

High Performance Transaction Processing on Non-Uniform Hardware Topologies

THÈSE N° 7023 (2016)

PRÉSENTÉE LE 7 JUILLET 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÉES MASSIVES

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Danica POROBIC

acceptée sur proposition du jury:

Prof. E. Bugnion, président du jury

Prof. A. Ailamaki, directrice de thèse

Prof. G. Alonso, rapporteur

Dr G. Swart, rapporteur

Prof. R. Guerraoui, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Curiosity is the lust of the mind.
— Thomas Hobbes

*To my family
Nada, Milan and Nikola*

Acknowledgements

This manuscript is signed with one name, but the pursuit of knowledge is never the work of one person. My PhD journey was no different and through it I was very fortunate to be surrounded by many amazing people to whom I owe a debt of gratitude.

First and foremost, I would like to thank my awesome advisor, Natassa Ailamaki. Throughout my PhD that often felt like a roller coaster ride, she was always there to tell me what I needed to hear. For years, I walked away from every one of our meetings energized and inspired, usually with a long TODO list. I really appreciate all the opportunities she gave me during my PhD and especially the encouragement to always aim higher. She is a truly inspiring researcher, teacher and leader, and I hope to continue learning from her in the future.

I am very grateful to Rachid Guerraoui, Gustavo Alonso and Garret Swart for finding the time to serve on my thesis committee. They have greatly improved this work through many insightful discussions and constructive feedback. I would also like to thank Ed Bugnion for accepting the role of my thesis jury president. In my opinion, one of the biggest strengths of EPFL is a very diverse and thriving community of systems researchers that supports many fruitful interactions among its faculty and students as well as with industry through many events that allow us to present our research and get feedback. I am also very grateful to Eric Sedlar and the Oracle Labs researchers for many insightful discussions over the years, especially during my internship at Oracle Labs. That summer convinced me that my PhD years were not wasted and restored my confidence in my research. Finally, I would like to thank Wolfgang Lehner and his team from TU Dresden for the fruitful collaboration on the energy efficiency project.

I have been very fortunate to be a member of the DIAS team together with: Adrian, Angelos, Ben, Cesar, Darius, Dimitra, Eleni, Erietta, Farhan, Georgios, Ippokratis, Iraklis, Lionel, Manos, Manos, Matt, Miguel, Mira, Odysseas, Pinar, Radu, Raja, Renata, Satya, Snow, Stella, Tahir, Thomas, Utku, Yannis, and many students and interns. They are the most amazing group of people who are always there to provide support and feedback, brainstorm about research ideas (or pretty much anything else) and be perfect travel companions. Renata has been a source of invaluable support from the time we both started at EPFL as we tackled in parallel all steps of our academic journeys. Pinar has been a great office mate for many years as well as my big little academic sister, great collaborator and conference buddy, as well as a perfect host during my recent trips to the Bay Area. I had a pleasure to learn a lot early in my thesis from

Acknowledgements

Ippokratis who kept giving me great feedback and looking out for me during the later stages of my career. Erietta made me question all assumptions about the problem we are solving and taught me a lot about doing rigorous research. I also learned a lot from my collaboration with Miguel and have enjoyed many insightful discussions with him that always inspired me to think bigger. I really enjoyed many discussions about different systems topics with Raja who always has great ideas. I had a pleasure to collaborate with and be a mentor to Iraklis and Utku. I would like to thank Utku for being very patient with my attempts at mentoring and for being a great office mate. I am very grateful to Mira and Manos for listening to my daily complaints and always having great and useful feedback. I owe special thanks to Erika and Dimitra for having answers for all administrative questions and for ensuring that lab functions smoothly. Finally, I thank Dimitra and Lionel for translating my thesis abstract to French.

The life in Lausanne would be far less enjoyable if I didn't have so many great friends. I was very fortunate to always have great neighbors. When I came to Lausanne, Bilja, Nataša, Laza and Zlatko welcomed me as a part of their household and made the first few months much less stressful. Sean and Giorgia tipped me off to a place in their building and were the perfect neighbors. When Maja and Petar moved in across the street, we spent countless enjoyable hours on their sofa, balcony and in the kitchen. Marija has been my go-to "sports buddy" and we enjoyed many hours skiing, rollerblading and ice skating. Even though our "social calendar" appeared crowded at times, I really enjoyed all the birthdays, barbecues, ski days, concerts and parties with the awesome ex-yu crew: Adrian, Aleksandar, Ana, Andrej, Azra, Baki, Bilja, Darko, Dražen, Edit, Ilija, Irena, Ivan, Laza, Lenka, Maja, Maja, Mara, Marija, Mia, Mica, Milenko, Milica, Miloš, Miloš, Mima, Mira, Mirko, Nataša, Pedja, Pedja, Petar, Renata, Sara, Sonja, Sloba, Stanko, Vojin, Zlatko, and many others. I will never forget the memorable weekends in Amsterdam, Madrid, Milano, Paris, and on the shores of Lago Maggiore.

Finally, I am eternally grateful to my parents, Nada and Milan, for their infinite patience and support throughout my education. From an early age, they have encouraged my curiosity, and supported my academic pursuits and career ambitions, even when they didn't agree with my choices. My brother Nikola has been my biggest cheerleader who believes his sister can accomplish anything. Finally, I am blessed to have the support of my uncle Miroslav, aunt Gordana and her family, and my late grandparents, who are always there for me.

This research has been supported by grants from Dotation (DIAS Lab), Oracle Labs, and the Swiss National Science Foundation (Grant No. 200021-146407/1).

Lausanne, May 2016

Danica Porobic

Abstract

Transaction processing is a mission critical enterprise application that runs on high-end servers. Traditionally, transaction processing systems have been designed for uniform core-to-core communication latencies. In the past decade, with the emergence of multisolet multicores, for the first time we have *Islands*, i.e., groups of cores that communicate fast among themselves and slower with other groups. In current mainstream servers, each multicore processor corresponds to an Island. As the number of cores on a chip increases, however, we expect that multiple Islands will form within a single processor in the nearby future. In addition, the access latencies to the local memory and to the memory of another server over fast interconnect are converging, thus creating a hierarchy of Islands within a group of servers.

Non-uniform hardware topologies pose a significant challenge to the scalability and the predictability of performance of transaction processing systems. Distributed transaction processing systems can alleviate this problem; however, no single deployment configuration is optimal for all workloads and hardware topologies. In order to fully utilize the available processing power, a transaction processing system needs to adapt to the underlying hardware topology and tune its configuration to the current workload. More specifically, the system should be able to detect any changes to the workload and hardware topology, and adapt accordingly without disrupting the processing.

In this thesis, we first systematically quantify the impact of hardware Islands on deployment configurations of distributed transaction processing systems. We show that none of these configurations is optimal for all workloads, and the choice of the optimal configuration depends on the combination of the workload and hardware topology. In the cluster setting, on the other hand, the choice of optimal configuration additionally depends on the properties of the communication channel between the servers. We address this challenge by designing a dynamic shared-everything system that adapts its data structures automatically to hardware Islands. To ensure good performance in the presence of shifting workload patterns, we use a lightweight partitioning and placement mechanism to balance the load and minimize the synchronization overheads across Islands.

Overall, we show that masking the non-uniformity of inter-core communication is critical for achieving predictably high performance for latency-sensitive applications, such as trans-

Abstract

action processing. With clusters of a handful of multicore chips with large main memories replacing high-end many-socket servers, the deployment rules of thumb identified in our analysis have a potential to significantly reduce the synchronization and communication costs of transaction processing. As workloads become more dynamic and diverse, while still running on partitioned infrastructure, the lightweight monitoring and adaptive repartitioning mechanisms proposed in this thesis will be applicable to a wide range of designs for which traditional offline schemes are impractical.

Keywords: Database management systems, Transaction processing systems, Multisocket multicore hardware, Hardware Islands, Non-uniform hardware topologies, Distributed transaction processing systems

Résumé

Le traitement de transactions est une application d'entreprise critique qui fonctionne sur des serveurs haut de gamme. Traditionnellement, les systèmes de traitement de transactions ont été conçus pour des latences de communication core-to-core uniformes. Ces dix dernières années, avec l'émergence de systèmes multi-sockets multi-cores, pour la première fois nous avons des Îles (Islands), à savoir, des groupes de cores qui communiquent rapidement entre eux et plus lentement avec d'autres groupes. Dans les serveurs traditionnels actuels, chaque processeur multi-cores correspond à une Île. Avec le nombre de cores par puce qui augmente, nous nous attendons à ce que plusieurs Îles soient formées au sein d'un seul processeur dans un futur proche. En outre, les latences d'accès à la mémoire locale et à la mémoire d'un autre serveur sur interconnexion rapide convergent, créant ainsi une hiérarchie d'Îles au sein d'un groupe de serveurs connectés sur un réseau à grande vitesse.

Les topologies matérielles non uniformes constituent un défi important pour l'évolutivité et la prévisibilité des performances des systèmes de traitement de transactions. Les systèmes de traitement de transactions distribués peuvent atténuer ce problème ; cependant, aucune configuration de déploiement n'est optimale pour toutes les charges de travail et les topologies matérielles. Afin d'utiliser pleinement la puissance de traitement disponible, un système de traitement de transactions doit s'adapter à la topologie du matériel sous-jacent et doit adapter sa configuration à la charge de travail courante. Autrement dit, le système devrait être capable de détecter toute modification de la charge de travail et de la topologie du matériel, et s'adapter sans perturber le traitement.

Dans cette thèse, premièrement nous quantifions systématiquement l'impact des latences de communication non uniformes sur les configurations de déploiement de systèmes de traitement de transactions distribués. Nous montrons qu'aucune de ces configurations n'est optimale pour toutes les charges de travail, et le choix de la configuration optimale dépend de la combinaison de la charge de travail et de la topologie matérielle. Dans le cadre d'un cluster, le choix de la configuration optimale dépend en plus des propriétés du mécanisme de communication entre les serveurs. Nous abordons ce défi en concevant un système dynamique shared-everything qui adapte automatiquement ses structures de données aux Îles matérielles. Pour assurer une bonne performance en présence de charges de travail changeantes, nous utilisons un mécanisme de partitionnement et de placement léger pour équilibrer la charge et réduire l'overhead de synchronisation à travers les Îles.

Résumé

Dans l'ensemble, nous montrons que le masquage de la non-uniformité de la communication inter-core est essentiel pour l'obtention prévisible de hautes performances pour les applications sensibles à la latence, tels que le traitement de transactions. Avec des clusters composés d'une poignée de systèmes comportant des puces multi-core avec de grandes mémoires principales qui remplacent les serveurs haut de gamme, embarquant de nombreux processeurs, les principes de déploiement, identifiées dans notre analyse ont le potentiel de réduire considérablement les coûts de synchronisation et de communication lors du traitement des transactions. Comme les charges de travail deviennent plus dynamiques et diversifiées, tout en fonctionnant sur une infrastructure partitionnée, le monitoring léger et les mécanismes de repartitionnement adaptatifs proposés dans cette thèse seront applicables à un large éventail de modèles pour lesquels les systèmes hors ligne traditionnels sont inadéquats.

Mots clefs : Système de gestion de base de données, Les systèmes de traitement de transactions, Systèmes multi-sockets multi-cores, Îles matérielles, Les topologies matérielles non uniformes, Les systèmes de traitement de transactions distribués

Contents

Acknowledgements	i
Abstract (English/Français)	iii
Table of Contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Modern Servers	2
1.2 Rack-Scale Computing Platforms	3
1.3 Transaction Processing	3
1.4 Thesis Statement and Contributions	4
1.5 Thesis Roadmap	6
2 Background	9
2.1 Online Transaction Processing Systems	9
2.2 Hardware Islands	12
2.2.1 Variable Communication Latencies	13
2.2.2 Abundant Hardware Parallelism	14
2.2.3 NUMA-aware System Design	16
2.2.4 Beyond Hardware Islands	17
2.3 Rack-scale Computing Platforms	18
3 OLTP on Hardware Islands	23
3.1 Islands: Hardware Topology-aware Shared-nothing OLTP Deployments	24
3.2 Experimental Setup and Methodology	26
3.2.1 Prototype Systems	28
3.2.2 Microbenchmark Workload and Experimental Methodology	29
3.2.3 Standard Workloads	31
3.3 Impact of Multisite Transactions	32
3.3.1 Distributed Transactions	33
3.3.2 Microarchitectural Behavior	35

Contents

3.3.3	Profiling	36
3.3.4	Impact of the Communication Channel	37
3.3.5	Different Topologies	38
3.3.6	Summary	40
3.4	Sensitivity Analysis with Microbenchmarks	40
3.4.1	Impact of the Size of Transaction	40
3.4.2	Increasing Hardware Parallelism	42
3.4.3	Tolerance to Skew	44
3.4.4	Increasing Database Size	45
3.4.5	Summary	46
3.5	Standard Workloads	46
3.5.1	TPC-B	47
3.5.2	Impact of Distributed Transactions on TPC-B	48
3.5.3	TPC-C	48
3.5.4	Impact of Distributed Transactions on TPC-C	49
3.5.5	Summary	50
3.6	Main-memory optimized system	50
3.6.1	Read-only Transactions	51
3.6.2	Update Transactions	53
3.6.3	Summary	53
3.7	Summary and discussion	53
4	Adaptive Transaction Processing	55
4.1	Design Trade-offs for OLTP on Multisocket Multicores	56
4.1.1	Design Options	56
4.1.2	Perfectly Partitionable Workloads	57
4.1.3	Workloads That Are Less Amenable to Partitioning	58
4.1.4	Accessing Remote Memory	60
4.2	Hardware-aware System Design	60
4.3	Workload-aware Partitioning and Placement	62
4.3.1	Factors Influencing Transaction Processing	62
4.3.2	Cost Model	66
4.3.3	Search Strategy	67
4.4	Adaptive Dynamic OLTP Design	68
4.5	Evaluation	70
4.5.1	Experimental Setup	71
4.5.2	Improving Throughput on Standard Benchmarks with ATrapos	71
4.5.3	Monitoring and Repartitioning Cost	73
4.5.4	Adaptive Behavior of ATrapos	74
4.6	Summary and Discussion	77

5	Toward Rack-scale OLTP	79
5.1	Setup and Methodology	80
5.1.1	Distributed OLTP Deployments	80
5.1.2	Hardware Platforms	80
5.1.3	Workloads	80
5.2	Scaling Out Across Rack-scale Nodes	81
5.2.1	Distributed Main Memory System	81
5.2.2	Read-only Transactions	81
5.2.3	Update Transactions	82
5.2.4	Sensitivity Analysis	82
5.2.5	Summary and Implications	83
5.3	Scaling Up on Rack-scale Nodes	84
5.3.1	Distributed Deployment Configuration	84
5.3.2	Scaling TPC-C Across Machines	85
5.3.3	Impact of Thread Binding	86
5.3.4	Sensitivity Analysis	86
5.3.5	Summary and Implications	88
5.4	The Impact of Network	88
5.4.1	Main Memory Optimized System	88
5.4.2	TPC-C Transactions	90
5.4.3	Microbenchmarks	91
5.4.4	Summary and Implications	92
5.5	A Step Toward Rack-scale OLTP	93
6	The Big Picture	97
6.1	What We Did	97
6.2	Impact	98
6.3	Looking Ahead	99
Bibliography		113
Curriculum Vitae		115

List of Figures

1.1	Communication latencies in a multsocket multicore	2
2.1	Impact of thread placement on performance for microbenchmark workload.	13
2.2	Impact of thread placement on performance of TPC-C workload.	14
2.3	Impact of the granularity of synchronization for microbenchmark workload.	15
2.4	Impact of the granularity of synchronization for TPC-C workload.	15
3.1	Different shared-nothing configurations on a four-socket four-core machine.	24
3.2	Islands performance model.	26
3.3	Topology of the three machines used in the experiments.	26
3.4	Performance of different IPC mechanisms.	28
3.5	Examples of microbenchmark transactions.	30
3.6	Main deployment configurations illustrated on the 4 socket server.	32
3.7	Microbenchmark: varying percentage of multisite transactions.	34
3.8	Microarchitectural profiling of cache and memory resident datasets.	35
3.9	Time breakdowns for cache and memory resident datasets.	36
3.10	Impact of fast communication on multisite transactions.	37
3.11	Time breakdowns for fast communication.	38
3.12	Impact of hardware topology on multisite transactions.	39
3.13	Cost of local and multisite read-only transactions.	41
3.14	Cost of local and multisite update transactions.	42
3.15	Scalability of different configurations as parallelism increases.	43
3.16	The impact of skew on multisite transactions.	44
3.17	The impact of disk resident datasets on multisite transactions.	45
3.18	Performance of standard and local-only deployment of the TPC-B benchmark.	46
3.19	TPC-B: varying percentage of multisite transactions.	48
3.20	Performance of standard and local-only deployments of TPC-C benchmarks.	49
3.21	TPC-C: varying percentage of multisite transactions.	50
3.22	Microbenchmark: performance of the main memory system	50
3.23	Time breakdowns for the main memory system.	51
3.24	Cost of microbenchmark transactions in the main memory system.	52
3.25	The impact of update multisite transactions on the main memory system	52
4.1	Instructions retired per cycle.	57

List of Figures

4.2	Throughput of the shared-nothing, centralized, and PLP architectures.	57
4.3	Throughput of different partitioned configurations.	58
4.4	Time breakdown for Island shared-nothing configuration.	59
4.5	Hardware-awareness improves scalability on Islands.	62
4.6	Simple transaction flow example.	63
4.7	Comparison of different partitioning and placement strategies.	64
4.8	Transaction flow graph for the TPC-C NewOrder transaction.	65
4.9	Adaptivity mechanism in ATraPos.	69
4.10	ATraPos partitioning and placement for TATP	71
4.11	ATraPos partitioning and placement for TPC-C	72
4.12	Scalability of ATraPos repartitioning mechanism.	73
4.13	Adapting to workload changes.	74
4.14	Adapting to sudden workload skew.	75
4.15	Adapting to hardware failures.	76
4.16	Adapting to frequent changes.	76
5.1	Impact of distributed transactions on Silo.	81
5.2	Abort rates of distributed transactions in Silo.	82
5.3	Increasing the number of servers for TPC-C transactions.	84
5.4	The impact of thread binding on cluster deployments.	85
5.5	Costs of distributed read-only transactions in cluster OLTP	86
5.6	Costs of distributed update transactions in cluster OLTP	87
5.7	Impact of network channel on read-only workload in Silo	89
5.8	Impact of network channel on update workload in Silo	89
5.9	Impact of network channel on TPC-C workload.	90
5.10	Impact of network channel on the read-only workload.	91
5.11	Impact of network channel on the update workload.	92

List of Tables

3.1	Description of the machines used.	27
4.1	Impact of memory allocation policy.	59
4.2	Overhead of ATrapos monitoring.	73

1 Introduction

Online Transaction Processing (OLTP) is a multi-billion dollar industry [51] and one of the most important and demanding database applications. Innovations in OLTP continue to garner significant attention, advocated by the recent emergence of appliances [116], startups [29, 100, 105, 112, 160], hosted cloud solutions [10], and research projects (e.g. [30, 71, 78, 81, 90, 118, 144, 157]). OLTP applications are mission-critical for many enterprises with little margin for compromising either performance or scalability. Thus, it is not surprising that all major OLTP vendors invest considerable effort in developing highly-optimized software releases, often with platform-specific optimizations.

OLTP workloads are characterized by many concurrent requests. Each transactional request typically reads about a dozen and writes a handful of data items in the database. The users of the system expect predictably low response times and high availability regardless of the degree of concurrency or the size of data. Increasing throughput and decreasing latency requirements, as well as the high cost of licenses for traditional database management systems when deployed in the web-scale scenarios gave rise to eventually consistent key-value stores [159]. Nowadays, OLTP infrastructure at Facebook processes 174 million transactions per second to service a wide variety of applications, including the main Facebook website, messaging and serving advertisements [25]. These transactions generate 12 billion reads and 65 million updates per second.

Increasing demands from the new generation of applications for both larger data sizes and higher throughput require ever more scalable transaction processing system designs. While key-value stores offered low latency and good horizontal scalability, they shift the burden of ensuring *ACID* (atomicity, consistency, isolation, and durability) properties traditionally provided by an OLTP system to the application developers, significantly increasing the application complexity. Thus, it is not surprising that the web-scale companies have reversed their position in recent years and invested heavily in developing highly scalable geo-distributed transaction processing systems such as Google's Spanner [30].

The rest of this chapter briefly motivates this work by first providing an overview of the changing hardware landscape and the rise of non-uniform hardware platforms before surveying recent work on transaction processing systems. Next, [Section 1.4](#) summarized the thesis statement and outlines intellectual and technological contributions. Finally, [Section 1.5](#) outlines the organization of this dissertation.

1.1 Modern Servers

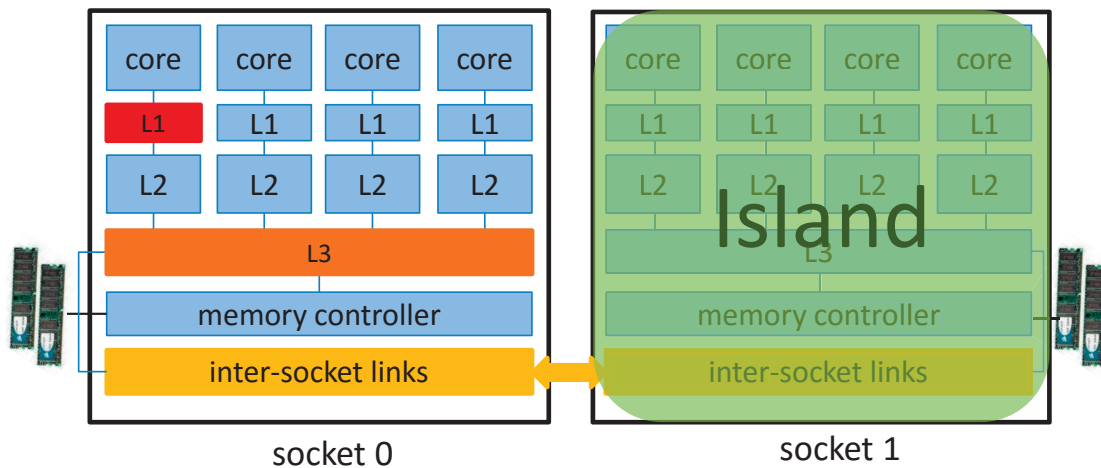


Figure 1.1: A schematic view of a multisocket multicore server. We identify a hardware *Island*: a group of cores that communicate faster with each other than with the cores from another island.

Hardware has long departed from uniprocessors, which have predictable and uniform performance. Due to thermal and power limitations, vendors cannot improve the performance of processors by clocking them to higher frequencies or by using more advanced techniques such as increased instruction width and extended out-of-order execution. Instead, vendors rely on two approaches that allow explicit parallelization of tasks to increase the processing capability of a machine. The first approach is to put together multiple processor chips that communicate through shared main memory. For several decades, such *multisocket* designs provided the only way to scale performance within a single node and the majority of OLTP systems have historically used such hardware. The second approach places multiple processing cores on a single chip, such that each core is capable of processing concurrently several independent instruction streams or hardware contexts. The communication between cores in these *multicore* processors happens through on-chip caches. In recent years, multicore processors have become a commodity.

Multisocket multicore systems are the predominant configuration for database servers today and are expected to remain popular in the future. [Figure 1.1](#) shows a simplified diagram of a typical machine that has two sockets with quad-core CPUs. Communication between the numerous cores happens through different mechanisms. For example, two threads running on

the same core can communicate very fast through the core's L1 cache. When they're running on different cores on the same socket, they communicate through the socket's last-level (L3) cache. Finally, two threads running on different sockets need to use inter-socket links (called *QPI* for Intel processors). The result is that the inter-core communication depends on the placement of communicating threads.

1.2 Rack-Scale Computing Platforms

In parallel with the increase in the number of processor cores, the bandwidth of network interconnects and main memory are converging [46]. Technologies such as Remote Direct Memory Access (RDMA) allow applications to access memory on a remote machine without involving either the operating system or the processor. High speed fabrics that support RDMA, such as Infiniband [63] and converged Ethernet [62], with bandwidths reaching up to 100Gbps, are already standard in supercomputers and high-end appliances.

Modern RDMA technology enables accessing the memory of another server within an order of magnitude of the latency to the local memory [156]. Relative remote access latency using RDMA is comparable to one in the traditional large shared memory machines, whose niche market are the mission-critical OLTP applications [60, 137]. Such high-end machines with more than 8 processors require specialized interconnects and controller chips to maintain cache coherence [53, 92]. However, maintaining cache coherence comes with high overheads. In the state-of-the-art systems, remote memory access latencies are 5.5x to 10.7x higher than the local memory ones [83].

Abundant parallelism and fast commodity networks are leading to the emerging class of commodity cluster computing platforms that will offer performance comparable to today's large shared memory machines. These *rack-scale platforms* achieve high compute density at low power budget and cost by eliminating unnecessary system components via highly customized system-on-a-chip (SoC) nodes. Individual nodes, which contain only processing cores, memory, and I/O interfaces, communicate using low-latency interconnect fabrics [38].

1.3 Transaction Processing

Efficiently utilizing the processing power available on modern hardware platforms for transaction processing remains a challenge despite the multitude of efforts from academia and industry. Until recently, shared-everything was the most popular deployment strategy on a single node, and OLTP has been studied extensively on shared-everything databases. It has been shown that shared-everything systems exhibit frequent shared read-write accesses [18, 54], which are difficult to predict [142]. Therefore, these systems enter numerous contentious critical sections even when executing simple transactions to ensure ACID properties [73]. The presence of critical sections affects single-thread performance, requires frequent inter-core communication, and causes contention among threads [71, 118, 119]. Recent work

suggests a departure from the traditional transaction-oriented execution model, to adopt a data-oriented one, circumventing the aforementioned properties - and flaws - of traditional shared-everything OLTP [118, 119].

Shared-nothing deployments [143], based on fully independent (physically partitioned) database instances that collectively process the workload, are an increasingly appealing design even within a single node [78, 136, 144]. The main advantage of shared-nothing deployments is the explicit control over the contention within each physical database instance. As a result, shared-nothing systems exhibit high single-thread performance and low contention. In addition, shared-nothing databases typically make better use of the available hardware resources whenever the workload contains transactions touching data on a single database instance. Systems such as H-Store [144] and HyPer [78] apply the shared-nothing design to the extreme, deploying one single-threaded database instance per CPU core.

Shared-nothing systems appear ideal from the hardware utilization perspective, but they are sensitive to the ability to partition the workload. Unfortunately, many workloads are not perfectly partitionable, i.e., it is hardly possible to distribute the data such that every transaction touches a single instance. Whenever multiple instances must collectively process a request, shared-nothing databases require using expensive distributed consensus protocols, such as two-phase commit, which many argue are inherently non-scalable [23, 58]. Similarly, handling data and access skew is problematic [151].

The overhead of distributed transactions urged system designers to explore partitioning techniques that reduce the frequency of distributed transactions [32, 121], and to explore alternative concurrency control mechanisms, such as speculative locking [74], multiversioning [19] and optimistic concurrency control (OCC) [87, 90], to reduce the overheads when distributed transactions cannot be avoided. Designers of large-scale systems have circumvented problems with distributed transactions by using relaxed consistency models such as eventual consistency [159]. Eventual consistency eliminates the need for synchronous distributed transactions, but it makes programming transactional applications harder, with consistency checks left to the application layer. The emergence of the non-uniform hardware platforms adds further complexity to the on-going debate between shared-everything and shared-nothing OLTP designs.

1.4 Thesis Statement and Contributions

The goal of this dissertation is to quantify the challenges posed by the non-uniformity of communication between different cores within a multsocket and across multsockets to transaction processing systems. We identify opportunities and propose techniques to address these challenges by dynamically tuning transaction processing systems to the underlying hardware and current workload properties.

Thesis Statement

Transaction processing scalability deteriorates as the number of cores is increased, due to real-time changes in workloads and to non-uniform communication latencies amongst hardware contexts. Using hardware-aware data structures and algorithms inside transaction processing engines is key to sustaining scale-up throughput during workload execution. Scale-up techniques are necessary but insufficient for scale-out deployments.

We analyze the impact of non-uniformity on transaction processing systems for a wide range of deployment configurations and workloads. We advocate for nimble system designs that dynamically adapt to the workloads and hardware based on the following insights:

- We identify *Hardware Islands* as the groups of cores that communicate faster with cores that belong to the same group and much slower with cores from other groups. We can use the *Islands* concept to model the horizontal non-uniformity in communication latencies in multsocket multicore servers and analyze its impact on transaction processing systems.
- Through experimental study, we show that the fine-grained shared-nothing deployments achieve significantly higher throughput than the shared-everything ones on a multsocket multicore when the workload is perfectly partitionable. By contrast, when the workload is not partitionable and/or exhibits skew, a shared-everything deployment has higher performance than a shared-nothing one. Therefore, there is no unique optimal deployment strategy for all workloads. In addition, when the workload or hardware topology change, switching to another configuration requires expensive physical repartitioning.
- We demonstrate that scalable transaction processing systems for multsocket servers must avoid modifying any centralized data structure in the critical path. With more cores that communicate less uniformly, any such access eventually becomes a bottleneck. We demonstrate that a shared-everything design can scale as well as a fine-grained shared-nothing one for perfectly partitionable workloads by using Island-aware data structures.
- We show that the scalable transaction processing designs for multicores face significant challenges when deployed in cluster environment due to messaging delays in the critical path of transaction execution. However, for the traditional system designs the messaging delays can be overlapped with other processing and the network is not the dominant factor impacting the throughput of different deployment configurations.

