transactions, making concurrency control more costly for all CC schemes compared to the narrow transaction mix. Second, at high core counts the heavier transactions dampen the impact of conflicts, allowing higher throughput especially for those CC schemes that can efficiently handle read-heavy transactions. This is not the case for DL DETECT, OCC, and HSTORE, as explained below.

The time breakdowns also provide insight into why DL DETECT, OCC, and HSTORE behave inconsistently with the other CC schemes, i.e., with increasing core counts these do not benefit (as much) from the heavier transactions. DL DETECT spends much more time waiting with the full transaction mix compared to the narrow mix, since waiting itself becomes more costly for DL DETECT due to traversing larger wait-for-graphs. OCC is initially slower due to more costly concurrency control like the other CC schemes, but at higher core counts aborting in OCC appears as new bottleneck. Remarkably, the time spent aborting increases for OCC despite lower abort rate for the full mix, i.e., for OCC aborting the heavier transactions is more costly and overshadows lower conflict. In contrast, HSTORE spends its time very similar for both transaction mixes, i.e., waiting time eventually dominates as conflicts overwhelm HSTORE's partition-based locking regardless the type of work. Consequently, the performance of HSTORE converges between the two transaction mixes due to similarly dominating waiting time. In contrast to the prior three CC schemes, MVCC performs exceptionally better with the full mix and indeed it spends more time for actual work and less for aborting or waiting, confirming its ability to prevent read-write conflicts (similar applies to TICTOC and SILO).

Insight: Under high conflict the heavier transactions of the full TPC-C transaction mix make concurrency control of all CC schemes more costly. However, at large scale, these heavier transactions also dampen the impact of conflicts, especially benefiting CC schemes that efficiently handle read-write conflicts.

POWER VS. HPE Figure 8.22 displays the throughput of the CC schemes for the full TPC-C transaction mix on Power9 and Power8. Additionally, this figure provides a comparison to the narrow mix on these hardware platforms and the difference to the comparison to the narrow mix on HPE. The behaviour of the CC schemes for the full transaction mix on Power8/9 broadly resembles their behaviour on HPE. The most noticeable difference is that throughput is generally lower, i.e., the heavier transactions reduce throughput on Power8/9

more than on HPE. Accordingly, at low core counts the full mix lags further behind the narrow mix and at high core counts it is less ahead on the Power platforms.

The general cause for the slowdown for the full transaction mix on Power is the same as on HPE, i.e., especially at low core counts the heavier transactions make concurrency control more costly. Furthermore, the following three differences between the Power and the HPE hardware platforms stand out:

1. As the number of cores increases on Power, especially the pessimistic locking schemes benefit far less from the full mix compared to HPE. HSTORE even degrades on Power9 and Power8, with increasing numbers of cores it increasingly falls behind its throughput for the narrow mix. That is, these CC schemes as well as SILO and TICTOC deviate further apart from their performance for the narrow transaction mix as the number of cores increases on Power. For the pessimistic locking schemes the time breakdowns reveal more time spent aborting rather than waiting. The time breakdowns of HSTORE, SILO, and TICTOC are very similar on the three hardware platforms, i.e., their behaviour is the same, but hardware performance makes the difference.

2. In contrast to the prior CC schemes, OCC copes better with the full mix on Power than on HPE. On Power9, OCC speeds up by 2.5x over the narrow mix, while on HPE it slows down by 0.4x. On Power8, OCC only reduces the performance gap, reaching 0.84x speedup at the maximum of cores.

3. Only MVCC does not diverge further, providing constantly less speedup (-0.10x) on Power than on HPE. Indeed, the time breakdowns of MVCC are similar for all three hardware platforms, i.e., its behaviour is the same, but hardware performance differs.

Insight: In conclusion, TICTOC and SILO handle high amount of conflicts best, regardless the hardware or workload type (full and narrow TPC-C mix), while MVCC proves its conflict handling advantageous for (read-)heavy workload (full TPC-C). Moreover, the specific performance of the CC schemes for heavier transactions as in the full TPC-C mix depends on the underlying hardware and the number of utilised cores, making for varying relative performance of the CC schemes across the hardware platforms and workload types.

8.4.3.2 *Full TPC-C under Low Conflict*

The previous experiment with the full TPC-C transaction mix under high conflict indicated that besides more conflicts also the higher load on the hardware impacts performance. That is, even for the high conflict workload that generally limits hardware utilisation, the increased load of the heavy transactions influences the CC schemes.

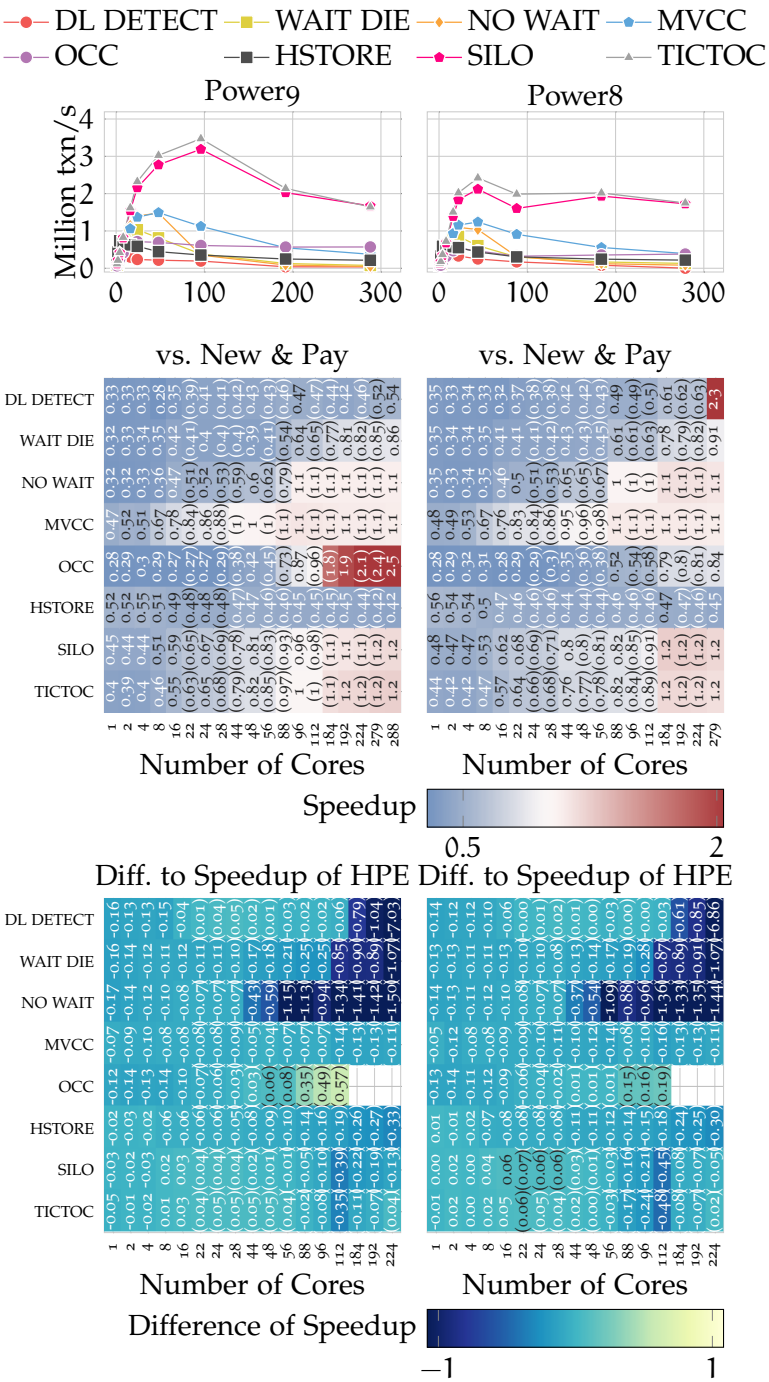


Figure 8.22: Throughput for full TPC-C mix under high conflict on Power9/8 with comparison to narrow mix (NewOrder & Payment) and in 3rd row difference to comparison of full vs. narrow mix on HPE (cf. Fig. 8.21a).

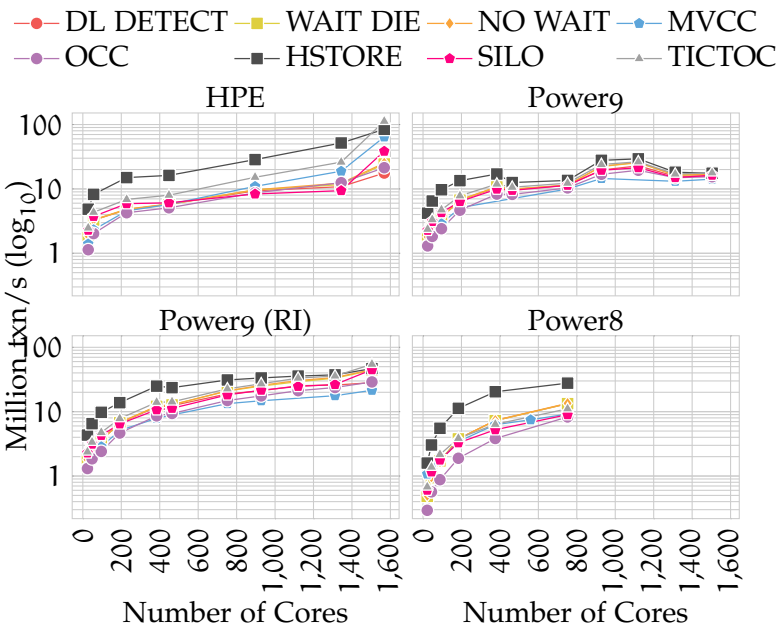


Figure 8.23: Throughput of optimised DBx1000 for full TPC-C transaction mix under low conflict on HPE, Power9, and Power8. The full mix causes a performance bug on Power9, which is removed in "Power9 (RI)" by replicating internal data structures.

Consequently, in the absence of conflicts (low conflict workload), hardware utilisation is the main factor determining the performance of the CC schemes.

First, we provide an overview of the performance of the CC schemes for the full mix on all three hardware platforms (indicating significant differences between those). In the next step, we then contrast the behaviour of the CC schemes for the full mix with the narrow mix on the same hardware platform (i.e., HPE) to identify divergences due to workload characteristics. In the final step, we then compare these divergences of the CC schemes for the full transactions mix across the three hardware platforms to distinguish trends relating to either workload or hardware characteristics.

COMPARISON OF CC SCHEMES Figure 8.23 provides an overview of the throughput of all CC schemes for the full TPC-C mix under low conflict on all three hardware platforms. As expected, it shows that the CC schemes have a generally positive scaling behaviour on HPE and Power8, i.e., throughput increases with increasing number of cores. However, in comparison to the narrow transaction mix, throughput is overall lower (cf. Figure 8.15). We will compare the narrow mix in detail below.

On Power9, though, the full mix causes anomalous behaviour for all CC schemes, due to a combination of caching and NUMA effects caused by the higher memory footprint. Specifically, pointers to access

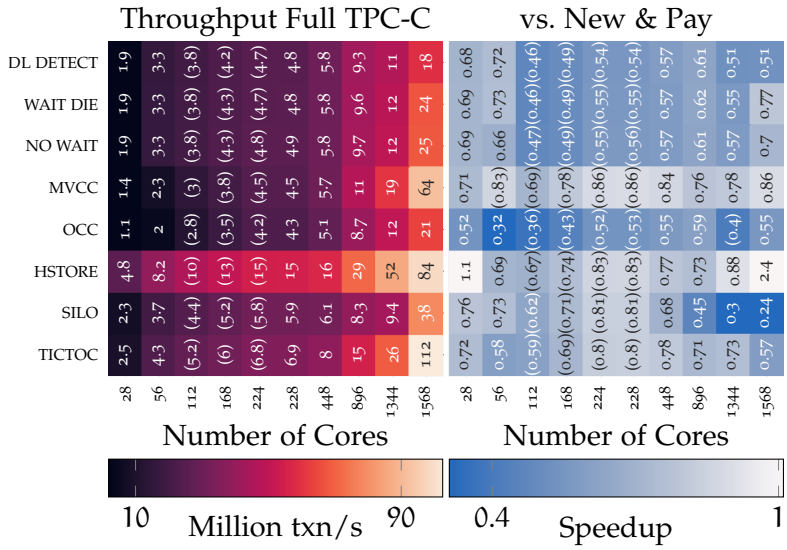


Figure 8.24: Detailed throughput for the full TPC-C transaction mix under low conflict on HPE with comparison to the narrow mix. The comparison indicates the speedup ratio by which the throughput differs from the narrow transaction mix (cf. Figure 8.15) for a CC scheme (on y-axis) at the same number of cores (on x-axis).

indexes drop out of the individual processor caches and have to be fetched from potentially distant memory, causing significant slowdown as number of cores increases and subsequently the NUMA distance between them.

We thus further optimised DBx1000 for Power9 by copying these crucial pointers into the local memory of each processor to reduce the memory access cost but have the cores of each processor share these pointers (at most one copy in each processor cache). The results of this optimised variant of Power9 (called *Power9 (RI)*) indeed show a similar behaviour to Power8 and HPE. Notably, this optimisation for Power9 has only minimal effect for the narrow transaction mix, due to the smaller footprint of the involved transactions.

FULL VS. NARROW MIX ON HPE In the following, we compare the full and the narrow mix on HPE. On HPE, the transactions of the full TPC-C mix, indeed show the expected benefit of MVCC, generally handling read-heavy transactions better. Also under low conflict MVCC becomes third best for the full mix at high core counts, which is different from the narrow mix. Conversely, SILO falls behind for this full mix.

In more detail, Figure 8.24 shows the detailed throughput for the full TPC-C mix on HPE and a comparison to the narrow mix. The full mix reduces the throughput of all CC schemes, but distinctly for the individual CC schemes. The best performing HSTORE and the improved MVCC slow down least (HSTORE: 0.69 - 0.88x, MVCC: 0.71 -

0.86x). At highest core count, HSTORE even speeds up by 2.4x and does not thrash as in the narrow mix. TICTOC follows with a slightly stronger slowdown, especially under NUMA effects at 56 cores and the highest core count. Next, the group of pessimistic locking schemes increasingly slows down until 1344 cores, even more so SILO with a significant slowdown of 0.24x at 1568 cores. Lastly, OCC is significantly affected by the full transaction mix (0.5x). A comparison of the time breakdowns reveals higher coordination costs as the main reason for the overall lower performance for the full transaction mix, i.e., even in this low conflict workload, the heavy transactions increase the time spent for coordination for all CC schemes.

Insight: As a major observation, the more involved (i.e., long-running) transactions of the full TPC-C mix do not simply increase the amount of actual work, but their increased footprint indeed impacts concurrency control, for both the high conflict and the low conflict workload. Besides a dampening effect on conflicts and the benefit of MVCC, the individual CC schemes slowdown distinctly. Hence, we conclude that the (read-)heavy transactions of the full TPC-C directly amplify the cost of the individual CC schemes.

POWER VS. HPE Finally, the comparison of the results for the full TPC-C under low conflict across the different hardware platforms confirms the general slowdown on the Power platforms and similar the slowdown trends of most CC schemes[5].

Importantly, the comparison across the hardware platforms confirms our observations on the relation of their hardware characteristics to the behaviour of the CC schemes, albeit leading to different performance. On one hand, the heavier transactions of the full mix cause stronger resource contention on Power, such that on both Power platforms the slowdown at low core counts is stronger than on HPE. On the other hand, the CC schemes scale better on Power, due to their better NUMA characteristics, finally reaching similar or less slowdown than on HPE at highest core counts. Notably, on HPE we previously observed a significant benefit of MVCC handling the read-heavy transactions of the full mix. On Power the resource contention cancels out this benefit of MVCC.

Insight: The full TPC-C transaction mix makes concurrency control even more costly on Power regardless the amount of conflicts in the workload, i.e., larger processor resources on HPE prove more beneficial than the better NUMA characteristics on Power for the full mix in contrast to the narrow mix. Overall, considering both transactions mixes, the CC schemes compare for low conflict workload as follows: (1) HSTORE provides the best performance (on any hardware) as long as there are barely any conflicts, i.e., even few conflicts inhibit its performance (e.g., even low conflict TPC-C at high core counts). (2)

5 Detailed figures omitted for brevity are available online [19]

TICTOC performs most reliably (even for both low and high conflict workloads). The remaining CC schemes compare diversely. Their performance depends on the characteristics of the individual hardware platforms (NUMA and cache capacity) and the workload. For example, due to the large memory footprint of the read-heavy transactions, MVCC does not prove advantageous on all hardware (i.e, on Power), despite targeting read-heavy transactions.

8.5 CONCLUSION AND FUTURE WORK

In this paper, we presented the results of our extensive analysis of concurrency control on real(ly) large multi-socket hardware as major component of OLTP DBMS. To conclude, we first summarise our major findings. Based on these findings, we then discuss our recommendations as well as possible future directions towards high and robust performing OLTP DBMSs.

8.5.1 *Discussion of Findings*

In the following, we summarise the main findings of the two evaluation parts and conclude with final insights.

FINDINGS OF PART ONE In the first part of our evaluation, we revisited the simulation of OLTP on then predicted large many-core hardware [206] and compared it to an Intel-based hardware which does provide "a 1000 cores" today but as multi-socket hardware. We identified several discrepancies between the original simulation and real hardware in our evaluation. Importantly, we showed that all CC schemes indeed scale well beyond 1000 cores for low conflict workload, when using state-of-the-art optimisations. Notably, due to shifting bottlenecks, combinations of optimisations were necessary, e.g., hardware-assisted timestamp allocation only improved performance with additional optimisations, as shifting contention then caused latch thrashing. Among the evaluated CC schemes, the now included TICTOC outperformed all other schemes for both low conflict and high conflict workloads. However, all of them including TICTOC still become overwhelmed by conflicts under high contention, degrading even more drastically on our real Intel-based hardware than in the simulation.

FINDINGS OF PART TWO In the second part of our evaluation, we then presented the results of our broadened evaluation, additionally comprising two IBM Power-based hardware platforms and the full transaction mix of TPC-C. With this broadened evaluation, we indeed confirmed our initial observations and further connected the performance of CC schemes with hardware and workload characteristics,

e.g., NUMA effects, processor resources, conflicts, and transaction footprint.

We observed common outstanding and complex nuanced effects of hardware and workload characteristics. First, under high contention, the previously observed thrashing caused by overwhelming conflicts persisted regardless of hardware or other workload characteristics, impeding adequate utilisation of all our large hardware. A major cause of this thrashing of all CC schemes is the simple inter-transaction parallel execution scheme commonly used in today's DBMS [65, 103, 106, 112, 183], as it can only utilise high hardware parallelism with high transaction concurrency, amplifying contention. Second, we observed more nuanced effects from the interaction between CC schemes, hardware, and workload. Different hardware characteristics proved significant depending on the design of the CC schemes, e.g., temporary copies of optimistic CC demand cache capacity and bandwidth, while locking of pessimistic CC is latency sensitive. Notably, we observed negative effects only when this demand exceeded the available cache capacity or bandwidth. That is, these capacity related effects did not appear as long as sufficient resources were available. Moreover, the workload further influenced this interaction between the CC schemes and the hardware. For example, the transaction footprint (accessed tuples) amplified the cache demand, degrading performance of optimistic CC when the cache was too small. However, at the same time, the transaction footprint also alleviated contention off latches indeed improving performance of pessimistic CC. Consequently, hardware and workload have complex effects on CC schemes and overall bottlenecks in the DBMS.

INSIGHTS FROM FINDINGS The bottom line of our findings is, that an agglomeration of bottlenecks in the system determines the cost of transaction execution and overall system performance. To reason about the scalability of DBMSs, it thus is important to understand how the cost of individual bottlenecks scales and how these bottlenecks interact. In this regard, our evaluation has shown complex effects of workload and hardware as well as complex interaction of bottlenecks adding up, amplifying each other but also dampening or hiding each other. Then, to achieve high performance, these costs must be balanced against the available hardware resources considering the workload at hand. To conclude, we argue that maintaining the ideal balance despite changing workloads and evolving hardware will enable robust DBMS performance.

8.5.2 *Recommendations and Peek into the Future*

Based on our findings, we now discuss our recommendations to achieve robust DBMS performance and point out according research avenues.

COMPREHENSIVE CONTENTION MANAGEMENT As discussed above, all evaluated CC schemes scale poorly, surrendering to conflicts and contention. Even advanced CC schemes with conflict mitigation mechanisms do not reliably withstand many conflicts, like TICTOC or beyond the evaluated ones CICADA [90, 120, 148, 201, 208]. Moreover, besides logical contention of transaction conflicts, also physical contention (e.g., on latches) significantly impacts performance and system components outside of the CC scheme strongly affect contention (logical & physical), especially the simple but common inter-transaction parallel execution scheme.

As a consequence, we recommend broader system-wide contention management far beyond the CC scheme, since system-wide many factors affect contention and there are more options to efficiently manage contention. For example, system-wide contention management (especially across concurrency control, execution scheme, and scheduler) could reduce logical contention with smarter parallel execution rather than with conflict mitigation in CC schemes, thereby reducing contention more efficiently. As such, we envision contention management throughout the entire system to better balance contention shifting across system components, which is a big challenge in achieving robust performance. While there is work in this direction [53, 143, 188, 199, 200, 205, 210], we believe that extending these to our notion of system-wide contention management and extending to stronger awareness of the underlying hardware would greatly benefit DBMS performance. In particular, interesting directions are broader forms of parallel execution besides inter-transaction parallelism and a transaction scheduler, which is aware of system-wide contention and the interaction with hardware like cache competition. A future route along this line would be to choose the appropriate form of parallel execution of transactions, e.g., inter-transaction parallel execution for uncontended workloads or where contention is "beneficial" (i.e., due to resource sharing), intra-transaction parallel execution for contending transactions, and even sequential execution under excessive contention.