This thesis makes the following technical contributions:

- We quantify the impact of non-uniform core topology on the performance of transaction processing systems and conclude that high performance software has to minimize contention among cores and avoid frequent communication between cores located on different hardware islands. We provide and validate the Islands performance model where we express the performance of an OLTP system as a function of the deployment configuration and the percentage of multipartition transactions on a wide variety of workloads and hardware topologies. The particular cross-over points that make a specific configuration optimal differ depending on the particular scenario. The relative performance trends, however, remain the same in all cases.
- We propose ATraPos, an OLTP system design that is aware of the non-uniform access latencies of multsocket systems. On top of its Island-aware data structures, ATraPos adopts a lightweight monitoring and repartitioning mechanism that adapts the partitioning strategy upon workload changes. It relies on the hardware and workload-aware partitioning and placement scheme to achieve balanced load and maximize locality of communication in the critical path.
- We quantify the similarities and differences between OLTP deployments on multsocket multicores and clusters with fast interconnects. We show that different configurations are optimal for different combination of workload and cluster properties. Choosing the right granularity of instances is essential for taking advantage of the fast network, yet, careful placement of threads to cores within an instance can substantially improve throughput.

1.5 Thesis Roadmap

This section outlines the structure of the thesis and summarizes the next chapters.

- [Chapter 2](#) provides the background and motivation for this work. We introduce the traditional transaction processing system designs in the single node and distributed deployments. Then we quantify the basic properties of *Hardware Islands* using microbenchmarks, and discuss the major distinguishing characteristics of emerging rack-scale hardware platforms and compare them against other high performance server designs. Finally, we survey the related work.
- [Chapter 3](#) quantifies the impact of *Hardware Islands* on a variety of transaction processing workloads and distributed deployment configurations. We use both microbenchmarks and standard TPC benchmarks to explore different dimensions including the impact of distributed transactions, hardware topology, skew, as well as the number and type of operations performed within a transaction. We conclude that no single deployment configuration is optimal for all scenarios and that the best configuration depends on the hardware topology and workload characteristics.

- [Chapter 4](#) presents *ATraPos*, a scalable shared-everything system design that minimizes the impact of inter-socket communication in the critical path of transaction execution by utilizing data oriented transaction execution and hardware-aware data structures. In order to adapt to different workloads, *ATraPos* relies on *precise data partitioning and placement* to maximize locality of data accesses and on *adaptive repartitioning* to maintain data locality when the workload changes.
- [Chapter 5](#) expands the analysis to clusters of multisoquets connected with fast interconnects. We analyze different distributed deployments using standard and synthetic benchmarks that include distributed transactions to quantify the challenges and opportunities for OLTP designs on rack-scale platforms.
- Finally, [Chapter 6](#) summarized the findings and their impact, and concludes the thesis by discussing possible avenues of future work.

2 Background

This chapter starts with a brief overview of the basic properties of transactions and transaction processing system designs, including a survey of related work. Then we discuss properties of modern multisolet multicore servers and quantify basic trade-offs involved in thread synchronization on these platforms before surveying recent processor trends. We conclude with the discussion of the characteristics of fast interconnects and the data management systems that take advantage of them.

2.1 Online Transaction Processing Systems

According to the popular database systems textbook by Ramakrishnan and Gehrke [130]: "A **transaction** is *any one execution* of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS." Each database transaction satisfies the following set of properties (commonly demoted as *ACID*):

- **Atomicity:** Execution of each transaction is atomic: either all operations are executed and visible to other transactions or none are.
- **Consistency:** Every transaction run by itself must preserve consistency of the database. Ensuring the consistency is the responsibility of the user while the system ensures that this property is preserved in the presence of concurrent transactions.
- **Isolation:** Execution of concurrent transactions is isolated, i.e., the effects of incomplete transactions are not visible to other in-progress transactions.
- **Durability:** Once a transactions completes successfully, its effects are persistent in the database.

In traditional database management systems, ensuring ACID properties is done in the storage manager that typically comprises the following four components [59]: the lock manager in

charge of concurrency control, the log manager in charge of recovery, the buffer pool in charge of caching the data pages in memory, and the access methods used for accessing data stored on the data pages. Concurrency control methods ensure consistency and isolation of transactions, while recovery manager guarantees atomicity and durability. In a traditional storage manager all of these components are centralized and accessed by a worker thread in the critical path of transaction execution. This leads to scalability issues on multicores as threads enter numerous contentious critical sections, e.g., a transaction that updates a single data item requires execution of over 70 critical sections [73].

Shared-everything OLTP. Within a database node, *shared-everything* is any deployment where a single database instance manages all the available resources. As database management systems have long been designed to operate on machines with multiple processors, shared-everything deployments assume equally fast communication between all processing cores, since each thread needs to exchange data with all of its peers. Until recently, shared-everything was the most popular deployment strategy for scale-up transactions processing and it is used by all major commercial database systems. OLTP has been studied extensively on shared-everything databases. For instance, the workload characterization studies that analyze micro-architectural behavior of the OLTP workloads demonstrate that shared-everything systems exhibit significant stalls during execution [4, 15, 54, 152]; a result we corroborate in [Section 3.3.2](#). These systems enter numerous contentious critical sections even when executing simple transactions, affecting single-thread performance, requiring frequent inter-core communication, and causing contention among threads [73, 71]. These characteristics make distributed memories (as those of multisoquets), distributed caches (as those of multicores), and prefetchers ineffective.

Many recent techniques aim to improve scalability of individual components of traditional systems, including locking, latching and logging on multicores, by specializing synchronization primitives to a particular component [70, 72, 76, 82]. Alternative to the traditional transaction-oriented execution model is a data-oriented execution model, that circumvents the aforementioned properties - and flaws - of traditional shared-everything OLTP [83, 118, 119]. Another promising direction is taking advantage of the hardware transactional memory support available in recent Intel processors to implement efficient concurrency control [91].

The large main memories available in modern servers have sparked a lot of interest in the main-memory optimized transaction processing designs for multisoquets. Such systems have been marketed by major vendors for many years, including IBM solidDB [95] and Oracle TimesTen [89], however, only now are they becoming mainstream. Modern multicore optimized main-memory transaction processing systems, such as Hekaton, Silo, Foedus, and Ermia, use multi-versioned latch-free data structures and optimistic concurrency control mechanisms to achieve good scalability by reducing the number of critical sections and their duration [80, 81, 90, 93, 157]. Yet, a recent study shows that none of the current concurrency control mechanisms scales to 1000 cores and suggests that extending hardware support is a promising way for overcoming this obstacle [171, 172].

Scalability concerns in face of increasing parallelism gave rise to a new generation of main memory optimized transaction processing designs, many of which adopt single threaded shared nothing execution model. In the shared-nothing systems, data is partitioned among a number of instances that collectively serve transactional requests.

Shared-nothing OLTP. Shared-nothing deployments [143], based on fully independent (physically partitioned) database instances, are an increasingly appealing design even within a single node [78, 136, 144]. This is due to the scalability limitations of shared-everything systems, which suffer from contention when concurrent threads attempt to access shared resources [73]. The main advantage of shared-nothing deployments is the explicit control over the contention within each physical database instance. As a result, shared-nothing systems exhibit high single-thread performance and low contention for workload that can be partitioned. Systems such as H-Store [144] and HyPer [78] apply the shared-nothing design to the extreme, deploying one single-threaded database instance per CPU core. This enables simplifications or removal of expensive database components such as locking and latching. However, if a transaction requires data from different instances, it typically executes a distributed transaction that uses a coordination protocol.

Two phase commit. The standard coordination protocol for distributed transactions is the *two-phase commit (2PC)* [52, 104]. One instance in the system acts as a *coordinator* of a distributed transaction while the others are *participants*. During the first phase, the coordinator sends messages containing operation requests to the participants and receives replies containing results of requested operations. After completing all operations, the coordinator sends a *prepare* message to each participant which replies with a *vote* containing an outcome of its part of the transaction. The first phase ends when the coordinator collects all votes. Based on the votes, the coordinator decides whether to commit the transaction (if all participants voted *commit* or *read-only*) or abort it (if at least one participant voted *abort*). In the second phase, the coordinator sends the decision to all participating instances who complete the transaction fragments locally and send the acknowledgement. When the coordinator collects all acknowledgments, it completes the distributed transaction.

Even a small percentage of distributed transactions in the workload severely harms the scalability of systems like H-Store as distributed transactions effectively block all partitions involved in that transaction [74]. An alternative approach is taken by the Multimed project, which views the multsocket multicore system as a cluster of machines [136]. Multimed uses replication techniques and a middleware layer to split database instances into those that process read-only requests and those that process updates. A similar approach is used in HyPer to support OLTP and OLAP in the same system by executing analytical queries on the snapshots of the transactional database [78].

OLTP partitioning mechanisms. One can reduce the negative impact of distributed transactions by minimizing the overhead of distributed transactions either by predicting which distributed transactions are effectively local [121] or by finding a good partitioning scheme

for a specific workload. Schism proposes a graph-based partitioning and replication method for OLTP workloads [32]. The graph is constructed from transaction access traces such that vertices represent tuples and edges connect the tuples used in the same transaction. The partitions are selected using the min-cut algorithm. An extension of Schism, Sword [129], proposes a different graph compression approach that allows incremental data movement between two partitioning solutions for different workloads. Another approach for automatic partitioning in shared-nothing OLTP systems is Horticulture [122], which utilizes large neighborhood search (LHS). It uses the database schema, the code of the stored procedures, the workload trace consisting of data items that were accessed, and timestamps. The output of the partitioning strategy is a set of decisions whether to range or hash partition a table or replicate it to all nodes. All these techniques can generate good initial partitioning, however, they are off-line methods that cannot be used to adapt to the workload changes at runtime.

Repartitioning and load balancing. On the other hand, one of the recent proposals for adaptive repartitioning algorithms targets physiologically partitioned shared-everything systems [151]. The load on each partition is monitored using histograms and work queues. Whenever a load imbalance exceeds the threshold, data is repartitioned. Similar approach can be used in distributed OLTP systems that rely on physiological partitioning [138].

A related set of challenges arise in multitenant shared-nothing deployments in the distributed system setting where load balancing and efficient tenant placement is essential to meet service level objectives (SLOs). ElasTras is a pioneering project that provides elasticity in multi-tenant environment with efficient live migration (repartitioning) in case of load imbalances [33]. Accordion focuses on efficient partitioning placement and minimal data repartitioning cost using mixed integer linear programming for fine-grained shared-nothing OLTP system [139]. E-store is a similar effort aimed at elasticity and load balancing in dynamic fine-grained shared-nothing systems [146].

Stored procedures. Even though transactions in general can contain an arbitrary sequence of SQL statements, in practice, they fall into one of the predefined transaction types and are executed using parametrized stored procedures [144, 149]. Furthermore, large majority of transaction types are *one-shot*, i.e., based on the values of the input parameters, one can determine a set of data items accessed by a transaction. For one-shot transactions, executing the first phase of the 2PC protocol requires a single message exchange: the coordinator sends the prepare message containing the input parameters to the participant who replies with both the result of the execution and the vote. In this work, we primarily consider one-shot distributed transactions.

2.2 Hardware Islands

In step with Moore's Law, hardware provides increasing opportunities for parallelism rather than faster processors since 2005. Therefore, instead of increasing frequency, we observe an increase in the number of cores on a processor. In addition, multiple such processors

are usually placed in the same server, creating *hardware islands*. Hence, there are two main trends in modern server hardware: *the variability in communication latencies* and *the abundance of parallelism*. In the following two subsections we discuss how each trend affects the performance of software systems before surveying related work.

2.2.1 Variable Communication Latencies

The impact of modern processor memory hierarchies on the application performance is significant because it causes variability in access latency and bandwidth, making the overall software performance unpredictable. Furthermore, it is difficult to implement synchronization mechanisms that are globally optimal for different applications and multicores and multisoquets with different topologies [35]. Using learning techniques to choose the optimal synchronization mechanism for a specific use case is a promising way to alleviate this problem [40].

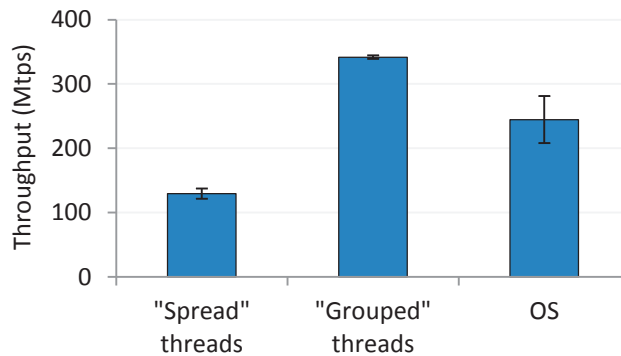


Figure 2.1: Results of a counter benchmark where groups of 10 threads increment a shared counter. Allocating threads and memory in a topology-aware manner provides the best performance and lowest variability.

We illustrate the impact of non-uniform topology on the efficiency of synchronization among threads with a simple microbenchmark. Figure 2.1 plots the throughput of a program running on a machine that has 8 CPUs with 10 cores each (the “Octo-socket” machine of Table 3.1). There are 80 threads in the program, divided into groups of 10 threads, where each group increments a counter protected by a lock in a tight loop. There are 8 counters in total, matching the number of sockets in the machine. We vary the allocation of the worker threads and plot the total throughput (million counter increments per second). The first bar (“*Spread*” threads) spreads worker threads across all sockets. The second bar (“*Grouped*” threads) allocates all threads in the same socket as the counter. The third bar lets the operating system do the thread allocation. Allocating threads and memory in a manner that maximizes locality results in the best performance and lowest variability. Leaving the allocation to the operating system leads to non-optimal results and higher variability. Although this has been an area of active research in recent years [17, 34], general purpose approaches do not work well for database systems due to their dynamic nature. Database-specific thread schedulers and interfaces that

enable the application to hint its requirements to the operating system are a very promising line of research [48].

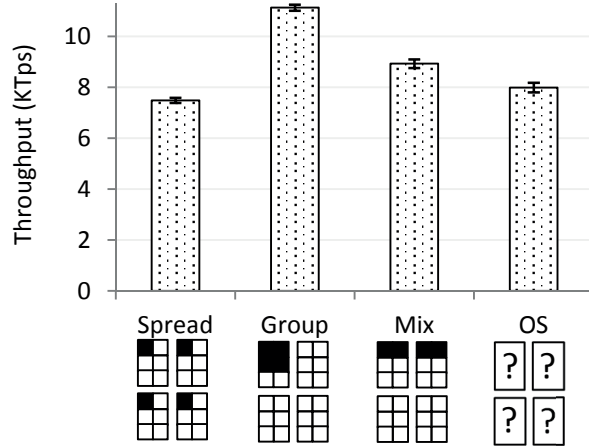


Figure 2.2: Throughput of the system when varying placement of 4 worker threads. Running the TPC-C-Payment workload with all threads on the same socket achieves 20-30% higher performance than other configurations.

We obtain similar results when running OLTP workloads. To demonstrate the impact of non-uniform communication latencies on OLTP, we run TPC-C Payment transactions on a machine that has 4 CPUs with 6 cores each (“Quad-socket” in Table 3.1). Figure 2.2 plots the average throughput and standard deviation across multiple executions on a database with 4 worker threads. In each configuration we vary the allocation of individual worker threads to cores. The first configuration (“Spread”) assigns each thread to a core in a different socket. The second configuration (“Group”) assigns all threads to the same socket. The configuration “Mix” assigns two cores per socket. In the “OS” configuration, we let the operating system do the scheduling. This experiment corroborates the previous observations of Figure 2.1: the OS does not optimally allocate work to cores, and a topology-aware configuration achieves 20-30% better performance and less variability. The absolute difference in performance is much lower than in the case of counter incrementing because executing a transaction has significant start-up and finish costs, and during transaction execution a large fraction of the time is spent on operations other than accessing data. For instance, studies show that around 20% of the total instructions executed during OLTP are data loads or stores (e.g., [15, 54]).

2.2.2 Abundant Hardware Parallelism

Another major trend is the abundant hardware parallelism available in modern database servers. Higher hardware parallelism potentially causes additional contention in multisocket multicore systems, as a higher number of cores compete for shared data accesses. Figure 2.3 plots the results obtained on the octo-socket machine when varying the number of worker threads accessing a set of counters, each protected by a lock. An exclusive counter per core

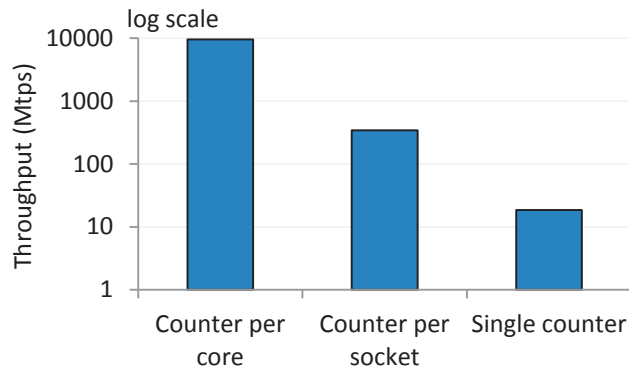


Figure 2.3: Results of a counter benchmark where we always use 80 threads and change the number of counters they increment. Improving locality of communication improves the performance by an order of magnitude.

achieves lower variability and 18x higher throughput than a counter per socket, and 517x higher throughput than a single counter for the entire machine. In both cases, this is a super-linear speedup. Shared-nothing deployments are better suited to handle contention, since they provide explicit control by physically partitioning data, leading to higher performance.

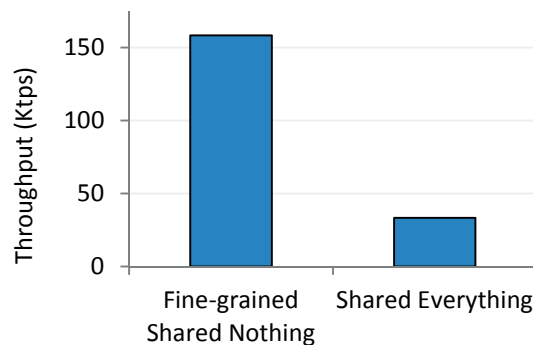


Figure 2.4: Running the TPC-C benchmark with only local transactions. Fine-grained shared-nothing deployment configuration is 4.5x faster than shared-everything.

Similarly, when the OLTP workload is perfectly partitionable, the fine-grained shared-nothing configuration provides better performance. As an example, we compare the performance of the shared-everything version of Shore-MT with the fine-grained shared-nothing version with 24 instances on the quad-socket machine. Both systems run a modified version of the TPC-C benchmark [154] Payment transaction, where all the requests are local and, hence, the workload is perfectly partitionable on Warehouses. We plot the results on Figure 2.4. The fine-grained shared-nothing configuration outperforms shared-everything by 4.5x, due in large part to contention on the Warehouse table in the shared-everything case.

2.2.3 NUMA-aware System Design

NUMA-aware operating systems. Adapting operating systems to non-uniform hardware is an area of active research. Some operating system kernels such as the Mach [2] and exokernel [43], or, more recently, Barrelfish [17], employ the message-passing paradigm. Message-passing potentially facilitates the development of NUMA-aware systems since the communication between threads is done explicitly through messages, which the operating system can schedule in a NUMA-aware way. Other proposals include the development of schedulers that detect contention and react in a NUMA-aware manner [22, 34, 147]. Such schedulers have recently been adapted to task-oriented analytical database engines [48, 126], however, they likely require extensive changes to a traditional database engine.

Synchronization. Adapting software systems to today's non-uniform hardware primarily requires efficient synchronization primitives [35]. Scalable synchronization structures typically rely on efficient inter-core communication using atomic operations. Since an atomic operation becomes much slower over inter-socket links, proposals for scalable NUMA-aware locks rely on hierarchically partitioned structures to maximize access locality [24, 39]. On the system level, a recent study on the performance of garbage collectors on multisoocket multicores analyzes synchronization patterns and systematically removes bottlenecks without completely redesigning the system [49]. We take inspiration from these efforts as we redesign our storage manager for multisoockets.

NUMA-aware data management systems. A lot of past work focuses on adapting databases for SMP systems. For instance, commercial database systems provide configuration options to enable NUMA support, but this setting is often optimized for legacy hardware where each individual CPU is assumed to contain a single core. With newer multisoocket servers, enabling NUMA support might lead to high CPU usage and degraded performance [28, 165]. Similarly, modern operating systems offer better support for NUMA architectures, however, they do not improve application performance out-of-the-box. Tuning existing database systems to multisoocket multicores is still a very challenging task [60, 161].

The majority of the proposals that target building NUMA-aware data management systems focus on removing memory bandwidth bottlenecks for analytical applications [127] and specifically devising efficient join and sorting algorithms that minimize data movement [5, 14, 94, 123]. However, OLTP workloads cannot saturate memory bandwidths and their main problem is ensuring efficient synchronization among threads [125]. Statistical analysis is another challenging data management task and a recent study explores different trade-offs in the design space to conclude that hardware topology-awareness can improve performance by an order of magnitude compared to the state-of-the-art systems [174].

2.2.4 Beyond Hardware Islands

Manycore designs. While hardware islands capture well the topology of today's mainstream servers, many other designs are more complex. Contemporary low power multicores have dozens of cores on the same chip: some of them have shared last level caches [27, 36, 164], while proposals for others argue for independent Islands on the same die [97]. However, with manycore designs, we cannot expect core-to-core latencies to be uniform. For example, the latest generation of low power chips from Tiler family [164] have cores that are organized in the form of a mesh. In this case, communication latency depends on the number of hops between two cores, e.g., for the 36 core chip, it ranges from 45 to 65 cycles [35].

Hierarchy of Islands. We expect that multicore chips with a large number of cores will have less uniform core-to-core communication latencies. For example, Oracle's latest M7 32-core chip [141], which is specifically designed for database appliances, features eight 4-core clusters with dynamically shared last level cache and data analytics accelerators. Each core in a cluster has private L1 data and instruction caches. The L2 instruction cache is shared by all the cores in a cluster, while each of the two L2 data caches is shared by a pair of cores, thus creating a hierarchy of inter-core latencies within a chip. Another interesting example is the latest generation of Intel's Xeon E5 chips [65]. In high core count configurations with 14 to 18 cores, it features three different cache coherence modes, including the new "cluster-on-die(COD)" setting. In this mode, two cache coherence rings on the same chip are completely independent which decreases the communication latency among the cores on the same ring. Practically, two rings form islands on the same chip and can increase performance for application that are island-aware. ARM server processor designs typically place a cluster of smartphone-optimized cores and add a shared last level cache and a memory controller. The resulting designs can have very complex topology and unpredictable latencies, e.g, cache hit latency on a Mars 64-core chip ranges from 2 to 70 cycles depending on the location of the cache line [175].

Dark silicon. A major challenge to continued scalability of multicore chips is the significant increase in power density that leads to dark silicon - the inability to power all cores simultaneously [55]. Limited number of pins and off-chip bandwidth pose additional challenges for future high performance chips. A promising way to overcome off-chip bandwidth issues is to stack multiple memory chips on top of the processor chip [21]. Silicon photonic interconnects between multiple smaller multicores offer a practical design for energy-efficient 1000-core systems [37, 85, 88].

Accelerators. As a consequence of dark silicon, heterogeneous chips containing specialized logic are becoming more appealing. Recent proposals range from custom chips that accelerate individual operations such as partitioning [167] and hashing [84], to query task accelerators [141], to designs that specialize chips to complete analytical queries [168]. These proposals are generally targeted at accelerating specific software codepaths. The more general approach is using reconfigurable chips, such as field programmable gate arrays (FPGAs), for offloading selected data processing paths [69, 166], or adding a layer of security through encryption [11].

This idea is already used in practice to accelerate parts of processing pipeline for Microsoft's Bing search engine [128]. Furthermore, Intel is already offering FPGAs integrated with its Xeon processor in the same coherent package that is socket-compatible with other Xeon processors [67].

Takeaways. In summary, modern hardware poses significant new challenges to software systems. Contention and topology have a significant impact on the performance and its predictability. Predictably fast transaction processing systems have to take advantage of the *hardware islands* in the system. They need to (a) avoid frequent communication between “distant” cores in the processor topology and (b) keep the contention among cores low. With ever more complex processor designs as well as heterogeneity within a core, making software aware of the underlying hardware is essential for sustaining high performance.

2.3 Rack-scale Computing Platforms

In addition to abundant parallelism, another major hardware trend is the decreasing distance between servers through fast interconnects. In this section, we discuss advances in networking in both datacenters and database appliances and forecast their impact on the general purpose transaction processing systems.

Fast network. Low latency interconnects are becoming mainstream [20, 135]. High-performance interconnect fabrics such as Infiniband, with bandwidths up to 100Gbps, are already standard in supercomputers and are making inroads into enterprise datacenters and database appliances [116, 101, 137].

Traditionally, commodity clusters use the TCP/IP software stack that poses a high overhead limiting the potential improvements of the faster interconnects. RDMA enables faster communication by allowing applications to access the main memory of a remote machine without involving an operating system or even a processor on a remote machine. With the growing popularity of Converged Enhanced Ethernet [62], RDMA capabilities are becoming available at lower price point in commodity datacenters. RDMA can enable access to the remote memory of another server in a rack at around 10x latency of the local memory node [156].

The impact of fast network on data analytics. As today's applications store more and more data, distributed data processing architectures are proliferating. They are typically deployed on commodity machines in datacenters and are using commodity Ethernet networks. Even though the conventional wisdom is that network is the bottleneck in distributed data analytics, a recent study demonstrates that the CPU is the bottleneck and that optimizing network performance can only improve median job completion time by 2% [117]. One recent proposal has demonstrated significant improvements in performance by focusing on improving CPU efficiency instead of network and disk I/O [31].

Large shared memory servers. Traditional large shared memory machines with more than 8 processors use specialized interconnects and controller chips to maintain cache coherence [53, 92, 115]. However, such complex designs add significant overheads: for example, remote memory access latencies on a modern 64 processor SGI UV 2000 machine, marketed as a database machine, are 5.5x to 10.7x higher compared to local memory accesses [83]. With the high latency overheads of maintaining cache coherence, software designers need to optimize their code for locality to achieve good performance on large shared memory machines, which in addition to high price limits them to niche applications [120].

Futhermore, the coherence overheads increase with the number of cores and in the near future high performance systems will consist of coherence domains connected by low-latency incoherent interconnect [57]. Each coherence domain will host independent operating system instances similar to commodity clusters today. However, the fast interconnects will decrease the gap between accessing local and remote memory by facilitating access to the remote memory without involving the operating system and with minimal CPU overhead.

Cluster consolidation. From the earliest days of computer clusters, system designers strove to provide an illusion of a single system by using shared file systems and peripheral devices [162, 169]. Server vendors have long embraced consolidation using blade server designs to improve energy efficiency and eliminate redundancies in datacenters. In the near future, rack-scale datacenter designs such as cluster-in-a-box with low power system-on-a-chip (SoC) multicore designs that integrate fast interconnect interfaces will further increase density and reduce energy consumption [38]. Integrating memory and network controllers is already becoming standard in low power server designs for highly parallel workloads [107, 158].

Global shared memory has been an appealing abstraction for decades, especially in the super-computing domain [8]. A number of recent proposals offer mechanism for achieving such functionality at lower cost. For instance, memscale design [106] allows dynamic sharing of memory among servers in a cluster through an add-on card that leverages existing processor interconnect such as HyPerTransport for accessing remote memory. Scale-out NUMA [111] goes a step further by introducing specialized chips and protocols that can bring remote memory to 4x latency of the local memory within a rack. BlueDBM uses custom network interfaces in addition to FPGA interface to flash arrays to improve performance of data analytics on flash by an order of magnitude [75]. Data processing in general is a very appealing target for specialized hardware/software co-design due to many data parallel tasks that are amenable to acceleration [7, 114].

Scaling out scale-up OLTP. Modern scale-up shared-everything designs optimized for multicores cannot be used directly on rack-scale hardware since they take advantage of cache coherent global shared memory. Their latch-free data structures and algorithms for concurrency control are optimized for short transactions and even shorter critical sections. Distributed transactions involve network delays that are significantly longer than the individual transactions and can introduce imbalances in the system. For example, Microsoft's Hekaton,

Chapter 2. Background

the state-of-the-art commercial in-memory transaction processing engine, does not support distributed transactions in SQL Server 2014 [102] which hints at the challenges of supporting this scenario.

Distributed transaction protocols. Any distributed transaction in a shared-nothing system needs to use a consensus protocol, such as two-phase commit (2PC) [104], to ensure ACID properties. Many have argued that two-phase commit is inherently unscalable [23, 58] and designers of many large-scale distributed systems have avoided these issues by using weaker consistency like eventual consistency [159]. While relaxed models remove the need for a synchronous consensus protocol, they significantly increase the complexity since consistency needs to be ensured in the application. One way to remove the need for agreement protocol at commit time in a transaction processing system is using a deterministic transaction execution protocol [148, 149]. Calvin deterministically schedules all incoming transactions to ensure conflict-free execution and achieve high throughput in a distributed system deployed in a datacenter [149].

An alternative to determinism for improving efficiency of transactions in distributed environment is using semantic information about the workload to avoid unnecessary coordination [13, 47]. The RAMP model [12] proposes atomic visibility as a weaker alternative to serializability to scale much better than the two-phase commit and satisfy requirements of web-scale companies whose systems typically use eventual consistency for better performance. MDCC is a recent commit protocol optimized for long roundtrip latencies in data centers [86] that improves upon 2PC by requiring only one round of communication when concurrent transactions are conflict-free. With a few additional assumptions, it is possible to design a one-phase commit protocol for general transactions [1]. It is an interesting avenue of future work to optimize such a protocol for modern scale-out clusters. However, all of these recent proposals are targeting datacenter deployments and thus utilize complex conflict detection algorithms that introduce too much overhead to be practical for low latency rack-scale deployments.

RDMA key-values stores. Many projects have explored network-specific optimizations for distributed data management systems, including Hadoop, HBase and memcached [61, 68, 145]. Simply using faster network significantly decreases time spent on network-related tasks, however, further improvements require optimization of the whole communication stack around RDMA. Several projects have gone a step further in designing distributed key-value stores specifically for RDMA using user level networking. Pilaf [103] uses one-sided RDMA reads to achieve high throughput and low CPU usage. FaRM [41] is a distributed computing platform that exposes memory of a cluster of machines as a shared address space which can be used for designing systems on top of it. HERD [77] is a recent design that improves performance of Pilaf and FaRM key-value stores by using two-sided RDMA reads and thus harnessing full potential of the current RDMA hardware.

RDMA and data analytics. Optimizing distributed data analytics for fast networks has recently gained popularity with ever increasing data sizes and network bandwidths that require

rethinking traditional distributed database architectures [20]. The HyPer team showed that their system can scale across multiple servers connected with Infiniband using the hybrid parallelism model that combines the morsel-driven parallelism with the communication multiplexer [133]. AnalyticsDB is a prototype analytical in-memory database system that leverages RAMCloud [113] cluster infrastructure to enable elastic sizing of memory per machine [150]. Large joins in distributed databases, that are very common in analytical workloads, severely stress bandwidth among machines in a cluster. Parallel hash join implementation can be optimized for rack-scale hardware by carefully tuning data exchange to the network characteristics [16]. CycloJoin is a proposal for clusters with fast interconnects that have ring topology that exploits the fast data movement to fully utilize available processors [45]. NeoJoin uses careful scheduling of network communication to improve locality and remove delays introduced by network saturation [132]. TrackJoin minimizes network traffic by scheduling processing on a per key basis [124]. Overlapping precisely inter-node communication with computation within a node also can be used for efficient sorting on clusters with fast interconnects [79]. While this line of work provides insight into the trade-offs between processing and communication for bandwidth sensitive data processing operations, it is orthogonal to transaction processing requirements that stress latency.

Takeaways. Rack-scale computer systems are the emerging commodity computing platform characterized by multiple multsocket servers with large main memories connected using a high-speed network [6]. Neither scale out nor scale up OLTP designs are optimal for rack-scale hardware platforms due to conflicting requirements. Scale-out designs completely ignore the opportunities for optimizing communication between instances located on the same physical machine. State-of-the-art scale-up systems, when deployed in a distributed way, are sensitive to the delays introduced by the network communication involved in executing distributed transactions. We also believe that transaction processing systems need to take holistic view when optimizing for rack-scale computing platforms.

3 OLTP on Hardware Islands

Multisocket multicores are highly parallel and characterized by the non-uniformity in the communication costs: sets, or *islands*, of processing cores that communicate with each other very efficiently through shared on-chip caches, and less efficiently with cores from other islands through bandwidth-limited and higher-latency links. Even though multisocket multicore machines are prevalent in modern data-centers, it is unclear how well software systems in general and OLTP systems in particular exploit multisockets.

In this chapter, we characterize the impact of non-uniformity of modern multisocket multicore servers on transaction processing systems. We use both microbenchmarks and standard benchmarks (TPC-B, TPC-C), with and without data skew. The workloads are executed using different deployments of distributed transaction processing systems of varying granularity as well as shared-everything deployments. We place particular emphasis on the impact of the percentage of multipartition transactions.

This chapter starts with an overview of the Islands performance model and the discussion about different deployment options in [Section 3.1](#). Next, it contains the detailed experimental setup description as well as the methodology in [Section 3.2](#). The next four sections outline experimental results grouped in the following way: [Section 3.3](#) quantifies the impact of multipartition transactions on the throughput in a variety of settings; [Section 3.4](#) expands the analysis to measure the sensitivity to varying the database size, number of processors, data access skew, and disk accesses; [Section 3.5](#) discusses the impact of distributed transactions in the context of more complex workloads, and [Section 3.6](#) expands the analysis in the context of the main-memory optimized system. Finally, [Section 3.7](#) summarizes the findings and discusses the implications.

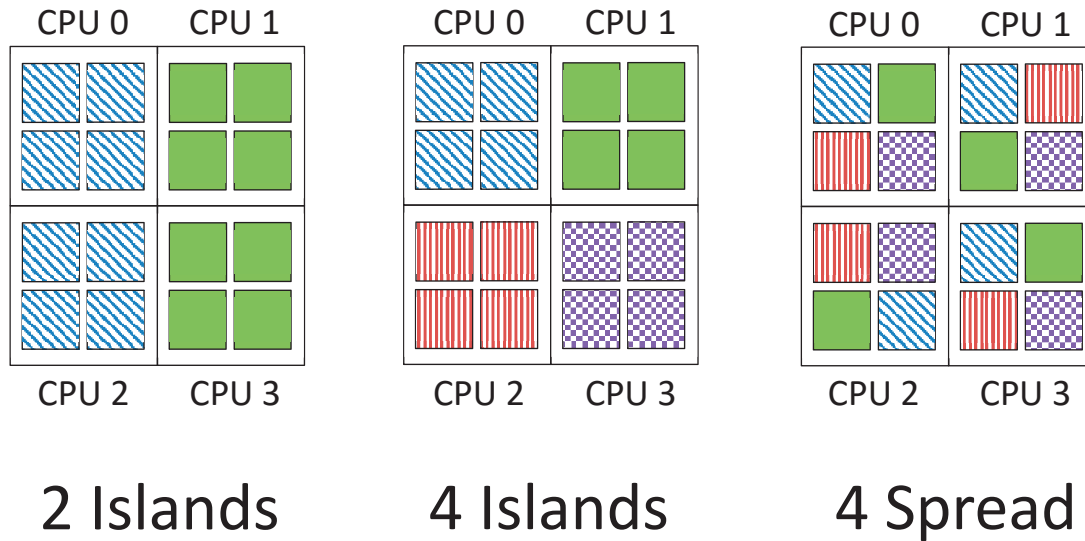


Figure 3.1: Different shared-nothing configurations on a four-socket four-core machine.

3.1 Islands: Hardware Topology-aware Shared-nothing OLTP Deployments

Traditionally, database systems fall into one of two main categories: shared-everything or shared-nothing. The distinction into two strict categories, however, does not capture the fact that there are many alternative shared-nothing deployment configurations of different granularities, nor how to map each shared-nothing instance to CPU cores.

Figure 3.1 illustrates three different shared-nothing deployment configurations. The two left-most configurations, labeled “2 Islands” and “4 Islands”, dedicate different number of cores per instance, but, for the given number of cores, minimize the communication cost as much as possible. Computation within an instance is done in close cores. The third configuration, “4 Spread” has the same size per instance as “4 Islands”. However, it does not minimize the communication cost, as it forces communication across sockets when it is strictly not needed. The first two configurations are islands in our terminology, where an island is a shared-nothing configuration where each shared-nothing instance is placed on the minimal number of sockets (in order to maximize locality). The third configuration is simply a shared-nothing configuration. As hardware becomes more parallel and less uniform, the design space over the possible shared-nothing configurations increases, and it is harder to determine the optimal deployment configuration.

On top of the hardware complexity, we have to consider that the cost of a transaction in a shared-nothing environment also depends on whether this transaction is *local* to a database instance or *distributed*. A transaction is local when all the required data for the transaction is stored in a single database instance. A transaction is distributed when multiple database instances need to be contacted and a distributed consensus protocol (such as two-phase

3.1. Islands: Hardware Topology-aware Shared-nothing OLTP Deployments

commit) needs to be employed. Thus, the throughput also heavily depends on the workload, adding another dimension to the design space and making the optimal deployment decision nearly “black magic.”

An oversimplified estimation of the throughput of a shared-nothing deployment as a function of the number of distributed transactions is given by the following. If T_{local} is the throughput of the shared-nothing system when each instance executes only local transactions, and T_{distr} is the throughput of a shared-nothing deployment when every transaction requires data from more than one database instances, then the total throughput T is:

$$T = (1 - p) * T_{local} + p * T_{distr}$$

where p is the fraction of distributed transactions executed.

In a shared-everything configuration all the transactions are local ($p_{SE} = 0$). On the other hand, the percentage of distributed transactions in a shared-nothing deployment depends on the partitioning algorithm and the system configuration. Typically, shared-nothing configurations of larger size, i.e., the ones deployed over more cores, execute fewer distributed transactions, as each database instance contains more data. That is, a given workload has a set of *local* transactions that access data in a single logical site, and *multisite* transactions that access data in multiple logical sites. A single database instance may hold data for multiple logical sites. In that case, multisite transactions can actually be physically local transactions, since all the required data reside physically in the same database instance. Distributed transactions are only required for multisite transactions whose data reside across different physical database instances. Assuming the same partitioning algorithm is used (e.g., [32, 122, 129]), then the more data a database contains the more likely for a transaction to be local.

Given the previous reasoning one could argue that an optimal shared-nothing configuration consists of a few coarse-grained (i.e., large-sized) database instances. This would be a naïve assumption as it ignores the effects of hardware parallelism and variable communication costs that we explore in Section 2.2. For example, if we consider contention, then the cost of a (local) transaction of a coarse-grained shared-nothing configuration C_{coarse} is higher than the cost of a (local) transaction of a very fine-grained configuration C_{fine} , because the number of concurrent contending threads is larger. That is, $T_{coarse} < T_{fine}$, since throughput is inversely proportional to the execution cost of a single transaction, i.e., $T = \frac{1}{C}$. If we consider communication latency, then the cost of a topology-aware islands configuration $C_{islands}$ of a certain size is lower than the cost of a topology-unaware shared-nothing configuration C_{naive} . That is, $T_{islands} > T_{naive}$.

Figure 3.2 illustrates the expected behavior of Islands, shared-everything, and fine-grained shared-nothing configurations as the percentage of multisite transactions in the workload increases. Islands exploit the properties of modern hardware by utilizing the sets of cores that communicate faster with each other. Islands are shared-nothing designs, but partially combine the advantages of both shared-everything and shared-nothing deployments. Similarly to a

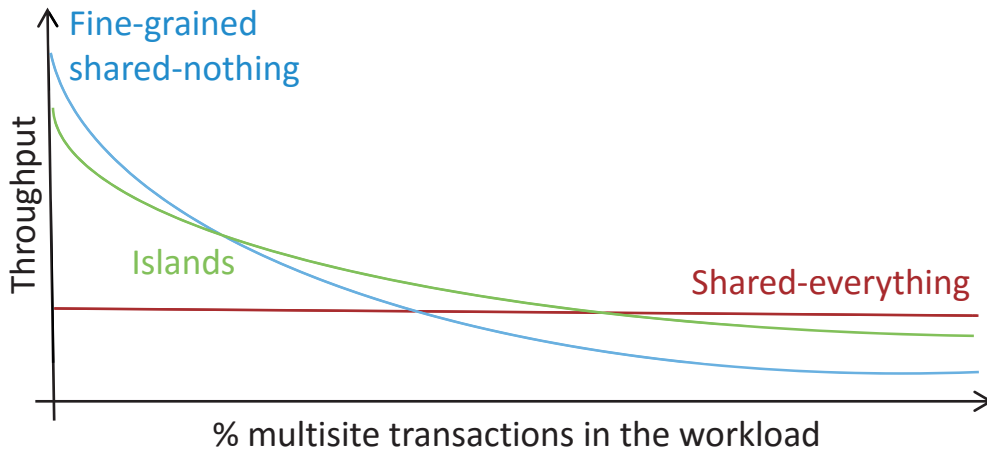


Figure 3.2: Performance of various deployment configurations as the percentage of multisite transactions increases.

shared-everything system, Islands provide robust performance even when transactions in the workload vary slightly. At the same time, performance on well-partitioned workloads should be high, due to less contention and avoidance of higher-latency communication links. Their performance, however, is not as high as a fine-grained shared-nothing system, since each node has more worker threads operating on the same data. At the other side of the spectrum, the performance of Islands will not deteriorate as sharply as a fine-grained shared-nothing under the presence of multipartition transactions.

3.2 Experimental Setup and Methodology

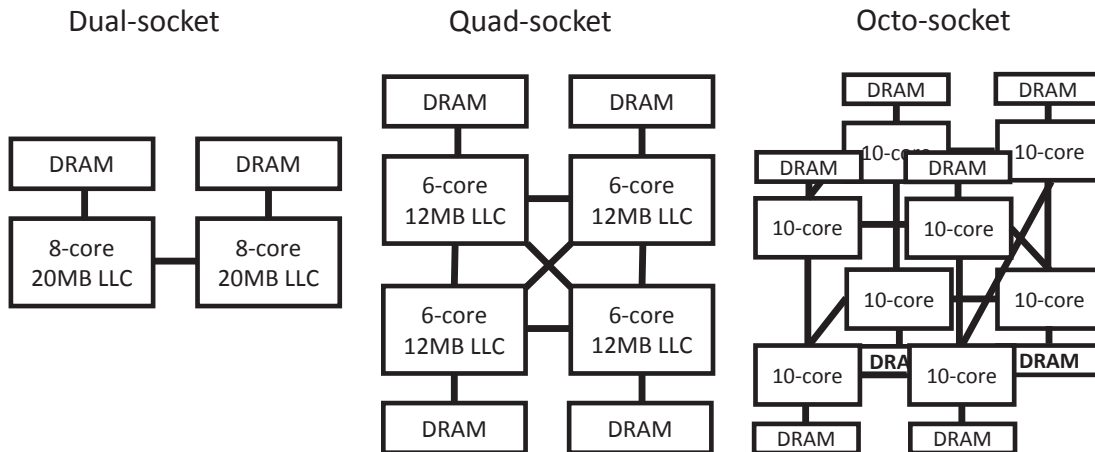


Figure 3.3: Topology of the three machines used in the experiments.

In this study we quantify the impact of non-uniform hardware topology using three modern multisocket multicore machines, one with two sockets of 8-core CPUs, one with four sockets

Table 3.1: Description of the machines used.

Machine	Description
Dual-socket	2 x Intel Xeon E5-2640 v2 @ 2.00GHz 8 cores per CPU Fully-connected with QPI 256 GB RAM 64 KB L1 and 256 KB L2 cache per core 20 MB L3 shared CPU cache
Quad-socket	4 x Intel Xeon E7530 @ 1.86 GHz 6 cores per CPU Fully-connected with QPI 64 GB RAM 64 KB L1 and 256 KB L2 cache per core 12 MB L3 shared CPU cache
Octo-socket	8 x Intel Xeon E7-L8867 @ 2.13GHz 10 cores per CPU Connected using 3 QPI links per CPU 192 GB RAM 64 KB L1 and 256 KB L2 cache per core 30 MB L3 shared CPU cache

of 6-core CPUs, and one with eight sockets of 10-core CPUs. The topology of these machines is depicted in [Figure 3.3](#): smaller machines are fully connected, while the octo-socket one uses the twisted cube topology such that each pair of sockets is at most two hops away¹. The two socket machine is a typical representative of the multisockets used by the major cloud service providers such as Amazon Web Services [9]. The four socket machine, that is used in an experiment unless otherwise noted, is an example of a current mainstream high performance server, while the eight socket one represents the type of servers used in high-end appliances marketed by major vendors [101, 116].

Hardware and tools. [Table 3.1](#) describes in detail the hardware used in the experiments. We disable HyperThreading to reduce variability in the measurements. The operating system is Red Hat Enterprise Linux 6.2 (kernel 2.6.32). In the experiment of [Section 3.4.4](#), we use two 146 GB 10krPM SAS 2,5" HDDs in RAID-0. We use Intel VTune Amplifier XE to collect basic micro-architectural and time-breakdown profiling results. VTune does hardware counter sampling, which is both accurate and lightweight [64].

IPC mechanisms. The performance of any shared-nothing system heavily depends on the efficiency of its communication layer. [Figure 3.4](#) shows the performance in the quad-socket machine of various inter-process communication (IPC) mechanisms provided by the operating systems using a simple benchmark that exchanges 256 byte messages between two processes which are either located in the same CPU socket or in different sockets using operating system

¹ For more details see <http://www.supermicro.com/manuals/motherboard/7500/X80BN-F.pdf>

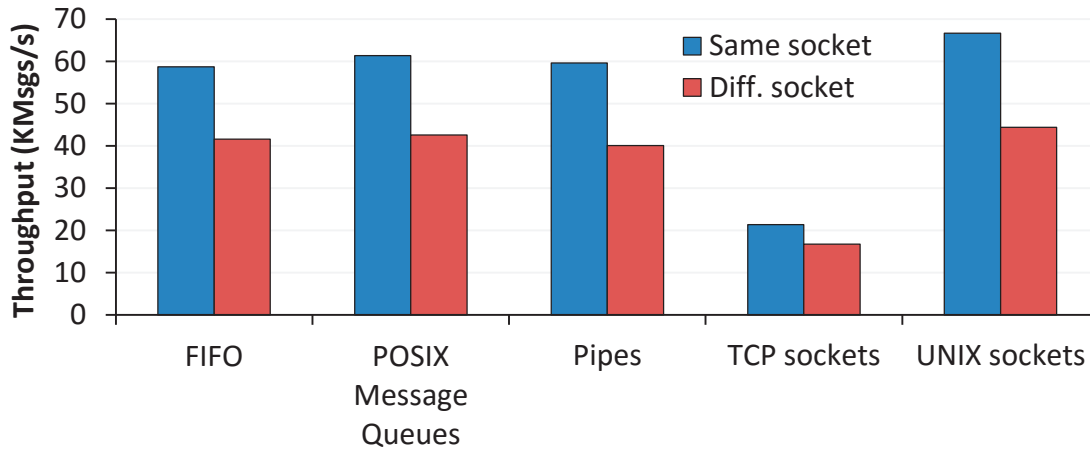


Figure 3.4: Throughput of message exchanging (in thousands of messages exchanged per second) for a set of inter-process communication mechanisms. Unix domain sockets are the highest performing.

facilities. Unix domain sockets achieve the highest performance and are used throughout the remaining evaluation. In Section 3.3.4 and with Silo, we use more efficient shared memory messaging implementation that bypasses the operating system, however, it does not change the trends in our experiments.

3.2.1 Prototype Systems

In order to evaluate the performance of deployments of different granularities, we prototype distributed transaction processing systems on top of two storage managers: Shore-MT² [71] and Silo³ [157]. Most of our experiments use Shore-MT and we use Silo to generalize our conclusions to the main memory systems. We use the same distributed transaction processing logic and communication mechanisms with both storage managers and apply the same optimizations.

We opted for Shore-MT as a representative traditional system since it is an open-source storage manager that scales very well on servers with a single multicore processor [71]. Shore-MT is the improved version of the SHORE storage manager, originally developed as an object-relational data store [26]. Shore-MT is designed to remove scalability bottlenecks, significantly improving Shore’s original single-thread performance. Its performance and scalability are at the highest end of open-source storage managers. Silo is an open source scalable shared-everything storage manager that is representative of the new wave of main-memory optimized transaction processing systems.

Both Shore-MT and Silo use shared-everything designs. Therefore, we extended them with

² <https://sites.google.com/site/shorem/>

³ <http://github.com/stephentu/silo>

the ability to run in shared-nothing deployments, by implementing a distributed transaction coordinator using the standard two-phase commit (2PC) protocol. Our 2PC protocol implementation includes an optimization for the execution of the read-only parts of the distributed transactions: if the execution site has decided that the transaction is read-only, it is committed at the end of the first phase and the site is not involved in the second round of communication.

Both systems used in this study are storage managers that do not include some components found in a typical commercial database system such as a query optimizer and a client communication library. Instead, the benchmark application directly accesses the storage manager through the API calls. We use hardcoded transaction execution plans for all benchmarks and implement distributed transactions in one-shot fashion [144] with local and remote transaction parts known apriori. This allows coordinator and participating instances to exchange only one message in the first phase of 2PC. These techniques are commonly used in commercial high performance deployments using stored procedures in order to eliminate unnecessary overheads.

Shore-MT includes a number of state-of-the-art optimizations for local transactions, such as speculative lock inheritance [70] and Aether holistic logging [72]. Speculative lock inheritance reduces the contention on the lock manager by caching locks acquired in the shared mode and reusing them for subsequent transactions. Aether reduces log buffer contention using cooperative log buffer insertions and flush pipelining to move system calls involved in writing log records to the durable storage off the critical path of transaction execution. We extended these features for distributed transactions, providing a fair comparison between the execution of local and distributed transactions.

Shore-MT-based system is compiled using GCC 4.4.7 with maximum optimizations, while experiments with Silo use version 5.1.0 as it requires the support for c++11 language features. In most experiments with Shore-MT, the database size fits in the aggregate buffer pool size. As such, the only I/O is due to the flushing of log entries. However, since the disks are not capable of sustaining the I/O load, unless otherwise noted, we use memory mapped disks for both data and log files. Overall, we exercise all code paths in the system and utilize all available hardware contexts. In the experiments with Silo, we use only main memory storage and do not generate any I/O requests.

3.2.2 Microbenchmark Workload and Experimental Methodology

In the experiments, we vary the number of instances of the database system. Each instance runs as a separate process. Within each experiment, we use the same input data size for all deployment configurations and range-partition the data into logical sites across all instances in the deployment. Sites are disjoint subsets of the dataset with one or more sites located in the same instance in the distributed deployment. We allocate one site to each processor core. For the majority of microbenchmark experiments, we use a small dataset with 10 000 rows per site (e.g., on a quad socket machine it amounts to 240,000 rows ~ 60 MB in Shore-MT), and

describe the specific larger datasets for other experiments. We show results using different deployment configurations, but we always use the same total amount of data, processor cores, and memory resources for every deployment in the experiment. Only the number of instances and the distribution of resources across instances change.

We ensure that each database instance is optimally deployed. That is, each database process is bound to the cores within a single socket when possible, and its memory is allocated in the nearest memory bank. We made this decision as allowing the operating system to schedule processes arbitrarily leads to suboptimal placement and frequent thread migration, which degrades performance, as explored in more detail in [Section 2.2](#).

In the experiments, we typically compare a number of deployment configurations of different granularities. The configurations on the graphs are labeled with "NISL" where N represents the number of instances. For example, in the experiments on a quad socket server with 24 cores, 8ISL represents the configuration with 8 database instances, each of which has 1/8th of the total data and uses 3 processor cores. The number of instances varies from 1 (i.e., a shared-everything system) to 24 (i.e., a fine-grained shared-nothing system). We tune all configurations, by turning on and off different optimizations when applicable and provide details when describing a particular experiment. For example, in Shore-MT experiments, fine-grained shared-nothing instances that run single-threaded do not latch data pages.

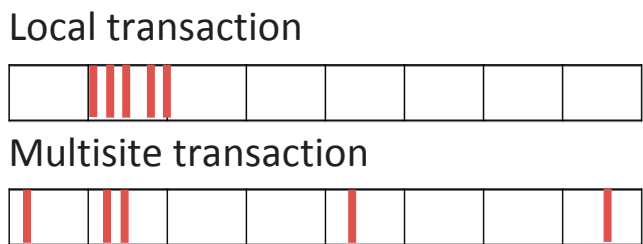


Figure 3.5: Examples of microbenchmark transactions with $N = 5$ where the second partition is the local one.

We use microbenchmarks that come in two flavors: (1) *read-only* where each transaction retrieves N rows, and (2) *update* where each transaction updates N rows. For each microbenchmark, we run two types of transactions, local and multisite. Intuitively, we assign a site (i.e., a subset of rows) to the processor core and then place in the same instance all rows assigned to the cores on which that instance runs. We illustrate this scheme in [Figure 3.5](#) and define the two transaction types as follows:

- **Local transactions** perform their action (read or update) on the N rows from the local site;
- **Multisite transactions** perform their action (read or update) on one row from the local site while the remaining $N - 1$ rows are chosen uniformly from the whole data range. Transactions are distributed if some of the input rows happen to be located in remote instances.

We chose these microbenchmarks because they allow us to quantify the impact of different factors on the cost of executing local and distributed transactions including the number of rows accessed in a transaction and the number of instances involved. The flexibility of the microbenchmark allows us to explore a wide range of workload types from the perfectly partitionable to the completely un-partitionable ones that access rows from many partitions requiring distributed transactions.

The percentage of multisite transactions that are executed as distributed transactions depends on the deployment configuration and the number of rows accessed. Let's illustrate the dependency through an example. For the transaction that accesses 2 rows in the configuration with N instances, there is a $\frac{1}{N}$ probability that the multisite transaction will be local since the remote row is chosen from a site residing in the local instance. In the case of 20% multisite transactions, we have 19.17% distributed transactions for the 24ISL configuration and 15% for the 4ISL one. In the case of transactions that access 20 rows, the probability of a multisite transaction being local is extremely low ($\frac{1}{N^{19}}$) and every distributed transaction involves all instances.

3.2.3 Standard Workloads

TPC-B [153] is a transaction processing benchmark that models debit and credit operations of a bank. It is designed as a stress test for OLTP systems, particularly their concurrency control and logging components. The TPC-B schema contains four tables: `Branch`, `Teller`, `Account`, and `History`. The TPC-B workload consists of a single transaction type, `AccountUpdate`, that updates one record in `Branch`, `Teller`, and `Account` tables and inserts one record to the `History` table. It is easily partitionable on the `BranchID` attribute of the `Branch` table. According to the benchmark specification, 85% of the transactions are local, i.e., they access data from the same branch, whereas the remaining remote transactions update one teller in the remote branch.

The more complex TPC-C [154] benchmark models a transactional database of the wholesale supplier. Its schema contains nine tables and can be partitioned on the `WarehouseID` key of the `Warehouse` table that is part of the primary key of six other tables [144]. The benchmark defines five different transactions, a mix of read-only and read-write ones, that each access at least three tables. We will focus only on the two read-write transactions, `NewOrder` and `Payment`, because 1) they comprise 88% of the transactions in the standard mix and 2) they are the only ones that potentially require distributed transactions in a shared-nothing deployment. `NewOrder` is a medium length transaction that models placing a new order for 5-15 items, where an item is selected from the remote warehouse with the probability of 1%. This leads around 10% of the transactions to be multisite. `Payment`, on the other hand, is a short transaction that updates customer's balance as well as the warehouse and district sales statistics. In 85% of the cases, the chosen warehouse represents home warehouse for the customer and district. In the remaining 15% of the cases, the chosen warehouse is a different

one which causes this transaction to be multisite, as it involves both logical sites associated with the home and remote warehouses.

3.3 Impact of Multisite Transactions

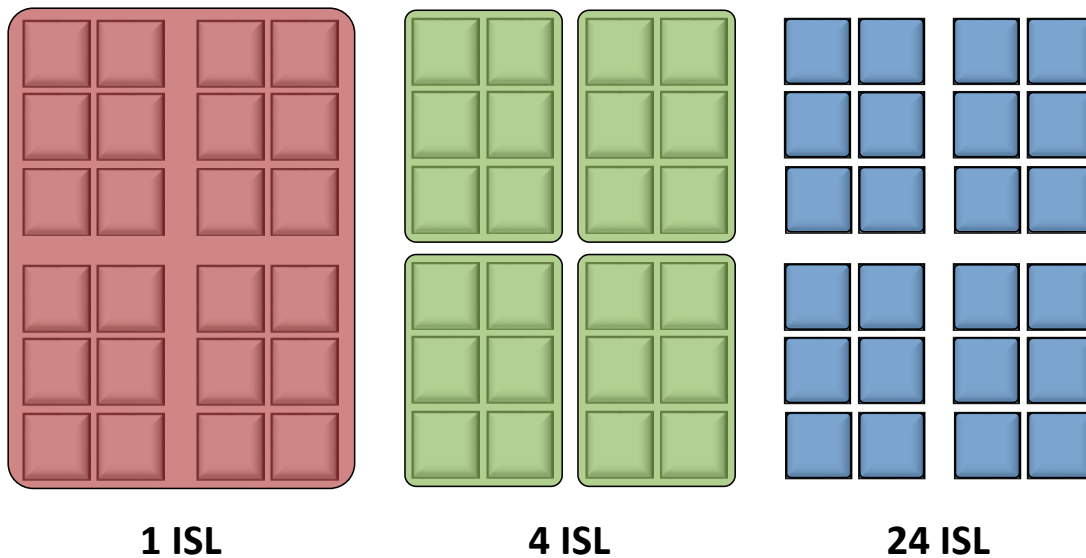


Figure 3.6: Main deployment configurations illustrated on the 4 socket server.

In this section we analyze in depth the impact of good partitioning scheme on performance of different configurations. If good partitioning scheme exists for a particular workload, there will be fewer multisite transaction and vice versa. The impact of partitioning is different for read-heavy, update-heavy and workloads whose transactions contains both reads and writes. We simulate different types of workloads by varying the percentage of multisite transactions for the microbenchmarks that read or update 10 rows. This setting gives us good baseline observations about the behavior of main configurations that we compare in this study (illustrated using the quad socket server and depicted in [Figure 3.6](#)):

- **Fine-grained shared-nothing** (labeled 24ISL) is a deployment configuration where data is divided into as many partitions as there are cores in the system. Each partition is assigned to a single database instance that serves all transactions accessing data from that partition. These instances are pinned to different cores of the machine with one instance per core. Each instance uses a single worker thread which eliminate the need to synchronize accesses to the data.
- **Island-sized shared-nothing** (labeled 4ISL) is a deployment configuration where data is divided into as many partitions as there are sockets in the system. Each partition belongs to a single database instance that is pinned to a particular processor socket. We use as many worker threads as there are cores on the processor and they collectively serve transactions

that access data belonging to a specific instance. Memory is allocated in the local memory node.

- **Shared-everything** (labeled 1ISL) is a deployment configuration with a single database instance that utilizes all cores in the system and processes all transactions. In contrast to the shared-nothing configurations, in this case all transactions are local and we never have to execute distributed transactions.

In the common case, we use a small dataset with 240,000 rows, Unix domain sockets as the communication mechanism and the system built on top of Shore-MT. We chose the small dataset because it is almost cache-resident which highlights the positive impact of data locality in shared-nothing configurations. We quantify the effects of dataset sizes by examining performance trends as well as microarchitectural behavior of different configurations when dataset does not fit in the caches. We also replace Unix domain sockets with shared memory communication mechanisms for inter-process communication and evaluate their impact on performance by breaking down the costs of local and multisite transactions into system components. Finally, we expand our analysis to different hardware platforms with varying numbers of sockets and cores per socket to quantify the impact of hardware topology on the behavior of different deployment configurations.