We further recommend adaptive concurrency control as part of comprehensive contention management. Adaptive CC is recognised for employing the most suitable CC scheme for a group of transactions or partition of tuples, e.g., CormCC [180] is an outstanding candidate with low overhead cooperation between a host of CC schemes. We believe that adaptive CC should determine the CC schemes as part

of our proposed system-wide contention management, as those CC schemes are strongly affected by many factors of the overall system but in turn strongly affect the system. For example, conflict frequency on a tuple (logical contention) is an important feature to determine the CC scheme, which again is strongly affected by transaction scheduling.

ADVANCED PERFORMANCE MODELS Other important aspects shown in our evaluation are the complex dependencies of DBMS performance, i.e., the system design, the workload and the underlying hardware interacting and jointly affecting thrashing points. Due to these complex dependencies, we argue for comprehensive (e.g., learned) performance models that can better reflect these complex effects. Current work on performance models facilitating synthesis of data structures [91] and recent progress on learned components [58, 86, 125, 163] spark our confidence in learned performance models [85] deriving complex dependencies beyond the ability of purely analytical models. Such models would then not only help to inform contention management as discussed above but also would open new opportunities, e.g., finding the optimal hardware for a given workload. In the long term, such performance models would further enable quick exploration of system performance without extensive benchmarking and could eventually lead to performance guarantees of DBMSs.

ADAPTIVE SYSTEM ARCHITECTURES Our findings point out, that optimal system performance requires the system design to ideally balance bottlenecks. However, this balance differs for workloads as well as hardware and changes over time, due to workload fluctuation but also progress of state-of-the-art (e.g., optimisations). Hence, we advocate for adaptive systems and especially adaptive system architectures capable of re-balancing the system design. Beyond the proposed adaptation and synthesis strategies [91, 93, 140, 180], we argue for flexible system-wide adaptation, which exceeds adaptation of individual components and opposes rigid instance optimisation. Towards effective system-wide adaptation, performance prediction and adaptation overhead are significant challenges. As recommended above, performance predictions will benefit from advanced performance models, whereas we consider flexible system architectures and execution models to drive efficiency. Specifically, we envision the decomposition of system designs into fine-grained building blocks which can be efficiently composed at runtime [18] (e.g., by new compilation techniques). This will enable flexible system architectures to broadly transform a system balancing bottlenecks across all system components. Thereby, we envision adaptive system architectures to successfully adjust to changing workloads and shifting hardware balances.

ARTEFACT AVAILABILITY Finally, another important route is that data of extensive evaluations like this should be made available for the community. We have released all artefacts [19, 20] of this evaluation,

making these available to the database community for further research. We believe that despite our extensive analysis in this paper the data itself is a valuable source for future research. So, we hope that this source helps researchers and system builders to further dig into details they find interesting and come up with their own conclusions.

ACKNOWLEDGEMENTS We would like to express our great gratitude to the authors of [206] for providing DBx1000 as open-source making this work possible. Furthermore, we also would like to thank IBM and HPE for providing us extensive access to the hardware used in this paper.

9

ROBUST PERFORMANCE OF MAIN MEMORY DATA STRUCTURES BY CONFIGURATION

ABSTRACT

In this paper, we present a new approach for achieving robust performance of data structures making it easier to reuse the same design for different hardware generations but also for different workloads. To achieve robust performance, the main idea is to strictly separate the data structure design from the actual strategies to execute access operations and adjust the actual execution strategies by means of so-called configurations instead of hard-wiring the execution strategy into the data structure. In our evaluation we demonstrate the benefits of this configuration approach for individual data structures as well as complex OLTP workloads.

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the peer-reviewed work *Robust Performance of Main Memory Data Structures by Configuration* by Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig in the *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. The contributions of the author of this dissertation are summarized in Part I Chaper 4 Section 4.1.

9.1 INTRODUCTION

MOTIVATION: Within the last decade, we have seen different hardware trends that significantly affected the design of single-node database systems: (1) Increases in main-memory capacities made it possible to hold even larger data sets in RAM, thus eliminating the I/O bottleneck of accessing secondary storage (e.g., hard drives). (2) Moore's Law and Dennard Scaling required processor designers to move from single-socket and single-core designs to multi-socket and multi-core designs. As a result of these trends, we have seen a rapid evolution of hardware designs differing in essential characteristics not only memory capacities but also the underlying topology of how cores and memory are connected as well as cache sizes and coherence protocols.

A considerable body of existing work in DBMS research has thus focused on optimising the design of core DBMS data structures such as indexes for specific hardware configurations and workloads. For example, there have been various design alternatives proposed for classical B-trees to adapt them to modern memory hierarchies and make them more cache-conscious for read-heavy workloads [150, 151] or to optimise their behaviour for high-contention scenarios [117] under write-heavy workloads. A significant issue with this manual tailoring of core DBMS data structures is that not only their redesign involves high effort and reintegration into the DBMS but also that a design optimal for one hardware generation and one workload might induce severe performance degradation on another hardware generation when underlying assumptions change.

An alternative to this approach is designing data structures that can provide robust performance [73]. At its core, robust performance means the ability of a data structure to provide acceptable performance for a wide variety of hardware configurations and environmental conditions without adjusting the fundamental data structure design. Achieving robust performance for a data structure, however, is a non-trivial problem because there can be many superimposed causes degrading its performance, not all of which are foreseeable given the speed modern hardware platforms evolve.

CONTRIBUTION: In this paper, we thus present a new approach for achieving robust performance. Instead of proposing a single design that is robust against different workloads and hardware characteristics, we suggest that data structures can be adapted to a workload and hardware by simple means of a configuration. The main idea to achieve this goal is to strictly separate the design of a data structure from the actual access operations and use a configuration policy for defining the strategy of how to execute access operations on a particular data structure in a declarative manner. This strict separation provides us then with the flexibility to control execution by simple means of a

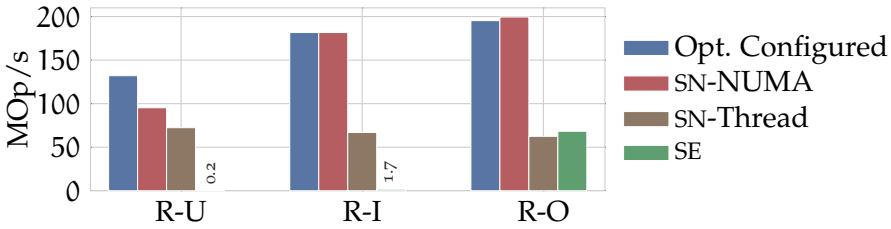


Figure 9.1: Robust throughput of the *FP-Tree* index on 8 sockets through individually optimal configuration using virtual domains (*Opt. Configured*) for Read-Update 50/50 (R-U), Read-Insert 95/5 (R-I), and Read-Only (R-O) YCSB workloads. Baselines are rigid partitioning strategies: Partition per NUMA region (*SN-NUMA*), partition per thread *SN-Thread*), and a shared-everything strategy (*SE*) without any partitioning.

configuration that determines how the access operations are actually executed, making the best use of the underlying hardware.

Clearly, the configuration policy is at the centre of our approach. Thus a key question is: How is it defined and what is its utility? The intuition behind a configuration policy is that it partitions the resources (CPU cores as well as memory) of a given multi-socket machine into so-called *Virtual Domains*. This configuration policy is then used by the runtime system to route tasks submitted by client threads to the responsible virtual domains and send the results of a task back to the client. One could now think that this sounds very much like NUMA-aware processing strategies which modern DBMS engines implement already today to split the resources of a machine and partition the data structures accordingly.

However, NUMA-aware processing strategies solely split the resources based on the hardware topology [103, 137, 141, 154] (i.e., by sockets with their local memory or by single cores). But, they ignore many important aspects of the software stack on top, such as the characteristics of a given data structure and the workload which may (heavily) degrade performance. For example, as we show in Figure 9.1, when using write-heavy workloads for a modern tree-based index structure design that leverages Hardware Transactional Memory (HTM) of modern CPUs [135] we can see that, however, when more than half of the cores of a socket concurrently access the index structure, the performance degrades heavily due to aborting memory transactions.

In contrast to classical NUMA-aware processing strategies, our approach based on virtual domains allows to split the resources of a given machine in arbitrary granularity (e.g., into virtual domains that span only half a socket) in order to control contention for the data structures in an optimal manner. As shown in Figure 9.1, our flexible configuration strategy can provide superior performance across different workloads over the rigid partitioning strategies.

OUTLINE: Section 9.2 discusses the basic intuition of how to provide robust performance for a real system which hosts many different data structures before we give an overview of our approach in Section 9.3. Sections 9.4 to 9.6 then present the details of our main building blocks. Afterwards, in Section 9.7, we present the evaluation showing the efficiency of our approach for different data structures as well as for executing a typical OLTP workload. To wrap up, Section 9.8 gives an overview of related work, and Section 9.9 concludes the paper.

9.2 THE ART OF ROBUST PERFORMANCE

There exist many different causes for degraded performance of core data structures in main-memory databases on multi-socket hardware. In this paper, we focus on OLTP databases whose workloads are mainly characterised by different mixes of read and write statements ranging from read-heavy to write-heavy mixes where these operations are typically executed over index structures such as modern versions of B-trees or hash-tables. In the following, we first discuss the main causes of performance degradation and how current approaches handle them before we elaborate on our approach to robust performance by (re-)configuration.

9.2.1 *Pitfalls of Rigid Architectures*

The sources of performance degradation can be manifold. One primary reason that causes performance degradation of core data structures such as B-trees or hash-tables in main-memory OLTP databases is the overly high-contention that results from concurrent accesses (reads and writes) to the same instance of an index structure [154]. Other reasons for performance degradation include increased latencies as a result of cross-socket memory accesses or high cache coherence traffic resulting from concurrent reads and writes to the same memory [202]. In order to mitigate these effects, different strategies have been devised.

A prevalent strategy to address the aforementioned issues is (as discussed in the introduction already) to use a NUMA-partitioned DBMS design to mitigate the negative side-effects of cache coherence and increased latencies caused by cross-socket traffic [116, 139, 146]. However, this design can still lead to degraded performance since partitioning data structures at the granularity of a socket can also turn out sub-optimal leading to a too high contention for some data structures and workloads [49, 63], as we demonstrate in our experiments.

Hence, another direction that systems like H-Store [103] or Orthrus [154] suggest is to partition the database in an even finer-grained manner per hardware thread. While this design avoids performance degradation due to high contention, it has several other drawbacks such as its sensitivity to skew or the fact that more complex workloads

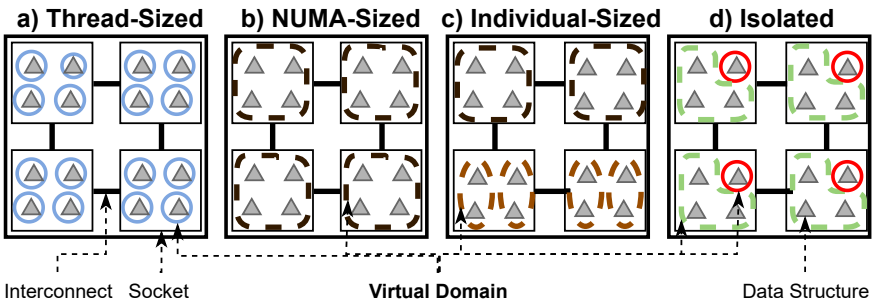


Figure 9.2: Flexible partitioning via configuration of *virtual domains* for a 4-socket machine: (a) *Thread-sized* with virtual domain per core. (b) *NUMA-sized* with a virtual domain per socket. (c) *Individual-sized* with two sizes of virtual domains. (d) *Isolated* with separate virtual domains for hot data structures.

cause an increased coordination overhead between partitions. Consequently, some systems such as Hekaton [52] even suggest avoiding partitioning and use a shared-everything approach instead, to mitigate the negative impacts of partitioning.

9.2.2 *Robust Performance By Configuration*

While all the afore-mentioned rigid partitioning strategies have their sweet spot(s), they can also cause severe performance degradation depending on the workload and data structures in use as we show in our experiments. In this paper, we thus propose a different route and suggest an approach enabling a flexible execution strategy that can adapt all these strategies ranging from thread-sized partitions to shared-everything by simple reconfiguration. The basic idea is that based on the mix of data structures and workload present in a concrete instance of a DBMS, we can provide a configuration using so-called virtual domains partitioning hardware resources in an optimal manner.

As shown in Figure 9.2 c) and d), virtual domains provide many more configuration options beyond what the rigid strategies (shared-everything or NUMA/thread-sized shared-nothing) can provide: First, when splitting the resources of a machine into virtual domains, not all virtual domains need to have identical sizes in terms of CPU or memory, but we can define virtual domains with different sizes to ideally support a mix of different data structures and workloads within a single system. Second, another configuration option provided by virtual domains is the isolation of hot data structures into separate virtual domains using a dedicated set of resources to enable more stable performance.

An important issue is that workloads in DBMS also might change over time and thus require reconfiguration of a hardware platform into larger or smaller virtual domains. At the moment, our approach

handles this by offline reconfiguration, i.e., all active operations in the system must complete before a reconfiguration can be applied and the system can then restart with a new configuration. This offline approach can be used for reconfiguration if changes in workloads are known a priory or can be predicted based on reoccurring patterns (e.g., for Black Friday). In the future, we plan to extend our approach further to support online reconfiguration at runtime and thus also support cases where the workload changes are less predictable.

9.3 SYSTEM OVERVIEW

In the following, we provide an overview of the main building blocks of our approach before discussing how to integrate our approach into a DBMS.

9.3.1 *Asynchronous Tasks & Configurations*

The two main building blocks an application needs to provide are asynchronous tasks implementing the access operations on data structures and a configuration that assigns data structures to optimally sized system partitions (virtual domains).

An asynchronous task is a container for an access method defined by the application, e.g., an insert or a lookup operation on a B-Tree. In contrast to operating system threads, tasks in our approach not only are much more lightweight but also are *data-aware*; i.e., a task is only executed inside the virtual domain where the data structure resides. This notion of tasks allows us to fully control contention and locality of access methods by simple means of a configuration.

In addition to tasks, the application can specify a configuration to control contention and locality of access methods for a given set of data structures. A configuration comprises two parts: (1) The first part of a configuration defines which *virtual domains* are being used to execute a given workload. Here the important aspect is the definition of how many domains are used and how resources are allocated to each virtual domain independent of the underlying hardware topology. (2) The second part of a configuration defines how data structure instances are mapped to virtual domains. Notably, an application may split a data structure into several instances and assign them to separate virtual domains to achieve higher throughput. In Section 9.5, we discuss an ILP-based approach to find an optimal configuration that maximises the overall system throughput given a workload and a set of data structures. According partitioning strategies for data structures are implemented by the application (i.e., the DBMS [15, 136]) on top of our runtime system. However, as we show in our experimental evaluation in Section 9.7 with our approach, DBMSs become less sensitive to the actual partitioning strategy being used

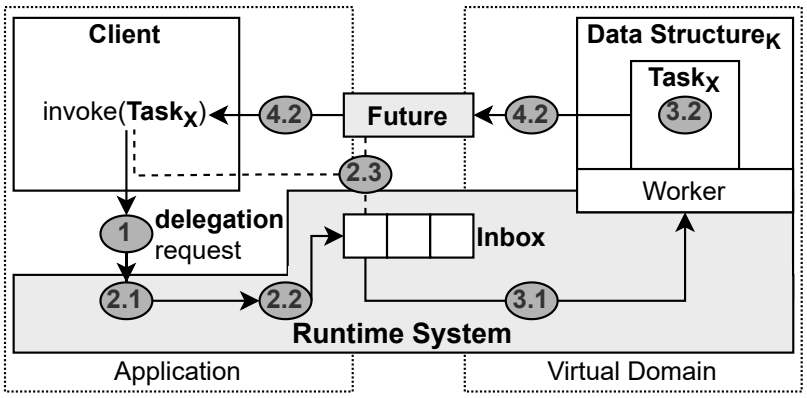


Figure 9.3: System Overview and Execution Flow: Client threads delegate *asynchronous tasks* to workers in a *virtual domain* which reply using *futures*. A configuration maps clients to virtual domains and workers.

since our approach seamlessly handles severe issues such as locality and contention.

9.3.2 *Runtime System*

The main objective of our runtime system is the efficient execution of tasks given a configuration. For efficient task execution, the runtime system provides a simple delegation mechanism based on highly optimised in-memory message passing. Noticeably, the aim of the runtime system is not to provide a full-fledged DBMS but to act as a thin *virtualisation* layer on top of the hardware providing the foundation for robust performance of a DBMS built on top. Below, we discuss the potential direction of how our runtime system can be integrated into a full DBMS.

Figure 9.3 presents an overview of our runtime system. A client thread submits an *asynchronous task* to be executed (step 1) and obtains an invocation handle, so-called *future*, on the submitted task (step 2.3) to consume the result of the task execution. Internally, the runtime system identifies the *virtual domain* responsible for the referenced data structure upon the invocation of an asynchronous task (step 2.1). It then places the task into the corresponding inbox (step 2.2) returning the future (step 2.3). For efficient message passing between virtual domains their inbox uses a fixed number of slots; details follow in Section 9.6.

The counterpart to the application's client threads are *worker threads* inside a virtual domain. These workers continuously poll the inbox for new tasks. Once a worker detects a new task (step 3.1), it executes the task (step 3.2) within the virtual domain on behalf of a client thread. Upon its completion, the *task* places its result in the earlier allocated *future* (step 4.1) from which the *client* retrieves the result (step 4.2).

9.3.3 *Discussion of DBMS Integration*

As mentioned before, the main contribution of this paper is not to provide a full-fledged DBMS. However, we believe that our delegation-based runtime system can be used for implementing a DBMS. In fact, we show in our experimental evaluation that we are able to execute typical OLTP workloads by implementing a "light-weight" OLTP engine on top of our runtime system. In the following, we discuss the main design choices involved in building an OLTP engine utilising our runtime system, though.

A first design choice for using our runtime system for OLTP is the mapping of transaction logic (i.e., the sequence of reads and writes) to tasks that can be executed by our runtime. A naïve way for this is to map every individual read/write operation of a transaction to a separate task to be submitted to our runtime system by the OLTP engine. Moreover, our programming model also allows more sophisticated implementations where transactions are chopped into sub-transactions and then are mapped to tasks as a whole. Studying the detailed effects of chopping is an interesting route for future work though. As we show in our experiments in Section 9.7.3, the naïve mapping already enables an efficient execution of OLTP.

A second design choice in addition to mapping transactions to tasks, is how tables of a database (and their indexes) are distributed across virtual domains. For this purpose, we introduce a configuration procedure in Section 9.5 that takes a set of data structures as input (i.e., the tables and indexes of a database) and compiles a configuration aiming to maximise the overall throughput for a given workload. Before applying this configuration procedure, the DBMS can still apply conventional partitioning strategies on tables as mentioned above and input these table partitions (as well as their indexes) as data structures to our configuration procedure.

In addition to these two main design aspects (i.e., mapping transactions to tasks as well as finding optimal configurations for a set of tables), further DBMS components need to be implemented, such as concurrency control as well as recovery mechanisms. The design of those components, however, is orthogonal to the contributions of this paper since many different schemes can be implemented on top of our runtime system. For instance, our runtime system allows DBMS to implement any concurrency control schemes ranging from pessimistic locking to various optimistic schemes. For our evaluation in Section 9.7.3, we hence omit these components for our "light-weight" OLTP engine as well as for all baselines (for a fair comparison), i.e., for concurrency control, we rely on latches to avoid data races but do not prevent other anomalies (e.g., lost updates). While this allows no direct comparison with other full-fledged DBMSs incorporating those components, it still allows us to compare the benefits of our execution

scheme for OLTP workloads compared to more classical OLTP engine designs where data is partitioned by NUMA regions and transaction managers directly execute operations without delegation.

Finally, an interesting future aspect when designing an OLTP engine on top of our runtime system is that the asynchronous execution model opens up many new opportunities for optimisations. For example, in a classical design of an OLTP engine, transaction manager threads execute only a single transaction at a time, whereas a design building on our runtime system could rethink this model allowing transaction manager threads to execute operations on behalf of multiple transactions at the same time. That is, when an operation of one transaction is submitted to our runtime, the transaction manager could submit operations on behalf of another transaction instead of blocking until the results of the first transaction are available. However, analysing these optimisations is beyond the scope of this paper and needs a more thorough investigation in our future work.

9.4 PROGRAMMING MODEL

In this paper, we propose a new approach for task-based programming. While asynchronous task-based programming is not new and has also direct support in different programming languages such as Erlang and C# [23, 57, 153, 186, 197] as a lightweight alternative over threads, we propose a novel abstraction called *asynchronous data-aware tasks*.

The essential aspect of a data-aware task is that it only allows accessing a data structure within a single *virtual domain* using precisely the configured resources of that domain. Therefore, a data-aware task must be executed by a worker thread inside a virtual domain. This concept allows us to control not only the degree of contention by simply re-configuring a virtual domain (i.e., by changing the size of the domain and thus the number of worker threads that have concurrent access) but also other transient properties such as cache state and cross NUMA-node traffic which are bound to the virtual domain as well.