3.3.1 Distributed Transactions

Distributed transactions are known to incur a significant cost, and this problem has been the subject of previous research, with e.g., proposals to reduce the overhead of the distributed transaction coordination [74] or to determine an initial optimal partitioning strategy [32, 122, 129]. Our experiment, shown in Figure 3.7, corroborates these results. We run two microbenchmarks whose transactions read and update 10 rows respectively on the quad-socket machine. As expected, the configuration 1ISL (i.e., shared-everything) is not affected by varying the percentage of multisite transactions. However, there is a drop in performance of the remaining configurations, which is more significant in the case of the fine-grained one.

Both fine-grained (24ISL) and island-sized(4ISL) shared-nothing configurations have high performance for the workloads that contain only local transactions. The performance improvement compared to the shared-everything is especially high for the read-only transactions and fine-grained configurations that run in the single-threaded mode without locking or latching. As the percentage of multisite transactions in the workload increases, the performance of 24ISL configuration decreases mainly due to the messaging overhead involved in the execution of distributed transactions. The trends for the 4ISL configuration are similar with progressively lower performance as the percentage of multisite transaction increases. However, the drop in performance is smaller due to fewer instances that participate in the execution of a single distributed transaction and, consequently, fewer messages that need to be exchanged. At the same time, performance for local-only transaction is not as high as in the 24ISL case because

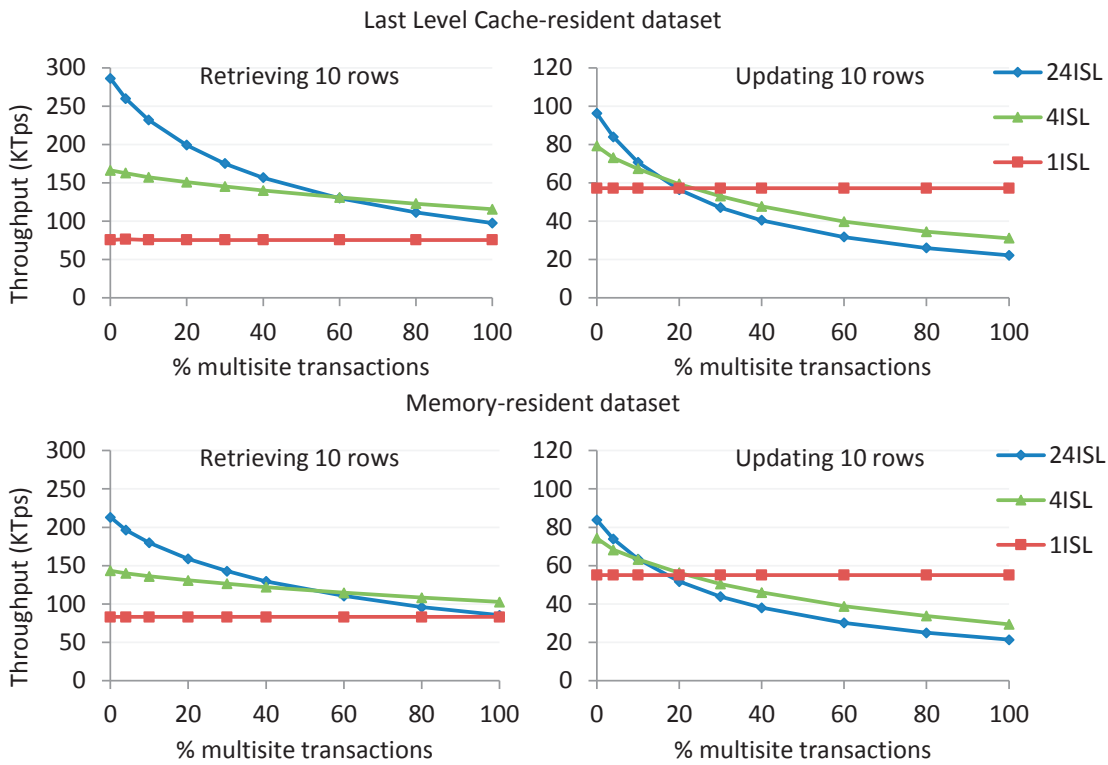


Figure 3.7: Performance as the percentage of distributed transactions increases on cache and memory resident datasets. While shared-everything remains stable, performance of share-nothing configurations decreases. Smaller instances benefit a lot from cache-resident datasets.

of multiple worker threads that execute transactions in the same instance and thus have to use locking and latching to ensure isolation.

While the trends for the update case (Figure 3.7, top right) are similar to the read-only one, the shape of the lines is different. As in the previous case, partitioned configurations have higher performance than the shared-everything one for local-only transactions, however, the difference is smaller because updates require logging that is more expensive than just accessing data. When the percentage of multisite transaction in the workload increases, distributed transactions cause performance to drop faster than in the case of read-only transactions. This is because distributed update transactions are more expensive due to the two rounds of messaging, additional logging after the first phase, and the increased contention as exclusive locks are held until the end of the second phase of the 2PC protocol.

In addition to lower synchronization costs compared to the shared-everything system, partitioned configurations in this experiment have a benefit of cache locality as the data set almost fits in the last level caches. To quantify the impact of locality on the performance, we repeat this experiment with a larger dataset of 2.4 million rows (~ 600 MB) and plot throughput on the lower half of Figure 3.7. We observe that while the performance of the shared-everything

system remains almost the same, the performance of partitioned configurations decreases by 5-25% with larger decrease for the fine-grained configuration. The relative decrease is larger for local-only transactions, since access to the data takes larger portion of the execution time compared to multisite transactions (as we show in more detail in [Section 3.3.3](#)).

3.3.2 Microarchitectural Behavior

To better understand the impact of thread synchronization and data locality for different types of configurations, we profile their behavior for local-only transactions by accessing hardware performance counters using VTune. For this experiment, we run read-only microbenchmark which accesses 10 rows from the local site and use both last level cache-resident ([Figure 3.8 \(top\)](#)) and memory-resident datasets ([Figure 3.8 \(bottom\)](#)).

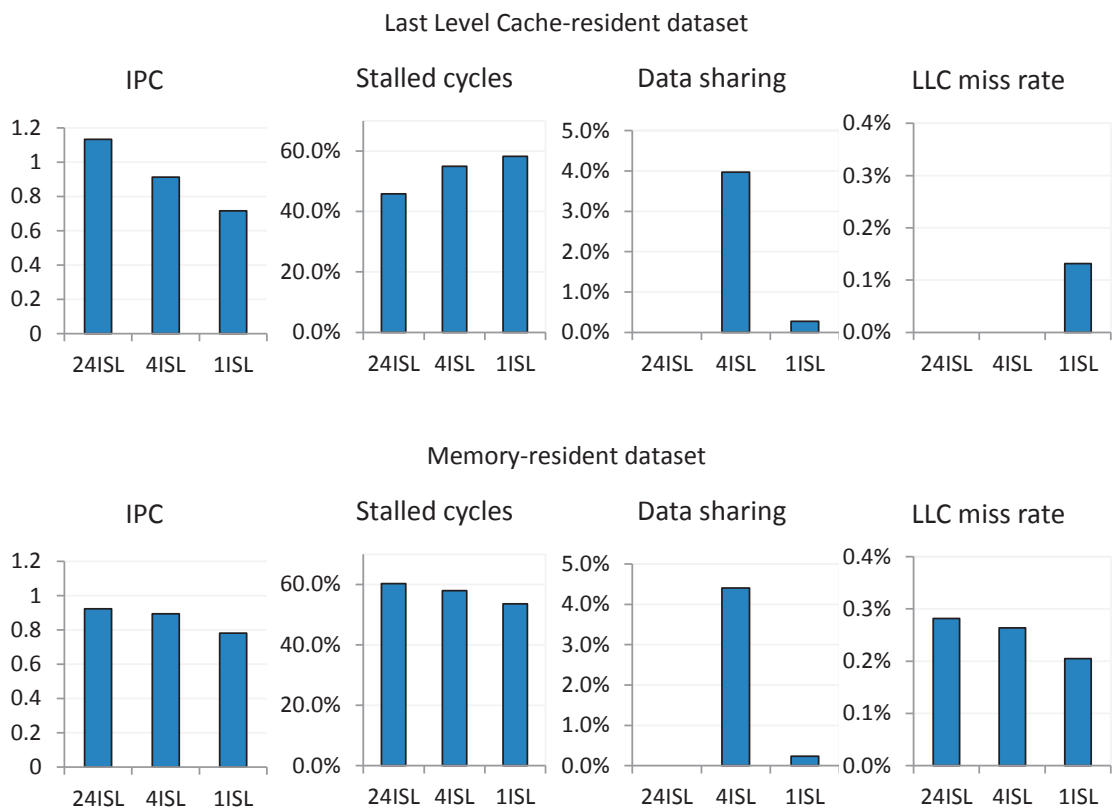


Figure 3.8: Microarchitectural data for different deployments and datasets: smaller instances benefit a lot from locality in the workload.

The leftmost graph of the top row in [Figure 3.8](#), which plots the number of instructions retired per cycle (IPC), shows that the shared-nothing configurations, whose instances have fewer threads, have better utilization of the CPU. Single-threaded instances, apart from not communicating with other instances, use simpler execution model leading to shorter code paths, which decreases the number of instruction misses. On the other hand, instances that span across sockets have a much higher percentage of stalled cycles (shown in the second

graph from the left of Figure 3.8 (top)). This is due to the presence of—expensive—last-level cache (LLC) misses (shown in the right-most graph in Figure 3.8 (top) as the percentage of all memory requests that result in LLC data misses). In contrast, shared-nothing instances have zero LLC misses as the data fits in the last level cache of each processor and all transactions are local. Finally, within the same socket, smaller instances have higher ratio of instructions per cycle due to fewer stalls while accessing shared data structures since fewer threads share the same data. This effect is observed on the “data sharing” graph in the Figure 3.8 (second from the right in the top row) that plots the ratio of cycles the system is accessing remote cache lines to all cycles.

The benefit of fewer threads per instance is reduced when the data does not fit in processor caches, which is the common case in real-life workloads, as shown in the bottom row of Figure 3.8. In this case, fine-grained shared-nothing instances still manage to retire more instructions per cycle compared to the larger instances, however, their IPC rates are lower than in the case with cache-resident data. This is due to the long latency LLC misses that cannot be effectively overlapped by the modern superscalar processors. LLC misses also increase for the coarse-grained shared-nothing instances leading to higher percentage of stalled cycles. Overall, the diminished locality in the workload, due to data not fitting in the LLC, causes the smaller instances to have more stalled cycles compared to the shared-everything instance. Finally, the data sharing patterns do not change compared to the case of cache-resident dataset, leading to the conclusion that the lower processor utilization for shared-nothing configurations is due to the reduced cache locality in the workload.

3.3.3 Profiling

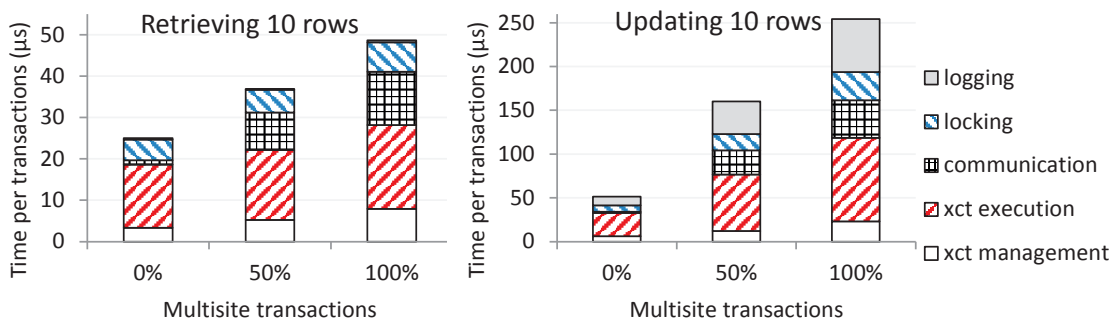


Figure 3.9: Time breakdown for a transaction that retrieves (left) or updates (right) 10 rows and uses unix domain sockets for communication. The cost of communication dominates in the cost of distributed transaction in the read-only case, while in the update case overheads are divided between communication and additional logging.

In order to characterize the overhead of inter-process communication costs in relation to the remaining costs of a distributed transaction, we profile the execution of a set of read-only and update transactions on the quad-socket machine, using the 4ISL configuration. Figure 3.9 plots time breakdown for the microbenchmark transaction which reads or updates 10 rows

from the small dataset. The messaging overhead is high in the read-only case, although it has a constant cost per transaction. The relative cost of communication can be seen by comparing the 0% multisite (i.e. local transactions only) and the 100% multisite bars. Also, we observe an increase in the cost of transaction management due to bookkeeping overheads.

Even though messaging overhead is high for the distributed read-only transactions, they require a single round of communication since we can use the following optimization of the 2PC protocol: if the transaction fragment contains only read-only operations, it sends a *read-only* vote at the end of the prepare phase and does not participate in the second phase. In contrast, update transactions have to vote either *commit* or *abort* at the end of the first phase. If they vote *commit*, i.e., the processing is successful, they have to hold all exclusive locks until they get the decision message from the coordinator in the second communication phase. These factors make the distributed transaction significantly more expensive than their read-only counterparts. Although distributed transactions require exchange of twice as many messages in the update case, this overhead is comparatively smaller because of additional logging, as well as increased contention which further increase the cost of a transaction.

3.3.4 Impact of the Communication Channel

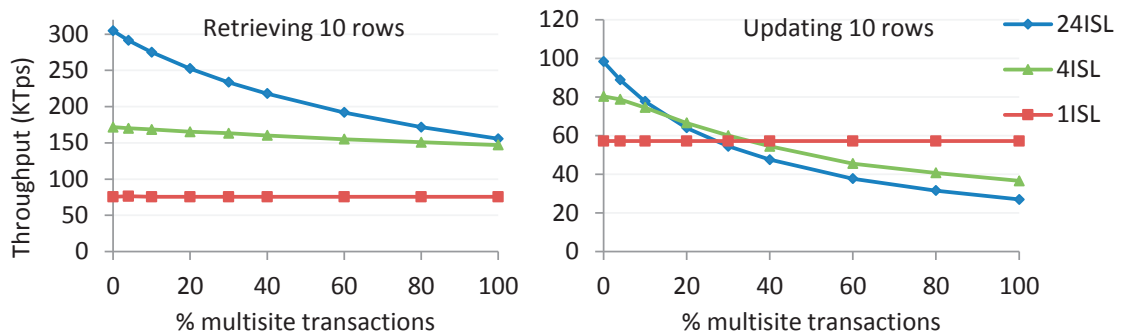


Figure 3.10: Performance as the percentage of multisite transactions increases using shared memory communication channel. Read-only distributed transactions benefit from faster communication much more than the update ones.

Although unix domain sockets are the fastest messaging mechanism provided by the operating system (Section 3.2), they still cause large communication overheads when executing distributed transactions (as we can observe in Figure 3.9). This is primarily due to the fact that they involve expensive system calls. In order to remove the overhead of system calls, we implement a prototype shared memory communication mechanism. While shared memory communication is more complicated to use and implement, it is used for inter-process communication in all major commercial database systems.

We repeat the experiment from Section 3.3.1 with a small dataset and plot the throughput in Figure 3.10. We observe that the performance trends of various configurations are the same as in the case of unix domain socket communication channels (Figure 3.7 (top)). However,

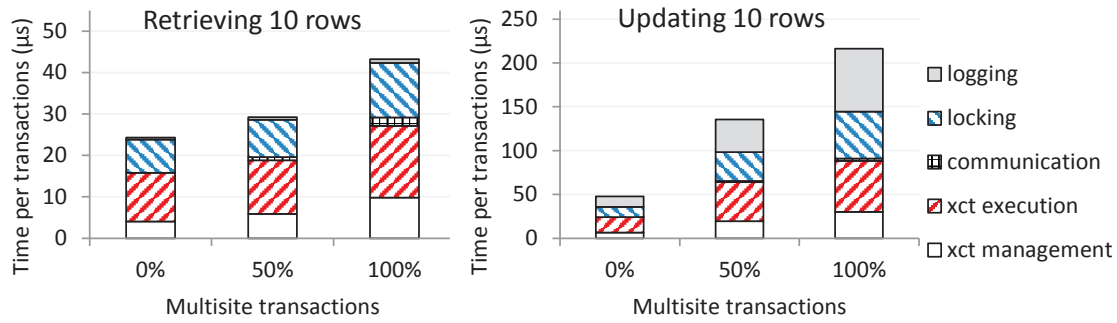


Figure 3.11: Time breakdown for a transaction that retrieves (left) or updates (right) 10 rows and uses shared memory channels for communication. Lower cost of communication decrease significantly decrease the costs in the read-only case, while other costs increase in the update costs as they cannot be overlapped with communication anymore.

the relative decrease in performance, as the percentage of the multisite transaction in the workload increases, is lower. For example, the throughput of 24ISL configuration for the read-only transactions improves by 60% and 4ISL by 25% for the workload consisting of 100% of multisite transactions. This improvement is smaller for the update case, measuring 22% and 12% respectively. Even though communication overhead represented significant part of the cost of distributed transactions (Figure 3.9) for both types of transactions, improved communication is more beneficial for the read-only ones.

To characterize the impact of faster communication mechanism, we repeat the profiling experiment from the Section 3.3.3 with the shared memory communication channel and show the results in Figure 3.11. Since in this case communication bypasses the operating system and the instances avoid making system calls, communication overhead diminishes significantly. The lower communication cost directly results in better throughput of read-only microbenchmark transactions. In the update case, however, the benefits are significantly smaller due to the other overheads of the 2PC protocol that cannot be overlapped with communication anymore, including additional logging and increased lock contention.

3.3.5 Different Topologies

The number of islands is one of the most important factors that determines their impact on the transaction processing systems. In this experiment, we extend our analysis to two very different multisocket machines with two and eight processors (their configuration is outlined in Table 3.1). We repeat the experiments with microbenchmark that reads and updates 10 rows and compare shared-everything and fine- and coarse-grained shared-nothing configurations.

Figure 3.12 (top) plots the throughput of the different configurations as we increase the percentage of multisite transactions on the octo-socket server. We use the cache-resident dataset with 10,000 rows per core for a total of 800,000 rows. Each of the eight processors in this machines has ten cores, hence we have 80 instances in the fine-grained (labeled 80ISL)

3.3. Impact of Multisite Transactions

and 8 instances in the island-sized shared-nothing deployment (labeled 8ISL). Similarly to the smaller, quad-socket, server used in the experiment in [Section 3.3.1](#), throughput of the shared-everything system is constant irrespective of the percentage of multisite transactions. However, it is much lower compared to the partitioned configurations. We further examine the scalability of different configurations as the number of sockets increases in [Section 3.4.2](#). As the percentage of multisite transaction increases in the read-only case, the performance of 80ISL configuration decreases more than the 8ISL one due to the higher communication overheads. In addition to more instances that are involved in the execution of a single distributed transaction, fine-grained deployment has higher static communication overheads due to the larger number of instances in the system. The trends are similar for the update case with larger decrease in performance due to higher overheads of distributed update transactions.

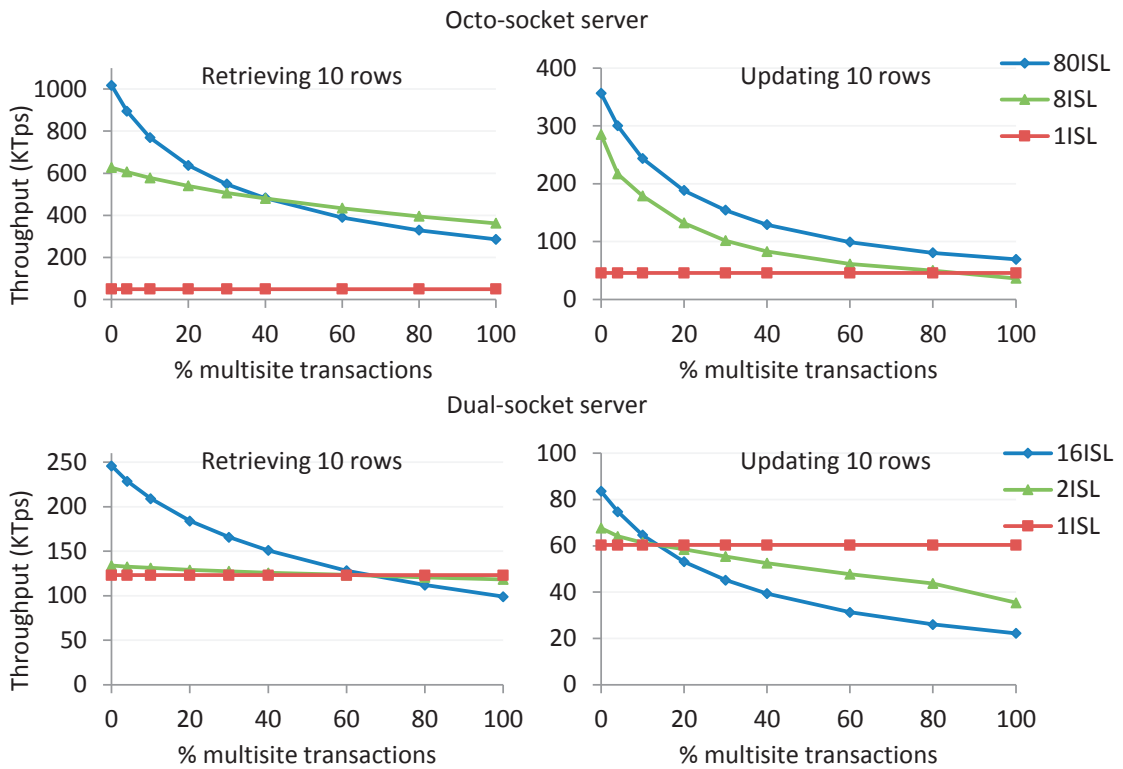


Figure 3.12: Performance as the fraction of multisite transactions increases on dual and octo socket servers. Trends are common across machines, however, hardware topology determines relative performance of different configurations.

In contrast to the octo-socket server, the impact of islands is much smaller on the dual-socket server. We plot the results of the dual-socket server experiment on the bottom part of [Figure 3.12](#). This server has two eight core processors, so we deploy fine-grained shared-nothing configuration with 16 instances and the island-sized one with 2 instances. In this case, the cache-resident dataset contains 160,000 rows. For the read-only microbenchmark, 16ISL configuration has almost two times better throughput compared to the other configurations for the local-only transactions. The performance drops with the increase in the percentage of

multisite transactions, however, this drop is smaller compared to fine-grained instances on the larger servers due to fewer instances in the system which lowers communication overheads. The 2ISL configuration has slightly better performance compared to the 1ISL one for local transactions due to the fairly large number of threads that need to synchronize their accesses to the shared data structures. At the same time, the overhead of distributed transactions is small as the percentage of multisite transaction increases since each distributed transaction requires exchange of a single pair of messages. The situation is different for the update microbenchmark where the overheads of distributed transactions cause sizable performance drop for partitioned configurations as the percentage of distributed transaction increases. This is the case even for 2ISL configuration as the main overheads related to additional logging and bookkeeping are proportional to the number of updated rows. The shared-everything deployment benefits from optimized logging to offer consistently good performance for update transactions.

3.3.6 Summary

Our experiments show that the performance trends for different deployment configurations are consistent with the Islands performance model. Finer-grained shared-nothing configurations have better throughput for mostly local transactions, while coarser-grained ones have higher throughput in the presence of many multisite transactions. The exact cross-over point depends on the type of operations as well as the hardware topology.

3.4 Sensitivity Analysis with Microbenchmarks

In this section we perform sensitivity analysis using microbenchmark workloads by varying a number of parameters. We start by expanding the range of configurations to include the ones larger and smaller than an island and measuring the cost of transactions as a function of the number of rows accessed. Next, we project how the deployments will scale with the increasing number of islands in the system and evaluate the tolerance to skew. Finally, we investigate the effects of dataset sizes that cannot entirely fit in the main memory.

3.4.1 Impact of the Size of Transaction

In this experiment we use the quad-socket machine and all reasonable configuration choices. We start with the configurations we introduced in the previous section: shared everything (1ISL), coarse-grained shared-nothing (4ISL), and fine-grained shared-nothing (24ISL). Additionally we introduce coarser-grained configuration whose instances span across sockets (2ISL) and two finer-grained configuration with multiple instances per socket (8ISL and 12ISL). We tune each configuration for the optimal performance: disable locking and latching for the single-threaded instances and enable Aether logging optimizations for larger instances where constructive sharing among threads decreases the pressure on the logging subsystem. We

focus on the costs as opposed to throughput since we analyze trends separately for the local and multisite transactions.

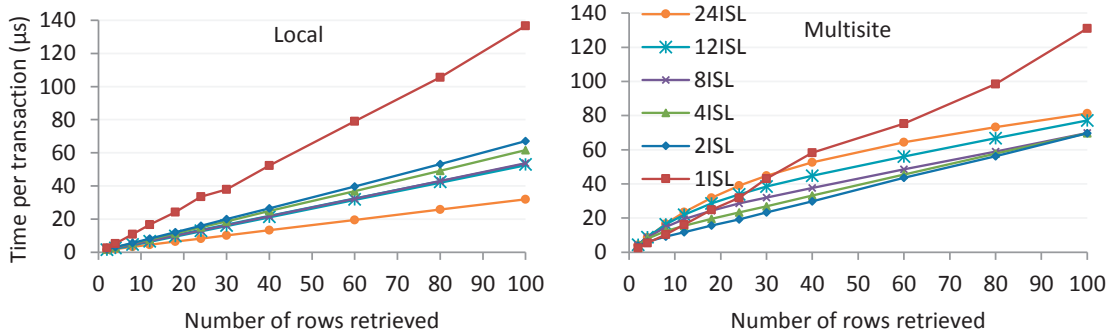


Figure 3.13: Cost of local and multisite transactions in the read-only microbenchmark. For multisite transactions, communication costs rise until all instances are involved in every transaction.

Read-only Case: Overhead Proportional to the Number of Participating Instances

Figure 3.13 (left) represents the time it takes to execute a single local read-only transaction in various database configurations as the number of rows retrieved per transaction increases. The 24ISL configuration runs with a single worker thread per instance, so locking and latching are disabled, which leads to roughly 40% lower costs than the next best configuration, corroborating previous results [56].

The costs of multisite read-only transactions (Figure 3.13 right) show the opposite trend compared to the local read-only transactions for shared-nothing configurations. First, for small number of rows per transaction, we observe super linear increase in cost as more instances become involved in the execution of a single transaction. This trend flattens out once all instances are involved in the execution of every transaction and the number of messages exchanged per transaction becomes constant. However, for the shared-everything case, the costs of accessing sharing data structures is so high that for large transactions, it has worse performance than all shared-nothing configurations which execute distributed transactions.

Update Case: Additional Logging Overhead Is Significant

The left graph of Figure 3.14 present the time it takes to execute a single local transaction of the update microbenchmark. The cost of a transaction increases with the number of threads in the system, due to contention on shared data structures. As in the read-only case, the 24ISL configuration runs without locks or latches and hence, has lower costs.

In contrast to the read case, multisite shared-nothing transactions (Figure 3.14, right) are significantly more expensive than their local counterparts. This is due to the overhead associated

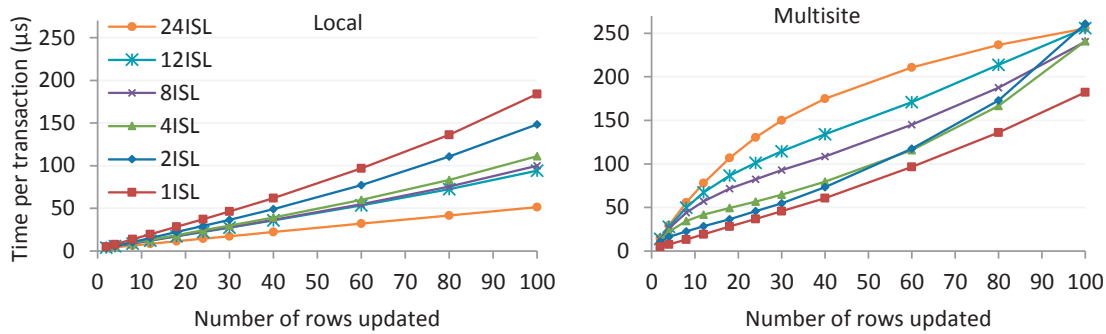


Figure 3.14: Cost of local and multisite transactions in the update microbenchmark. Shared-everything can take advantage of consolidated logging that is especially significant for multisite transactions.

with distributed transactions and to the (mandatory) use of locking. Any configuration that requires distributed transactions is more expensive than the shared-everything configuration. We can observe the same trend as in read-only case with super linear increase in costs as number of instances involved in the transaction rises which later flattens out. In addition, we have another trend of the increase in costs of transaction that access the large number of rows since holding locks for a longer period of time increases contention. Finally, for the shared-everything configuration costs rise linearly and quickly become smaller than all the other configurations, primarily due to use of efficient logging with Aether [72].

3.4.2 Increasing Hardware Parallelism

Hardware parallelism as well as communication variability will likely continue to increase in future processors. Therefore, it is important to study the behavior of alternative database configurations as hardware parallelism and communication variability grow. In Figure 3.15, we run the microbenchmark which reads (left) or updates (right) 10 rows with fixed percentage of multisite transactions to 20%, while the number of cores active in the machine is increased gradually. Results are shown for both the quad-socket and the (more parallel and variable) octo-socket machine.