Listing 9.1: API of an *Asynchronous Task*.

```
1 class Task {
2     Task(void* dataStructure, Args... args)
3     void operator()(Result& res);
4 };
```

In order to implement a data-aware task, the outlined API of a task (see Listing 9.1) only requires the first parameter of the constructor to be the targeted data structure and to implement the function *operator()* to return results of the task using the *Result* object. Additionally, this abstraction must also encapsulate all input parameters for the contained operations. In combination, this simple API enables the

runtime system to route the task to the corresponding virtual domain based on the referenced data structure.

Listing 9.2: A task to insert a record into a table.

```
1   class TaskInsertRecord {
2       Table* tab; // Pointer to table
3       Record* rec; // Pointer to buffer of record
4       TaskInsertRecord(Table* table, Record* record):
5           tab(table), rec(record){};
6       void operator()(Result &res){
7           // Read buffer and insert record
8           RowID rowID = tab->insert(*rec);
9           delete rec; // Delete buffer
10          res.set(rowID); // Return inserted row id
11      }
12  };
```

To show that this programming model can also be used to implement typical operations of a transaction, the example in Listing 9.2 implements a task to insert a record into a (partition of a) table. While Listing 9.2 is a simple and illustrative example, the DBMS could also use tasks to implement more complex operations involving several data structures within the same virtual domain or fusion of several operations.

9.5 ROBUSTNESS BY CONFIGURATION

The main aspect for configuration is the definition of virtual domains partitioning the resources of a given hardware platform. In this section, we define the configuration options of virtual domains before we outline the process of how to find a configuration for a workload and the set of data structures comprising the overlaying application.

9.5.1 *Virtual Domains*

A *Virtual Domain* is defined as a set of dedicated logical (SMT) cores, a worker thread placement policy (i.e., if it allows thread migration or requires strict pinning to cores), and a memory allocation policy (e.g., strictly local to individual workers or interleaved across all workers). In this regard, we *virtualise* NUMA-regions which directly represent the hardware topology into flexibly configurable regions. We establish these *virtualised* hardware regions as *domains* to control worst-case contention and data locality, thus the name *virtual domains*.

In particular, only worker threads of a virtual domain are allowed to execute tasks on the data structure instances assigned to that virtual domain, and thus, no side effects can cross its boundaries. Moreover, since all operations on the data structure instance are executed as tasks by dedicated workers of a virtual domain, cache state exclusively

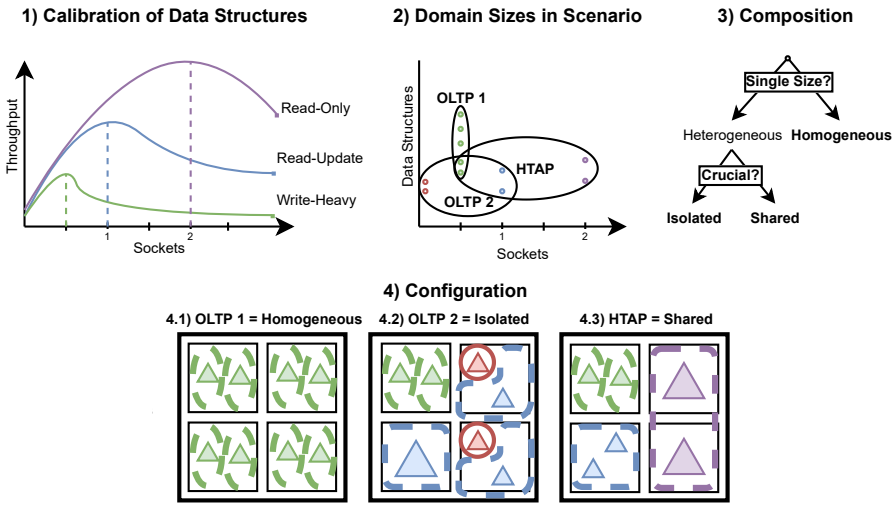


Figure 9.4: Configuration Process: 1. Calibration of domain sizes for the best trade-off between contention and locality. 2. Optimal domain sizes for different scenarios (OLTP1, OLTP2, HTAP) based on calibration. 3. Composition of virtual domain as *homogeneous* or *heterogeneous* configurations. 4. Resulting configurations for exemplary scenarios.

resides in the respective CPUs (cache locality) and only these CPUs synchronise for cache-coherence on that cache state [95].

Consequently, virtual domains limit the *(1) worst-case contention* of tasks in a virtual domain to the number of worker threads and *(2) worst-case locality* using the placement policies. Thus, virtual domains provide configurable contention control and locality which can be flexibly specialised for distinct data structure instances expose to diverse conditions through co-existing virtual domains in a single system.

9.5.2 *Configuration Process*

Having the means to control contention and locality of distinct data structure instances through virtual domains, the configuration process is about finding the individually optimal domain sizes and their composition into a single configuration. The overall process of finding a configuration that defines which virtual domains should be used and how data structures are mapped into the domains is shown in Figure 9.4.

The first step of the process (step 1, Fig. 9.4) is a calibration phase that gathers performance metrics and quantifies the performance behaviour of the different data structures involved in a workload. The goal is to find the optimal domain size for each data structure instance involved in that workload individually (step 2, Fig. 9.4). Subsequently,

we start the composition process (step 3, Fig. 9.4) which, based on the calibration information, divides the system resources and maps data structure instances into virtual domains to produce a configuration (step 4, Fig. 9.4).

Notably, the configuration does not partition the data structures themselves. Instead, we expect the application to partition the data structures utilising application-specific knowledge (e.g., partitioned indexes in a DBMS) while the goal of the configuration process is to find an optimal assignment of those partitions to virtual domains. For finding an optimal configuration for these partitioned data structures, the application can define constraints which data structure instances should be mapped into the same virtual domain to realise co-location of data structure partitions (e.g., an OLTP DBMS could co-locate data of several tables in one virtual domain avoiding transactions across virtual domains).

CALIBRATION OF DOMAIN SIZES: The calibration phase executes a given workload under growing domain sizes (i.e., with an increasing number of threads) for each data structure instance individually. This calibration typically results in a common throughput pattern as sketched in step 1 of Figure 9.4. The reason is that with increasing domain size the contention increases and locality is getting worse if domains span multiple NUMA nodes. As a result of the calibration, we derive the domain size maximising the overall performance up to the point after which the slope of the throughput becomes negative. As sketched in step 1 of Figure 9.4, the domain size providing maximum performance is typically larger for read-heavy workloads than for write-heavy workloads: that is $\frac{1}{2}$ socket for a write-heavy workload and 2 sockets for a read-only workload in our example.

Since the calibration phase determines the domain sizes for individual data structure instances, it ignores side-effects that might occur when several data structure instances share a virtual domain. Since sharing a virtual domain means sharing its worker threads, contention on individual data structure instances may only decrease, hence does not violate contention control. In contrast to contention, locality gets worse when multiple data structures share the same domain since CPU caches are also shared. Our composition process (discussed next) thus aims to balance the load equally across all virtual domains such that the negative effect of decreased locality is equally distributed across domains.

COMPOSITION OF DOMAINS: We now discuss the second step of the configuration process deciding the composition of virtual domains. In this step, we partition the hardware resources and assign data structure instances given from the application to individual virtual domains. In the following, we use three typical workloads as examples

to explain the composition approach: (1) *OLTP 1* as a typical OLTP scenario where indexes are accessed with a write-heavy workload; (2) *OLTP 2* as a mixed OLTP scenario where indexes are accessed with a mix of write-heavy and read-update statements; (3) *HTAP* as an HTAP scenario where indexes are accessed with write-heavy, read-update, and read-only statements.

As shown in step 3 of Figure 9.4, we distinguish two high-level cases for the composition: (1) *homogeneous* and (2) *heterogeneous* composition.

The *homogeneous* composition applies, when the calibration indicates a single optimal domain size for all data structures instances (as for OLTP1). Hence, a configuration may coincide with state of the art, e.g., Shared Nothing partitioning schemes, but it may also yield better-performing configurations, e.g., half a socket instead of a full socket as shown with configuration 4.1 in Figure 9.4 for *OLTP1*.

The second case (*heterogeneous* composition) applies if the calibration shows the data structure instances require different domain sizes for a particular workload; e.g., in HTAP workloads some data structures are used in a read-heavy manner and can use larger domains while others are write-heavy and thus need smaller domains. In this case, we differentiate the *isolated* and *shared* heterogeneous composition: (1) *Isolated* is used for crucial data structure instances necessitating predictable performance (e.g., a lock table where latency matters). The idea behind isolation is that these data structure instances do not share a virtual domain with other data structure instances. Configuration 4.2 in Figure 9.4 demonstrates *isolation* with thread-sized domains for two crucial indexes (red) in the OLTP2 workload. (2) For all other data structure instances, we apply the *shared* heterogeneous case composing domains of different sizes which can be shared by multiple data structure instances as shown in configuration 4.3 (Figure 9.4) for the exemplary HTAP workload.

For the *shared* heterogeneous composition, we formulate the problem as a variation of a General Assignment Problem with Minimal Quantities (GAP-MQ) [110] in form of an Integer Linear Program (ILP). Intuitively, the ILP should fulfil the following goals: (1) Most importantly, data structure instances should reside in domains of at most the calibrated optimal domain size. (2) The number of domains should be minimised because a higher number of domains increases the sensitivity to skew. (3) The load between all domains should be balanced.

For the input of our ILP, we introduce the data structure instances of an application as n data structure instances $i \in I = \{1, ..., n\}$ with calibrated optimal domain sizes $s_i \in S \subseteq \mathbb{N}^+$. Further, we specify the number of available worker threads in the system as $w \in \mathbb{N}^+$. Then we define the multiset B as all possible domain sizes comprising any potential configuration within the limits of the given workers w where each domain size $s \in S$ appears $\lfloor w/s \rfloor$ times (multiplicity of s). For example, assuming 192 workers as a system size ($w = 192$)

for our *OLTP2* scenario in Figure 9.4 and optimal domain sizes of $S = \{24, 48\}$ that we identified by calibration, the multiset is $B = \{24_1, 24_2, ..., 24_8, 48_9, 48_{10}, 48_{11}, 48_{12}\}_b$. Based on B, the domains to choose for a configuration are $d \in D = \{1, ..., |B|\}$ with domain size $b_d \in B$, where binary variables y_d indicate the choice of d. In our *OLTP2* scenario, this could be the choice $y_1, y_2, y_9, y_{10}, y_{11} = 1$, i.e., 2 domains of size 24 and 3 domains of size 48. Subsequently, the binary variables $x_{i,d}$ denote the assignment of a data structure instance i to a domain d in the resulting configuration.

For load balancing, we assign an abstract expected load of an instance as $l_i \in \mathbb{R}^+$ as well as a minimum and maximum load of a domain as q_d and $r_d \in \mathbb{R}^+$, where the minimum load avoids domains without any load while the maximum load avoids overloading domains. Finally, we incentivise choosing larger domains by assigning the profit in proportion to the domain size as $p_d = P^{b_d}$ with a large P, s.t. $p_1 \ll ... \ll p_{|D|}$.

$$\max \quad \sum_{d \in D} p_d y_d \tag{9.1}$$

$$\text{s.t.} \quad n y_d - \sum_{i \in I} x_{i,d} \leqslant n - 1, \qquad \forall d \in D \tag{9.2}$$

$$\sum_{d \in D} x_{i,d} = 1, \qquad \forall i \in I \tag{9.3}$$

$$b_d x_{i,d} \leqslant s_i, \qquad \forall i \in I, \forall d \in D \tag{9.4}$$

$$\sum_{d \in D} b_d y_d \leqslant w \tag{9.5}$$

$$q_d y_d \leqslant \sum_{i \in I} l_i x_{i,d} \leqslant r_d, \qquad \forall d \in D \tag{9.6}$$

$$x_{i,d}, y_d \in \{0, 1\}, \qquad \forall i \in I, \forall d \in D \tag{9.7}$$

Equations 9.1-9.7 formulate the ILP for our configuration problem based on the GAP-MQ problem. The objective function formalises an optimal configuration as a choice y_d of domains d maximising the profit through large domain sizes and consequently a minimal number of domains, where the constraint in Equation 9.2 connects that choice of a domain to the assignment of data structures $x_{i,d}$. The constraint in Equation 9.3 requires the assignment of each instance to precisely one domain. Equation 9.4 constrains the assignment of an instance to domains of at most the calibrated optimal domain size to satisfy the calibrated worst-case contention and locality while Equation 9.5 restricts the choice of domains to the available workers. Finally, in Equation 9.6, we constrain the assignment of instances to domains, such that the sum of the load of a domain is within the required bounds if the domain is chosen. Solving this ILP determines y_d and $x_{i,d}$ establishing a configuration of domains with assigned instances for our runtime system. Additionally, our ILP can simply reflect application-specific requirements on the configuration by additional constraints. For example, further constraints can incorporate co-location of specific data structure instances to place secondary indexes into the same domain.

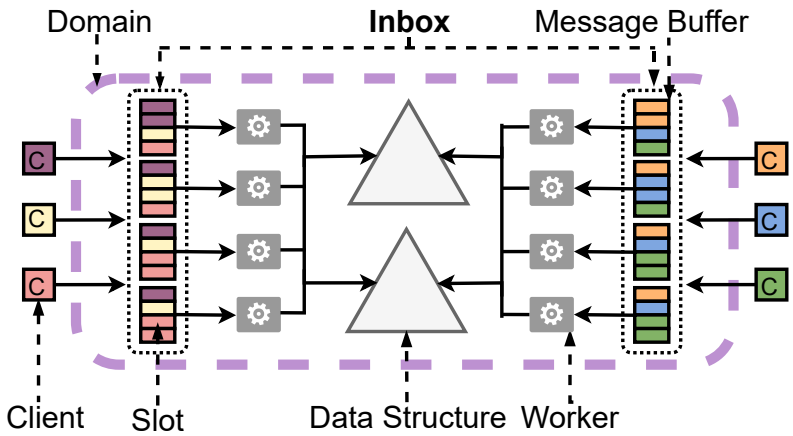


Figure 9.5: Flexible delegation through an *inbox* constructed from message buffers of workers in which clients obtain ownership of slots (colour coded).

9.6 RUNTIME SYSTEM

Given a configuration, our runtime system realises efficient execution of *data-aware asynchronous tasks* on generic data structures in freely configurable *virtual domains* via *delegation* and *futures*.

EFFICIENT AND FLEXIBLE DELEGATION: In order to achieve robust performance with an optimal configuration via delegation, the communication between clients and workers must be as efficient as possible, especially it should not cause contention which we seek to reduce through optimal configuration.

Therefore, we implement the delegation as efficient in-memory message passing based on *fast, fly-weight delegation* (FFWD) [157]. At the core of FFWD is a message passing scheme minimising cache coherence traffic for synchronous communication between multiple clients and a single worker. It enables highly efficient communication outperforming common concurrent data structures with shared memory synchronisation primitives and latch-free designs, e.g., queues with NUMA-aware *MCS* latches and latch-free queues. In detail, FFWD allocates a contiguous message buffer for the worker in which each client has a dedicated slot for a message at the position of the client id. Then, FFWD minimises cache coherence traffic through efficient detection of new messages via embedded toggle bits and batching of responses for up to 15 clients. Furthermore, their design simply includes common optimisations, e.g., NUMA-aware memory allocation as well as memory alignment to 128 bytes to prevent false sharing of adjacent cache lines [95] and incorporates optimisations not generally possible for concurrent data structures, i.e., complete absence of atomic instructions and memory ordering fences.

Beyond the original FFWD, we extend the messaging scheme to reach the necessary flexibility for our approach of optimal configuration. Specifically, we break the strong relation between a client and a worker in FFWD while maintaining the same optimisations. Figure 9.5 outlines how we establish an *inbox* for a virtual domain from which clients obtain ownership of slots to delegate to and physically construct this inbox out of the message buffers of the configured workers. Consequently, clients are only loosely coupled with (workers in) virtual domains enabling any number of clients to be transparently serviced by the independently configured number of workers within a virtual domain (limited by the total number of slots of the inbox). Additionally, we enable asynchronous delegation of several tasks to virtual domains via futures by handling responses to delegated tasks and returning ownership of a slot after returning the results.

As optimisation for virtual domains spanning multiple NUMA nodes (e.g., two sockets), the runtime system assigns ownership of a slot in the inbox, such that the backing worker has minimal NUMA distance to the requesting client. For example in Figure 9.5, the purple client on the left gets assigned ownership of the purple slots from message buffers on the left from the inbox of a virtual domain spanning two sockets and vice versa for the orange client on the right. Thereby both clients communicate locally with workers instead of communicating through an interconnect.

Notably, the implementation of delegation across virtual domains puts little requirements on the underlying hardware platform. For example, delegation can be implemented in NUMA systems with access to shared memory (which is our main focus in this paper) but can also be used in distributed systems with RDMA or future systems like Gen-Z [71]), which we aim to study in future work.

OPTIMISED DELEGATION MODE(S): On top of the efficient communication scheme, we introduce enhanced delegation that allows clients to asynchronously delegate numerous tasks and only eventually request their results which we utilise to optimise task delegation for bursting behaviour.

We enable the client to announce bursting delegation for a specific data structure instance to the runtime system. Then, the runtime system pre-allocates futures and slots from the inbox of the according virtual domain for a maximum number of outstanding tasks (burst size) specified by the client, thereby we clear the critical path from resource allocation and minimise the overhead for delegation in bursts. Moreover, we provide a delegation mode to maximise throughput based on these bursts. The runtime system can manage a burst for the client in a way that the client can continuously delegate independent tasks and only needs to process the result of the oldest tasks when the burst is completely filled. This delegation mode maximises

overlap of pending tasks in addition to minimising the overhead and consequently maximises delegation throughput for the client to the specified data structure instance. Additionally, further extended delegation modes are possible to cover other application-specific delegation patterns. With *bulk bursting*, for example, multiple tasks are delegated under a single synchronisation phase. This mode optimises delegation for a fixed number (i.e., bulk) of parallel operations within a transaction requiring a common synchronisation point.

9.7 EXPERIMENTAL EVALUATION

In the following experiments, we evaluate the efficiency of our approach and illustrate its robust performance by re-configuration for a range of data structures and workloads.

BASELINES AND SETUP: As baselines, we consider a wide range of fixed partitioning strategies from naïve shared everything to extreme shared nothing: (1) *SE* and *SE-NUMA* represent shared everything strategies, where all threads access all data structure instances. The former is a naïve setting which solely relies on the OS for data placement of its partitioned data structures into NUMA regions. Whereas the latter (*SE-NUMA*) setting is NUMA-aware, but only for memory allocations of the individual partitions. However, all threads are still allowed to operate on all the partitions, i.e., execution is not NUMA-aware. (2) *SN-NUMA* and *SN-Thread* correspond to state of the art shared nothing strategies [141], which we apply to the configuration of data structures in our framework. *SN-NUMA* represents NUMA-aware system partitioning that explicitly dedicates data structures to specific NUMA nodes. *SN-Thread* is an extreme shared nothing strategy, with thread-granular partitioning, where a single thread is exclusively accessing a partition of a data structure.

We compare all these baselines against our approach (*Opt. Configured*), where we use an optimal partitioning strategy that results from applying our configuration process in Section 9.5. In all our experiments, we use the bursted execution mode with a burst size of 14 for our approach. While the burst size is a configuration parameter, this size has shown on average the best performance across all experiments with only a minimal increase in latency. For both *shared everything* strategies bursting cannot be applied since clients directly access data structure instances.

HARDWARE: We conduct our experiments on an *HPE MC990 X* system with two hardware partitions each containing four *Intel Xeon E7-8890 v4* CPU (24 cores, 60MB L3), i.e., 192 physical cores and 384 logical (SMT) cores (with HyperThreading). A *NUMAlink* controller combines these hardware partitions to a single, cache-coherent NUMA

system [60]. The resulting system has four levels of NUMA, for which we measure memory latencies of 114, 217, 265, and 487ns. In order to assess the robustness for different hardware architectures, we use this system to simulate different architectures by restricting the number of sockets ranging from small-scale NUMA systems connected via one hop to large-scale NUMA systems that need to cross the NUMAlink.

9.7.1 *Exp. 1: Efficiency for Various Data Structures and Workloads*

In the first experiment we show the ability of our approach to enable robust performance across a wide range of data structures and workloads.

WORKLOADS AND METRICS: To assess the performance of different data structures, we use YCSB [47]. We use workloads A (Read-Update 50/50), C (Read-Only), and D (Read-Insert 95/5) of YCSB. These workloads allow us to investigate the performance of a data structure with increasing contention due to the varying amount of modifications (inserts or updates). We changed the distribution of workload D from *Latest* to *Zipfian* to keep the distribution of records and operations identical across all three workloads for direct comparison. Moreover, we use records of 64-bit integer keys and values, which potentially allow more caching but also may cause higher contention. This allows us to evaluate complex effects of locality and contention in both software and hardware.

We define the number of records as ten times the cumulative last level cache size of all sockets in the hardware mentioned before resulting in 314 M records. We generate the complete workload (records and operations) through the official Java implementation [27, 47, 94], which we then simply replay using our C++ based prototypes. For each experiment, we execute 2M key/value-operations per client thread.

Experimental measurements are presented as the median out of seven executions. We assess the reliability of our measurements in terms of Coefficient of Variation (CV) (ratio of standard deviation to mean) and consider a CV $\leqslant$ 5% as reliable. Since all our measurements fit this reliability requirement, we do not present error bars in our plots.

DATA STRUCTURES: In all experiments, we use a set of data structures commonly used for indexing in main-memory DBMS with different synchronisation schemes, listed in Table 9.1[1].