The shared-nothing configurations scale linearly, with *CG* (coarse-grained shared-nothing) configuration being competitive with the best case across different machines and across different levels of hardware parallelism. The configuration labeled *SE* (shared-everything) does not scale linearly, particularly on the machine with 8 sockets. In the *SE* configuration, there is no locality when accessing the buffer pool, locks, or latches. To verify the poor locality of *SE*, we measured the QPI/IMC ratio, i.e. the ratio of the inter-socket traffic over memory controller traffic using Intel's PCM tool [66]. A higher QPI/IMC ratio means the system does more inter-socket traffic while reading (i.e. processing) less data overall: it is less NUMA-friendly. The QPI/IMC ratio for the experiment with read-only workload on octo-socket server using all 80 cores is 1.73 for *SE*, 1.54 for *CG*, and 1.52 for *FG*. The *FG* and *CG* configurations still

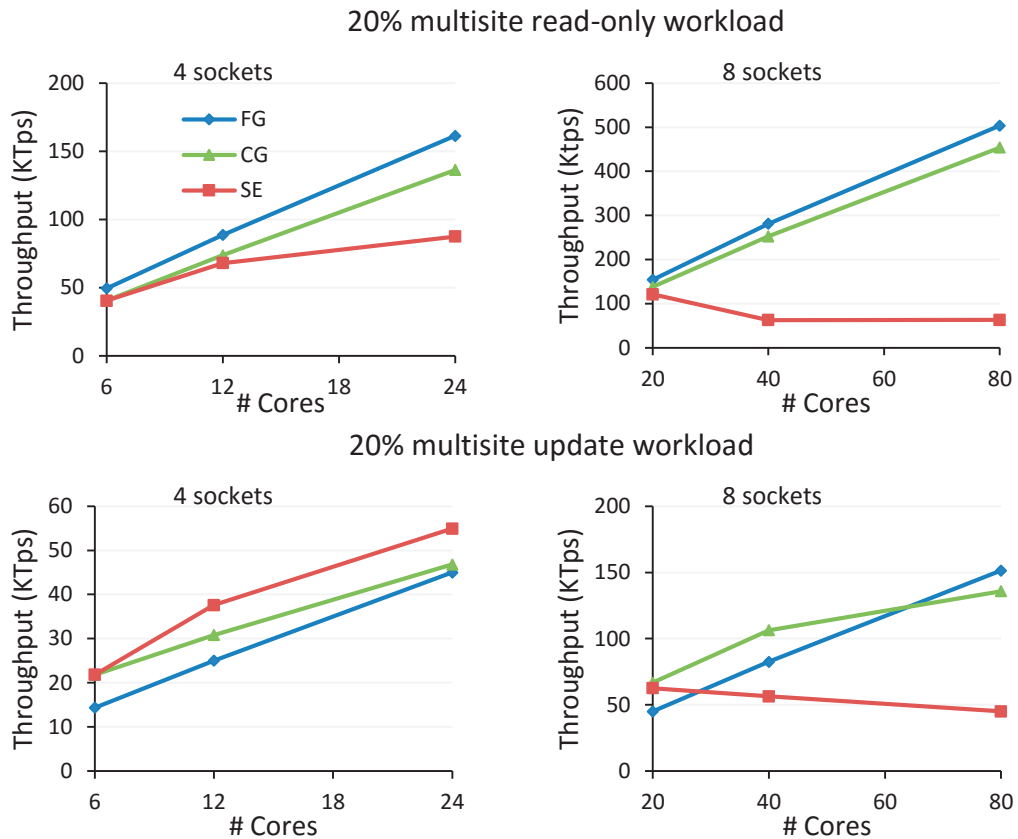


Figure 3.15: Performance of alternative configurations as the hardware parallelism increases. Coarser-grained shared-nothing provides an adequate compromise between performance and predictability.

have a relatively high ratio due to multisite transactions but, unlike *SE*, these consist of useful work. When restricting all configurations to local transactions only, we observe a steady data traffic of 100 Mb/s on the inter-socket links for *FG* and *CG* (similar to the values observed when the system is idle), while *SE* exceeds 2000 Mb/s.

Clearly, to scale the *SE* configuration to a larger number of cores, data locality has to be increased. Additionally, one of the main reasons for poor performance of *SE* configuration is high contention on locks and latches. Using partitioned shared-everything designs with data-oriented execution can significantly improve locality of accesses and remove or minimize the overheads coming from locking and latching components in the system [118, 119]. We explore this direction further in Chapter 4 and design ATraPos, a system that uses NUMA-friendly data structures and data-oriented execution to minimize inter-socket synchronization in the critical path of transaction execution.

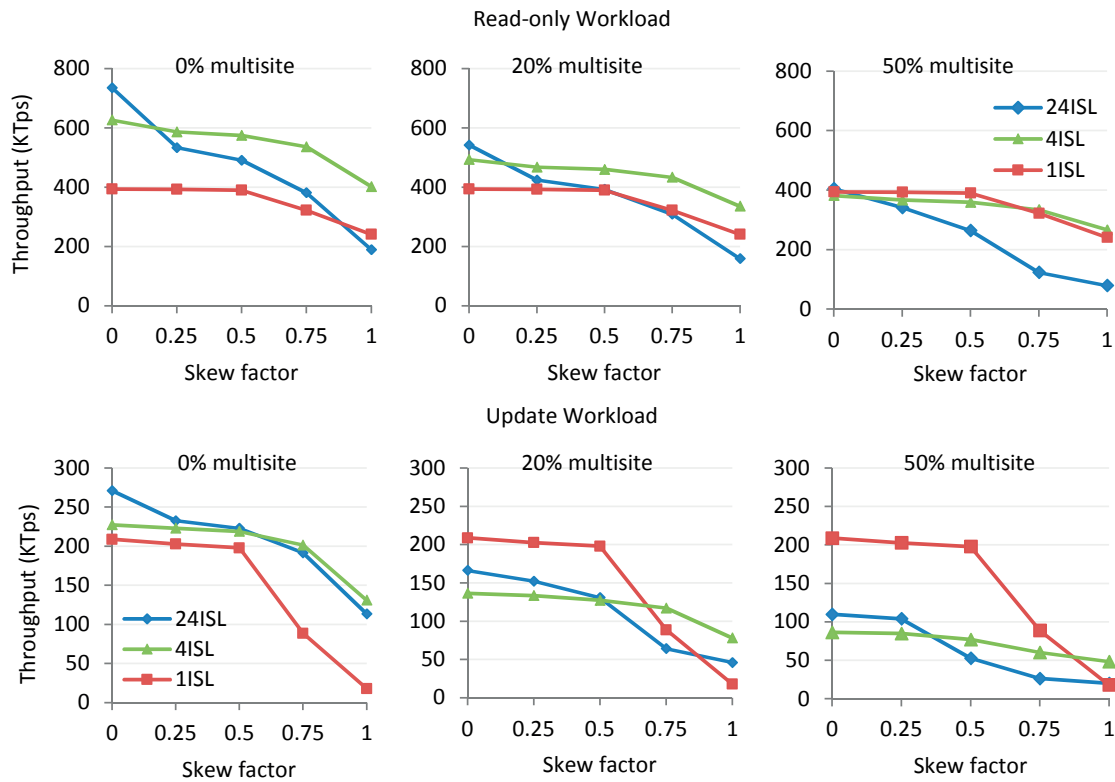


Figure 3.16: Performance of read-only (top) and update (bottom) workloads with skewed accesses. As skew increases, shared-everything suffers from increased contention, while fine-grained shared-nothing suffers from a highly-loaded instance that slows others. Island-sized shared-nothing configurations cope better with highly loaded instances, due to multiple internal threads.

3.4.3 Tolerance to Skew

In many real workloads, skews on data and requests, as well as dynamic changes are the norm rather than the exception. For example, many workloads seem to follow the popular 80-20 distribution rule, where the 80% of requests accesses only the 20% of the data. This subsection describes experiments with workloads that exhibit skew.

The following microbenchmark reads or updates two rows chosen with skew over the whole data range. We use Zipfian distribution, with different skew factors s , shown on the x-axis of Figure 3.16. The figures show the throughput for varying percentages of multisite transactions. We employ similar optimizations as described in Section 3.4.1.

Skew has a dramatic effect on the performance of the different configurations. For shared-everything, heavily skewed workloads result in a significant performance drop due to increased contention. This effect is apparent particularly in the update case. When requests are not strongly skewed, shared-everything achieves fairly high performance in the update microbenchmark, mainly due to optimized logging, which significantly improves the perfor-

mance of short read-write transactions [72]. In coarser-grained deployments, the increased load due to skewed accesses is naturally distributed among all worker threads in the affected instance. With fine-grained instances, which have a single worker thread, the additional load cannot be divided and the most loaded instance becomes a bottleneck. Furthermore, as the skew increases to the point where all remote requests go to a single instance, the throughput of other instances drops significantly as they cannot complete transactions involving the overloaded instance.

3.4.4 Increasing Database Size

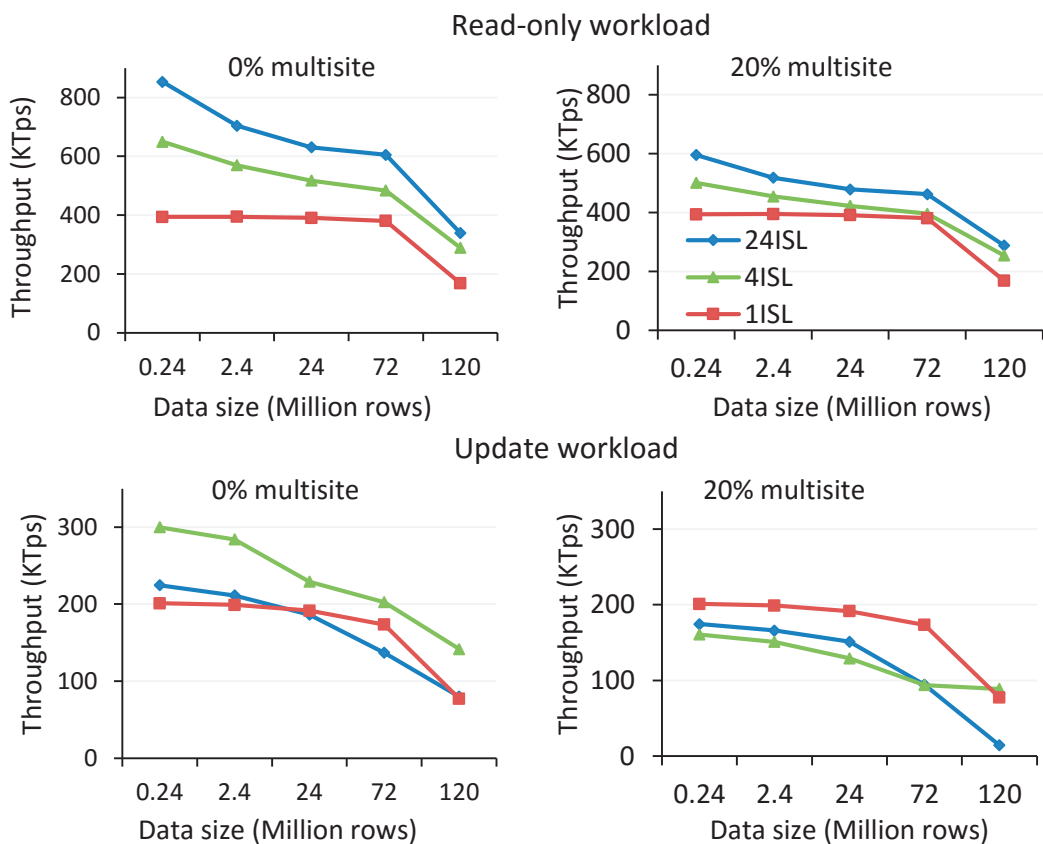


Figure 3.17: Performance of the various configurations on workloads, as we gradually increase the database size from almost cache-resident to I/O-resident.

Although main memory sizes in modern servers continue to grow, there are many workloads that are not main memory resident and rely on disk-resident data. To evaluate various database configurations on growing dataset sizes, we gradually increase the number of rows in the dataset from 240,000 to 120,000,000 (i.e., from 60 MB to 33 GB). Contrary to previous experiments, we place the database on two hard disks configured as a RAID stripe. We use a 12 GB buffer pool, so that the smaller datasets completely fit in the buffer pool. In the shared-nothing configurations, the buffer pool is proportionally partitioned among instances, e.g. in

the 4ISL case each instance has 3 GB buffer pool. We run read and update microbenchmarks with two rows accessed and 0% and 20% multisite transactions.

In [Figure 3.17](#), we plot the performance of the read-only microbenchmark on the left-hand side and the update microbenchmark on the right-hand side as the number of rows in the database grows. For the smaller dataset, shared-nothing configurations exhibit very good performance as a significant part of the dataset fits in last-level caches of the processor. Since the instances do not span multiple sockets, there is no inter-socket traffic for cache coherence. As data sizes increase, the performance of shared-nothing configurations decrease steadily, since smaller portions of the data fit in the caches. Finally, when the dataset becomes larger than the buffer pool, the performance drops sharply due to disk I/O. These effects are less pronounced when the percentage of multisite transaction is higher, since the longer latency data accesses are overlapped with the communication.

3.4.5 Summary

The size of a transaction and the number of instances in the deployment are the main factors that determine the relative impact of 2PC overheads on the cost of a distributed transaction. Island-sized shared-nothing configurations exhibit good performance in the presence of skew, as they suffer less from increased contention and are more resistant to load imbalances. Finally, relative performance of different configurations does not change as parallelism or data sizes increase.

3.5 Standard Workloads

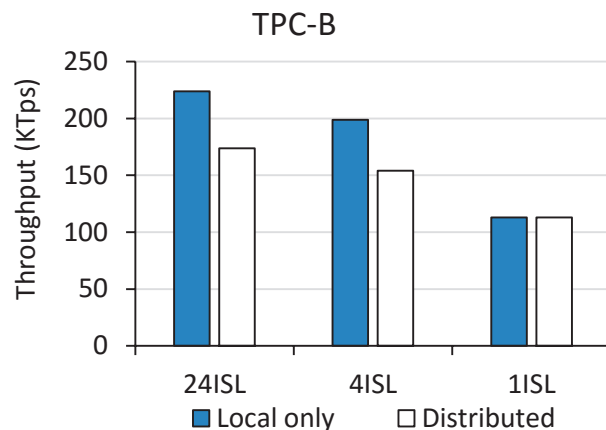


Figure 3.18: Performance of the AccountUpdate transaction in the TPC-B benchmark for the local only and standard-benchmark settings.

In this section we expand our analysis of the impact of hardware islands on transaction processing systems with the characterization of the behavior of industry standard benchmarks,

TPC-B and TPC-C, and, in particular, their remote transactions. The main difference compared to the microbenchmarks discussed in the previous sections lies in the relative distribution of the work among different instances involved in the execution of a distributed transactions.

In the case of microbenchmarks, the work in the distributed transactions is split roughly equally among the participating instances (as discussed in [Section 3.2.2](#)). Whenever a transaction involved accessing more than 2 rows, it was likely to involve more than 2 instances in fine-grained shared-nothing configuration. On the other hand, distributed transactions defined by the TPC-B and TPC-C specifications share the property that the local part of the transaction contains many more operations compared to the remote one. Also, the number of participating instance is two for all TPC-B `AccountUpdate` and TPC-C `Payment` and the vast majority of TPC-C `NewOrder` transactions.

In the majority of microbenchmark experiments presented in [Section 3.3](#) and [Section 3.4](#), rows were selected randomly from a single table. In contrast, in the TPC benchmarks, transactions involve multiple tables, including the ones containing few rows. Furthermore, these transactions typically update the hot rows. When the hot rows are involved in a distributed transaction, they are locked until both phases of the 2PC protocol are completed which prevents any other transaction from accessing them. We run benchmarks with only local transactions as well as varying percentage of distributed transactions and analyze their behavior.

3.5.1 TPC-B

[Figure 3.18](#) compares the throughput of different configurations when they run only local or a mix of local and remote TPC-B transactions. We run the experiment on the quad socket server and use the dataset with 24 branches equally partitioned among instances in the shared-nothing configurations. In this experiment we compare shared-everything (1ISL) and coarse (4ISL) and fine-grained shared-nothing (24ISL) configurations. Shared-everything configuration benefits from the Aether logging optimizations and 24ISL is configured without latching. We use unix domain sockets as the communication mechanism. The remote version of the TPC-B `AccountUpdate` transaction updates one row in the `Teller` table chosen randomly from a remote branch. We use the mix that has 15% of the remote transactions as this percentage is defined in the TPC-B specification.

`AccountUpdate` is a transaction that stresses the concurrency control and logging components of the transaction processing system. Thus, it is not surprising that the partitioned configurations have higher throughput for the local only transactions due to less synchronization among threads in the same instance. However, their performance drops by 22% when we introduce distributed transactions. Even though the distributed version of the `AccountUpdate` transaction involves only two instances, and hence, does not have high communication and bookkeeping overhead, it increases the time that the hot row in the `Branch` table is locked, thus increasing contention.

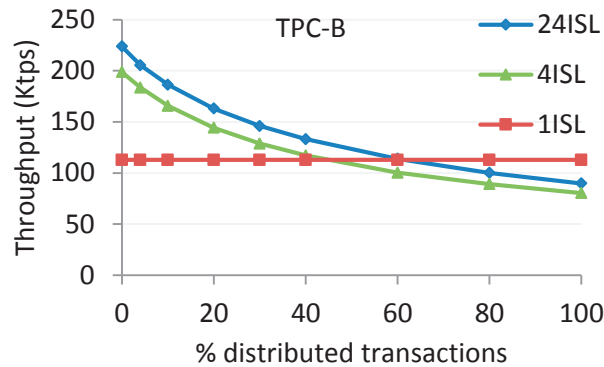


Figure 3.19: Performance of different configurations as the percentage of distributed transactions increases for the TPC-B workload. Distributed transactions increase contention for hot data causing the drop in performance for shared-nothing configurations.

3.5.2 Impact of Distributed Transactions on TPC-B

We further examine the impact of distributed transactions on the performance of TPC-B for different configurations by running an experiment with varying percentage of remote transactions in the workload. We use the same setting as in the experiment in [Section 3.5.1](#), but we gradually increase the percentage of remote transactions from 0% to 100% and plot the throughput in [Figure 3.19](#).

The shared-everything system is not affected by the remote transactions and its stable performance benefits from optimized logging, similarly to the update version of the microbenchmark. Also, we observe the trend of deteriorating performance of shared-nothing configurations as we increase the percentage of distributed transactions. However, in contrast to the update microbenchmarks, here both coarse-grained and fine-grained shared-nothing configurations follow the same trend. This is due to the fact that the number of participating instances in both cases is the same, thus, making the relative cost of remote to local transactions constant.

3.5.3 TPC-C

In this experiment, we quantify the impact of remote transactions for TPC-C benchmark by separately looking at the Payment and NewOrder transactions. We use the quad socket server and the dataset with 24 warehouses. For shared-nothing configurations, we partition the data with one warehouse per core. We compare shared-everything (1ISL), and coarse (4ISL) and fine-grained shared-nothing (24ISL) system configurations. Since both of these transactions contain updates, we enable Aether logging optimization for the shared-everything configuration and disable latching for the fine-grained shared-nothing configuration. We use unix domain sockets as the communication mechanism. We compare the setting with only local transaction and the mix of local and remote transactions using the percentages of remote

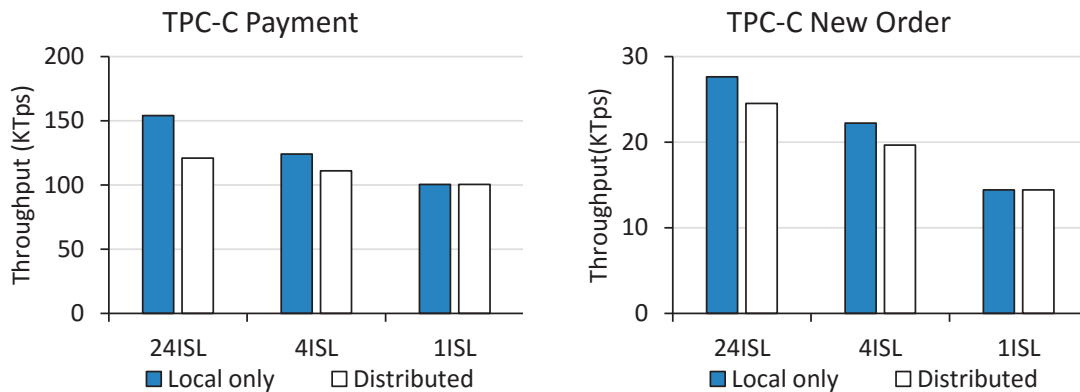


Figure 3.20: Performance of different transactions in the TPC-C benchmark for their local only and standard-benchmark settings. Distributed transactions are more expensive than their local counterparts and they have higher impact on the finer-grained configurations.

transactions defined in the benchmark specification: 15% for the Payment transaction and 10% for the NewOrder.

Figure 3.20 (left) plots the throughput for different configurations of the Payment workload, while Figure 3.20 (right) plots the throughput for the NewOrder case. Similarly to the TPC-B workload, shared-everything system is oblivious to the remote transactions, while the performance of the shared-nothing configurations drops with distributed transactions. The drop is higher for the Payment workload since it has higher percentage of distributed transactions. Also, Payment workload is more sensitive to the distributed transactions as it updates one row of the Warehouse table. On the other hand, NewOrder transactions update one row in the District table that contains 10 rows for each warehouse. In practice, this means that we can have more concurrent transactions in the system for the NewOrder workload (up to the number of District rows) compared to the Payment one (up to the number of Warehouse rows).

3.5.4 Impact of Distributed Transactions on TPC-C

Finally, we characterize the impact of distributed transactions on the TPC-C Payment workload as we gradually increase the percentage of distributed transactions in the workload. We plot the throughput in Figure 3.21 and observe the sharp drops in the performance of shared-nothing configurations as the contention on the hot rows increases with more distributed transactions. At the same time, the performance of shared-everything configuration remains stable.

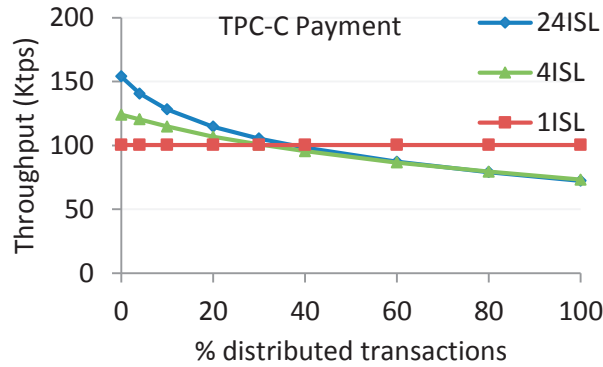


Figure 3.21: Performance of different configurations as the percentage of distributed transactions increases for the TPC-C Payment transactions. Shared-everything configuration offer robust performance in the presence of remote transactions which cause throughput drops for partitioned systems.

3.5.5 Summary

The impact of distributed transaction overheads on TPC-B and TPC-C workloads is lower than for microbenchmark workloads due to fewer participants in the execution of a distributed transaction. When varying the percentage of distributed transactions, we observe trends consistent with the microbenchmarks.

3.6 Main-memory optimized system

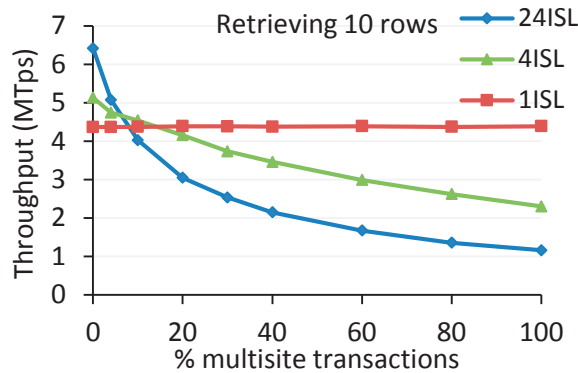


Figure 3.22: Performance of different deployments of Silo as the number of multisite transactions increases. It shows the same trends as the deployments based on Shore-MT.

In this section we quantify the impact of hardware islands on the performance of different deployments of a modern main-memory optimized system. We use Silo [157] which is a multicore optimized system that utilizes cache-conscious multiversioned Mass-tree design [98] as the data storage and employs optimistic concurrency control protocol that scales well on multicores.

We use the same distributed coordination layer as for the Shore-MT experiments. Since Silo does not support distributed transaction out of the box, we split its commit processing into a pre-commit and a post-commit phase. The pre-commit phase, which performs all the validation checks and locks rows that have been changed in a transaction, is executed at the end of the first phase of the 2PC protocol, while the post-commit phase, which applies the changes on the Mass-tree, is executed in the second phase of 2PC. As Silo is a main-memory optimized system that achieves very high throughput, we only run experiments with shared memory communication channels tuned with appropriately sized buffers. We implement the same microbenchmark described in Section 3.2 and use the same transaction execution logic as in the Shore-MT experiments. We run all experiments on a quad socket machine and use a dataset with 240 000 rows. As in the previous experiments, we compare shared-everything (1ISL), and coarse (4ISL) and fine-grained shared-nothing (24ISL) deployment configurations.

3.6.1 Read-only Transactions

Figure 3.22 plots the results of the experiment with increasing percentage of multisite transactions in the workload for the microbenchmark that reads 10 rows. We observe that the smaller instances have higher performance for local-only transactions as data is accessed by fewer cores and hence, the accessed have more locality. Even though Silo’s transaction execution protocol does not have any global synchronization points, it does not use any partitioning and the data is shared by all the threads in the instance. As the percentage of multisite transactions in the workload increases, throughput of partitioned configurations decreases sharply since distributed transactions are more expensive.

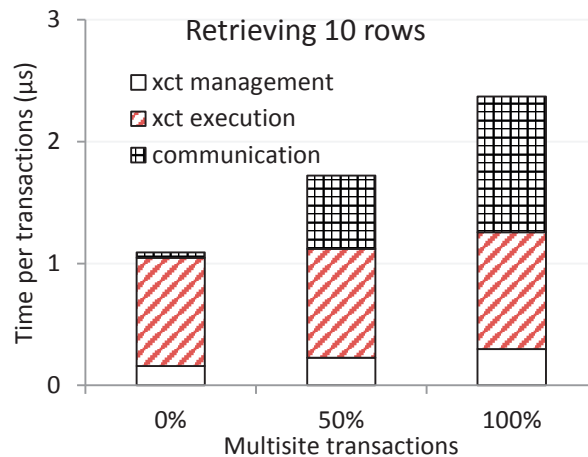


Figure 3.23: Time breakdown for a transaction that retrieves 10 rows in 4ISL deployment. Communication costs determine the overall cost of a transaction.

In order to characterize the impact of communication, we profile the execution of a 4ISL deployment with different percentages of multisite transactions for the microbenchmark that reads 10 rows. Figure 3.23 breaks down the time needed to execute one transaction into

transaction execution, transaction management and communication. As we increase the percentage of multisite transactions, the time required for communication rises while the other two components remain the same. This trend shows that the communication costs are the dominant factor in the cost of the distributed transactions.

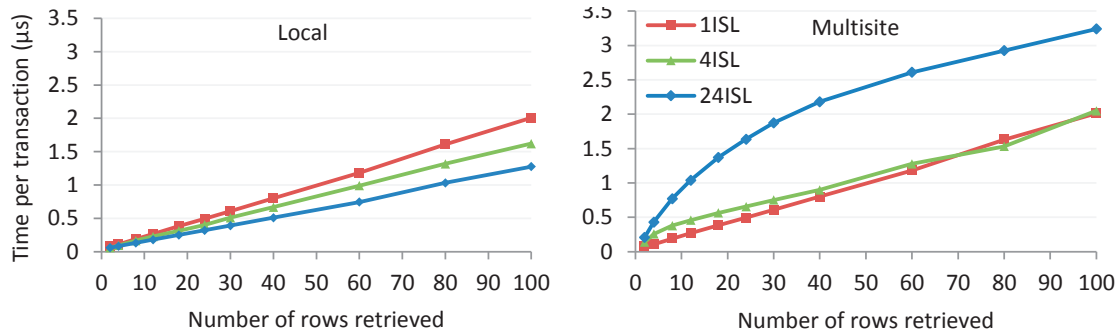


Figure 3.24: Cost of local and multisite transactions in the read-only microbenchmark. The cost of multisite transactions rises until all instances participate in every transaction.

Next, we quantify the impact of transaction size on the cost of local and multisite transactions by increasing the number of rows read, using the same methodology as in Section 3.4.1. The left hand side of Figure 3.24 shows the time it takes to execute a single local transaction. All deployments show linear increases in costs as the number of rows accessed per transactions increases with smaller instances having lower costs. The relative performance trend for the multisite case, presented on the right hand side of Figure 3.24, is completely opposite. Smaller configurations have higher costs that increase with larger number of rows accessed. The increasing trend flattens after all instances in the configuration become involved in every transaction.

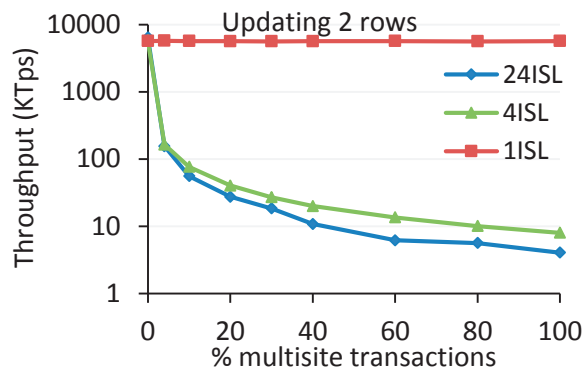


Figure 3.25: Performance of different deployments of Silo as the percentage of multisite transactions increases. The trend is the same as with read-only transaction, however, update transactions are much more expensive.