1 During our experiments, we discovered skew in the hash of the *Hash Map* and extended it with an additional XOR of the upper half of the key with the lower half resulting in a more even occupation of hash buckets, e.g., standard deviation of bucket size 1.2 instead of 4.7.

Data structure	Synchronisation scheme
STX B-Tree [26]	none by default, modified: atomic load/store + global lock for inserts
FP-Tree [135]	HTM + global lock for fallback
Open BW-Tree [202]	Copy-On-Write + atomic CAS
Hash Map [181]	Fine-grained locking + spin lock

Table 9.1: Data structures employed in experiments with specifics about their synchronisation scheme.

Workload	Read-Only	Read-Update	Read-Insert
B-Tree	48	24	24
FP-Tree	48	24	24
BW-Tree	48	48	48
Hash Map	1	1	1

Table 9.2: Optimal size (no. of workers) of virtual domains for data structures and workloads.

In the following, we evaluate the performance using an optimal configuration for each of the before-mentioned data structures and workload and compare it to rigid approaches ranging from Shared Everything to fine-granular Shared Nothing configurations. The overview of data structures and workloads we used in this experiment is shown Table 9.2.

9.7.1.1 *Exp. 1a - Performance for Various Data Structures*

We begin our evaluation by applying the optimal configuration as described in Section 9.5 to the largest system size (i.e., a machine with 8 sockets) and investigate the throughput of the index data structures under all workloads. Figure 9.6 shows, that *Opt. Configured* reaches the best throughput for all data structures under various workloads ranging from read- to write-heavy. Moreover, we see that there is no single rigid approach that dominates all other rigid approaches under all workloads and data structures, e.g., *SN-Thread* performs well with *Hash Map* but significantly worse with *FP-Tree* and *BW-Tree* whereas for *SN-NUMA* the opposite is the case.

Insight: Configuration of individually optimal domain sizes yields robust (best or close to best) throughput for any of the evaluated index data structures and workloads.

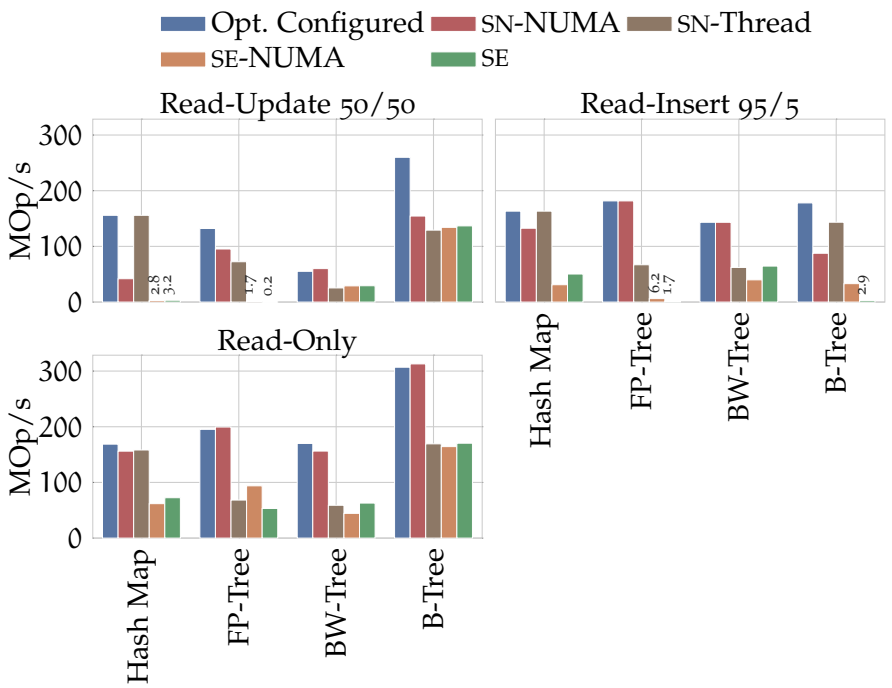


Figure 9.6: Performance of our approach across a range of YCSB workloads and data structures on largest system size (8 sockets) through individually optimal configuration. Baselines are rigid partitioning schemes.

9.7.1.2 *Exp. 1b - Robust Performance for Various System Sizes*

In the following experiment, we examine the performance of our approach on different system sizes; i.e., by varying the system size from 1 up to 8 sockets of the machine outlined in the beginning of this section. We illustrate the resulting number of virtual domains used across different system sizes with alternating white and grey shadings in the plots, e.g., the first shading represents the first virtual domain, the second shading represents the second virtual domains.

READ-UPDATE WORKLOADS: For this experiment, we first asses the effect of configuration on different system sizes with an equal mix of reads and updates. The updates are in-place modifications to index records, which do not cause any maintenance, such as node splits in a tree. Thus, these are of high locality, providing an opportunity for low contented, efficient synchronisation. Still, the high update rate puts pressure on synchronisation and causes physical contention.

Figure 9.7 depicts the throughput of the read-update workload for the same 4 index data structures as in the previous experiment. Our *Opt. Configured* provides robust scalability on the read-update workload for all the data structures, i.e., best or close to best throughput at each scale. The *Shared Everything* settings scale only with the *B-Tree*

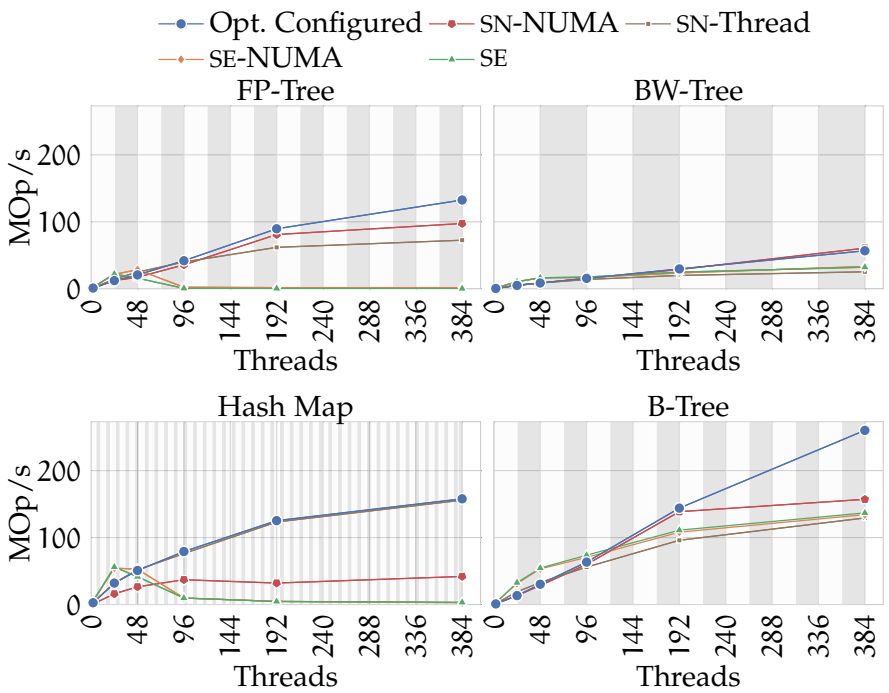


Figure 9.7: Throughput of read-update workload for various system sizes from 1 - 8 sockets (each 48 threads).

and *BW-Tree*, whereas *Shared Nothing* settings at most perform as good as our *Opt. Configured*.

While providing robust performance across all data structures, with *FP-Tree Opt. Configured* even improves performance by 560x over *SE*, 1.8x over *SN-NUMA*, and 1.4x over *SN-Thread* at 384 threads. Specifically, both *Shared Everything* settings stagnate after 24 threads and significantly drop in performance for larger system sizes, i.e., performance collapses by over 90% between 1 and 2 sockets. Instead, our *Opt. Configured* setting scales best because it retains the best scale-up performance of 24 threads in virtual domains and efficiently scales these to the largest system size. The other settings either insufficiently limit contention for HTM to perform well or incur much overhead, as detailed below.

For better understanding of the root causes for the performance degradation and the lack of scalability, we analyse the abort rate of HTM transactions and cache locality presented in Figure 9.8. The performance of *Shared Everything (SE)* settings and *SN-NUMA* setting is tied to the sensitivity of HTM to high conflict ratios (i.e., workload with 50% updates) and length of HTM transactions, amplified by longer NUMA distances, as investigated in [31]. The consequence is high abort ratios of HTM transactions inflicting the substantial performance degradation for these three settings. In contrast, *SN-Thread* does not cause any aborts, but increased L2 (and L3) cache misses instead, thus indicating overhead of its extreme partitioning which inflates competi-

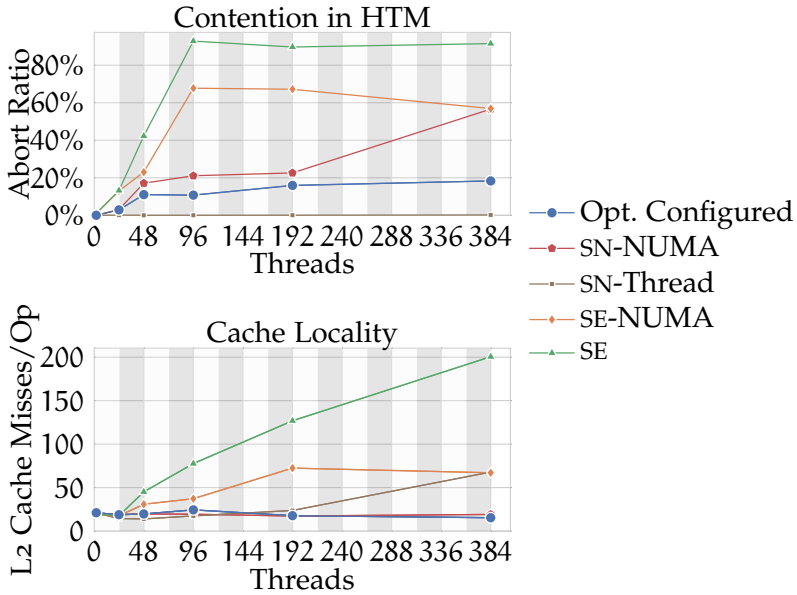


Figure 9.8: Hardware metrics indicating contention and cache locality of FP-Tree on read-upadte workload.

tion between the data structure and the delegation procedure for the private L2 cache of the responsible CPU core. Finally, *Opt. Configured* keeps the abort ratio and cache misses low, such that it performs well for the contended read-update workload. This confirms the benefits of our apt configuration with adequate contention management yet minimal overhead.

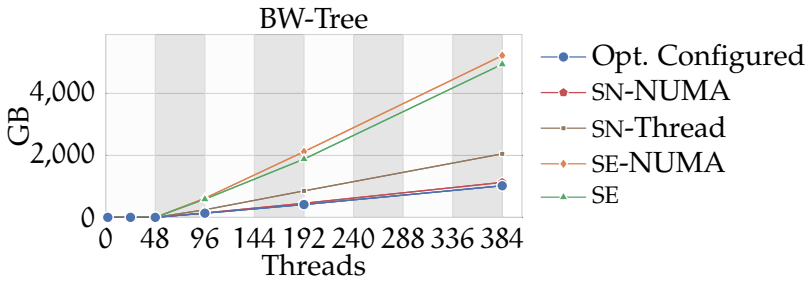


Figure 9.9: Communication volume on interconnects between sockets for BW-Tree on read-update workload.

In the contrary to FP-Tree, *BW-Tree* manages to scale with the *SE* settings due to its conflict resistant Copy-On-Write (COW) synchronisation scheme but the performance of *Opt. Configured* is superior at larger scales, i.e., up to 1.9x. Figure 9.9 shows that the COW synchronisation scheme induces high communication overhead on the interconnects of up to 5 TB. Here, our the efficient in-memory messaging pays off with about 5x lower communication volume for *Opt. Configured* and *SN-NUMA* as well as 2.5x less for *SN-Thread*.

The *Hash Map* exhibits a behaviour similar to the *FP-Tree* under *SE*. For the smallest deployments up to a single socket, SE provides high performance. However, for larger deployments the performance of *SE* collapses similar to the FP-Tree. Our profiling analysis of the *Hash Map* indicates that the bottleneck is highly contended synchronisation. This high contention also explains the mediocre performance of SN-NUMA, whose partitioning per NUMA region insufficiently controls contention. Therefore, SN-Thread and the Opt. Configured provide robust performance for the *Hash Map* when scaling to larger deployments. This highlights the benefit of our configurable approach, which allows us to partition data structures into optimally sized domains: for the *Hash Map*, our approach uses many small domains similar to SN-Thread, while for the other data structures before (FP-Tree, BW-Tree) we use a configuration that is closer to the SN-NUMA.

Finally, for *B-Tree Opt. Configured* performs as good as the NUMA-partitioned strategy. However, we use synchronisation with just atomic operations on the record level since the B-Tree itself does not include any synchronisation. This synchronisation is unfair as it does not protect modifications of the structure of the *B-Tree*. Hence, this mainly serves as an upper bound for possible performance with the simplest synchronisation that we could achieve with all strategies.

Insight: Optimal configuration establishes robust scalability for indexes under high contention. In contrast, both *Shared Everything* settings suffer steep performance cliffs beyond one socket already and the *Shared Nothing* approaches scale well just for some indexes at larger scale. Only our configurable approach (*Opt. Configured*) handles contention effectively for all data structures providing locality at different system scales.

READ-ONLY WORKLOADS: Next, we show the effect of configuration on a read-only workload (YCSB C) for the same set of data structures and system sizes. This workload is favourable for a *Shared Everything* strategy, as there is little to no contention and maximum opportunity for high cache utilisation. Therefore, we expect the benefits of our approach over Shared Everything to be limited to the better locality in case of memory accesses or remaining synchronisation, e.g., reader side of latches.

Figure 9.10 presents the experimental results. For *FP-Tree* the *SE* as well as *SN-Thread* settings scale only up to 96 threads (2 sockets), after which their throughput stagnates and *SE-NUMA* follows stagnating only after 4 socket. In contrast, *Opt. Configured* and *SN-NUMA* manage to scale linearly up to 8 sockets with a maximal throughput improvement of 3.2x over *SE*. Only our approach (and SN-NUMA) provides efficient execution of access methods even for large deployments. Moreover, *BW-Tree* and *B-Tree* present similar behaviour to *FP-Tree*,

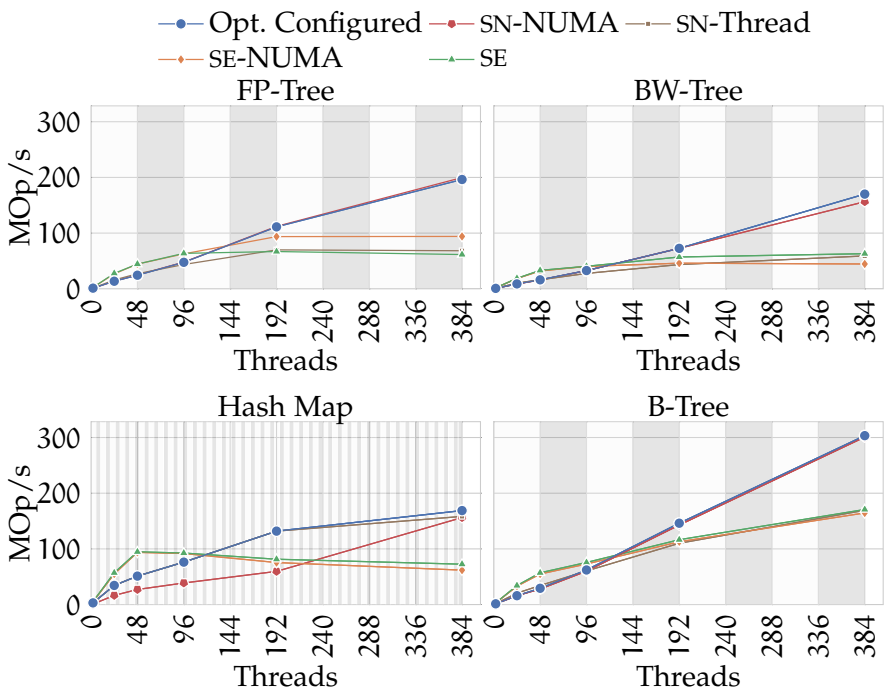


Figure 9.10: Throughput of read-only workload for various system sizes from 1 - 8 sockets (each 48 threads).

only that *BW-Tree* is slightly slower and *B-Tree* is faster due to their differing synchronisation.

The data structure *Hash Map* performs well only on a single socket with SE, whereas *Opt. Configured* enables robust performance of *Hash Map* reaching 2.3x higher throughput than SE at 8 sockets. This improvement results from a bottleneck in the general-purpose implementation of the *Hash Map* rooted in the reader coordination of the reader-writer mutex for synchronisation on the hash buckets. The locality within our virtual domains optimises the execution of the atomic increment to register readers on the mutex.

Insight: For read-only workloads where contention is not as prevalent as for the read-update workload before, our approach (Opt. Configured) shows competitive performance as well as low overhead for all index structures. Most importantly, Opt. Configured again is the only approach that can provide robust performance for all data structures when scaling out: while Opt. Configured performs on par with SN-NUMA offering the best performance for the 3 tree-based data structures, Opt. Configured employs smaller domain sizes for the Hash Map, and thus behaves more like SN-Thread, which is best performing in this case.

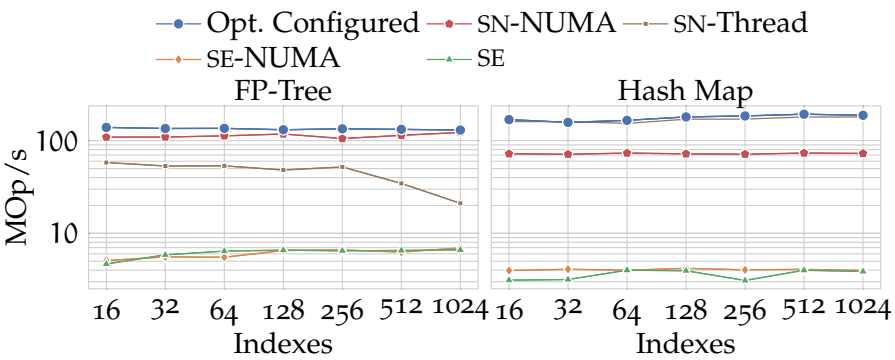


Figure 9.11: Agg. throughput for increasing no. of indexes (i.e., application size) for read-update workload.

9.7.1.3 *Exp. 1c - Robustness for Different Application Sizes*

In the following experiment, we extend the perspective of robustness by application size using an increasing number of index instances. As the system is under full load even with a single index instance already, there cannot be major improvement of throughput when increasing the number of index instances. On the contrary, we expect this experiment to expose bottlenecks, as it may amplify overheads or impact important performance factors such as cache locality or contention inherent in our framework.

We setup this experiment as previously but increase the number of indexes by separating the prior indexes into smaller indexes (16 - 1024) with the identical total data volume. Moreover, we configure our framework with the same number of optimally sized virtual domains, i.e., 16 domains of size 24 threads, but additionally now instances share domains, e.g., for a total of 1024 indexes 64 instances share one domain.

Figure 9.11 presents stable throughput under increasing number of indexes for most settings with both index types. Exceptions are on FP-Tree a minor positive trend for both shared everything settings (SE: 1.4x, SE-NUMA: 1.3x) and degrading of SN-Thread by up to 50% beyond 256 indexes. Importantly, the individual configuration of indexes within our framework (Opt. Configured) is stable and provides the best throughput for all numbers of indexes.

Insight: The benefits of configuration within our framework persists for large numbers of indexes.

9.7.2 *Exp. 2: Cost-Benefit Breakdown of Configurability*

Having demonstrated the potential of configurability in Section 9.7.1, we now detail its associated overhead. Indeed, our runtime system, which delegates tasks to virtual domains, causes computational overhead in addition to baselines which directly execute the access meth-

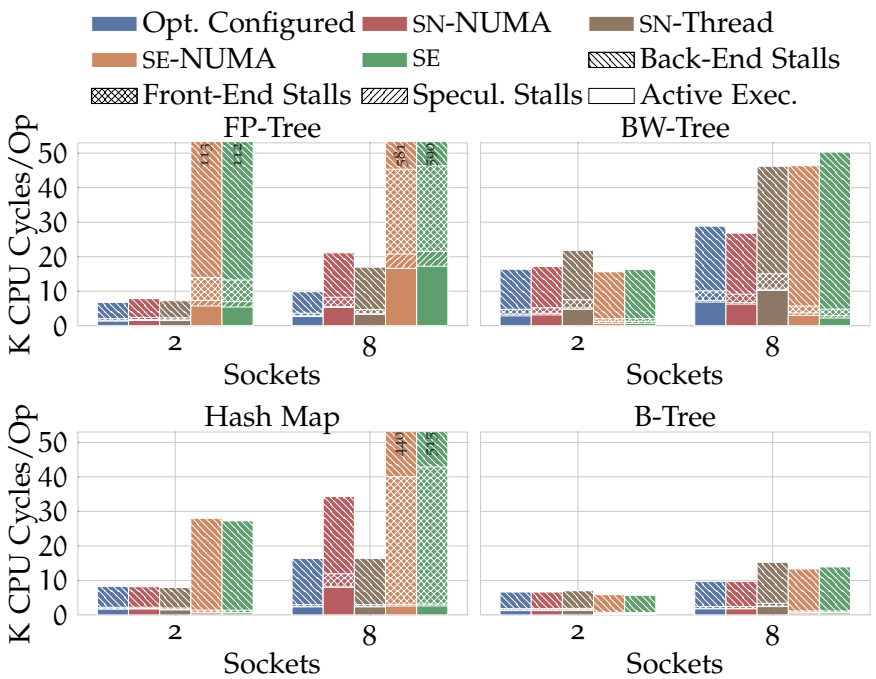


Figure 9.12: Execution cost breakdown into active execution cycles vs. stall cycles per operation for system sizes 2 vs. 8 sockets with read-update workload.

ods. However, as we show in the following, this overhead is negligible even for small system sizes and provides significant benefits for robust performance, especially when scaling to larger deployments.

WORKLOAD AND BASELINES: In this experiment, we use the *YCSB Workload A* (Read-Update) from the previous experiment to show the overhead and benefits of workloads with a mix of operations that is common for OLTP workloads. Moreover, we run this experiment on a small system size (2 sockets) and the largest system size (8 sockets) to show the cost breakdown for small versus larger systems as mentioned before. For each system, we compare the same data structures and baselines as in Section 9.7.1.

PERFORMANCE RESULTS: Figure 9.12 shows the average cost for one operation (read/update) in *CPU cycles per operation* as stacked bars for the different data structures and system sizes. The cost is broken down into the main categories of the *Top-down Microarchitecture Analysis Method* (TMAM) [95] commonly used to guide performance optimisation [166] (e.g., by Intel VTune). Based on TMAM, we distinguish active execution cycles (solid part of bars) and wasted execution time (striped parts of bars), i.e., stalls in the *Back-End* of the CPU (mainly memory accesses), in the *Front-End* (e.g., instruction decoding), and stalls due to bad speculation (e.g., branch misprediction).

Notably, in this representation, *lower cost means better throughput per thread*.

Again, in this experiment, we can observe robust performance (i.e., lowest cost or close to lowest cost) for our approach (Opt. Configured) compared to the baselines. That is, as expected for the small system size (2 sockets), our runtime system has comparable performance to the shared-nothing and shared-everything baselines, while we achieve significant benefits for larger system sizes, especially for the FP-tree. Moreover, as observed before, Opt. Configured achieves its robustness by configurability: it resembles the execution cost of SN-NUMA or SN-Thread (which are the best performing) while its distinct configuration of half a socket (i.e., in between the partition sizes of those other approaches) even achieves better cost for *FP-Tree*.

Finally, when observing cost in detail, we can confirm our runtime system adds negligible active execution overhead in comparison to the SE-baselines, which can be mainly attributed to the additional instructions for delegation on top of the bare data structure operations. However, comparing the amount of stall cycles, our runtime efficiently executes these additional instructions (comparable *Front-End* and *Speculation Stalls*) and effectively decreases overall memory (*Back-End*) stalls below the stalls of the bare data structure operations (i.e., improves locality and contention), which benefits overall cost per operation.

Insight: Our cost-benefit analysis shows that our approach (Opt. Configured) has highest active cycles stemming from the additional overhead of our runtime system. However, importantly, this allows our approach to reduce stalls efficiently. As a result, our approach has the overall lowest execution cost (active cycles + stalls), thus can provide the highest performance across different data structures and system sizes.

9.7.3 *Exp. 3: Efficiency for OLTP Workloads*

In the last experiment we show that our approach also provides performance benefits beyond single data structures and supports the efficient execution of more complex OLTP workloads. For this purpose, we implement a light-weight OLTP engine on top of our runtime system and observe the performance of transactions of TPC-C – a typical OLTP benchmark.

LIGHT-WEIGHT OLTP ENGINE AND BASELINE: The light-weight OLTP engine provides basic functionality for partitioning tables and executing transactions as tasks: (1) For partitioning, our light-weight OLTP engine supports a typical hash-partitioning scheme. In particular, it partitions tables including their primary and secondary indexes that are then configured as composite data structure by our config-

uration procedure from Section 9.5 to place tables and their indexes jointly in the same virtual domains. As data structures for tables and indexes, we have chosen the *FP-Tree* and the *BW-Tree* from the previous experiments since both are specifically designed for supporting OLTP workloads in main memory DBMS, notably they use very different synchronisation mechanisms (cf. Table 9.1). (2) For executing transactions as task, our OLTP engine uses a scheme where each individual statement of a transaction is mapped to a task (i.e., we do not do any chopping which could further improve the performance). Finally, we omit higher order components for recovery and concurrency control. As already discussed in Section 9.3, all these components are orthogonal and different schemes can be implemented on top of our runtime system. Analysing the effects of these different schemes (e.g., for concurrency), however, is beyond the scope of this paper.

As baseline, we compare our light-weight OLTP engine that is based on our runtime system with an OLTP engine based on the design of [141] that uses a NUMA-aware *shared-nothing* design where transactions are directly executed by transaction managers (i.e., not by delegating tasks to our runtime system). To allow a fair comparison, we also omit concurrency control and recovery in the baseline.

OLTP WORKLOAD: In order to compare our light-weight OLTP engine with the shared-nothing baseline, we use the TPC-C benchmark. For this experiment, we implemented the *New-Order* and *Payment* TPC-C transactions as tasks, which represent 88% of the workload. As data, we generate a TPC-C database with 8 warehouses (i.e., one for each NUMA-region considered to be favourable for the shared-nothing baseline used in this experiment). We partition the database across different system sizes by warehouse IDs ranging from 1 to 8 NUMA regions (i.e., system sizes from 48 to 384 threads). Moreover, we vary the fraction of *New-Order* and *Payment* transactions that need to access remote warehouses from 0% to 75% to simulate workloads that range from perfect locality to almost no locality. Finally, we configure tables into virtual domains with the procedure outlined in Section 9.5.

PERFORMANCE RESULTS: The results of this experiment are shown in Figure 9.13. Observing the TPC-C throughput for increasing system size on the left of Figure 9.13 reveals that using the FP-Tree results in brittle performance in the NUMA-aware system, i.e., degrading from the best throughput at the smallest system size (48 threads) to the worst throughput for larger system sizes ($\geqslant$ 96 threads), which is in line with our results in Section 9.7.1. Whereas, the BW-Tree is more robust across different system sizes in the NUMA-aware system design. Nevertheless, the OLTP-engine based on our runtime system increases the overall performance for both indexes (the FP- and BW-tree) due to our effective contention management and efficient communication

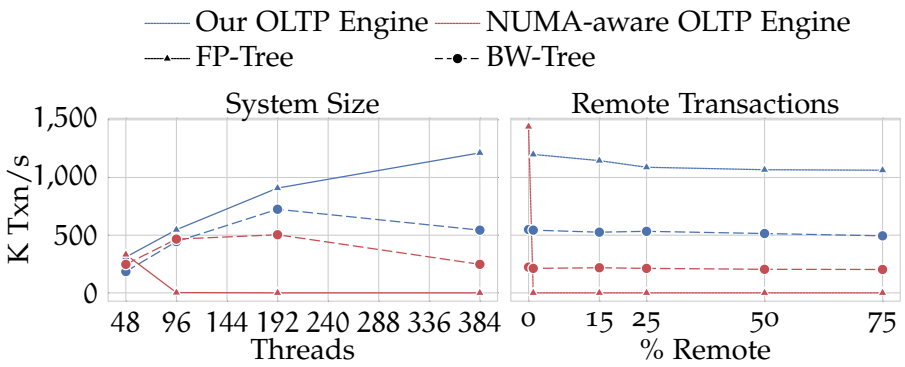


Figure 9.13: Throughput of TPC-C New-Order and Payment with 8 warehouses for increasing system size with 1% remote transactions (left) and for increasing remote transactions at largest system size (right).

as observed earlier. Even more interestingly, our approach in combination with the FP-Tree establishes robust performance scaling the throughput of TPC-C transactions linearly with the system size.

Now, we present the TPC-C throughput at the largest system size under an increasing proportion of remote transactions on the right of Figure 9.13. The results indicate the sensitivity of the two different OLTP engines w.r.t. partitionability and locality of OLTP workloads; i.e., commonly OLTP workloads (especially TPC-C) are partitioned into exclusive partitions to achieve high throughput on large system sizes rendering them sensitive to remote transactions. The performance of the NUMA-aware system with FP-Tree perfectly demonstrates this sensitivity to remote transactions dropping from 1.5M txn/s at 0% remote transactions to barely any throughput at 1%. In contrast, our OLTP engine provides high throughput regardless the remote transactions, i.e., 1.2 - 1.1M txn/s. Again, BW-Tree improves the robustness of the NUMA-aware system. Still, in this case, our approach also enables better throughput for BW-Tree compared to the NUMA-partitioned baseline since our approach can further increase locality and reduce contention.

Insight: As we have shown in this experiment, using our runtime that relies on the configuration of virtual domains can also provide significant benefits for executing OLTP workloads. An interesting insight is that opposed to classical OLTP engines where optimal partitioning is crucial to maximise locality, partitioning does not play a significant role anymore when using our runtime system since delegation can provide high locality (and thus high throughput) even for non-partitionable workloads.

9.8 RELATED WORK

In this paper, we make the case for configuration to achieve robust performance for a wide range of workloads and a variety of hardware platforms.

As hardware development advances, a huge body of research proposes solutions for new challenges and reiterates optimisations on all levels of system design ranging from synchronisation primitives to data structure designs and all the way to entirely new DBMS architectures. In response to increasing core counts and main memory capacity, systems like H-Store [103] and DORA [137] propose to partition the database in a fine-grained manner per hardware thread to avoid contention on data structures, where the latter actually applies delegation of transactions between worker threads.

Yu et al. [206] give a projection on the arising challenges when hardware with more than 1000 cores becomes commonplace for DBMS. They identify challenges of prior in-memory DBMS which they say will only be amplified as the number of cores increases. Since then, many proposals take different directions on how to adapt DBMS architectures to hardware with so many cores spread over many sockets. For example, ERIS [107] takes the DORA approach from multi-core OLTP to multi-socket OLAP and adds load balancing between system partitions to address skew in the workload. Instead, Hekaton [52] suggests to reject partitioning and to use a shared-everything approach to avoid the negative impacts of partitioning, e.g., skew. Porobic et al. [141] analyse the effect of hardware topologies with different core and socket counts on partitioning strategies and conclude that DBMS must be aware of the underlying hardware.

We propose to decouple the implementation of data structures (and larger components) from the system architecture, such that the configuration can adapt to new hardware and the existing implementations are reused.

Our general approach to adapt to hardware development follows other systems research. Node Replication [34] devises an automatic approach to transform any sequential data structure into a concurrent data structure for large scale hardware through a combination of shared memory and distributed computing approaches. They replicate data structures in a distributed manner across NUMA nodes and apply flat-combining delegation to share and synchronise a distinct data structure between workers on a NUMA node. [33] evaluate approaches to combine message passing known from distributed systems and common shared memory programming for NUMA-aware systems. They expect benefits from message passing in combination with delegation when communication of data and cache coherence between NUMA node becomes too expensive, which we show is already the case for index operations on the BW-Tree from two sockets

and more (cf. Figure 9.9). But they point out that efficient message passing is crucial for delegation. To this end, FFWD [157] devises a delegation scheme with minimal cache coherence transactions between participating CPU cores, outperforming prior delegation schemes including flat-combining [54, 123], shared memory synchronisation primitives [48, 55], and latch-free algorithms. We extend FFWD with broader flexibility to enable efficient configuration and robust performance.

9.9 CONCLUSION

In this paper, we proposed a new approach for achieving robust performance of fundamental data structures. The main idea is to strictly separate the data structure design from the actual strategies how access operations are executed and to adjust the execution strategies by means of a configuration. In our evaluation, we demonstrated that reconfiguration establishes robust performance across diverse workloads and hardware sizes. While we believe that our abstractions (tasks and configurations) allow an efficient adaption of existing DBMS to make use of our approach, showing this would be beyond the scope of this paper though and is an interesting avenue of future work.

10

ANYDB: AN ARCHITECTURE-LESS DBMS FOR ANY WORKLOAD

ABSTRACT

In this paper, we propose a radical new approach for scale-out distributed DBMSs. Instead of hard-baking an architectural model, such as a shared-nothing architecture, into the distributed DBMS design, we aim for a new class of so-called architecture-less DBMSs. The main idea is that an architecture-less DBMS can mimic any architecture on a per-query basis on-the-fly without any additional overhead for reconfiguration. Our initial results show that our architecture-less DBMS AnyDB can provide significant speedup across varying workloads compared to a traditional DBMS implementing a static architecture.

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the peer-reviewed work *AnyDB: An Architecture-less DBMS for Any Workload* by Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig in the *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*. The contributions of the author of this dissertation are summarized in Part I Section 5.1.

10.1 INTRODUCTION

MOTIVATION: Scale-out distributed architectures are used today by many academic and commercial database systems such as SAP HANA, Amazon Redshift / Aurora, and Snowflake [9, 66, 72, 75, 194] to process large data volumes, since these allow scaling compute and memory capacities by simply adding or removing processing nodes. The two predominant architectural models used in academic and commercial distributed databases are the shared-nothing (aggregated) and the shared-disk (disaggregated) architecture [172].

While the shared-nothing (aggregated) architecture provides high performance in case the data and workload are well partitionable, its performance degrades significantly under skew, overloading some resources while others are idle [22]. Moreover, dealing with requirements such as elasticity in the shared-nothing architecture is hard, since this always requires repartitioning the data even if compute is the bottleneck [61]. This renders the shared-nothing architecture less suited for modern environments such as the cloud where elasticity is a key requirement.

On the other hand, the shared-disk (disaggregated) architecture tackles the drawbacks of the shared-nothing architecture by disaggregating storage and compute [122, 194]. This disaggregation provides many new potentials especially for better skew handling as well as providing elasticity independently for compute and storage. Yet, the shared-disk (disaggregated) architecture has other downsides. One major downside is that data always needs to be pulled into the compute layer, resulting in higher latencies. While this additional latency often does not matter for OLAP workloads, it renders the shared-disk (disaggregated) architecture less suitable for OLTP workloads, which require low latency execution to reduce the potential of conflicts and provide high throughput.

Another observation is that these architectural models (shared-nothing or shared-disk) are statically baked into the system designs of today's databases [22], expecting certain workload characteristics. However, modern workloads are versatile, e.g., HTAP containing a mix of OLTP and OLAP queries [147]; even more, workloads are often evolving over time or as in the cloud are not even foreseeable for cloud providers. Consequently, databases following a static architectural model are always kind-of-a compromise and cannot provide optimal performance across a wide-spectrum of workloads.

CONTRIBUTIONS: In this paper, we propose a radical new approach for scale-out distributed DBMSs. Instead of hard-baking an architectural model statically into the DBMS design, we aim for a new class of so-called *architecture-less DBMSs*.

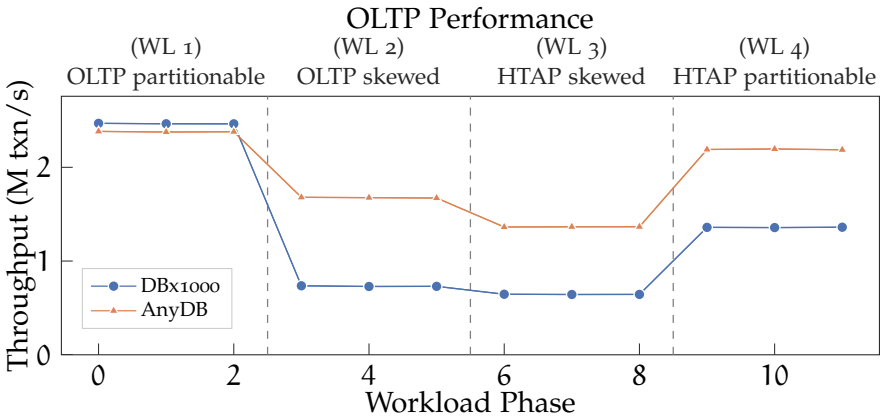


Figure 10.1: Performance of AnyDB across a workload evolving from partitionable OLTP (phase 0-2), over a skewed OLTP (phase 3-5), to skewed HTAP (phase 6-8), and then to partitionable HTAP (phase 9-11). The y-axis only shows the throughput of the OLTP transactions excluding the OLAP queries in the HTAP phases.

The main idea of an architecture-less database system is that it is composed of a single generic type of component where multiple instances of this component "act together" in an optimal manner on a per-query basis. To instrument these generic components at runtime and coordinate the overall DBMS execution, each component consumes two streams: an event and a data stream. While the event stream encodes the operations to be executed, the data stream shuffles the state required by these events to the executing component, such that a component may act as a query optimizer at one moment for one query but for the next as a worker executing a filter or join operator. Essentially, this instrumentation of generic components by event and data streams flexibly shapes the "architecture" of an architecture-less DBMS.

A key aspect of this execution model is that by simply changing the routing of event and data streams between generic components, an architecture-less DBMS can mimic different distributed architectures and form traditional architectures as well as completely new architectures. Another important aspect is, since we decide this routing on a per-query basis, an architecture-less DBMS can simultaneously act as a shared-nothing system for one query while also acting as a shared-disk (disaggregated) DBMS for another query that runs concurrently with the first one. This opens up interesting opportunities for executing mixed workloads (e.g., HTAP) or adapting to evolving workloads.

Also, an interesting aspect of this execution model is that it cannot only mimic architectures on the macro-level (shared-nothing vs. shared-disk) but also can adapt execution strategies on the micro-level. For example, query execution in an architecture-less DBMS can mimic various query processing models at runtime (tuple-wise pipelined

vs. vectorized vs. materialized [105]) and degrees of parallelism by simply instrumenting the generic components with different event and data streams. The same holds also for other components such as transaction execution and concurrency control.

The potential of the proposed architecture-less database system is shown in Figure 10.1. Here, we compared the performance when running an evolving workload in a static shared-nothing architecture (blue line) based on an extended version of DBx1000 [206] with AnyDB (orange line) our prototypical implementation of an architecture-less database system. As we see, AnyDB is either able to match or outperform DBx1000 depending on whether DBx1000's static architecture happens to suit a workload or not.

OUTLINE: The remainder of this paper is structured as follows. First, in Section 10.2 we give an overview of how we envision an architecture-less database system. Second, in Sections 10.3 and 10.4 we then discuss the opportunities that an architecture-less database provides for OLTP, OLAP as well as HTAP and present initial experimental results in each of these sections using our prototypical architecture-less database system AnyDB. Finally, we conclude with a discussion of future directions in Section 10.5.

10.2 AN ARCHITECTURE-LESS DBMS

In the following, we first give an overview of the general execution model of an architecture-less DBMS. Then we present how typical database workloads can be mapped to this execution model and discuss the main challenges.

10.2.1 *Overview of Execution Model*

As shown in Figure 10.2, the main idea of an architecture-less database system such as AnyDB is that the DBMS is composed only of generic components, so-called *AnyComponents* (ACs). These generic ACs can provide any database functionality, varying over time. That is, by routing events and their required data to an AC, the AC can act as a query optimizer in one moment and in the next moment as a query executor or any other component (e.g., log writer, etc.). This gives an architecture-less DBMS the flexibility to shift its architecture just in an instant without any downtime for reconfiguration, as we discuss later.

To execute a complete SQL query (or a transaction composed of several of database operations), multiple events and data streams are routed through AnyDB from one AC to another. For example, as outlined in Figure 10.2 (a), when executing a query in AnyDB calling the query optimizer is one event that can trigger follow-up events for executing the operators, e.g., scans and joins. The accompanying data

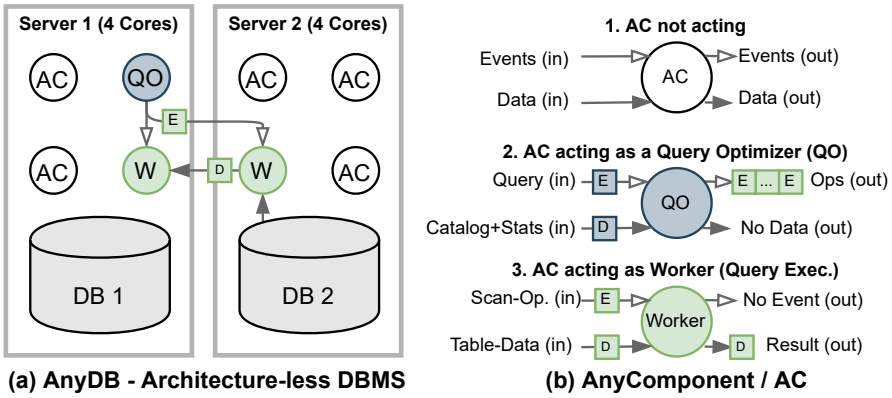


Figure 10.2: AnyDB is an architecture-less DBMS of generic components called AnyComponents (ACs), executing arbitrary logic. ACs are instrumented by events and data streams. Depending on the incoming events an AC can act as a Query Optimizer (QO) or a Worker (W) executing a scan or a join operator or any other component, e.g., log writer, etc. An AC can also produce new data and event streams for other ACs. For example, an AC that acts as a scan operator produces a data stream with results of the scan operation.

streams are responsible to shuffle the required state of an event to the executing AC. Next, we focus on two important key design principles that underpin the architecture-less DBMS:

- **Fully Stateless / Active Data:** ACs are designed to be fully stateless meaning that events can be processed in any AC and all state required to execute an event is being delivered to the AC via data streams, including table data but also catalog data, statistics, and any other state. By designing ACs fully stateless, we gain a high degree of freedom as any DBMS function can be executed anywhere. This feature allows mimicking diverse architectures but also to support elasticity for all database functions individually, i.e., additional ACs can execute any DBMS function at any time. Moreover, in architecture-less DBMSs data is active, meaning that data is not pulled after an event is scheduled but it is pushed from data sources actively to the ACs before it is actually needed (We further discuss active data in Section 10.2.3).