3.6.2 Update Transactions

Finally, we investigate the behavior of the update distributed transactions as we increase the percentage of multisite transactions. In contrast to the read-only case, in this experiment we use the microbenchmark that updates 2 rows and plot the throughput in [Figure 3.25](#). We use fewer rows because heavier transactions increase contention even further resulting in a very low throughput. [Figure 3.25](#) shows the same trends as the update microbenchmarks that runs on top of Shore-MT, however, the performance degradation is much more severe. This is due to much bigger impact of the communication delays which increase contention and cause many aborts. In Silo, when distributed update transaction successfully finishes the first phase of the 2PC protocol, it locks the affected rows until it completes the second phase. Any transaction attempting to access the locked rows will be aborted.

3.6.3 Summary

Overall, different distributed deployments of Silo, a main-memory optimized system, exhibit the same behavior as Shore-MT, in line with the model described in [Section 3.1](#), even though the designs of these two systems are very different. Furthermore, performance trends in the experiments with increasing percentage of multisite transactions in Silo are even more clear as it is a much leaner system with fewer components that interact with each other. For example, when we switched to shared memory communication mechanism for Shore-MT prototype, the communication overhead has diminished significantly. On the other hand, even shared memory communication adds significant overhead to the read-only distributed transactions when many instances need to be involved in a transaction. Additionally, since Silo relies on short critical sections to achieve high performance, it is very sensitive to the increase in their effective length caused by the distributed update transactions.

3.7 Summary and discussion

Modern multisoocket multicore servers are characterized by abundant hardware parallelism and variable communication latencies. This non-uniformity has an important impact on OLTP databases and neither shared-everything configurations, nor shared-nothing designs, are an optimal choice for every class of OLTP workloads on modern hardware. In fact, our experiments show that no single optimal configuration exists: the ideal configuration is dependent on the hardware topology and workload, but the performance and variability between alternative configurations can be very significant, encouraging a careful choice. There is, however, a common observation across all experiments: **the topology of modern servers favors a configuration we call Islands**, which groups together cores that communicate quicker, minimizing access latencies and variability.

We show that **OLTP Islands provide robust performance under a variety of scenarios**. Islands, being hardware topology-aware, provide some of the performance gains of shared-

nothing databases while being more robust to changes in the workload than shared-nothing. Their performance under heavy skews and multisite transactions also suffers, but overall, Islands are robust under the presence of moderate skews and multisite transactions.

As for previous approaches, our experiments corroborate previous results in that **shared-everything OLTP provides stable but non-optimal performance**. Shared-everything databases are robust to skew and/or updates in their workloads. However, their performance is not optimal and in many cases, significantly worse than the ideal configuration. In addition, **shared-everything OLTP is likely to suffer more on future hardware**. As the hardware parallelism continues to increase, it becomes increasingly important to make shared-everything databases NUMA-aware. Also, **extreme shared-nothing OLTP is fast but sensitive to the workload**. Extreme shared-nothing databases, as advocated by systems such as H-Store, provide nearly optimal performance if the workload is perfectly partitionable. Shared-nothing databases, however, are sensitive to skew and multisite transactions, particularly in the presence of updates.

The percentage of distributed transactions in the workload is one of the main factors that determine the performance of any OLTP deployment. It directly depends on the partitioning scheme of data into logical sites that determine which transactions are going to be multisite. Depending on the number of multisite transactions in the workload, different hardware topologies favor different deployments. For perfectly partitionable workloads, such as single row reads or updates that are very common in web applications, fine grained configurations are an ideal choice since they incur no synchronization overheads. Many common workloads such as TPC-B and TPC-C we analyzed in [Section 3.5](#) have few multisite transactions and favor partitioned deployments whose optimal granularity depends on the specifics of the workload. In this case, socket-sized islands are a good choice and are commonly used in practice to improve scalability of IBM DB2 deployments [\[99\]](#). Finally, many complex workloads, including TPC-E benchmark [\[155\]](#), are not easily partitionable as they contain multiple tree schemas and transactions that access data from many different tables [\[152\]](#). In this case, even a very good partitioning scheme will generate many multisite transactions [\[32, 122\]](#). We further discuss the impact of good partitioning scheme on throughput in [Chapter 4](#).

4 Adaptive Transaction Processing

In [Chapter 3](#), we have demonstrated that different workloads favor different deployment configurations. Changes in the workload characteristics cause optimal configuration to change, which requires costly repartitioning of data across many physically partitioned instances. However, instead of paying the price of physical repartitioning, we can incorporate adaptivity inside a single transaction processing system. In this chapter, we present *ATraPos*, a scalable shared-everything system that minimizes the impact of inter-socket communication in the critical path of transaction execution (i.e., the sequence of actions that determine the duration of the transaction). *ATraPos* relies on *precise data partitioning and placement* to maximize locality of data accesses and on *adaptive repartitioning* to maintain data locality even when the workload changes.

ATraPos first partitions the data *logically*, by allowing only specific threads to access each data item, and then *physically*, by partitioning tables and indices with respect to the logical parts. It puts emphasis on the data locality by keeping the system state in hardware-aware data structures. These data structures are specially designed to require only socket-local data accesses in the critical path. *ATraPos* ensures stable performance by choosing the appropriate partitioning scheme, which maximizes resource utilization and balances the load. The choice is based on a cost model that takes into account a) the static data dependencies, b) the dynamic workload information, and c) the underlying hardware topology. Finally, *ATraPos* uses a lightweight monitoring mechanism to continuously track the transaction behavior. When it detects that the workload has changed, it adjusts the data partitioning and partition placement to guarantee high and predictable performance.

In this chapter, we first present the different design trade-offs for scalable transaction processing systems on multisoquets, in [Section 4.1](#). That analysis motivates our design of hardware-aware system components that we show to scale linearly for perfectly partitionable workloads in [Section 4.2](#). We present our hardware and workload-aware partitioning and placement scheme in [Section 4.3](#) and the adaptation mechanism in [Section 4.4](#). [Section 4.5](#) details an experimental evaluation of different aspects of the *ATraPos* prototype. Finally, we summarize the findings and explore directions of future work in [Section 4.6](#).

4.1 Design Trade-offs for OLTP on Multisocket Multicores

In this section, we analyze the behavior of various system designs when running transactional workloads on multisocket multicore servers. We compare centralized shared-everything, shared-nothing, and physiologically partitioned shared-everything designs on workloads that are perfectly partitionable and less amenable to partitioning. We show that none of these designs can fully exploit the multisockets due to data sharing across sockets.

We run the experiments presented in this section on the octo-socket server described in [Table 3.1](#). We use the Shore-MT storage manager [71] (introduced in [Section 3.2.1](#)) and Intel's VTune Analyzer XE [64] for profiling.

4.1.1 Design Options

Centralized shared-everything. We evaluate the traditional shared-everything configuration by running Shore-MT as a single process using on available processor resources. In this case, all data structures accessed by transaction execution threads are centralized, e.g., the lock manager, the log, and the buffer pool. We enable the optimizations that are beneficial to the workloads we run, including speculative lock inheritance [70] and optimized logging using Aether [72].

Shared-nothing. We benchmark two shared-nothing configurations by running multiple instances of Shore-MT. All instances communicate using the thin distributed transaction execution layer described in [Section 3.2.1](#). Specifically, we simulate the *fine-grained shared-nothing* architectures, by running one instance of Shore-MT per processor core. Each record and page are touched by a single thread, while locking and latching are disabled for read-only workloads. For workloads that contain updates, we still need to use locking. We also test a *Island shared-nothing* deployment configuration, having one instance per processor socket, where locking and latching are enabled.

PLP. One of the main problems of centralized shared-everything systems on multicores is the contention in the lock manager. This problem can be eliminated by using physiological partitioning (PLP) [118, 119]. PLP first logically partitions the data and assigns each partition to a separate thread. Transactions are decomposed into small actions, which are routed to the relevant threads. Each thread contains a local lock table that eliminates the need to access the centralized lock manager for the majority of locks that each transaction needs to acquire. Eliminating the lock manager bottleneck exposes the bottleneck of latching on database pages. PLP removes this bottleneck by using multi-rooted B-trees and seamlessly changing the record insert operation. Multi-rooted B-trees partition the original B-tree by having one root per each logical partition. All data pages are pointed by a single leaf page. Since subtree accesses are thread-local, both B-tree and data page accesses can be latch-free. PLP scales very well on single processor systems [119].

4.1.2 Perfectly Partitionable Workloads

We start with a simple perfectly partitionable workload where each transaction reads one row from a table that contains 10 integer columns. Different transactions in this workload have no dependencies or conflicts, so the performance of a scalable system should increase linearly with more resources. We choose this workload because it clearly illustrates structural problems of shared-everything designs on multisockets. We run the benchmark for the shared-nothing and island shared-nothing configurations, the traditional centralized shared-everything configuration, and PLP. We use a dataset of 800K rows, equally divided between the participating instances, for various numbers of processors (1, 2, 4, and 8 processor sockets).

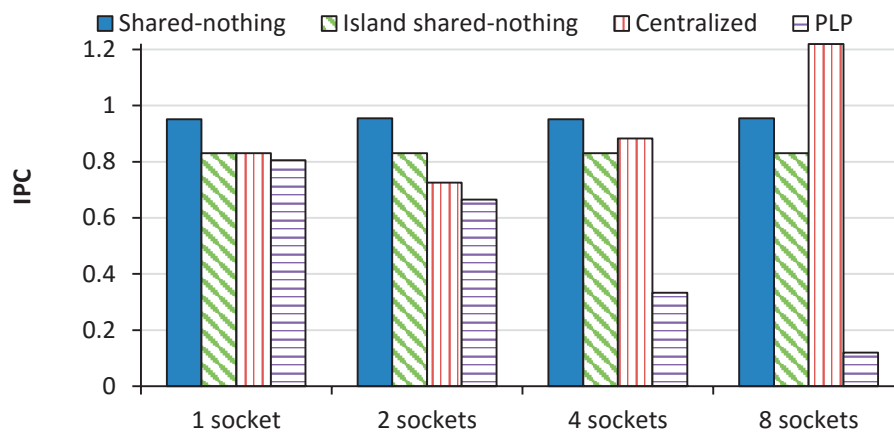


Figure 4.1: Instructions retired per cycle.

In Figure 4.1, we evaluate how well the above configurations use the available processor resources by measuring the number of retired instructions per cycle (IPC). Although we use a processor that can achieve up to 4 IPC, OLTP workloads can barely exceed 1. Low IPC is a general characteristic of OLTP [140, 152] due to the large instruction footprints and unpredictable data accesses.

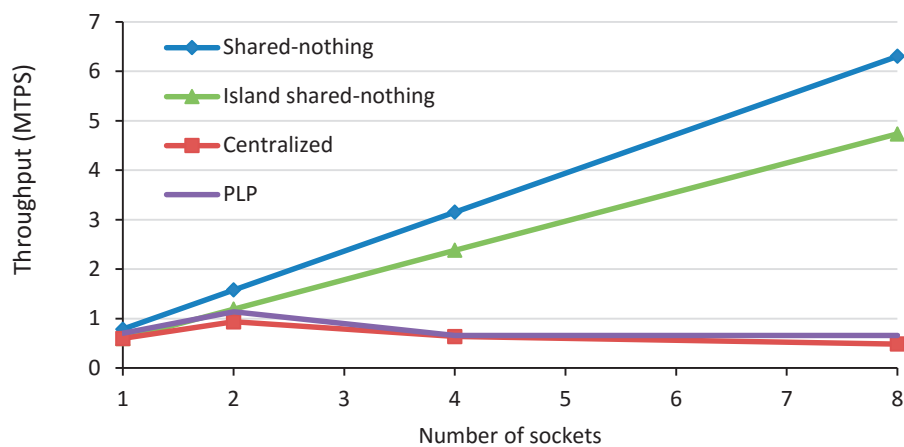


Figure 4.2: Throughput of the shared-nothing, centralized, and PLP architectures.

Chapter 4. Adaptive Transaction Processing

The shared-nothing architectures have constant IPC for all configurations. As we see in [Figure 4.2](#), which shows the throughput of the four configurations as we increase the number of sockets, they scale linearly because the requests are completely independent from each other and instances do not need to exchange messages to execute them.

When we examine the traditional centralized architecture, we observe a slight decrease in IPC when we go from 1 to 2 sockets followed by an increase when we go to 4 and 8 sockets, where IPC exceeds 1.2. However, in these cases, high IPC is due to high cache hit rates while waiting to acquire contended locks. The time wasted on waiting is the reason why the throughput decreases with more sockets in [Figure 4.2](#). This effect is more pronounced as the number of threads in the system increases.

When we run the perfectly partitionable microbenchmark using PLP on more than one socket, we observe a performance degradation similar to the centralized configuration. However, the trends on the IPC graph are completely different. The striped bars in [Figure 4.1](#) indicate large drops in IPC due to accesses to centralized data structures that are implemented using atomic compare-and-swap (CAS) instructions. While CAS instructions are executed efficiently on the same socket, they become very expensive across sockets, as they require accessing cache lines on remote processors.

Implication: Accessing any centralized data structure in the critical path is a potential bottleneck on multisockets.

4.1.3 Workloads That Are Less Amenable to Partitioning

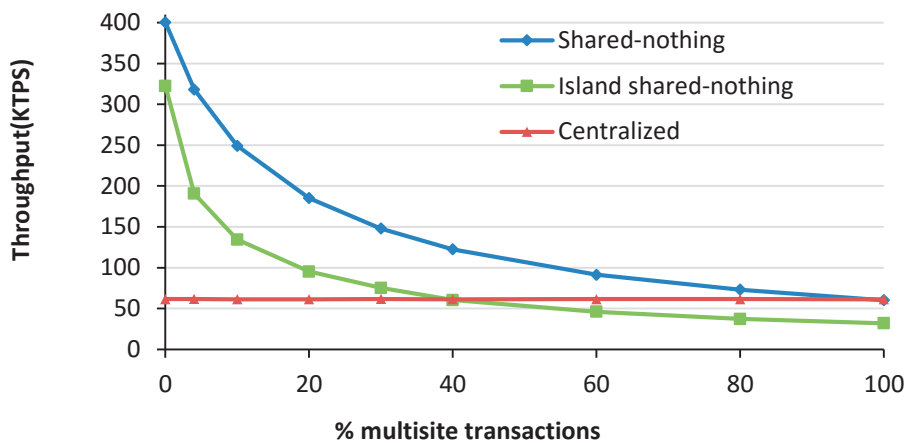


Figure 4.3: Throughput of different deployment configurations as percentage of multisite transactions increases.

While the shared-nothing architectures exhibit great performance on perfectly partitionable workloads, they suffer when the workload is not as partitionable. We illustrate this problem with a microbenchmark that updates 10 rows (see [Section 3.2.2](#) for the detailed description).

4.1. Design Trade-offs for OLTP on Multisocket Multicores

Table 4.1: Throughput (in transactions per second) for various memory allocation policies.

Policy	Socket1	Socket2	Socket3	Socket4	Socket5	Socket6	Socket7	Socket8
Local	6992	7028	6913	7075	6991	7029	7016	7036
Central	6591	6643	6774	6645	6578	6839	6816	7018
Remote	6521	6774	6532	6775	6752	6588	6773	6575

We run these transactions on the shared-nothing configuration, the Island shared-nothing configuration, and the traditional shared-everything configuration. In all cases, we use a dataset of 800K rows, equally divided between the participating instances.

In Figure 4.3, we plot the throughput when we vary the percentage of multi-site transactions from 0 to 100. We use shared memory communication channels, which are significantly faster than other communication mechanisms that involve the operating system, such as UNIX domain sockets and named pipes. However, we still observe a significant drop in the performance of partitioned systems as the fraction of multisite transactions increases. The reason is that they execute multisite transactions as distributed transactions.

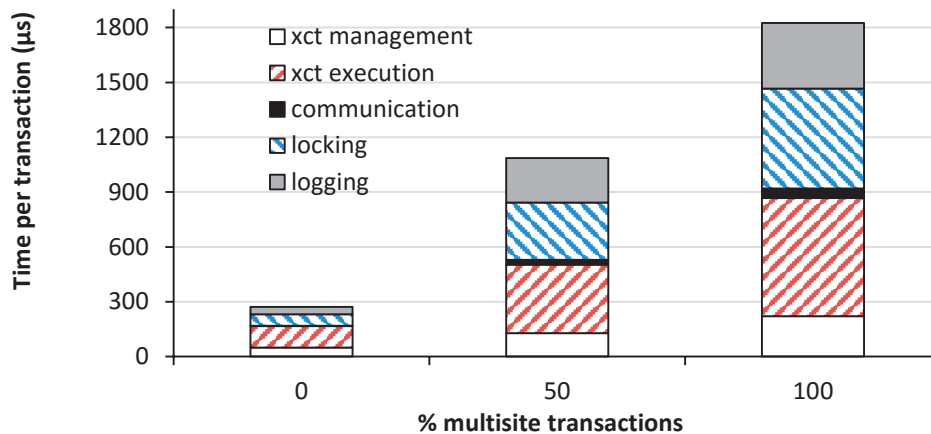


Figure 4.4: Time breakdown for Island shared-nothing configuration.

In Figure 4.4, we analyze the overheads of distributed update transactions by breaking down the execution time to different system components as we vary the percentage of multisite transactions for the Island shared-nothing configuration. The breakdowns are similar for the shared-nothing configuration. As we increase the percentage of multisite transactions, we see a significant increase in time spent in all components, especially in logging and locking.

Implication: Even with fast interprocess communication, the overhead of distributed update transactions limits the benefits of shared-nothing designs to perfectly partitionable or read-only workloads.

4.1.4 Accessing Remote Memory

One significant advantage of shared-nothing configurations, where instances run within a single processor socket, is the ability to achieve perfect NUMA locality by allocating all memory in the local NUMA node. In this section, we quantify the impact of memory allocation on the performance. We run one Shore-MT instance per socket and change memory allocation policy using the Linux utility `numactl` [96]. We test the system in 3 different settings: 1) each instance allocates memory in the local NUMA node, 2) all instances allocate memory in one NUMA node, and 3) every instance allocates memory in a different remote NUMA node.

We use a microbenchmark that reads 100 rows chosen randomly from a 1 million row dataset (1.5GB), which is enough to fill the memory of a large NUMA node in our server (32GB). We choose data randomly to 1) minimize the chance of a data hit in the last level cache and 2) limit the effectiveness of data prefetchers. We summarize the performance in terms of the throughput in Table 4.1. When memory is allocated locally (Local), throughput of each instance is within 1% of the average for all instances. When we allocate all memory on a single node (Central), for example on Node 8, instance 8 achieves throughput similar to all local cases, while other instances lose 2.5-6.2% of the performance. Finally, when every instance accesses remote memory (Remote), the performance is 3.3-7% worse compared to the local case. Experiments with transactions that read fewer rows show smaller differences in throughput, while the ones that read more rows show similar performance drops.

To explore the causes of these performance drops for different configurations, we use the Intel's Performance Counter Monitoring tool [66] to examine the interconnect utilization. We measure that the ratio of interconnect (QPI) to memory controller (IMC) data traffic is 0.01 for the local case, in contrast to 1.36 for the central case, and 1.49 for the remote case. Total utilization of all QPI links for accessing memory and maintaining cache coherence increases from 13Gb/s for local node allocation to 21 Gb/s and 22 Gb/s, respectively. Even in the case where all instances allocate memory on a single node, QPI links are lightly utilized with the most used link being utilized at 14%. The behavior of analytical queries is completely opposite: with memory bandwidth being the critical resource [127].

Implication: In contrast to the performance bottlenecks of accessing the shared data structures that are often found in remote caches, the performance impact of accessing remote main memory is limited to less than 10% and is not critical.

4.2 Hardware-aware System Design

As we show in the previous section, the state-of-the-art techniques that achieve scalability on multicores are not sufficient for multisoquets. This is caused by the bottleneck of accessing the centralized data structures in the critical path, e.g., the list of active transactions and various mutexes. Sharing data among threads that run on different sockets is expensive due to the cost of cache coherence and high latency of accesses to cache lines on remote sockets. ATraPos

solves this problem by partitioning these structures among sockets to increase the locality of accesses. This section details our general approach to hardware-aware data structures.

Critical path. Most centralized data structures in a typical storage manager are used for maintaining the global system state and are protected by read/write locks. Typically a transaction acquires a lock in read mode for a short period of time in order to change state, e.g, a transaction acquires volume read lock during the initialization phase. This is a fairly inexpensive operation on a single chip, but becomes increasingly expensive when we need to update data that is located on a remote chip or in memory. These locks are never acquired in write mode in the critical path of transaction execution. They are only used in write mode by threads performing background tasks, e.g., checkpointing, to ensure that no transaction changes state during this operation. Hence, we use the insight that the thread executing a transaction does not require a global view of the system state in the critical path. Instead, it accesses the local view only taking advantage of the locality.

Shared locks. We reduce the cost of acquiring read locks by replacing centralized read/write locks with partitioned NUMA-aware ones. In this design, we have one read/write lock for each processor socket. This way, acquiring a read lock entails accessing data cached on the local socket or stored in the local memory node. Additionally, there is less contention as the lock is shared only by the threads running on a specific processor socket. Acquiring write locks is a significantly less frequent operation and does not occur in the critical path. For example, a write lock on the checkpoint mutex is required only when the checkpointing procedure is running to ensure that no transaction has changed state (committed or aborted). In the centralized case, acquisition requires grabbing one write lock, while in the partitioned case it requires grabbing a write lock on every socket.

List of transactions. When a transaction starts, it is added to the list of active transactions and it stays there until it is completed. In Shore-MT, this structure is a lock-free list that requires a transaction to do one compare-and-swap on the list head to add itself to the list. When the system is running over many sockets, and especially when it is executing short-lived transactions, this operation becomes very expensive. ATraPos greatly reduces this cost by using a separate list of transactions for each socket, which makes the process of adding and removing elements from the list socket-local. In this way, accessing the list of transactions in the critical path never requires inter-socket memory access. Background operations that need to traverse the whole list of active transactions, such as checkpointing and page cleaning, simply need to go through all local lists. Furthermore, these accesses can be parallelized by using multiple threads that perform background operations on a single socket or a group of sockets.

Thread binding. In ATraPos we exploit information about the underlying hardware to further improve scalability and performance. On top of data partitioning to ensure locality, we bind threads to specific processor cores and cache information about their socket. This ensures that each thread always accesses the same partition of any partitioned data structure to guarantee

correctness. For example, each transaction is removed from the list of active transactions by the same thread that added it, which ensures that both operations are performed on the same partition. Each partition is always local to the socket where the thread is running on.

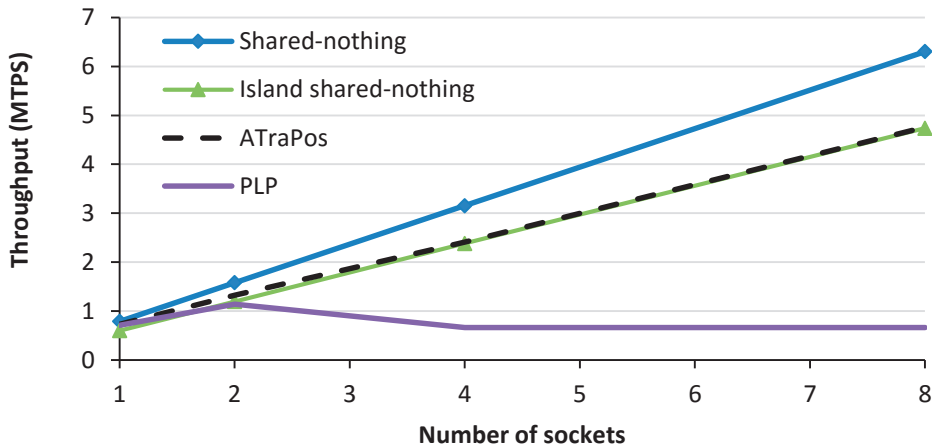


Figure 4.5: Throughput of a perfectly partitionable workload.

Proof of concept. In Figure 4.5, we repeat the experiment of Figure 4.2 and include ATraPos. Since we remove expensive accesses to the centralized data structures from the critical path, ATraPos can scale over multiple sockets and make full use of the fact that the workload is perfectly partitionable. In this case, we use the *naïve partitioning scheme* where a table is range partitioned across cores with one partition per core. ATraPos matches the performance of the Island shared-nothing configuration that has perfect locality because it runs one instance per socket. Both of these architectures scale similarly to the shared-nothing architecture.

4.3 Workload-aware Partitioning and Placement

Hardware-aware data structure enable scalability for perfectly partitionable workloads. In that case, ATraPos scales linearly since each worker thread operates independently on its own data partition. For more complex workloads, however, we need to partition and place the data on cores in a way that reduces the inter-socket data exchange as much as possible.

In this section, we first discuss the intuition behind our partitioning scheme. Then, we present the cost model and the search strategy that ATraPos uses to decide the appropriate partitioning and placement scheme.

4.3.1 Factors Influencing Transaction Processing

There are a number of factors that we have to consider when choosing a partitioning scheme for an OLTP workload. Typically, the database schema is fixed and known a priori. In addition, most or all transactions fall into one of the predefined transaction classes expressed as parameterized stored procedures [144]. Furthermore, the input parameters of a transaction point

to all data items a transaction is going to access (with the exception of the items accessed through the secondary indices). ATraPos exploits all this knowledge about the workload and the underlying hardware topology to efficiently choose a good partitioning scheme.

The goal of ATraPos is twofold: a) to maximize the CPU utilization and b) to minimize the transaction synchronization cost. We express the CPU utilization as the sum of work done by its individual cores. We model the synchronization cost of a transaction based on the placement of partitions that need to communicate at each synchronization point. We present the cost model in more detail in [Section 4.3.2](#).

Static workload information. We use database schema information, such as foreign keys, to extract the static data dependencies. We automatically infer the following static information about transaction classes from the transaction code: a) the number of actions that access each table, b) the dependencies between pairs of actions (via foreign keys of the tuples they access), and c) the number of synchronization points. A synchronization point in the transaction flow graph is the point where two or more actions need to exchange data. Its cost depends on which sockets the actions are running on and on the size of data they need to exchange. The synchronization cost of a transaction is the sum of the costs of all the individual synchronization points it includes.

Dynamic workload information. We track the dynamic aspect of a transactional workload by tracking the amount of work that is done by each partition and which partitions are involved in each synchronization point. This information allows us to estimate the core utilization and synchronization costs for any partitioning and placement scheme and to choose the best scheme for the current workload.

Hardware topology. The static and dynamic workload information already provides valuable pointers for deciding a good partitioning scheme. Additionally, ATraPos takes into consideration the underlying non-uniform hardware topology to specialize the partitioning scheme for each machine. This information can also be dynamic; as in the case that the system is running on a virtual machine whose available computing resources change over time.

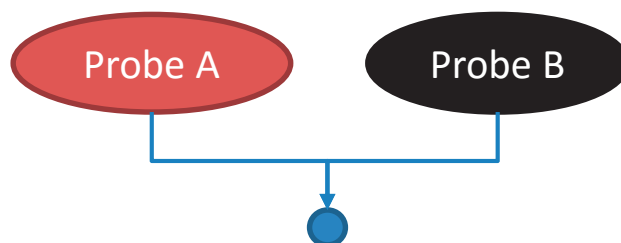


Figure 4.6: Simple transaction flow example.

Simple transaction example. The following example illustrates the impact of the various factors in our partitioning scheme. We use two tables, A and B, and the following transaction whose input parameters are ID_a and ID_b:

Chapter 4. Adaptive Transaction Processing

```
select * from A where pk_a = ID_a;
select * from B where pk_a = ID_a
        and pk_b = ID_b;
```

We illustrate the execution plan of this transaction in Figure 4.6. Figure 4.7 shows the throughput on various configurations. We use the centralized shared-everything and the PLP designs as baselines. We compare them against the naïve partitioning scheme introduced in Section 4.2 and the ATraPos model using the criteria discussed above.

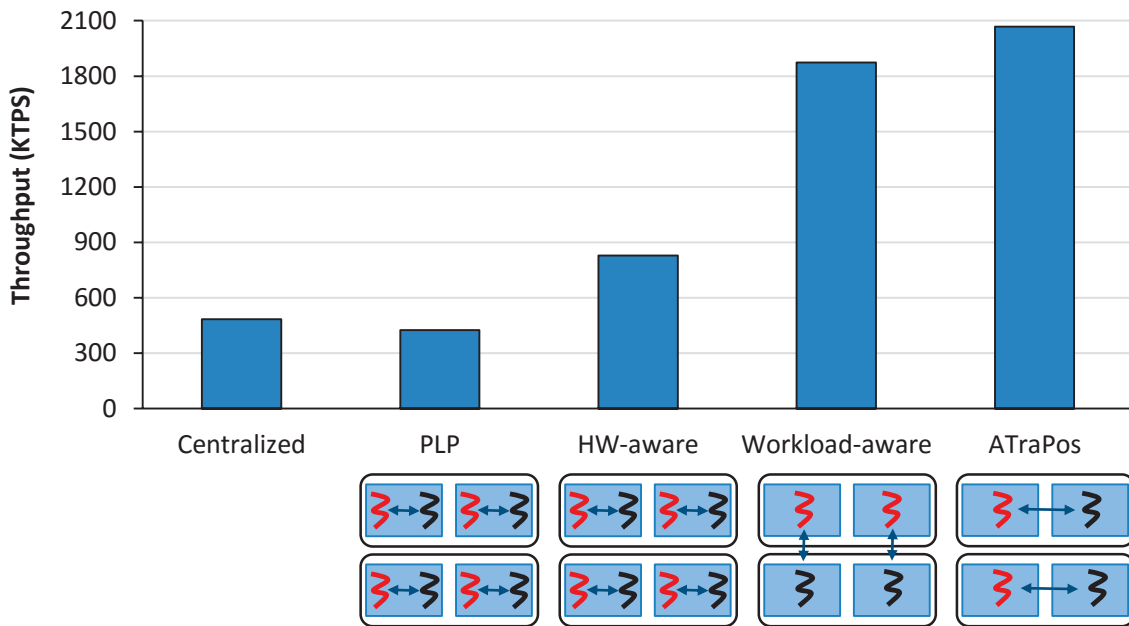


Figure 4.7: Throughput of a simple transaction with varying partitioning and placement strategies.

The naïve partitioning scheme (*HW-aware*) creates one partition of each table per processor core. As both tables have the same number of rows and we use range partitioning, this scheme achieves perfect locality for this simple workload. The hardware-awareness of the underlying storage manager produces 1.7-2x better performance compared to the baseline configurations. However, it suffers from oversaturation as in every core there are two partitions that contend for resources. To eliminate oversaturation, we place only one partition per core. In this case, we create 40 partitions for each table and compare two placement strategies: 1) the partitions are placed in a hardware-oblivious manner (*Workload-aware*) and 2) the partitions are placed in a workload and hardware-aware way (*ATraPos*). By removing oversaturation, we achieve 2.3x better performance even though the partitions of tables A and B are spread over 4 sockets each. However, this placement incurs inter-socket synchronization for every transaction. Therefore by placing dependent partitions on the same socket, the performance improves by 10%. Overall, for this workload we can get over 4x performance improvement by using hardware and workload-aware partitioning and placement.

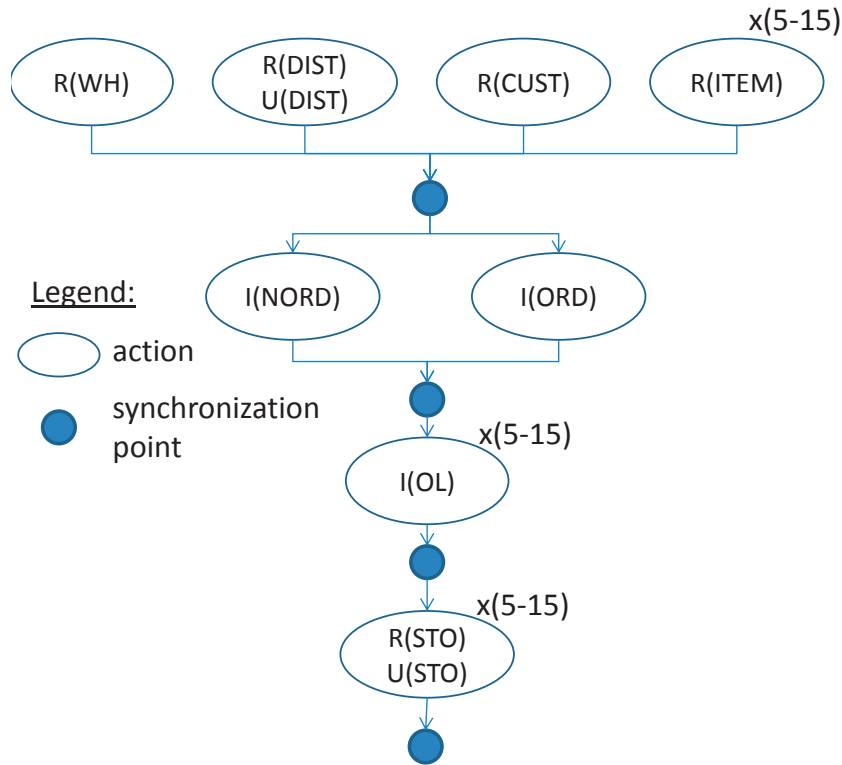


Figure 4.8: Transaction flow graph for the TPC-C NewOrder transaction.

Complex transaction example. In this example, we briefly illustrate a more complex scenario, i.e., the `NewOrder` transaction in the TPC-C benchmark and explain the challenges in choosing a good partitioning and placement scheme. This transaction models the ordering for 5 to 15 items from one warehouse and Figure 4.8 depicts its execution plan.

The `NewOrder` transaction accesses 8 tables, and has fixed and variable parts. Both of these parts contain read, insert, and update operations, denoted as R, I, and U, respectively. The fixed part accesses one tuple each from 5 different tables, while the variable part accesses one tuple per ordered item from 3 different tables. Furthermore, in our transaction flow graph, we have four synchronization points that all, except for the second, involve a variable number of partitions. The number of partitions that need to synchronize depends on the number of items in the order. Regarding the partitioning decision, we have to assign more CPU cores to tables that are accessed more times. Finally, regarding the partition placement policy, we should place the partitions that are involved in the same synchronization point on cores that belong to the same socket to reduce the synchronization overhead.

Conclusions from examples. From the previous two examples, we can conclude that using the naïve partitioning scheme is not enough; both workload and hardware-awareness of the partitioning mechanism are important for achieving high performance. Our system uses the data-oriented transaction execution model [118] where each worker thread operates on a single partition of a specific table. Using well-known partitioning schemes for TATP and

TPC-C workloads (which are practically identical to the naïve partitioning in our system) causes severe overloading.

4.3.2 Cost Model

ATraPos uses a partitioning and placement scheme that achieves two goals: maximal resource utilization and minimal transaction synchronization overhead. One of our main metrics is *balanced resource utilization*. In the case of multicore systems, we define balanced resource utilization as the ability to avoid overloading any particular core. If some of the cores are 100% utilized, they cannot process more requests. By balancing the load, we aim to leave the same amount of free resources on each core so that they can process proportionally more requests and the system can achieve higher throughput. Our other metric is the *transaction synchronization overhead*. We assess the quality of a placement scheme according to its ability to reduce the inter-socket communication costs; i.e., the smaller these costs are, the better the placement scheme is.

We express the resource utilization metric RU for the workload trace W and the partitioning and placement scheme S as:

$$RU(S, W) = \sum_c |RU(c) - RU_{avg}|$$

where $RU(c)$ is the utilization of a particular core c and

$$RU_{avg} = \frac{\sum_c RU(c)}{N}$$

is the average utilization for all N cores. We compute the utilization of one core c as:

$$RU(c) = \sum_{p \in P_c} \sum_{a \in A(p)} C(a)$$

where P_c is the set of partitions that are placed on core c , $A(p)$ is the set of all actions that use partition p , and $C(a)$ is the time we need to execute action a .

We compute the transaction synchronization overhead $TS(S, W)$ for the workload trace W and the partitioning and placement scheme S as

$$TS(S, W) = \sum_{T \in W} Sync(T)$$

where $Sync(T)$ is the synchronization cost of a single transaction T . We express this cost with the following formula:

$$Sync(T) = \sum_{s \in S(T)} Cost(s)$$

where $Cost(s)$ represents the synchronization cost for a particular synchronization point s . We express cost $Cost(s)$ of the synchronization point s as:

$$Cost(s) = (n_{socket}(s) - 1) * Data(s)$$

where $n_{socket}(s)$ is the number of unique sockets that actions in s run on and $Data(s)$ is the cost of the data exchange operation in this synchronization point. The synchronization cost of two actions that are running on the same socket is zero, while when they are on different sockets it can be a considerable cost depending on their distance. The data exchange cost is expressed as:

$$Data(s) = Distance(s) * Size(s)$$

where $Distance(s)$ is the average communication cost between the participating sockets and $Size(s)$ is the size of data that has to be exchanged.

Algorithm 1 Choose Partitioning

```

1: // Greedily choose initial partitioning S
2: repeat
3:    $Good \leftarrow true$ 
4:   for all underutilized core  $c$  do
5:      $S_c \leftarrow$  move a sub-partition to  $c$ 
6:     if  $RU(S_c, W) < RU(S, W)$  then
7:        $S \leftarrow S_c$ 
8:        $Good \leftarrow false$ 
9:     break
10: until  $Good$ 
11:  $S_{part} \leftarrow S$ 

```

4.3.3 Search Strategy

The goal of the ATraPos partitioning and placement mechanism is to be able to quickly find a good solution that will maximize the throughput of the system for the current workload. To that end, we use a two step exhaustive search strategy that first chooses the partitioning scheme and then decides a good partition placement.

In the first step, we use information about the current load for sub-partitions of every existing partition to choose a new partitioning scheme. As shown in Algorithm 1, we group sub-partitions into new partitions that balance the resource utilization according to our cost model. We initially assign one new partition per core in a greedy fashion: we first estimate the target average utilization and keep adding sub-partitions until we exceed that load. Then, move to the next core. Next, we iteratively try to improve the assignment by choosing a new partition placed on a core with the highest under-utilization, moving a sub-partition of the same table to that partition, and recomputing the utilization metric. If an under-utilized core contains the only partition of a table, we place a sub-partition of another table on that core to

Chapter 4. Adaptive Transaction Processing

improve overall utilization. If the global utilization balance improves, we use this solution as the current best case and restart the search. We conclude the search when we cannot improve the overall utilization of the scheme by moving sub-partitions to under-utilized cores.

Algorithm 2 Choose Placement

```
1:  $S \leftarrow S_{part}$ 
2: repeat
3:    $Good \leftarrow true$ 
4:   for all  $s$  such that  $C(s) > 0$  do
5:      $S_s \leftarrow$  switch partitions to minimize  $C(s)$ 
6:     if  $TS(S_s, W) < TS(S, W)$  then
7:        $S \leftarrow S_s$ 
8:        $Good \leftarrow false$ 
9:     break
10: until  $Good$ 
11:  $S_{opt} \leftarrow S$ 
```

After finding the partitioning that balances the resource utilization, we choose the placement that aims to reduce the synchronization overhead using Algorithm 2. We start from a placement that evenly distributes partitions of every table to different sockets. We iteratively examine various alternatives that move the partitions involved in a costly synchronization point to the same socket by switching them with other partitions. If the switch lowers the global synchronization cost, we keep the placement as the new best and restart the search. We reach the solution when we can no longer improve the placement.

4.4 Adaptive Dynamic OLTP Design

In this section we illustrate how we leverage the cost model described in Section 4.3.2 to adapt to any changes in the workload properties or hardware topology. While the hardware topology and the static workload characteristics are inferred beforehand, the dynamic properties are measured at runtime. Our goal is to trace all the required information we use in our cost model in a lightweight manner.

Monitoring overhead. We minimize the monitoring overhead by storing the traces in thread-local data structures and aggregating system-wide traces periodically. In this way, we do not add unnecessary inter-socket accesses in the critical path. The global traces are collected by a special monitoring thread that is also in charge of deciding the best partitioning and placement scheme for the captured traces. To minimize the storage overhead, we discard the traces after each computation.

Monitoring data structures. Since both the number of tuples in a table and the number of transactions that arrive in a time period vary greatly across different workloads, the space overhead of the tracing structure should not depend on the dynamic characteristics of the workload. Hence, we choose to have fixed-sized tracing structures tied to the number of

elements in a partition. We use two thread-local arrays per partition: a) one that stores the cost of all actions executed by a specific sub-partition, and b) one that keeps the number of synchronization points executed for each local sub-partition. We initialize arrays based on the number of sub-partitions upon a new partition creation. In our experiments we use 10 sub-partitions per partition as it offers a good trade-off between the size of the arrays and the number of repartitioning operations needed to adapt to even the most drastic changes in the workload. Using more sub-partitions would increase precision at the cost of quadratic space increase required for keeping synchronization information. It will also increase the time it takes to evaluate the cost model, however, as this is done in a separate thread, it does not impose overheads on the normal processing. Fewer sub-partitions reduces overheads, however, it might require an additional round of repartitioning to adapt to extreme skew, e.g., when 50% of the load is targeting a single sub-partition.

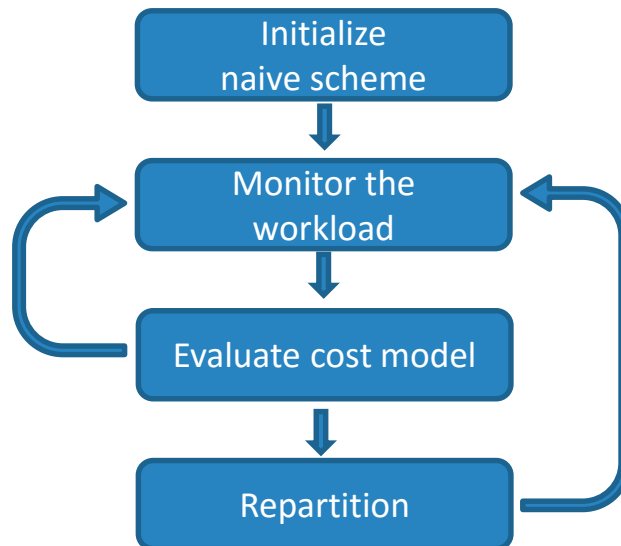


Figure 4.9: Adaptivity mechanism in ATraPos.

Detecting changes. ATraPos uses the lightweight monitoring mechanism to be able to adapt to any change in the workload. The workflow is illustrated in Figure 4.9. When the system initializes, it has no information about the dynamic aspects of the workload so it sets up the partitions using the naïve partitioning scheme described in Section 4.2. ATraPos continuously monitors the workload using the array-based approach described above. It periodically aggregates the trace information using the monitoring thread and decides the optimal partitioning and placement scheme according to the cost model. Since changes in the workload may happen during different time intervals, ATraPos uses an adaptive approach where it tunes the time interval length based on the frequency of the workload fluctuations. When the workload is stable for a long time it increases the intervals, while upon having frequent workload changes it shortens them.

ATraPos starts from a 1 second interval and monitors the throughput. If the throughput is within 10% of the average of the previous 5 measurements it doubles the monitoring interval. After each monitoring interval, it checks if the throughput difference has exceeded the threshold; if it has, it evaluates the model, otherwise it increases the monitoring interval. If the result of the evaluation is the decision to repartition, ATraPos resets the monitoring interval to 1 second. Since repartitioning in ATraPos is a lightweight operation, it makes a decision to repartition whenever the optimal partitioning and placement scheme for the observed traces differs from the current scheme. We prevent needless repartitionings for very dynamic workloads by evaluating the cost model only if the workload patterns are stable.

Repartitioning. One of the design goals of ATraPos is to quickly adapt to any change. To that end, when we decide on the new partitioning and placement scheme, we generate a set of repartitioning actions and pause the execution of regular actions while we execute them. We do not interleave the execution of repartitioning and regular actions because interleaving different types of actions causes dependencies between actions that add unpredictable delays. A repartitioning action can either be a *split* or a *merge*, and it modifies both the logical and physical representation of the data. The split action divides an existing partition into two new partitions at a specific key, while the merge action creates a new partition by merging two existing partitions. These operations modify the physical multi-rooted B-trees, the logical partition-local structures such as action queues and lock tables, and the global partitioning information. The multi-rooted B-tree structure makes the repartitioning actions very lightweight as they require only one traversal of the tree structure and modification of a couple of nodes on each level of the tree. After we complete all the repartitioning actions, we empty the partition-local monitoring data structures and restart the monitoring operation.

The downside of this approach is the fact that the regular actions are completely stopped during repartitioning which causes noticeable drop in the throughput. Also, since we only modify the tree structure during repartitioning and do not access majority of the data in the new partitions, the first few data accesses to the new partitions would incur additional latency as data is moved to the caches local to the new partition. An alternative approach to repartitioning that would incur less disruption to the normal processing involves performing repartitioning actions interleaved with regular actions one at a time to minimize the negative impact on normal processing. Exploring interleaved schedules of repartitioning and regular actions is an interesting direction for future work.

4.5 Evaluation

In this section, we present the detailed experimental evaluation of the system using both microbenchmarks and standard benchmarks such as TPC-C and TATP. We designed and implemented ATraPos on top of Shore-MT [71]. We show that ATraPos exploits hardware resources better than the state-of-the-art, providing a significant performance boost even when the workload changes.

4.5.1 Experimental Setup

Our experimental platform is the octo-socket server described in Table 3.1. We use memory mapped disks for both data and log files. All experiments run on Red Hat Enterprise Linux 6.4 (kernel 2.6.32) and we compile using GCC 4.4.7 with maximum optimizations.

We use microbenchmarks and the standard OLTP benchmarks TATP [110] and TPC-C [154]. The TATP benchmark models a mobile phone provider. Its schema contains 4 tables that are perfectly partitionable on the `SubscriberID` attribute. TATP uses a set of 7 transactions of 3 different classes. It contains read-only transactions that access only a single table (e.g., `GetSubData`), read-only transactions that access multiple tables (e.g., `GetNewDest`), and update transactions that access multiple tables (e.g., `UpdLocation`). In all experiments with TATP, we use a dataset with 800K subscribers (1.8GB). The more complex TPC-C benchmark models a wholesale supplier. There, we have 9 tables and 5 different transactions. In contrast to TATP, all TPC-C transactions require data from 3 or more tables. We use the TPC-C dataset with scaling factor 80 (13GB) in all experiments.

4.5.2 Improving Throughput on Standard Benchmarks with ATrapos

In our first experiment we demonstrate the significant performance boost that ATrapos achieves on the standard benchmarks TATP and TPC-C. The performance metric used is throughput, i.e., how many transactions the system executes per second. We compare ATrapos using partitioning and placement scheme against its version that only employs Islands-aware data structures and the state-of-the-art, PLP, both of which assign one partition of each table per processor core.

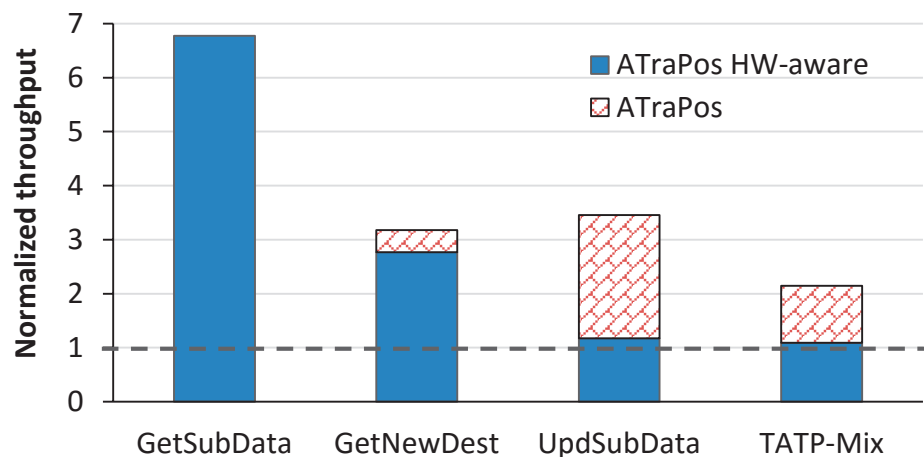


Figure 4.10: Improving throughput on the standard TATP benchmark with ATrapos compared to the state-of-the-art ($y = ATrapos/PLP$).

The graph in Figure 4.10 shows the behavior of ATrapos on the TATP benchmark. The y-axis depicts the throughput of ATrapos normalized over the throughput of PLP. In this way, the

y-axis represents the throughput improvement achieved by different components of ATrapos. We show results both for individual transaction types and the standard TATP transaction mix (denoted as TATP-Mix). As the `GetSubData` transaction is perfectly partitionable and all configurations place one partition of the `Subscriber` table per core, ATrapos achieves 6.7x improvement due to Island-aware data structures. Similarly, for the read-only `GetNewDest` transaction, here we need to access data from two tables, Island-aware data structures achieve improvement of 2.7x that rises to 3.2x when using the ATrapos partitioning and placement scheme. For other transactions, ATrapos achieves most of the throughput improvements due to a good partitioning and placement scheme. The improvement in performance for update transactions comes in large part from the decreased contention on the log since the better partitioning scheme of ATrapos creates fewer partitions, hence fewer threads are competing for the access to the log manager.

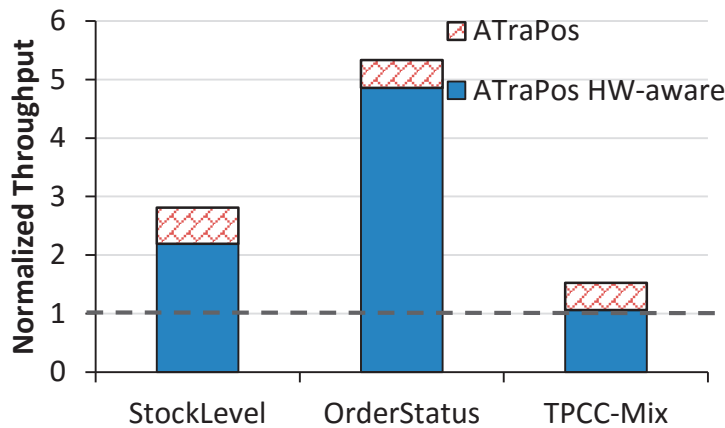


Figure 4.11: Improving throughput on the standard TPC-C benchmark with ATrapos compared to the state-of-the-art.

Figure 4.11 depicts the throughput improvement on the TPC-C benchmark. We plot the normalized performance of two configurations of ATrapos (over PLP) for the two read-only transactions of TPC-C as well as for TPCC-Mix. We observe a larger performance improvement of 5.3x for the lightweight `OrderStatus` transactions, compared to the 2.8x improvement for the heavyweight `StockLevel` transactions. This variation in performance improvement stems from the fact that the `OrderStatus` transactions benefits more from the Island-aware data structures. On the other hand, `StockLevel` benefits more from the better data partitioning that improves locality of the join that requires many data accesses. Finally, the throughput of TPCC-Mix improves by 50%.

Summary. ATrapos brings a significant improvement compared to the state-of-the-art for various types of workloads due to Island-aware data structures and its data partitioning and placement scheme.

Table 4.2: ATrapos monitoring brings negligible overhead.

Workload	No monitoring	Monitoring	Overhead (%)
GetSubData	4461960.1	4313524.2	3.32
GetNewDest	326249.9	325890.6	0.11
UpdSubData	64650	63994.5	1.01
TATP-Mix	276601.3	274019	0.93

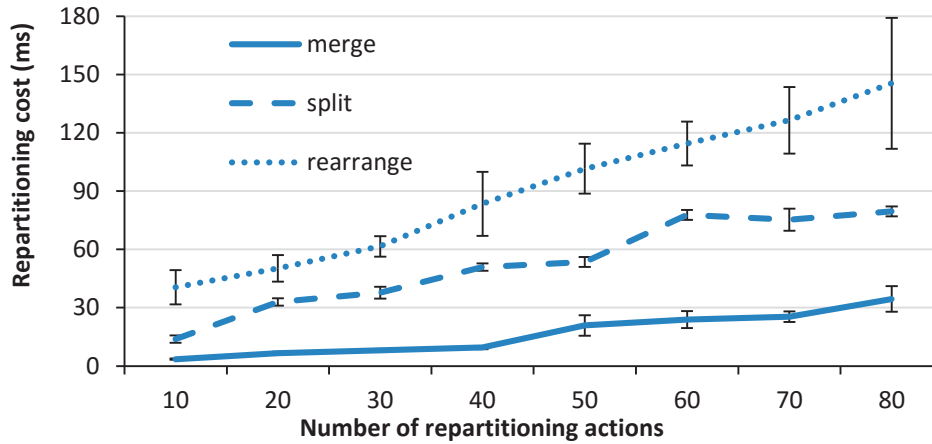


Figure 4.12: Scalability of ATrapos repartitioning mechanism.

4.5.3 Monitoring and Repartitioning Cost

Next, we demonstrate that the ATrapos monitoring and repartitioning mechanisms pose a negligible overhead.

First we quantify the monitoring overhead. To achieve this we test ATrapos in two modes: a) with monitoring enabled and b) with monitoring disabled. Table 4.2 shows the performance while running various transactions and the workload mix of the TATP benchmark as well as the overhead in percentages. In all cases, the monitoring mechanism poses a minimal overhead on throughput. The only transaction that is slightly affected is the GetSubData transaction where the throughput deteriorates by at most 3.32%. This occurs because GetSubData is a notably short transaction, hence the total number of actions that needs to be tracked per second by the monitoring subsystem corresponds to the worst-case scenario.

To quantify the repartitioning overhead, we use the following experiment. On a table of 800K rows and 10 integer attributes, we vary the number of repartitioning actions we trigger and measure the time it takes to complete each individual action. Figure 4.12 shows the results. For each case we show the average time of 10 repeated measurements with standard deviation. The merge operation combines two trees into one, the split divides one tree into two, and the rearrangement performs one split and one merge. As we see in Figure 4.12, the cost of all repartitioning sequences increases linearly with the number of repartitioning actions needed. The merge operation is always cheaper compared to the split operation. This is because the

latter performs more updates to the metadata. A rearrangement consists of one split and one merge. In this way, a sequence of rearrangements is hard to predict, because of the interference of splits and merges. In [Figure 4.12](#), we observe the trend of slowly increasing costs as we increase the number of operations. However, even the costliest repartitioning scenario (i.e., 80 rearrangements in our 80-core system) completes in less than 200 milliseconds.

Summary. ATraPos monitoring mechanism poses negligible overhead on the system performance. In addition, the repartitioning operations are lightweight and complete in a fraction of a second to ensure that ATraPos can quickly adapt the partitioning scheme to workload changes.

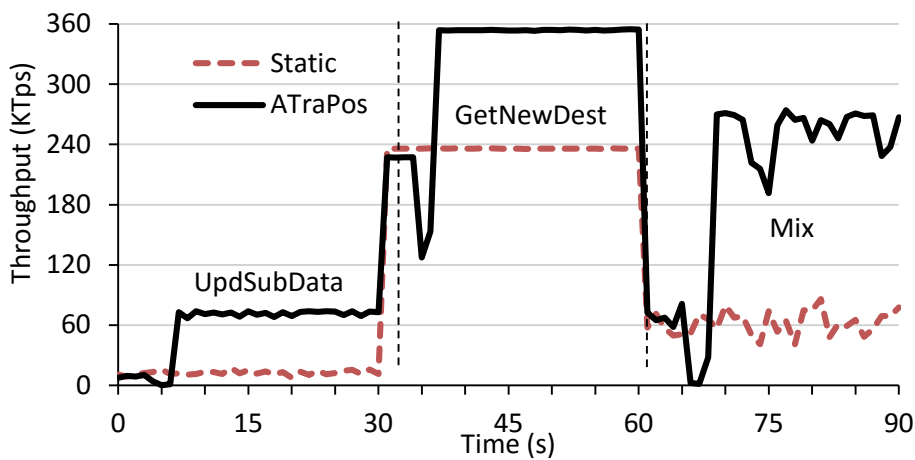


Figure 4.13: Adapting to workload changes.

4.5.4 Adaptive Behavior of ATraPos

Here, we demonstrate that ATraPos can successfully adapt to a) changes in the workload characteristics, b) skewed accesses to data, c) changes in the underlying hardware topology, and d) different frequencies of workload changes. As we have already shown that ATraPos outperforms the state-of-the-art approach, in this set of experiments we compare ATraPos to its static version where monitoring and adaptation are disabled.

Workload Characteristics

First, we test the behavior of ATraPos when the workload changes. We use TATP and every 30 seconds we switch to a different transaction type. Specifically, for the first 30 seconds we run only UpdSubData transactions; then for the next 30 seconds we run only GetNewDest transactions; and for the last 30 seconds we run the standard TATP-Mix. [Figure 4.13](#) depicts the results.

Every time the workload changes, ATrapos quickly adapts, i.e., within 5 seconds, boosting the throughput of the system significantly. For example, when during the first workload change throughput is 220 KTPS (thousands of transactions per second) for the first 5 seconds, ATrapos increases the throughput to 360 KTPS by monitoring and quickly detecting the workload change and subsequently reoptimizing data and thread placement.

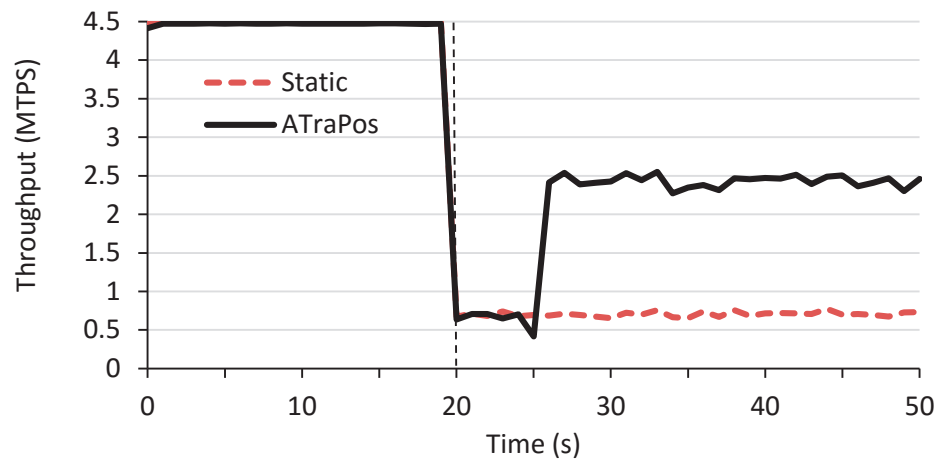


Figure 4.14: Adapting to sudden workload skew.

Data Skew

Figure 4.14 depicts the benefits of the adaptive ATrapos behavior when skew appears in the workload. In this experiment, we use the `GetSubData` transaction from the TATP benchmark. This transaction initially chooses uniformly distributed values from the whole dataset. After 20 seconds, we introduce skew by specifying that 50% of the requests go to the 20% of the data. The heavy skew causes the throughput to drop by $\sim 80\%$. ATrapos quickly detects the change and optimizes for the new workload characteristics. It manages to achieve 3x better performance than the static system.