- **Non-blocking / Asynchronous Execution:** A second key aspect is that ACs are executing events in a non-blocking manner. This means that an AC never waits for data of an event if data is not available yet. Instead, another event with available data is being processed. For example, a filter or a join operator is only processed once its input data, a batch of tuples, is arriving via the data stream. To provide this non-blocking execution, ACs use queues to buffer input events and data items. In addition, these queues decouple the execution between ACs as much as possible;

i.e., ACs can process events asynchronously from each other. This asynchronous execution model, which is only implicitly synchronizing the execution across ACs through events and data streams, opens up many new opportunities, as we discuss later in this paper.

At a first glimpse, the execution model of an architecture-less DBMS seems to have similarities with existing approaches such as scalable stream processing systems, function-as-a-service (FaaS) or serverless DBMSs. However, there are crucial differences.

(1) First and foremost, while AnyDB also uses streams as a major abstraction, AnyDB is different from stream processing engines, since we target classical database workloads that process relations but employ streams as a vehicle to on-the-fly adapt the database architecture on a per-query basis. Still, our approach benefits from techniques of scalable stream processing such as efficient data routing or for implementing fault-tolerance in AnyDB [50].

(2) Similar to function-as-a-service (FaaS), AnyDB relies on a fully stateless execution model to provide elasticity. However, in architecture-less DBMSs data (i.e., state) does not come as an afterthought. In FaaS as offered today, a function is scheduled first and then data must be pulled in from storage before the execution can actually start [138]. Instead, as mentioned before, in architecture-less DBMSs data is active, meaning that data is actively pushed from data sources to the ACs before the event is actually being processed. Moreover, while architecture-less DBMSs logically disaggregate the DBMS execution into small functions like function-as-a-service, we still allow executing events in a physically aggregated manner and also allow shipping events to the data to make use of locality.

(3) Finally, there also exist serverless DBMS offerings such as Amazon Aurora Serverless [4, 194], Snowflake Serverless, Azure SQL Database Hyperscale [8, 131]. These aim to provide elasticity (similar to FaaS), though, they still rely on static architectures and are thus restricted to very distinct workloads (e.g., OLAP only). This is very different from our architecture-less approach that can optimally adapt to a given workload and provide elasticity at the same time.

10.2.2 *Supporting OLAP and OLTP*

In the following, we give a brief overview of how the execution model above can be used to execute OLAP and OLTP workloads. Later in Sections 10.3 and 10.4, we discuss the opportunities for these workloads arising from our execution model.

SUPPORTING OLAP: The basic flow when executing an OLAP query in an architecture-less DBMS is shown in Figure 10.3. The initial event is typically a SQL query that is sent from a client to any AC of

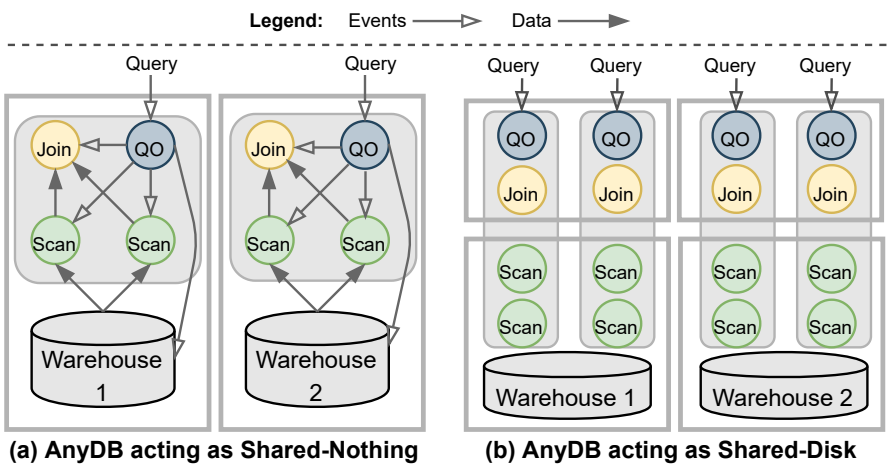


Figure 10.3: AnyDB can mimic diverse architectures simply by using different routing schemes for events and data streams. In (a), two servers act as a shared-nothing database while in (b) additional resources (i.e., two servers with additional 4 ACs per server) are added and AnyDB acts as a disaggregated architecture to deal with a higher query load. For simplicity, we only show the events and data streams for (a). The gray-shaded boxes around the ACs, however, indicate in (b) which ACs execute events of the same query.

the DBMS – never mind which one – which then acts as the Query Optimizer (QO). The main task of the QO is to come up with an efficient execution plan like a traditional query optimizer in a static DBMS architecture.

In contrast to traditional optimizers, however, the QO in an architecture-less DBMS produces an event stream and initiates the data streams that instrument the ACs for query execution. Importantly, the QO also determines the routing of a query's events and data through the architecture-less DBMS. Consequently, by these routing decisions the QO defines the DBMS architecture perceived by individual queries.

For example, as shown in Figure 10.3 (a), if a query touches only one partition and there is moderate load in the system, then the QO can route events of a query such that the architecture-less DBMS acts as a shared-nothing architecture. However, in case the query load in the system increases, servers with additional ACs are added and the architecture-less DBMS executes queries in a disaggregated mode simply by routing events differently, as shown in Figure 10.3 (b).

SUPPORTING OLTP: The basic flow of executing OLTP transactions is similar to executing OLAP read-only queries. Transactions are also decomposed into event and data streams where routing decisions define the architecture. A key difference to read-only OLAP, however, is that in OLTP (1) transactions need to update state and (2) concur-

rently running transactions need to coordinate their operations to guarantee correct isolation. Both these aspects are discussed below in the following (cf. *Concurrency and Updates*).

10.2.3 *Key Challenges*

There are different key challenges to enable efficient execution in an architecture-less DBMS. One of them is the optimal routing of events and data for a given workload. Another one is to handle concurrency and updates. In the following, we briefly discuss the main ideas how we aim to address these challenges. Some of these ideas are already built into our prototype AnyDB while others represent future routes of research.

EVENT AND DATA ROUTING: As mentioned before, a key challenge of an architecture-less DBMS is to decide how to handle a query and how to route its events, as part of query optimization. Depending on requirements of an application (e.g., latency guarantees), load in the system, and the workload, the query optimizer has to define an optimal event routing. In our current prototype, we do not focus on this problem but use an optimal decision to showcase the potential of our approach. We believe however that this is an interesting avenue for learned query optimizers.

A second challenging aspect is the efficient data routing. As mentioned before, this aspect is important for latency hiding. We utilize the decoupling of data streams from events in our execution model to solve this challenge. The main observation is that in DBMS execution one often knows which data is accessed way ahead of time before the data is actually being processed. For example, complex OLAP queries need to be optimized and compiled, often taking up to 100ms in commercial query optimizers in our experience, while we already know which tables contribute to a query before query optimization. In AnyDB we make use of this fact and initiate data streams as early as possible. Once initiated a data stream actively pushes data to the AC where, for example, a filter operator will be executed once query optimization finished. We call this feature *data beaming* as data is often available at an AC before the according event arrives, entirely hiding latencies of data transfers. We analyze the opportunities of data beaming for OLAP later in Section 10.4.

CONCURRENCY AND UPDATES: In general, updates are supported in AnyDB by event streams directed towards the storage which ingests these events and produces acknowledgment events when the updates have been processed, as required for transaction coordination in OLTP. A major challenge in handling updates thus is to hide latencies of updates as much as possible. Again, to hide latencies of updates and

decrease the overall latency of executing a transaction, operations of one transaction are represented as events and executed asynchronously by ACs. For example, an update can be sent to the storage by one AC while other (independent) operations of the transaction can progress on other ACs. Only the commit operation at the end of a transaction needs to know if the write successfully persisted and thus needs to wait for the acknowledgment event coming from the storage. As we show later in Section 10.3, this asynchronous model for OLTP provides many interesting opportunities and results in higher performance under various workloads.

Another challenge that is harder to solve is to efficiently handle concurrency. A naïve way would be to implement a lock manager using events representing lock operations and data streams providing the state of the lock table. A more clever way, however, is to rethink concurrency protocols and route events and data streams such that their processing order already captures the requirements of a particular isolation level for concurrency control, as we discuss later in Section 10.3.

FAULT-TOLERANCE AND RECOVERY: Fault-tolerance and recovery are two major challenges any DBMS needs to address. For an architecture-less DBMS this is a challenge due to the asynchronous (decoupled) execution of multiple ACs where individual ACs might fail.

Again, a naïve approach would be to implement standard write-ahead logging by sending log events from ACs to durable storage. For recovery the DBMS could be stopped and the log could be used to bring the DBMS into a correct state. Again, in an architecture-less DBMS we believe that we can do better and learn from the streaming community. For example, as the entire execution of a DBMS is represented as streams, another direction is to make the streams reliable, such that upon AC failure the streams (events and data) can be rerouted to another AC [50]. Applying these ideas is again an interesting avenue of future research.

10.3 OPPORTUNITIES FOR OLTP

In the following, we discuss the various opportunities emerging from an architecture-less DBMS when executing OLTP workloads and show initial results when compared to existing execution models of static architectures (shared-nothing and shared-disk). For all initial experiments in this paper, we use the two dominant transactions of the TPC-C benchmark (i.e., payment and new-order) [185].

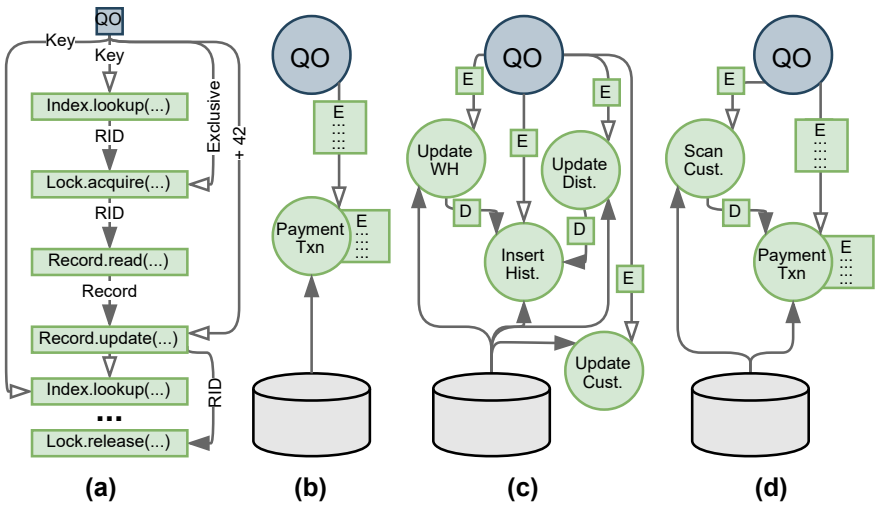


Figure 10.4: Duality of Disaggregation. (a) shows how a transaction is logically disaggregated into individual events for each operation. (b) shows physically aggregated execution of events by routing the stream to one AC. (c) shows event routing for *fully* intra-transaction parallel execution. (d) shows *balanced* intra-transaction parallel execution.

10.3.1 *Opportunity 1: Duality of Disaggregation*

As indicated earlier in Figure 10.1, for partitionable OLTP workloads an architecture-less DBMS can achieve nearly the same throughput as an (aggregated) shared-nothing architecture. Key to this is the duality of disaggregation in the architecture-less DBMS. The architecture-less DBMS distinguishes logical disaggregation of the DBMS design and physical disaggregation of the DBMS execution. Logically, the DBMS is entirely disaggregated into independent fine-grained functionality interacting via events and data streams. However, while the logical execution is disaggregated into many small events, physically the events can still be executed in an aggregated manner if desired. This opens up the opportunity to achieve high data locality if desired as all events can be executed close to the data, e.g., a partition of the database.

For example, an OLTP transaction may consist of an event stream like in Figure 10.4 (a). Yet, this logical disaggregation does not mandate disaggregated execution. In the contrary, any sub-sequence of these events can be physically aggregated and executed by a single AC. As shown in Figure 10.4 (b), the entire event stream of a transaction can be executed by a single AC. In fact, this physical aggregation of events establishes a shared-nothing architecture that performs on par with the static shared-nothing architecture of DBx1000 as shown earlier in Figure 10.1.

the gist: Through the duality of logical vs. physical disaggregation, we believe that an architecture-less DBMS can efficiently mimic diverse architectures ranging from entirely aggregated shared-nothing to fine-grained disaggregated as required, simply shifting between those by adapting event and data routes.

10.3.2 *Opportunity 2: Execution Strategies*

Along with the freedom of achieving different architectures on the macro-level, the execution model in an architecture-less DBMS also provides broad freedom to layout parallel execution strategies in an optimal manner on the micro-level.

Generally, as explained earlier, transactions are represented as event streams flowing through the architecture-less DBMS. Importantly for transaction execution, this event-based execution allows diverging from typical execution models in OLTP that aim for *inter*-transaction parallel execution and allows investigating also other forms of parallelism. For example, event-based execution naturally brings opportunities to ad hoc parallelize execution within a single transaction to achieve *intra*-transaction parallelism, especially when contention prohibits inter-transaction parallel execution.

The efficiency of this freedom to change the transaction execution on the micro-level becomes also visible in Figure 10.5. When running a partitionable OLTP workload in the first phase (0-2), AnyDB mimics not only a shared-nothing architecture but also uses classical inter-transaction parallelism. Afterwards it uses intra-transaction parallelism in the second phase (3-5) for the highly skewed, contended OLTP workload, where 100% of TPC-C payment transactions operate on one warehouse only.

The baseline DBx1000 in this experiment uses a shared-nothing model and is thus bound by the resources that are assigned to one partition resulting in 0.7 M txn/s. In such a case, our architecture-less DBMS allows to simply shift from inter- to intra-transaction parallelism by routing events of a single transaction to several ACs, i.e., shifting from Figure 10.4 (b) to (c). Such intra-transaction parallel execution may accelerate transactions in contended OLTP workload, as already proven by architectures such as DORA [137].

However, just like any design decisions in a static architecture, also static intra-transaction parallelization does not always prove beneficial either. For example, Figure 10.5 shows that with naïve intra-transaction parallelism, where every independent operation of a transaction is farmed out to a different AC, AnyDB only achieves 0.8 M txn/s (orange squares), barely exceeding the inter-transaction parallel execution of the DBx1000 baselines (blue lines). The main reason is that the overhead of parallelizing within a transaction dominates the execution.

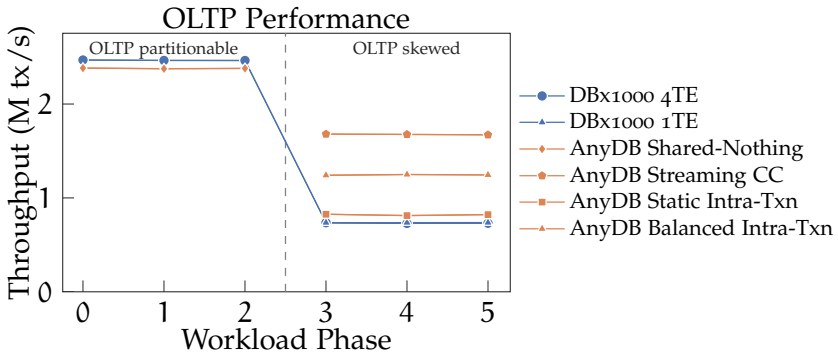


Figure 10.5: OLTP performance of AnyDB versus the shared-nothing DBx1000 under partitionable and skewed OLTP. In phases 0-2 AnyDB acts as a shared-nothing DBMS using an inter-transaction parallel execution model while in phases 3-5 AnyDB acts as a shared-disk DBMS using an intra-transaction parallel execution model. Note that for DBx1000, 4 transaction executors (TEs) perform like a single TE due to high contention between transactions.

The architecture-less DBMS addresses the challenging parallelization of transactions in the following way: generally, the representation of transactions as event streams allows the architecture-less DBMS to route independent sub-sequences of events (i.e., sub-sequences of operations) to multiple ACs for parallel execution. Thus, the challenge in an architecture-less DBMS is to split a transaction into suitable sub-sequences of events and route them to different ACs, balancing the amount of work versus overhead. In our experiments, for example, we partition the TPC-C payment transaction into one sub-sequence with several brief update statements and a second sub-sequence with a long range scan, as depicted in Figure 10.4 (d).

Finding an optimal splitting and routing of event sequences of transactions depends on many factors. One important point is that the individual sub-sequences of operations have similar execution latencies, such that the overall latency of the transaction is minimized. Finding such an optimal routing of events is an opportunity for learning query optimization and scheduling, as mentioned before. As shown in Figure 10.5, with this balanced intra-transaction parallelization AnyDB achieves 1.2 M txn/s (orange triangle) with only 2 ACs, outperforming the baseline (blue lines) and AnyDB's naïve parallelization (orange square) by 3.2x and 3x in throughput per thread, respectively.

the gist: Generally, we envision, that the freedom of the execution models on the micro-level in an architecture-less DBMS can enable new parallel transaction execution models spanning any design between pure inter-transaction to aggressive (fine-grained) intra-transaction parallelism on a per-transaction level.

10.3.3 *Opportunity 3: Concurrency Control*

In OLTP workloads, especially under high contention, concurrency control (CC) causes significant coordination effort and challenges efficient parallelization [17]. In an architecture-less DBMS, the event-based nature of transactions provides the opportunity to transform CC to a streaming problem. Thereby, the architecture-less DBMS can improve the efficiency of traditional CC schemes and opens many opportunities for novel CC schemes, as discussed next.

TRANSFORMING TRADITIONAL CONCURRENCY CONTROL: Interestingly, many traditional CC protocols are stream-like already and thus benefit from a direct mapping to the asynchronous (non-blocking) execution model of the architecture-less DBMS. For example, a pessimistic lock-based CC scheme [24] needs to match incoming lock requests with its lock state. This can be mapped to a streaming join on an event stream containing lock requests and a data stream containing the lock state of the requested item. Similarly, verification in optimistic CC protocols [111] joins the read/write set of a transaction which is one data stream with the current state of the database which is another data stream. Despite these benefits for traditional CC protocols, the architecture-less DBMS offers opportunities for novel coordination-free CC schemes vastly outperforming the traditional approaches, as explained in the following.

NOVEL STREAMING CONCURRENCY CONTROL: The key idea of rethinking CC schemes is that they can be enabled by efficiently ordering events of (conflicting) transactions flowing through the architecture-less DBMS, rather than actively synchronizing execution of concurrent transaction using traditional CC schemes causing high coordination overhead especially under high contention. Here, the streaming execution of transactions brings new opportunities despite high contention.

Concurrency control in AnyDB can be implicitly and coordination-free encoded into event routes. That is, for consistency of concurrent transactions it suffices to route their events in a consistent order through ACs which execute the conflicting operations. For example, considering two TPC-C payment transactions accessing the same warehouse, AnyDB can guarantee consistency by simply routing their events to all involved ACs in the same order. Thereby, AnyDB enables intra-transaction parallelism, routing independent events through different ACs and also provides CC without the need to actively synchronize operations at the same time.

Note that event-ordering does not violate our non-blocking (asynchronous) execution model of ACs. Operations (i.e., events) of conflicting transactions simply remain in the ACs' input queues for ordering while other events can still be executed. In Figure 10.5, we see that this

instantiation of AnyDB called *streaming CC* (orange pentagon), yields 1.7 M txn/s for TPC-C payment under high contention (phases 3-5). This is much closer to the performance of the uniform (partitionable) execution of TPC-C payment in phases 0-2.

the gist: Along the properties of event streams, we envision novel CC protocols, that avoid active synchronization, as discussed. Moreover, the streaming CC enables new directions where events are gradually rerouted depending on the load, e.g., at first all events for a specific transaction are routed to a single AC until this AC becomes overloaded. Then AnyDB may transparently repartition event streams while still guaranteeing consistent event order.

10.4 OPPORTUNITIES FOR OLAP AND HTAP

Previously, we have described that the execution model of an architecture-less DBMS provides many opportunities for OLTP. In the following, we discuss further opportunities for OLAP as well as mixed HTAP workloads.

Especially for OLAP, operations encoded in the event streams are data intensive (e.g., a join of two large tables). Therefore, data streams must efficiently bring data to wherever events are executed as if data access was local to facilitate the architecture-less DBMS. While this aspiration of "omnipresent" data appears challenging, we observe that in DBMSs one often knows data to be accessed way ahead of time before actually processing it. For example, in OLAP data is only accessed and processed after several milliseconds of query optimization and compilation. Hence, we propose *data beaming*, a technique initiating data streams early and pushing data to ACs where events will be executed.

In the following experiment, we demonstrate the effect of data beaming with a simple OLAP query. Based on CH-benCHmark Q3 [46], our query reports all open orders for all customers from states beginning with "A" since 2007 via 3 (filtered) scans and 2 joins. In Figure 10.6, we display the effect of data beaming in several degrees for this query: (1) In blue, the baseline does not utilize beaming, but instead passively pulls in data when needed. (2) In orange, the build sides are beamed during query compilation, then joins are executed. (3) In green, build and probe sides are beamed.

To detail the efficiency of data beams, we implement two variants: one where data beams only need to shuffle locally (not over the network) and one where they shuffle data across the network. The solid lines in Figure 10.6 demonstrate the runtime using local beaming via shared-memory queues [88] (e.g., to hide NUMA latencies) and the dashed lines show the beaming across the network for a disaggregated architecture using DPI-flows [76]. On the x-axis as reference point for

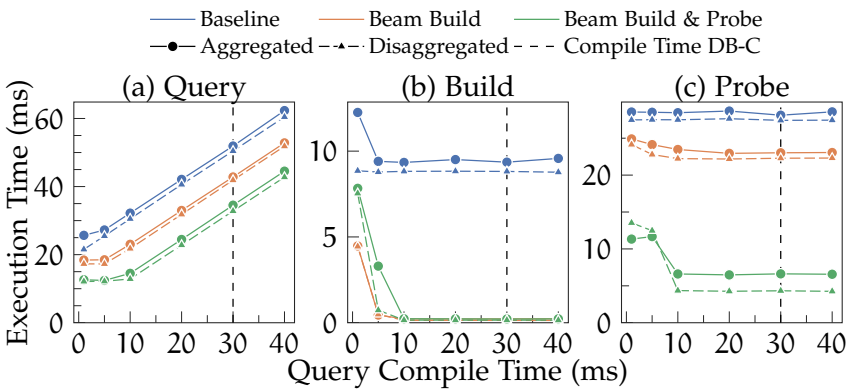


Figure 10.6: Data beaming can effectively shorten query execution for disaggregated execution of OLAP workloads and in the best case hide latencies of data shuffling completely.

query compile time, 30ms marks the time taken by a commercial DBMS (DB-C) to compile this query.

Figure 10.6 (a) shows that the overall query execution time with beaming is only slightly higher than query compile time (green line), whereas without beaming (blue line) the query execution time has additional latency of 20ms, since data transfer is not overlapped with query compile time. In detail, Figures 10.6 (b) and (c) show the individual effects of data beaming on the build and probe side (without query compilation overhead), respectively. We see that beaming can reduce the runtime of the smaller build side almost to 0ms. For the larger probe side, beaming also reduces the runtime from 30ms to less than 10ms. Notably, the disaggregated architecture (which needs to shuffle data across the network) performs even better than the aggregated architecture, as DPI offloads event and data transfers to the InfiniBand NICs acting as a co-processor in AnyDB.

The previous experiment demonstrated the utility of data beaming to hide data transfer latencies in OLAP workloads. Besides hiding transfer latencies, data beaming can also be used to achieve other goals such as resource isolation, e.g., for HTAP workloads. The idea is that in HTAP workloads, we can use data beams to route data intensive analytical queries to additional compute resources disaggregated from storage while latency-sensitive transactions are executed close to the data.

The HTAP workloads in Figure 10.1 (phase 6-11) outline such scenarios, where the OLAP query of the previous experiment is executed in parallel to the OLTP workload. Here, AnyDB executes the OLAP query independently of the OLTP workload, only sharing storage resources, whereas the OLAP query in the static DBx1000 uses the same transaction resources for OLAP queries as for the OLTP workload. Thereby, AnyDB simultaneously provides higher OLTP and OLAP performance than DBx1000.

the gist: In general, we envision AnyDB to establish flexible architectures through described data beaming as well as optimal per query/transaction event routing, opening up new paths to hybrid architectures and supporting various types of deployments.

10.5 CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we have proposed architecture-less DBMSs, a radical new approach for scale-out distributed DBMSs. In addition to the discussed opportunities, we see many further interesting research directions arising from the architecture-less approach:

ELASTICITY FOR FREE: Flexible event routing and data beaming open up opportunities, apart from resource isolation, for transparent elasticity without additional latencies. Considering events are self-contained and state is always beamed, elasticity for execution of event streams just means consistent routing of events and their state to an elastic number of ACs. Even more, as mentioned before since all events and data are both delivered as streams to ACs, these streams could be repartitioned or rerouted to distribute load in the system adaptively.

TRANSPARENT HETEROGENEITY: The stateless execution model in conjunction with the opportunity for elasticity further facilitates transparent (ad hoc) integration of heterogeneous compute resources per query, including but not limited to accelerators (e.g., FPGAs or GPUs) and programmable data planes (e.g., programmable NICs or switches). Moreover, since events fully describe what to do and data streams deliver all required state, event execution (*how to do it*) can be specialized for FPGAs, etc. without any impact nor dependencies on the rest of the DBMS.

CROSSING CLOUDS AND MORE: Finally, data beaming is an interesting concept generally hiding data transfer cost not only within but also across data centers. This opens up opportunities for DBMS deployments across on-premise, cloud offerings, and the edge without paying significant latencies for data transfer. For example, an architecture-less DBMS for HTAP workload may run transactions for daily business on-premise and may ad hoc beam data to cloud resources for sporadic reporting, combining the benefits of both platforms efficiently.

BIBLIOGRAPHY

[1] Advanced Micro Devices Inc. *AMD64 Technology: AMD64 Architecture Programmer's Manual Volume 2: System Programming (Revision 3.38)*. 2021. URL: https://www.amd.com/system/files/TechDocs/24593.pdf.

[2] Advanced Micro Devices, Inc. *AMD EPYC™ 7003 SERIES PROCESSORS*. 2021. URL: https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf.

[3] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. "Tackling Hardware/Software Co-design from a Database Perspective." In: *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*. URL: https://www.cidrdb.org/cidr2020/papers/p30-alonso-cidr20.pdf.

[4] Amazon Web Services, Inc. or its affiliates. *Amazon Aurora Serverless*. 2020. URL: https://aws.amazon.com/rds/aurora/serverless/.

[5] Amazon Web Services, Inc. or its affiliates. *Amazon Redshift Serverless (preview)*. 2022. URL: https://aws.amazon.com/redshift/redshift-serverless/.

[6] Amazon Web Services, Inc. or its affiliates. *AWS Graviton Processor*. 2022. URL: https://aws.amazon.com/ec2/graviton/.

[7] Mikkel Møller Andersen and Pinar Tözün. "Micro-architectural Analysis of a Learned Index." In: *CoRR* abs/2109.08495 (2021). DOI: 10.48550/arxiv.2109.08495.

[8] Panagiotis Antonopoulos et al. "Socrates: The New SQL Server in the Cloud." In: *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. ACM, 2019. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3314047.

[9] John Appleby. *SAP HANA - Scale-up or Scale-out Hardware?* URL: https://blogs.saphana.com/2014/12/10/sap-hana-scale-scale-hardware/.

[10] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. "Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads." In: *Proceedings of the VLDB Endowment* 11.2 (2017). ISSN: 2150-8097. DOI: 10.14778/3149193.3149194.

[11] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. "The Case For Heterogeneous HTAP." In: *8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*. 2017. URL: http://www.cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf.

[12] Arm Limited or its affiliates. *Arm® Instruction Set Reference Guide*. 2018. URL: https://documentation-service.arm.com/static/5e7b694616d2907d594029eb?token=.

[13] Arm Limited or its affiliates. *Neoverse V1 reference design*. 2021. URL: https://developer.arm.com/tools-and-software/development-boards/neoverse-reference-design.

[14] M. M. Astrahan et al. "System R: Relational Approach to Database Management." In: *ACM Trans. Database Syst.* 1.2 (1976). ISSN: 0362-5915. DOI: 10.1145/320455.320457.

[15] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. "Optimal Column Layout for Hybrid Workloads." In: *Proceedings of the VLDB Endowment* 12.13 (2019). ISSN: 2150-8097. DOI: 10.14778/3358701.3358707.

[16] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, 2000. ISBN: 1-58113-217-4. DOI: 10.1145/342009.335420.

[17] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "The Tale of 1000 Cores: An Evaluation of Concurrency Control on Real(ly) Large Multi-Socket Hardware." In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. ACM, 2020. DOI: 10.1145/3399666.3399910.

[18] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "AnyDB: An Architecture-less DBMS for Any Workload." In: *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*. 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper10.pdf.

[19] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. *The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware — Measurements, Logs, Plots*. archived: https://doi.org/10.48328/tudatalib-726, browsable: https://github.com/DataManagementLab/VLDBJ_1000_cores_measurements. 2021.

[20] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. *The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware — Source Code*. archived: https://doi.org/10.48328/tudatalib-727, browsable: https://github.com/DataManagementLab/DBx1000. 2021.

[21] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. "The Full Story of 1000 Cores: An Examination of Concurrency Control on Real(ly) Large Multi-Socket Hardware." In: *The VLDB Journal*. 2022. DOI: 10.1007/s00778-022-00742-4.

[22] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. "Robust Performance of Main Memory Data Structures by Configuration." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, New York, NY, USA: ACM, 2020. DOI: 10.1145/3318464.3389725.

[23] Nick Benton, Luca Cardelli, and Cédric Fournet. "Modern Concurrency Abstractions for C#." In: *ACM Trans. Program. Lang. Syst.* 26.5 (2004). ISSN: 0164-0925. DOI: 10.1145/1018203.1018205.

[24] Philip A. Bernstein and Nathan Goodman. "Concurrency Control in Distributed Database Systems." In: *ACM Comput. Surv.* 13.2 (1981).

[25] Philip A. Bernstein and Nathan Goodman. "Multiversion Concurrency Control—Theory and Algorithms." In: *ACM Trans. Database Syst.* 8.4 (1983).

[26] Timo Bingmann. *STX B+ Tree C++ Template Classes*. 2013. URL: http://panthema.net/2007/stx-btree/.

[27] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. "HOT: A Height Optimized Trie Index for Main-Memory Database Systems." In: *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. ACM, 2018. DOI: 10.1145/3183713.3196896.

[28] Nils Boeschen and Carsten Binnig. "GalOP: Towards a GPU-Accelerated OLTP DBMS." In: *DAMON'21: Proceedings of the 17th International Workshop on Data Management on New Hardware*. ACM, 2021. ISBN: 978-1-4503-8556-5. DOI: 10.1145/3465998.3466007.

[29] Peter Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[30] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. "SAHARA: Memory Footprint Reduction of Cloud Databases with Automated Table Partitioning." In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, Paolo

Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang. OpenProceedings.org, 2022. DOI: 10.5441/002/edbt.2022.02.

[31] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. "Investigating the Performance of Hardware Transactions on a Multi-Socket Machine." In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 2935796: ACM, 2016. DOI: 10.1145/2935764.2935796.

[32] Paolo Bruni, Rafael Garcia, Sabine Kaschta, Josef Klitsch, Ravi Kumar, Andrei Lurie, Michael Parbs, Rajesh Ramachandran, et al. *DB2 10 for z/OS Technical Overview*. IBM Redbooks, 2014. ISBN: 978-0-7384-3511-4. URL: https://www.redbooks.ibm.com/abstracts/sg247892.html?Open.

[33] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. "Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores." In: *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*. OPODIS 2013. Springer-Verlag, 2013. DOI: 10.1007/978-3-319-03850-6_7.

[34] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. "How to implement Any Concurrent Data Structure." In: *Commun. ACM* 61.12 (2018). ISSN: 0001-0782. DOI: 10.1145/3282506.

[35] Wei Cao et al. "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers." In: *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD'21)*. ACM, 2021. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3457560.

[36] Joseph O. Celtruda, William R. Crosthwait, John G. Earle, Roy F. Henderson, and John W. (JR.) Fennel. *APPARATUS AND METHOD FOR SERIALIZING INSTRUCTIONS FROM TWO INDEPENDENT INSTRUCTION STREAMS*. 1972. URL: https://worldwide.espacenet.com/patent/search?q=pn%5C%3DCA954227A.

[37] S. Chandrasekaran and R. Bamford. "Shared Cache - The Future of Parallel Databases." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2003. DOI: 10.1109/ICDE.2003.1260883.

[38] P. Y. Chang and W. W. Myre. "OS/2 EE Database Manager overview and technical highlights." In: *IBM Systems Journal* 27.2 (1988). DOI: 10.1147/sj.272.0105.

[39] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. "Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine." In: *Proc. VLDB Endow.* 15.1 (2021). ISSN: 2150-8097. DOI: 10.14778/3485450.3485461.

[40] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. "New Directions in Cloud Programming." In: Chaminade, USA, 2021. URL: http://www.cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf.

[41] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. "Scalable Address Spaces using RCU Balanced Trees." In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012. DOI: 10.1145/2150976.2150998.

[42] Kevin Closson. *You buy a NUMA system, Oracle says disable NUMA! What gives?* 2009. URL: https://kevinclosson.net/2009/05/14/you-buy-a-numa-system-oracle-says-disable-numa-what-gives-part-ii/.

[43] Alibaba Cloud. *Alibaba Cloud Unveils New Server Chips to Optimize Cloud Computing Services*. 2021. URL: https://www.alibabacloud.com/press-room/alibaba-cloud-unveils-new-server-chips-to-optimize-cloud-computing-services.

[44] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM* 13.6 (1970). ISSN: 0001-0782. DOI: 10.1145/362384.362685.

[45] Edgar F Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-14192-2.

[46] Richard Cole et al. "The Mixed Workload CH-benCHmark." In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. ACM, 2011. DOI: 10.1145/1988842.1988850.

[47] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010. ISBN: 978-1-4503-0036-0. URL: https://github.com/brianfrankcooper/YCSB.

[48] Travis Craig. *Building FIFO and Priority Queuing Spin Locks from Atomic Swap*. Technical Report. TR 93-02-02, University of Washington, 1993.

[49] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "Everything You Always Wanted to Know About Synchronization But Were Afraid to Ask." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2522714: ACM, 2013. DOI: 10.1145/2517349.2522714.

[50] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Portland, OR, USA: ACM, 2020. ISBN: 9781450367356. DOI: 10.1145/3318464.3389723.

[51] Departments of Informatics, University of Klagenfurt. *Oracle V2*. URL: http://cs-exhibitions.uni-klu.ac.at/index.php?id=403.

[52] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL Server's Memory-optimized OLTP Engine." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM Press, 2013. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463710.

[53] Dave Dice and Alex Kogan. "Avoiding Scalability Collapse by Restricting Concurrency." In: *Euro-Par 2019: Parallel Processing*. Lecture Notes in Computer Science. Springer International Publishing, 2019. ISBN: 978-3-030-29399-4. DOI: 10.1007/978-3-030-29400-7_26.

[54] Dave Dice, Virendra J. Marathe, and Nir Shavit. "Flat-Combining NUMA Locks." In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2011. DOI: 10.1145/1989493.1989502.

[55] David Dice, Virendra J. Marathe, and Nir Shavit. "Lock Cohorting: A General Technique for Designing NUMA Locks." In: *ACM Trans. Parallel Comput.* 1.2 (2015). ISSN: 2329-4949. DOI: 10.1145/2686884.

[56] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Batabases." In: *Proc. VLDB Endow.* 7.4 (2013). ISSN: 2150-8097. DOI: 10.14778/2732240.2732246.

[57] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, and Jon Webb. *The CMU Task Parallel Program Suite*. Report. Carnegie-Mellon University Pittsburgh, Dept. of Computer Science, 1994.

[58] Jialin Ding et al. "ALEX: An Updatable Adaptive Learned Index." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. ACM, 2020. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3389711.

[59] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[60] Markus Dreseler, Thomas Kissinger, Timo Djürken, Eric Lübke, Matthias Uflacker, Dirk Habich, Hasso Plattner, and Wolfgang Lehner. "Hardware-Accelerated Memory Operations on Large-Scale NUMA Systems." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2017. URL: http://www.adms-conf.org/2017/camera-ready/adms-gru.pdf.

[61] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. "Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Melbourne, Victoria, Australia: ACM, 2015. ISBN: 9781450327589. DOI: 10.1145/2723372.2723726.

[62] Ian Essling. *"Big 3" Holiday Days Exceed $22 Billion in Online Spending, Grow 25 Percent Year-Over-Year*. 2020. URL: https://www.comscore.com/Insights/Blog/Big-3-Holiday-Days-Exceed-22-Billion-in-Online-Spending.

[63] Jose M. Faleiro and Daniel J. Abadi. "Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?" In: *In Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*. 2017. URL: http://www.cidrdb.org/cidr2017/papers/p121-faleiro-cidr17.pdf.

[64] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. "High Performance Transactions via Early Write Visibility." In: *Proceedings of the VLDB Endowment*. 2017. DOI: 10.14778/3055540.3055553.

[65] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA Database: Data Management for Modern Business Applications." In: *SIGMOD Rec.* 40.4 (2012). ISSN: 0163-5808. DOI: 10.1145/2094114.2094126.

[66] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database–An Architecture Overview." In: *IEEE Data Eng. Bull.* 35.1 (2012). URL: http://sites.computer.org/debull/A12mar/p28.pdf.

[67] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. "Make the Most out of Last Level Cache in Intel Processors." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019. DOI: 10.1145/3302424.3303977.

[68] Brad Fitzpatrick. *livejournal*. 2003. URL: https://changelog.livejournal.com/637455.html.

[69] Elvis C. Foster and Shripad Godbole. "Overview of DB2." In: *Database Systems: A Pragmatic Approach*. Apress, 2016. ISBN: 978-1-4842-1191-5. DOI: 10.1007/978-1-4842-1191-5_23.

[70] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux." In: *AUUG Conference Proceedings*. Vol. 85. AUUG, Inc. Kensington, NSW, Australia, 2002. URL: https://www.kernel.org/doc/mirror/ols2002.pdf%5C#page=479.

[71] *Gen-Z Core Specification 1.1*. Gen-Z Consortium. 2019.

[72] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. "Towards Scalable Real-Time Analytics: an Architecture for Scale-Out of OLxP Workloads." In: *Proc. VLDB Endow.* 8.12 (2015). ISSN: 2150-8097. DOI: 10.14778/2824032.2824069.

[73] Goetz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler, eds. *Robust Query Processing, 19.09. - 24.09.2010*. Vol. 10381. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010. DOI: 10.4230/DagRep.2.8.1.

[74] Vincent Gramoli. "More than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms." In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. New York, NY, USA: ACM, 2015. ISBN: 978-1-4503-3205-7. DOI: 10.1145/2688500.2688501.

[75] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. "Amazon Redshift and the Case for Simpler Data Warehouses." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, 2015. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742795.

[76] Alonso Gustavo, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. "DPI: The Data Processing Interface for Modern Networks." In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019. URL: http://www.cidrdb.org/cidr2019/papers/p11-alonso-cidr19.pdf.

[77] James Hamilton. *Four DB2 Code Bases?* 2020. URL: https://perspectives.mvdirona.com/2017/12/1187/.

[78] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer-Verlag, 1999. ISBN: 3-540-65040-7.

[79] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. "An Evaluation of Distributed Concurrency Control." In: *Proc. VLDB Endow.* 10.5 (2017). ISSN: 2150-8097. DOI: 10.14778/3055540.3055548.

[80] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. "Architecture of a Database System." In: *Foundations and Trends(R) in Databases* 1.2 (2007). ISSN: 1931-7883, 1931-7891. DOI: 10.1561/1900000002.

[81] John L. Hennessy. *Computer Architecture: a Quantitative Approach*. 5. Morgan Kaufmann Publ., 2012. ISBN: 978-0-12-383872-8.

[82] Hewlett Packard Enterprise. *The Unique Modular Architecture of HPE Superdome Flex: How it Works and Why It Matters*. 2018. URL: https://community.hpe.com/t5/Servers-The-Right-Compute/The-unique-modular-architecture-of-HPE-Superdome-Flex-How-it/ba-p/7001330%5C#.XnsMbEBFyAg.

[83] Hewlett Packard Enterprise Development LP. *HPE Superdome Flex, Intel Processors Scale SAP HANA*. 2018. URL: https://www.intel.com/content/www/us/en/big-data/hpe-superdome-flex-sap-hana-wp.html.

[84] Hewlett Packard Enterprise Development LP. *HPE Superdome Flex Server Architecture and RAS*. 2020. URL: https://assets.ext.hpe.com/is/content/hpedam/documents/a00036000-6999/a00036491/a00036491enw.pdf.

[85] Benjamin Hilprecht and Carsten Binnig. "One Model to Rule them All: Towards Zero-Shot Learning for Databases." In: *11th Annual Conference on Innovative Data Systems Research (CIDR '22)*. 2022. URL: http://www.cidrdb.org/cidr2022/papers/p16-hilprecht.pdf.

[86] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. ACM, 2020. ISBN: 978-1-4503-6735-6. URL: http://doi.org/10.1145/3318464.3389704.

[87] Benjamin Hilprecht et al. "DBMS Fitting: Why should we learn what we already know?" In: *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*. 2020. URL: http://www.cidrdb.org/cidr2020/papers/p34-hilprecht-cidr20.pdf.

[88] Bo Hu and Jordan DeLong. *Folly single-producer-single-consumer queue*. 2019. URL: https://github.com/facebook/folly/blob/d2c64d94c7e892925a02a080c886ab3df3f5c937/folly/ProducerConsumerQueue.h.

[89] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. "X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing." In: *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. ACM, 2019. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3314041.

[90] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. "Opportunities for Optimism in Contended Main-Memory Multicore Transactions." In: *Proc. VLDB Endow.* 13.5 (2020). ISSN: 2150-8097. DOI: 10.14778/3377369.3377373.

[91] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn." In: *9th Annual Conference on Innovative Data Systems Research (CIDR '19)*. 2019. URL: http://www.cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf.

[92] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. "MonetDB: Two Decades of Research in Column-oriented Database Architectures." In: *IEEE Data Eng. Bull.* 35.1 (2012). URL: http://sites.computer.org/debull/A12mar/monetdb.pdf.

[93] Stratos Idreos et al. "Learning Key-Value Store Design." In: *CoRR* abs/1907.05443 (2019). arXiv: 1907.05443. URL: http://arxiv.org/abs/1907.05443.

[94] *Index Microbench*. 2017. URL: https://github.com/speedskater/index-microbench.

[95] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 (Order Number: 325462-077US)*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[96] International Business Machines Corporation. *POWER8 Processor User's Manual for the Single-Chip Module (Version 1.3)*. 2016. URL: https://openpowerfoundation.org/?resource_lib=power8-processor-users-manual.

[97] International Business Machines Corporation. *POWER9 Processor User's Manual (Version 2.1)*. 2019. URL: https://openpowerfoundation.org/?resource_lib=power9-processor-users-manual.

[98] International Business Machines Corporation. *Power ISA™ Version 3.1*. 2020. URL: https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0.

[99] International Business Machines Corporation. *DB2 for IBM i*. 2021. URL: https://www.ibm.com/support/IGNpages/db2-ibm-i.

[100] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. "Eliminating Unscalable Communication in Transaction Processing." In: *The VLDB Journal* 23.1 (2014). ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-013-0312-3.

[101] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. "Exploiting Coroutines to Attack the "Killer Nanoseconds"." In: *Proc. VLDB Endow.* 11.11 (2018). ISSN: 2150-8097. DOI: 10.14778/3236187.3236216.

[102] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. "DB2's use of the coupling facility for data sharing." In: *IBM Systems Journal* 36.2 (1997). DOI: 10.1147/sj.362.0327.

[103] Robert Kallman et al. "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System." In: *Proc. VLDB Endow.* 1.2 (2008). ISSN: 2150-8097. DOI: 10.14778/1454159.1454211.

[104] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots." In: *2011 IEEE 27th International Conference on Data Engineering*. 2011. DOI: 10.1109/ICDE.2011.5767867.

[105] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask." In: *Proc. VLDB Endow.* 11.13 (2018). ISSN: 2150-8097. DOI: 10.14778/3275366.3284966.

[106] Hideaki Kimura. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, 2015. DOI: 10.1145/2723372.2746480.

[107] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. "ERIS: A Numa-Aware In-Memory Storage Engine for Analytical Workloads." In: *Proceedings of the VLDB Endowment* 7.14 (2014). DOI: 10.1.1.475.3378.

[108] Barbara Klein, Diane Goff, John Butterweck, Margaret Wilson, Moira McFadden Lanyi, Rick Long, Sandy Sherrill, Steve Nathan, and Kenny Blackmann. *An Introduction to IMS: Your Complete Guide to IBM Information Management System*. 2nd. ed. IBM Press, 2012. ISBN: 978-0-13-288687-1.

[109] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning." In: *CoRR* abs/1808.03196 (2018). DOI: 10.48550/arXiv.1808.03196.

[110] Sven O. Krumke and Clemens Thielen. "The Generalized Assignment Problem with Minimum Quantities." In: *European Journal of Operational Research* 228.1 (2013). ISSN: 0377-2217. DOI: 10.1016/j.ejor.2013.01.027.

[111] H. T. Kung and John T. Robinson. "On Optimistic Methods for Concurrency Control." In: *ACM Trans. Database Syst.* 6.2 (1981). ISSN: 0362-5915.

[112] Tirthankar Lahiri and Markus Kissling. *Oracle's In-Memory Database Strategy for OLTP and Analytics*. 2015. URL: https://www.doag.org/formes/pubfiles/7378967/2015-K-DB-Tirthankar_Lahiri-Oracle_s_In-Memory_Database_Strategy_for_Analytics_and_OLTP-Manuskript.pdf.

[113] Mahesh Lal. *Neo4j Graph Data Modeling*. Packt Publishing Ltd, 2015. ISBN: 978-1-78439-344-1.

[114] Hai Lan, Zhifeng Bao, and Yuwei Peng. "An Index Advisor Using Deep Reinforcement Learning." In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. CIKM '20. ACM, 2020. ISBN: 978-1-4503-6859-9. DOI: 10.1145/3340531.3412106.

[115] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. "Parallel Replication across Formats in SAP HANA for Scaling out Mixed OLTP/OLAP Workloads." In: *Proc. VLDB Endow.* 10.12 (2017). ISSN: 2150-8097. DOI: 10.14778/3137765.3137767.

[116] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. Snowbird, Utah, USA: ACM, 2014. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610507.

[117] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. "The Bw-Tree: A B-tree for New Hardware Platforms." In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013. DOI: 10.1109/ICDE.2013.6544834.

[118] Feifei Li. "Cloud-native Database Systems at Alibaba: Opportunities and Challenges." In: *Proceedings of the VLDB Endowment* 12.12 (2019). DOI: 10.14778/3352063.3352141.

[119] Sam S. Lightstone, Toby J. Teorey, and Tom Nadeau. *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*. Morgan Kaufmann, 2010. ISBN: 0-08-055231-5.

[120] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. "Cicada: Dependably Fast Multi-Core In-Memory Transactions." In: *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. ACM, 2017. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064015.

[121] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. "IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability." In: *IEEE Data Eng. Bull.* 36.2 (2013). URL: http://sites.computer.org/debull/a13june/a13jun-cd.pdf%5C#page=16.

[122] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. "On the Design and Scalability of Distributed Shared-Data Databases." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Melbourne, Victoria, Australia: ACM, 2015. ISBN: 9781450327589. DOI: 10.1145/2723372.2751519.

[123] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications." In: *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi.

[124] Redis Ltd. *Redis: REmote DIctionary Server*. URL: https://redis.io/.

[125] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. "Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems." In: *Proc. VLDB Endow.* 12.12 (2019). ISSN: 2150-8097. DOI: 10.14778/3352063.3352112.

[126] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. "BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications." In: *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. SIGMOD '17. New York, NY, USA: ACM, 2017. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3035959.

[127] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. "Benchmarking Learned Indexes." In: *Proc. VLDB Endow.* 14.1 (2020). ISSN: 2150-8097. DOI: 10.14778/3421424.3421425.

[128] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. "Bao: Making Learned Query Optimization Practical." In: *Proceedings of the*

2021 ACM SIGMOD International Conference on Management of Data (SIGMOD'21). New York, NY, USA: ACM, 2021. ISBN: 9781450383431. DOI: 10.1145/3448016.3452838.

[129] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. "Cloud computing - The Business Perspective." In: *Decision Support Systems* 51.1 (2011). ISSN: 0167-9236. DOI: 10.1016/j.dss.2010.12.006.

[130] *Memcached*. URL: https://memcached.org/.

[131] Microsoft. *Hyperscale Service Tier*. 2020. URL: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale.

[132] David L. Mulnix. *Intel® Xeon® Processor Scalable Family Technical Overview*. 2017. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[133] Felix Naumann. *Genealogy of Relational Database Management Systems*. 2018. URL: https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/projekte/RDBMSGenealogy/RDBMS_Genealogy_V6.pdf.

[134] Michael A Olson, Keith Bostic, and Margo I Seltzer. "Berkeley DB." In: *USENIX Annual Technical Conference, FREENIX Track*. 1999. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/olson/olson.pdf.

[135] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory." In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. ACM, 2016. DOI: 10.1145/2882903.2915251.

[136] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. "Workload-Aware Storage Layout for Database Systems." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, 2010. ISBN: 9781450300322. DOI: 10.1145/1807167.1807268.

[137] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. "Data-Oriented Transaction Execution." In: *Proceedings of the VLDB Endowment* 3.1-2 (2010). ISSN: 21508097. DOI: 10.14778/1920841.1920959.

[138] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. "Starling: A Scalable Query Engine on Cloud Functions." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Portland,

OR, USA: ACM, 2020. ISBN: 9781450367356. DOI: 10.1145/3318464.3380609.

[139] Orestis Polychroniou and Kenneth A. Ross. "A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, 2014. DOI: 10.1145/2588555.2610522.

[140] Danica Porobic, Erietta Liarou, Pinar Tozun, and Anastasia Ailamaki. "ATraPos: Adaptive Transaction Processing on Hardware Islands." In: *2014 IEEE 30th International Conference on Data Engineering*. 2014. DOI: 10.1109/ICDE.2014.6816692.

[141] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. "Characterization of the Impact of Hardware Islands on OLTP." In: *The VLDB Journal* 25.5 (2016). ISSN: 1066-8888. DOI: 10.1007/s00778-015-0413-2.

[142] Danica Porobic, Pınar Tözün, Raja Appuswamy, and Anastasia Ailamaki. "More than a network: distributed OLTP on clusters of hardware islands." In: *Proceedings of the 12th International Workshop on Data Management on New Hardware - DaMoN '16*. ACM Press, 2016. ISBN: 978-1-4503-4319-0. DOI: 10.1145/2933349.2933355.

[143] Guna Prasaad, Alvin Cheung, and Dan Suciu. "Improving High Contention OLTP Performance via Transaction Scheduling." In: *arXiv:1810.01997 [cs]* (2018). URL: http://arxiv.org/abs/1810.01997.

[144] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. "Interleaving with Coroutines: A Systematic and Practical Approach to Hide Memory Latency in Index Joins." In: *The VLDB Journal* 28.4 (2019). ISSN: 0949-877X. DOI: 10.1007/s00778-018-0533-6.

[145] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement." In: *The Proceedings of the VLDB Endowment* 8.12 (2015). DOI: 10.14778/2824032.2824043.

[146] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. "Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores." In: *Proc. VLDB Endow.* 10.2 (2016). ISSN: 2150-8097. DOI: 10.14778/3015274.3015275.

[147] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. "Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling." In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2014. ISBN: 978-3-319-15350-6. URL: https://link.springer.com/chapter/10.1007/978-3-319-15350-6_7.

[148] Thamir M. Qadah and Mohammad Sadoghi. "QueCC: A Queue-oriented, Control-free Concurrency Architecture." In: *Proceedings of the 19th International Middleware Conference*. ACM, 2018. ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274810.

[149] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 2nd. McGraw-Hill, Inc., 2000. ISBN: 0-07-244042-2.

[150] Jun Rao and Kenneth A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 1999.

[151] Jun Rao and Kenneth A. Ross. "Making B^+-Trees Cache Conscious in Main Memory." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00), May 16-18, 2000, Dallas, Texas, USA*. 2000. DOI: 10.1145/342009.335449.

[152] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. "Automating Physical Database Design in a Parallel Database." In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, 2002. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564757.

[153] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc., 2007. ISBN: 1449390862.

[154] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. "Design Principles for Scaling Multi-core OLTP Under High Contention." In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. ACM, 2016. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882958.

[155] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases: New Opportunities for Connected Data*. Second Edition. O'Reilly Media, Inc., 2015. ISBN: 978-1-4919-3200-1.

[156] *RocksDB: A Persistent Key-Value Store for Fast Storage Environments*. URL: http://rocksdb.org/.

[157] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. "Ffwd: Delegation is (Much) Faster Than You Think." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017. DOI: 10.1145/3132747.3132771.

[158] SAP America, Inc. *Certified and Supported SAP HANA® Hardware*. 2022. URL: https://www.sap.com/dmc/exp/2014-09-02-hana-hardware/enEN/%5C#/solutions?filters=v:deCertified%5C&id=s:87.

[159] SAP SE. *SAP HANA Hardware and Cloud Measurement Tools (HCMT)*. 2020. URL: https://help.sap.com/viewer/02bb1e64c2ae4de7a11369f4e70a6394/2.0/en-US.

[160] *Private conversation with Derek Schumacher, Russ Anderson, and Dimitri Sivanich of Hewlett Packard Enterprise (HPE)*. 2021-11-16. Nov. 16, 2021.

[161] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL: Optimization, Backups, and Replication*. " O'Reilly Media, Inc.", 2012.

[162] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. "Transaction Chopping: Algorithms and Performance Studies." In: *ACM Trans. Database Syst.* 20.3 (1995). DOI: 10.1145/211414.211427.

[163] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. "Scheduling OLTP Transactions via Learned Abort Prediction." In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. aiDM '19. ACM, 2019. ISBN: 978-1-4503-6802-5. DOI: 10.1145/3329859.3329871.

[164] Utku Sirin, Raja Appuswamy, and Anastasia Ailamaki. "OLTP on a Server-grade ARM: Power, Throughput and Latency Comparison." In: ACM Press, 2016. DOI: 10.1145/2933349.2933359.

[165] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. "Micro-architectural Analysis of In-memory OLTP." In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. ACM, 2016. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882916.

[166] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. "A Methodology for OLTP Micro-Architectural Analysis." In: *Proceedings of the 13th International Workshop on Data Management on New Hardware*. DAMON '17. New York, NY, USA: ACM, 2017. ISBN: 9781450350259. DOI: 10.1145/3076113.3076116.

[167] R.D. Sloan. "A Practical Implementation of the Data Base Machine-Teradata DBC/1012." In: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*. Vol. i. 1992. DOI: 10.1109/HICSS.1992.183180.

[168] W. J. Starke, J. S. Dodson, J. Stuecheli, E. Retter, B. W. Michael, S. J. Powell, and J. A. Marcella. "IBM POWER9 Memory Architectures for Optimized Systems." In: *IBM J. Res. Dev.* 62.4–5 (2018). ISSN: 0018-8646. DOI: 10.1147/JRD.2018.2846159.

[169] W. J. Starke, J. Stuecheli, D. M. Daly, J. S. Dodson, F. Auernhammer, P. M. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner. "The Cache and Memory Subsystems of the IBM POWER8 Processor." In: *IBM Journal of Research and Development* 59.1 (2015). DOI: 10.1147/JRD.2014.2376131.

[170] M. Stonebraker and U. Cetintemel. ""One size fits all": An Idea Whose Time Has Come and Gone." In: *21st International Conference on Data Engineering (ICDE'05)*. 2005. DOI: 10.1109/ICDE.2005.1.

[171] M. Stonebraker, L.A. Rowe, and M. Hirohama. "The Implementation of POSTGRES." In: *IEEE Transactions on Knowledge and Data Engineering* 2.1 (1990). DOI: 10.1109/69.50912.

[172] Michael Stonebraker. "The Case for Shared Nothing." In: *IEEE Database Eng. Bull.* 9.1 (1986). URL: http://sites.computer.org/debull/86MAR-CD.pdf.

[173] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. "The Design and Implementation of INGRES." In: *ACM Trans. Database Syst.* 1.3 (1976). ISSN: 0362-5915. DOI: 10.1145/320473.320476.

[174] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. "The End of an Architectural Era: It's Time for a Complete Rewrite." In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. Ed. by Massachusetts Institute of Technology and Michael L. Brodie. 1st ed. ACM, 2018. ISBN: 978-1-947487-19-2. DOI: 10.1145/3226595.3226637.

[175] Michael Stonebraker and Ariel Weisberg. "The VoltDB Main Memory DBMS." In: *IEEE Database Eng. Bull.* 36.2 (2013). URL: http://sites.computer.org/debull/a13june/voltdb1.pdf.

[176] Esther Swilley and Ronald E. Goldsmith. "Black Friday and Cyber Monday: Understanding consumer intentions on two major shopping days." In: *Journal of Retailing and Consumer Services* 20.1 (2013). ISSN: 0969-6989. DOI: 10.1016/j.jretconser.2012.10.003.

[177] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. "E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems." In: *Proceedings of the VLDB Endowment* 8.3 (2014). ISSN: 2150-8097. DOI: 10.14778/2735508.2735514.

[178] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. "Choosing a Cloud DBMS: Ar-

chitectures and Tradeoffs." In: *Proc. VLDB Endow.* 12.12 (2019). ISSN: 2150-8097. DOI: 10.14778/3352063.3352133.

[179] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. "An Analysis of Concurrency Control Protocols for In-memory Databases with CCBench." In: *Proceedings of the VLDB Endowment* 13.13 (2020). ISSN: 2150-8097. DOI: 10.14778/3424573.3424575.

[180] Dixin Tang and Aaron J. Elmore. "Toward Coordination-free and Reconfigurable Mixed Concurrency Control." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/tang.

[181] *Threading Building Blocks (TBB)*. URL: https://www.threadingbuildingblocks.org/.

[182] Teradata. *Teradata Vantage™ - Database Introduction*. 2021. URL: https://docs.teradata.com/r/Teradata-VantageTM-Database-Introduction/July-2021.

[183] The PostgreSQL Global Development Group. *PostgreSQL 14 Released!* 2022. URL: https://www.postgresql.org/about/news/postgresql-14-released-2318/.

[184] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 14: Chapter 27. High Availability, Load Balancing, and Replication, 27.1. Comparison of Different Solutions*. 2022. URL: https://www.postgresql.org/docs/14/different-replication-solutions.html.

[185] The Transaction Processing Council. *TPC-C Benchmark (Revision 5.11)*. 2021. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[186] Peter Thoman et al. "A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing." In: *The Journal of Supercomputing* 74.4 (2018). ISSN: 1573-0484. DOI: 10.1007/s11227-018-2238-4.

[187] Michael Freitag Thomas Neumann. "Umbra: A Disk-Based System with In-Memory Performance." In: *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*. 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[188] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. "Contention-aware Lock Scheduling for Transactional Databases." In: *Proceedings of the VLDB Endowment* 11.5 (2018). ISSN: 21508097. DOI: 10.1145/3187009.3177740.

[189] Mike Travis. *x86/platform/UV: Add check of TSC state set by UV BIOS*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=97d21003df3e7504c899b1701546f18ff475966f.

[190] Mike Travis. *x86/tsc: Add option that TSC on Socket 0 being non-zero is valid*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=341102c3ef29c33611586072363cf9982a8bdb77.

[191] Mike Travis. *x86/tsc: Provide a means to disable TSC ART*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6c66350d0a482892793b888b07c1177fc6d4b344.

[192] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. "Speedy Transactions in Multicore In-Memory Databases." In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 2013. DOI: 10.1145/2517349.2522713.

[193] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. "DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes." In: *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. 2000. DOI: 10.1109/ICDE.2000.839397.

[194] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases." In: *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. ACM, 2017. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3056101.

[195] Scott Vetter, Alexandre Bicas Caldeira, YoungHoon Cho, James Cruickshank, Bartłomiej Grabowski, Volker Haug, Andrew Laidlaw, and Seulgi Yoppy Sung. *IBM Power Systems E870 and E880 Technical Overview and Introduction*. IBM Redbooks, 2017. ISBN: 9780738454016. URL: http://www.redbooks.ibm.com/abstracts/redp5137.html?Open.

[196] Scott Vetter, James Cruickshank, Volker Haug, Yongsheng Li (Victor), and Armin Röll. *IBM Power Systems E980: Technical Overview and Introduction*. IBM Redbooks, 2020. ISBN: 9780738457123. URL: http://www.redbooks.ibm.com/abstracts/redp5510.html?Open.

[197] Robert Virding, Claes Wikström, Mike Williams, and Joe Armstrong. *Concurrent Programming in ERLANG (2nd Ed.)* Prentice Hall International (UK) Ltd., 1996. ISBN: 013508301X.

[198] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. *Intel Memory Latency Checker v3.8*. 2020. URL: https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[199] Donghui Wang, Peng Cai, Weining Qian, and Aoying Zhou. "Discriminative Admission Control for Shared-everything Database under Mixed OLTP Workloads." In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021. ISBN: 978-1-72819-184-3. DOI: 10.1109/ICDE51399.2021.00073.

[200] Jixin Wang, Jinwei Guo, Huan Zhou, Peng Cai, and Weining Qian. "Adaptive Transaction Scheduling for Highly Contended Workloads." In: *Database Systems for Advanced Applications*. Ed. by Guoliang Li, Jun Yang, Joao Gama, Juggapong Natwichai, and Yongxin Tong. Springer International Publishing, 2019. ISBN: 978-3-030-18590-9. DOI: 10.1007/978-3-030-18590-9_90.

[201] Tianzheng Wang and Hideaki Kimura. "Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores." In: *Proc. VLDB Endow.* 10.2 (2016). ISSN: 2150-8097. DOI: 10.14778/3015274.3015276.

[202] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. "Building a Bw-Tree Takes More Than Just Buzz Words." In: *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. ACM, 2018. DOI: 10.1145/3183713.3196895.

[203] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. "On the Calculation of Optimality Ranges for Relational Query Execution Plans." In: *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. ACM, 2018. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3183742.

[204] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An Empirical Evaluation of In-memory Multi-version Concurrency Control." In: *Proceedings of the VLDB Endowment* 10.7 (2017). ISSN: 2150-8097. DOI: 10.14778/3067421.3067427.

[205] Zhengming Yi, Yiping Yao, and Kai Chen. "FTSD: A Fissionable Lock for Multicores." In: *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 2021. ISBN: 978-1-4503-8698-2. DOI: 10.1145/3476886.3477518.

[206] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores." In: *Proc. VLDB Endow.* 8.3 (2014). ISSN: 2150-8097. DOI: 10.14778/2735508.2735511.

[207] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. *DBx1000 Github (Commit: b40c09a)*. 2020. URL: https://github.com/yxymit/DBx1000/tree/b40c09a27d9ab7a4c2222e0ed0736a0cb67b7040.

[208] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. "TicToc: Time Traveling Optimistic Concurrency Control." In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. New York, NY, USA: ACM, 2016. DOI: 10.1145/2882903.2882935.

[209] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. "Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System." In: *Proceedings of the VLDB Endowment* 11.10 (2018). ISSN: 2150-8097. DOI: 10.14778/3231751.3231763.

[210] Tieying Zhang, Anthony Tomasic, Yangjun Sheng, and Andrew Pavlo. "Performance of OLTP via Intelligent Scheduling." In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018. DOI: 10.1109/ICDE.2018.00132.

[211] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. "A Learned Query Rewrite System Using Monte Carlo Tree Search." In: *Proc. VLDB Endow.* 15.1 (2021). ISSN: 2150-8097. DOI: 10.14778/3485450.3485456.