Underlying Hardware Topology

The next experiment demonstrates the ability of ATrapos to gracefully adapt to hardware changes. In this case, we test the behavior when a processor fails. We simulate the failure of a processor P by excluding all cores of P and leaving them idle. We use the `GetSubData` transaction from TATP since it is a very short transaction that is sensitive to the changes in the environment. Figure 4.15 shows that at the time of the simulated processor failure (one 10-core processor fails at the 20th second), the static system fails to optimally use the rest of the available hardware. It still uses a partitioning plan that assumes 80 processor cores are available. Therefore, it implicitly overloads 1 full processor (with 10 cores) that now needs to satisfy not only its own requests but also the requests that would normally go to the processor that failed. This causes a 22% drop in throughput. On the other hand, ATrapos detects the

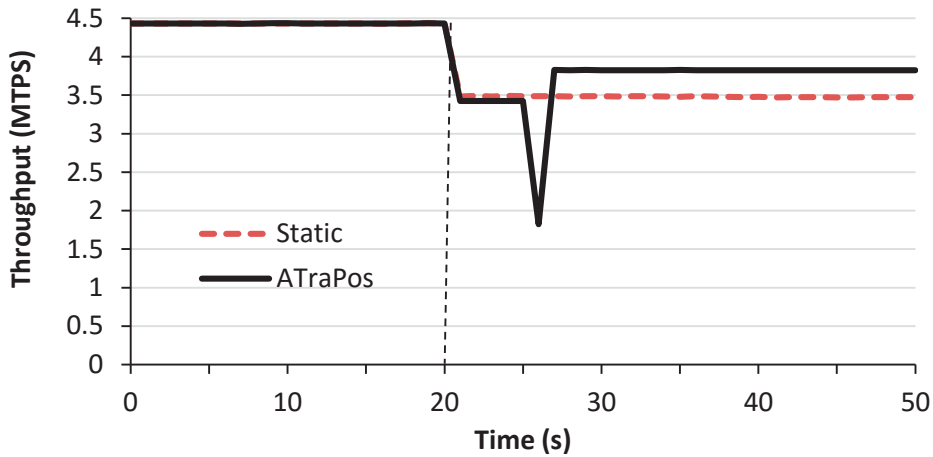


Figure 4.15: Adapting to hardware failures.

change in the underlying hardware topology and repartitions the data to create one partition for each of the 70 available cores. The optimized repartitioning removes the overloading effects and improves throughput by 11%.

Frequency of Changes

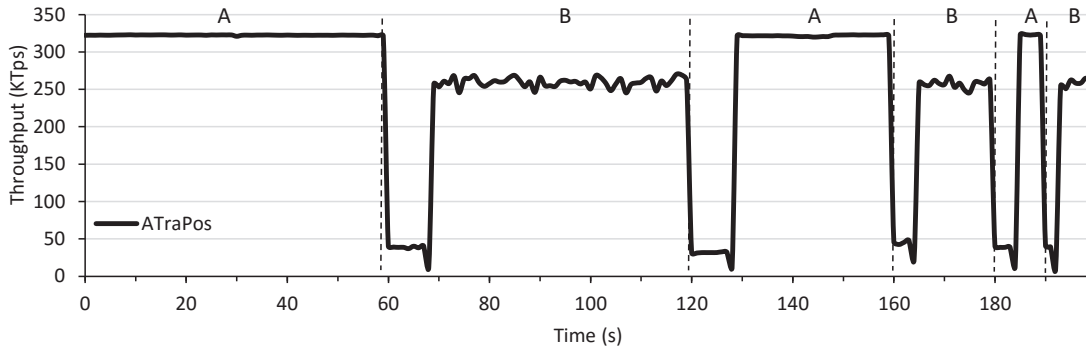


Figure 4.16: Adapting to frequent changes.

In our last experiment, we demonstrate how ATraPos gracefully adapts to workload fluctuations. We test a dynamic scenario that consists of workloads *GetNewDest* and *TATP-Mix* from the TATP benchmark, denoted as *A* and *B*, respectively, in Figure 4.16. Workload *A* is active for the first 60 secs. ATraPos continuously monitors the throughput and as long as it remains stable, it relaxes its monitoring interval; during the first 60 secs the interval is 1 sec and it gradually becomes 8 sec (this is the upper bound in our prototype). When the workload shifts to workload *B* at the 60th sec, ATraPos manages to identify the throughput degradation within 8 seconds. Then, it adjusts to the optimal partitioning scheme for workload *B* and it sets its monitoring interval back to 1 sec so it can be more alert until it realizes that the workload is stabilized; when this happens, it gradually increases the monitoring interval again.

As [Figure 4.16](#) depicts, when frequent workload fluctuations occur, ATraPos remains alert (keeping the monitoring interval low) and it quickly adapts to the changes. For example, in the last two workload shifts ATraPos adapts within about 2 seconds. Overall, ATraPos manages to continuously adapt and autonomously reconfigure its monitoring setup to follow the workload fluctuations.

Summary. By monitoring the workload and available resources in longer intervals, and by graciously adapting its data and thread placement, ATraPos provides predictable performance for a wide variety of dynamic workloads.

4.6 Summary and Discussion

In this chapter, we show that ignoring non-uniform hardware topology severely limits scalability of transaction processing systems. We identify the main shortcoming of the state-of-the-art shared-everything transaction processing systems on multsocket multicore servers as the existence of centralized data structures in the critical path. We address this problem in ATraPos by systematically making all data structures accessed in the critical path hardware-aware. This allows us to achieve linear scalability for perfectly partitionable workloads.

To address the workloads that are not perfectly partitionable, ATraPos includes a dynamic lightweight monitoring and repartitioning mechanism. Our partitioning mechanism takes into account static and dynamic workload characteristics as well as the hardware topology to choose a good partitioning and placement scheme for the current workload. When workload or hardware characteristics change, it quickly adapts the current partitioning scheme to the new environment. In this way, ATraPos offers robust performance on a variety of dynamic transactional workloads on today's and upcoming non-uniform hardware platforms. The presented adaptivity techniques can also be applied to other transaction processing architectures, with the modifications we describe in the next two paragraphs.

Coarse-grained shared-nothing. We can apply the ATraPos cost model to the physically partitioned shared-nothing architecture with a few modifications. Since data is physically partitioned, the primary cost in the model is the cost of distributed transactions, as in previously proposed partitioning methods for the physically partitioned systems [32, 122]. Similarly, the cost of repartitioning includes the cost of physical data movement from one instance to another. This cost is generally much higher than the repartitioning cost in the logically partitioned systems. The resource estimation part of the model can be used to determine sizes of individual instances in the system if amended with the cost model for the contention among different threads in larger instances. Given the larger repartitioning cost, interleaving repartitioning and regular actions would be more beneficial [138].

Fine-grained shared-nothing. The ATraPos model can also be applied to fine-grained shared-nothing systems that are aware of the hardware topology. Such systems could detect a situation where all the participating instances of a distributed transaction are located on the same

Chapter 4. Adaptive Transaction Processing

machine. Then they are able to switch to a more efficient communication channel, e.g., shared memory. In that case, the cost model could include information about the relative cost of two types of distributed transactions to choose the partitioning scheme that reduces the number of more expensive distributed transactions.

5 Toward Rack-scale OLTP

In this chapter, we investigate similarities and differences between multsocket multicore and rack-scale hardware system designs that represent current and future hardware platforms for high performance transaction processing. We analyze different distributed deployments using standard and synthetic benchmarks that include distributed transactions to quantify the challenges and opportunities for OLTP designs on rack-scale platforms. [Section 5.1](#) details the experimental setup and methodology. Based on our analysis, we characterize the requirements for rack-scale OLTP designs.

Since each node in a rack-scale system is a small scale multicore system and the complete rack-scale system can be viewed as a large partially cache coherent multsocket system, one would expect multicore optimized designs to perform well. The first question we answer is how the current state-of-the-art multicore optimized scale-up designs behave when deployed in a distributed configuration ([Section 5.2](#)). We also investigate whether it is better to deploy one instance per node and scale out across the cluster or use the fine-grained deployments with one instance for each processor core in the system.

In [Chapter 3](#) we show that instances of different granularities are optimal for different types of workloads in the multsocket multicore environment. However, communication latencies within a multsocket are much smaller than among machines in a cluster. In [Section 5.3](#) we investigate whether multsocket topology matters in the cluster environment or the network communication costs dominate.

One of the main differentiating properties of the rack-scale systems is the fast network between nodes. In [Section 5.4](#), we use different systems and workloads as well as different communication mechanisms to quantify the impact of network on performance on OLTP deployments.

Finally, we discuss the findings in [Section 5.5](#) and provide outlook toward transaction processing designs that scale up and out to efficiently utilize rack-scale hardware platforms.

5.1 Setup and Methodology

5.1.1 Distributed OLTP Deployments

In this study, we use two state-of-the-art open-source OLTP systems: Shore-MT and Silo. We choose Shore-MT as the representative traditional storage manager and Silo as the main-memory optimized one. Both of these systems use scale-up designs that we extend with a thin distributed transaction layer (see [Section 3.2.1](#) for more details). We implement different communication mechanisms to execute distributed transactions using the standard two phase commit (2PC) protocol. Distributed transactions in our deployment fit into predefined transaction classes and are one shot [144] with local and remote transaction parts known apriori which removes the need for more than one message in the first phase of 2PC. Local transaction site acts as a coordinator in the 2PC protocol. Unless noted otherwise, we bind threads to cores and allocate memory in local memory node when possible to improve locality.

5.1.2 Hardware Platforms

In order to better approximate rack-scale platforms, we use two different hardware platforms: a cluster and a large multisoocket server. Our cluster consists of 8 machines with 2 Intel Xeon X5660 processors each, connected using 10Gbps Ethernet network. Each processor has 6 cores with private L1 (32KB each for data and instructions) and L2 (256KB) caches, as well as 12MB of shared L3 cache. Each machine has 48GB of RAM that we use for both data and log files through memory mapped disks. All experiments are run using Ubuntu 12.04.4 LTS (kernel version 3.2.0-34) and the software is compiled using GCC 4.6.3 with maximum optimizations.

The large multisoocket server we use in this study is the octo-socket server described in [Table 3.1](#). We run all experiments using Red Hat Enterprise version 6.7 (kernel version 2.6.32) and compile the software using GCC 5.1.0 with maximum optimizations.

5.1.3 Workloads

In this study, we use a synthetic microbenchmark defined in [Section 3.2.2](#) and an industry-standard TPC-C benchmark [154]. The microbenchmark enables us to precisely quantify the impact of different types of operations and the different percentages of multisite transactions.

To characterize the impact of more complex transactions, we use Payment and NewOrder transactions from the TPC-C benchmark that comprise 88% of the benchmark mix. Both of them are read-write transactions that access data either from the local or the remote warehouse. The benchmark specifies that 15% of the Payment and 10% of the NewOrder transactions access remote warehouses. We partition the data using the well known scheme [144], where the data associated with a particular warehouse are placed in the same instance and the `Item` table is replicated in every instance. In contrast to microbenchmark experiments,

TPC-C transactions involve at most two instances, always involve updates, and use the fixed percentage of remote transactions.

5.2 Scaling Out Across Rack-scale Nodes

Each node in a rack-scale system is a multsocket multicore with a large main memory and they are connected using low latency network. In this section we answer the question how do distributed deployments of the scale-up main memory optimized design compare to the scale-out deployments of the same system. We use Silo main memory OLTP system and approximate the rack-scale system using the multsocket multicore server with each socket representing a rack-scale node.

5.2.1 Distributed Main Memory System

Silo uses an optimistic concurrency control protocol that scales well on multicores as it avoids any centralized synchronization points. In order to adapt Silo for distributed deployment, we split its commit processing into two phases: 1) the validation phase that we perform at the end of the first phase of 2PC and 2) the actual commit that we perform in the second phase. Between these two phases, the updated rows are locked and any transactions attempting to read them is aborted.

We deploy a distributed version of Silo using a shared memory communication channel. The dataset size is 8 million rows and partitioned equally among all instances in the deployment. We compare *scale-out* deployments with one instance per processor core and *scale-up* deployment with one instance per processor socket and use 80 and 8 instances respectively. We distinguish the behavior of the distributed deployment for the read-only and update workloads and identify factors that cause differences between the two.

5.2.2 Read-only Transactions

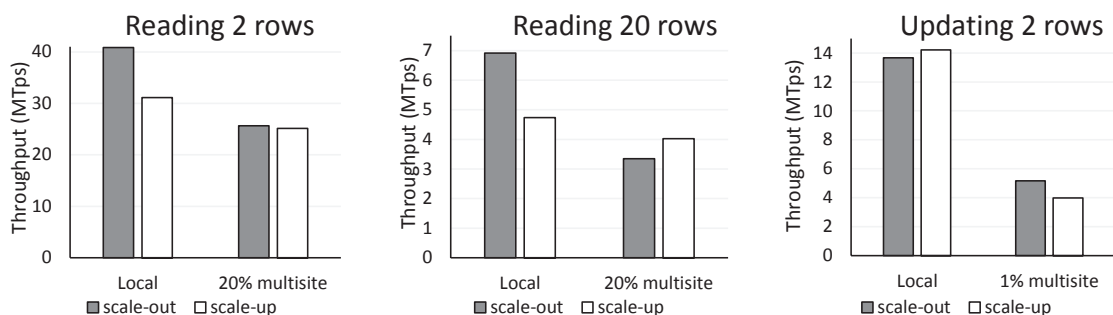


Figure 5.1: The impact of distributed transactions on the throughput of the scale-up main memory system for read-only and update workloads.

We start with the read-only microbenchmark for 2 and 20 rows for local only transactions and a mix with 20% multisite transactions and plot the results in [Figure 5.1](#) (left and center). In both cases, the fine-grained scale-out deployment has higher throughput for local transactions due to better locality of data accesses and absence of thread synchronization. However, in the presence of multisite transactions, the number of rows accessed has noticeable impact on the relative throughput. For the lightweight transactions, scale-out deployment has higher throughput, while the situation reverses for heavier transactions. In that case, scale-up deployment perform better due to fewer instances that participate in a single transaction which leads to lower communication overheads.

5.2.3 Update Transactions

To quantify the impact of updates, we run a microbenchmark that updates 2 rows and compare two settings: 1) only local transactions and 2) 1% multisite transactions. We plot the results in [Figure 5.1](#) (right). We use significantly smaller percentage of multisite transactions compared to the previous experiment since distributed update transactions have much higher cost. In contrast to the read-only distributed transactions, the update ones increase contention due to the prolonged commit phase that leads to abort rates of 11.5% for 1% of multisite transactions. The pessimistic choice to abort transactions that attempt to update the locked row has a negligible effect when running only local transactions due to the short commit phase. However, its impact is much higher in the presence of distributed transactions that increase the length of the commit phase significantly.

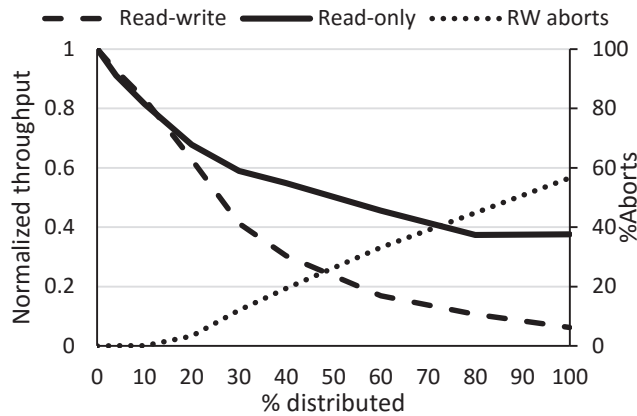


Figure 5.2: Increasing the duration of commit processing in distributed transactions significantly increases abort rates.

5.2.4 Sensitivity Analysis

In order to characterize the difference between the read-only and update distributed transactions on the throughput, we distinguish the impact of communication overheads and increased contention. We use a modified microbenchmark on a deployment that has only

2 instances, each using a single core. We vary the percentage of distributed transactions from 0 to 100%. The dataset size is 200 000 rows evenly split between the instances. We use two microbenchmarks: 1) the one that reads 2 rows and 2) the one that reads 1 rows and updates 1 row. For the local transactions, both rows are chosen from the local instance. For the distributed transactions, one row is chosen from the local and the other is chosen from the remote instance. In the read-only case, distributed transactions will incur only the communication overheads. In the read-write case, we choose the row that is updated from the local instance and the row that is retrieved from the remote instance. In both cases, distributed transaction have the same communication overheads since the remote fragment is read-only and does not require any processing in the second phase. However, for the read-write case, the update row is locked until the remote fragment is processed causing any concurrent requests accessing that row to conflict.

We plot the normalized throughput of both microbenchmarks as well as the abort rates for the read-write one in [Figure 5.2](#). For small percentages of distributed transactions, the relative throughputs of the read-only (solid line) and the read-write (dashed line) microbenchmarks follow the same trend as long as abort rates (dotted line) are negligible. However, with 10% or more of distributed transactions in the workload, the throughput of read-write microbenchmark starts dropping faster. At the same time, abort rates steadily increase reaching 55% when all transactions are distributed while throughput plummets to the 6% of the peak.

This experiment emphasizes the reliance of main-memory-optimized scale-up designs on the short critical sections for achieving good performance. The delay introduced by a distributed transaction that artificially lengthens a critical section leads to a large increase in contention and high abort rates. Similar effect can be observed in the case of long running update transactions and workloads that exhibit high contention [80].

5.2.5 Summary and Implications

Distributed transactions are more expensive than their local-only counterparts as they require communication among multiple instances in the system and their relative cost depends on the type of accesses within a transaction. For the read-only distributed transactions, communication is the main overhead. Hence, distributed transactions affect the coarser grained configurations less since they potentially involve fewer instances in the execution of a transaction.

The impact of distributed transactions is much higher for the transactions that contain updates. Main-memory optimized systems achieve high performance by accessing only a small number of short critical sections in the critical path of transaction execution. Adding communication step in the middle of the commit processing of the efficient OCC protocol increases abort rates significantly and has detrimental effect on performance.

5.3 Scaling Up on Rack-scale Nodes

In the previous section, we show that multicore optimized main memory designs that scale well on multisoockets face challenges when deployed in distributed setting. We identify the trade-offs between different distributed deployment configurations on multisoockets in [Chapter 3](#). However, the network poses much higher communication overheads across the cluster of machines compared to a single multisoocket which potentially overshadows the impact of multisoocket topology in cluster deployments. In this section, we use a cluster of machines and different workloads to quantify the impact of multisoocket topology on the performance of different deployments. As we require TCP/IP communication channel for cluster deployment, we use the traditional OLTP system.

5.3.1 Distributed Deployment Configuration

In this set of experiments, we use a distributed transaction processing system built on top of the Shore-MT [71] open-source storage manager. Shore-MT provides near linear scalability on machines with a single multicore chip [71] and includes a number of the state-of-the-art optimizations for local transactions, such as speculative lock inheritance [70] and Aether holistic logging [72].

In all experiments, we choose a configuration for a machine and deploy it across all machines in the cluster. We use *scale-out* (one per core), *scale-up* (one per machine), and *hybrid* (one per socket) deployments with 12, 1, and 2 instances per machine, respectively. We scale dataset sizes to 1 warehouse per core for experiments with TPC-C benchmark and 10 000 rows per core for microbenchmarks. We use TPC-C to investigate whether the granularity of instances and careful placement within a machine matter in a rack-scale setting. Then we quantify the impact of the size of a transaction and the type of accesses on performance of cluster deployments using microbenchmarks.

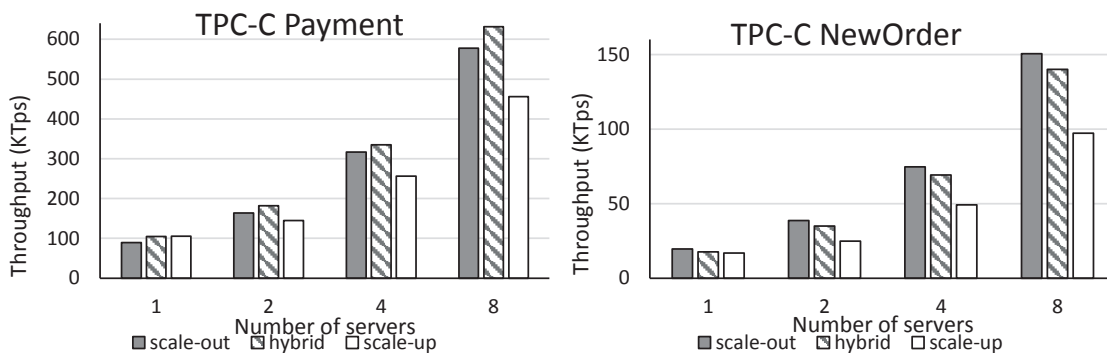


Figure 5.3: Throughput of different deployments as the number of servers increases for Payment and NewOrder transactions.

5.3.2 Scaling TPC-C Across Machines

We start by analyzing scalability of TPC-C benchmark as we increase the number of machines from 1 to 8. We plot the throughput in Figure 5.3 for both Payment (left) and NewOrder (right) transactions. On a single server, the larger instances perform better for Payment transactions, while the smaller ones perform better for NewOrder transactions. The difference stems from the type of write operations done by a transaction. In the Payment case, larger instances profit from constructive sharing of a single log whereas each scale-out instance needs to write its own log and issue expensive system calls. On the other hand, the NewOrder transactions perform many insertions to the `OrderLine`, `Order`, and `NewOrder` tables, which require a lot of synchronization among threads in the same instance. Also, the scale-up deployment greatly benefits from the fact that it does not need to execute any distributed transactions.

When we increase the number of servers, smaller instances scale better than the scale-up deployment which requires executing distributed transactions when deployed over multiple servers. Scale-out deployments scale better than the hybrid ones for NewOrder, while the situation is reversed for Payment. Also, when deployed on 8 servers, both achieve on average 7.7x better throughput for NewOrder and only 6.2x for Payment. The difference in scalability comes from the type of updates performed by each transaction. Namely, Payment transactions update one row from the `Warehouse` table, which limits the number of concurrent transactions in the system to the number of warehouses. In contrast, NewOrder updates a row from the `District` table, that has 10 rows for each warehouse, thus permitting 10 times more concurrent transactions. Distributed transactions holding locks on the updated rows until the end of the second phase of the 2PC protocol lead to lower concurrency in Payment, which severely limits scalability.

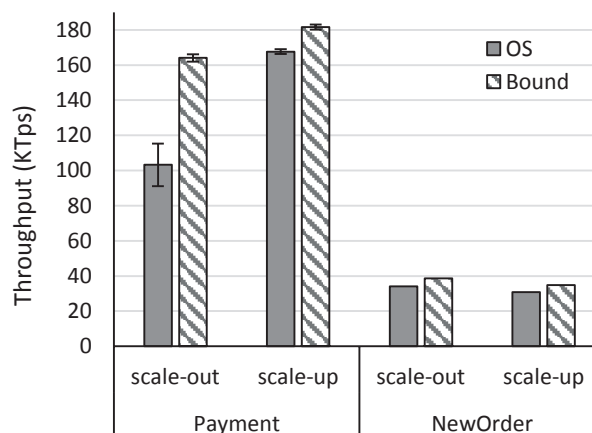


Figure 5.4: The impact of thread binding on throughput for different configurations and TPC-C benchmark.

5.3.3 Impact of Thread Binding

Careful thread binding is an important prerequisite for achieving predictable high performance on multisockets as it maximizes locality and disallows thread migrations (see [Section 2.2](#) for more details). In this set of experiments we investigate whether thread binding has any impact on performance of cluster deployments. We use 2 servers and either bind the instances to specific cores or sockets, or leave the placement to the operating system. We repeat the experiments with both Payment and NewOrder transactions.

The left hand side of [Figure 5.4](#) shows throughput for the Payment transaction with solid bars representing threads placed by the operating system and striped bars representing manual binding with each core-sized scale-out instance on a separate core and each scale-up socket-sized instance on a separate socket. We run the experiment three times and show standard deviation on the bars. Binding instances to sockets improves performance of the scale-up deployment by 8%. This effect is more pronounced for the scale-out deployment, where binding instances to cores improves performance by 60% and reduces variability from 11.7% to 1.2%. The right hand side of the figure shows the result for the NewOrder transaction, which is much more predictable with standard deviations of less than 1% in all cases. However, even in this case, binding the instances and disabling migrations improve performance by 13.5% and 12.9% respectively for scale-out and scale-up deployments.

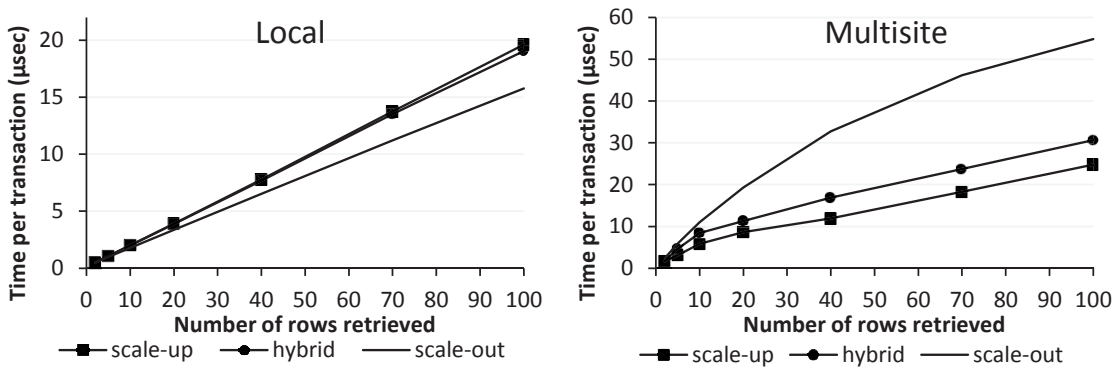


Figure 5.5: The cost of local and multisite transactions for different deployment configuration as the number of retrieved rows increases. Multisite transactions in coarser-grained deployments have up to 2 times lower cost compared to the fine-grained scale-out one for the large number of rows.

5.3.4 Sensitivity Analysis

In easily partitionable TPC-C benchmark, each distributed transaction involves at most two instances, all transactions involve updates, and the percentage of distributed transactions is fixed. To better quantify the costs of arbitrary distributed transactions we perform a sensitivity analysis using microbenchmarks. The cost of a transaction is expressed as the time it takes to execute a single transaction. We use a 4 machine cluster and measure the cost of local

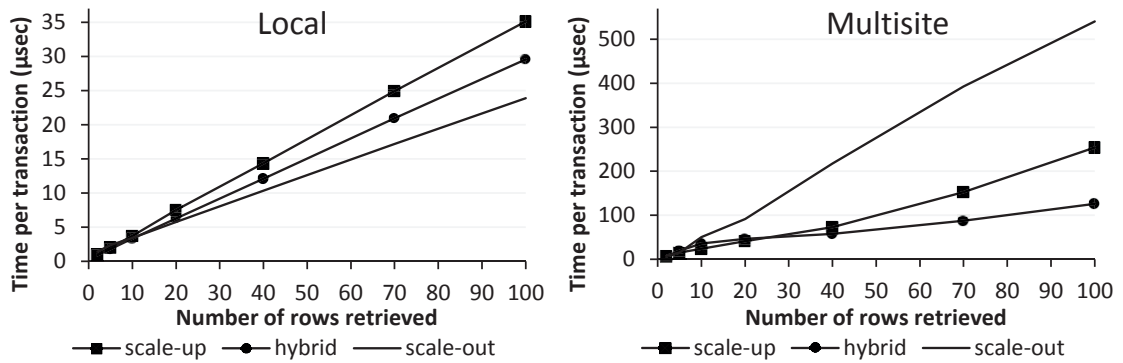


Figure 5.6: The cost of local and multisite transactions for different deployment configuration as the number of rows updated increases. Multisite transactions in coarser-grained deployments have 2-4x times lower cost compared to the fine-grained scale-out one for the large number of rows.

and multisite transactions, in read and update versions, as we increase the number of rows accessed from 2 to 100. With the higher number of rows per transactions, multisite transactions require data from multiple instances and thus have to exchange more messages to complete a single transaction.

Read-only case. Figure 5.5 (left) plots the time it takes to execute a local read-only transaction for different deployments as the number of rows retrieved per transaction increases. The scale-out deployment has the lowest cost since each instance runs single-threaded and, hence, pays no thread synchronization overhead. Larger instances have higher costs due to these overheads. The cost trend is reversed for the multisite case (Figure 5.5 right) where scale-out instances have significantly higher costs compared to the larger ones. The increase in cost is primarily due to the number of messages needed for a multisite transaction. Since these transactions are read-only, we use the optimized version of the 2PC protocol that requires only one roundtrip per participant. For every configuration, after the number of rows surpasses the number of instances in the system, every multisite transaction typically involves all instances in the system. This results in the flattening lines as the distributed transaction overheads become constant.

Update case. Figure 5.6 shows the time required to execute one local (left) and one multisite (right) transaction as the number of rows updated per transaction increases from 2 to 100. For the local transactions, the increase in cost is linear with the number of rows per transaction with larger instances having higher cost. The differences between configurations are more pronounced due to the higher synchronization overhead involved in the operations that modify data. In the multisite case, while the number of instances involved in a transaction increases at the same rate as in the read-only case, the costs increase faster. This effect is due to the higher communication costs (as update transactions require both roundtrips in the 2PC protocol) and increased contention since locks are held until the end of a transaction. Even though scale-up and hybrid deployments, with multiple threads per instance, use optimized

logging, their cost trends do not flatten out for higher number of rows due to increased contention. The increase is higher for the scale-up deployment because of more threads in the instance.

5.3.5 Summary and Implications

In this section, we show that the network overheads do not eliminate the requirement to optimize deployment configuration at the level of a rack-scale node and that choosing the best configuration requires considering the information about the whole cluster. For the workloads that access many rows, overheads such as communication, as well as logging and additional contention due to locks being held longer for the update case, make distributed transactions 2-4x more expensive for the fine-grained scale-out deployments compared to the coarser ones.

If the number of instances required in a distributed transactions is small, as it is the case for the TPC-C workload, the impact of communication is less significant and the optimal configuration depends on the trade-off between the overheads of thread synchronization and the opportunities for constructive sharing between threads within an instance. Finally, our experiments show that adjusting placement of the individual instances within a machine can significantly improve performance, especially for scale-out deployments, by improving locality.

5.4 The Impact of Network

Network communication represents significant component of the cost of distributed transactions. Its performance is determined by two factors: the hardware channel and the software stack. With rack-scale systems using high-speed low latency interconnects and enabling RDMA-based messaging that bypasses the operating system, communication overheads significantly diminish and can potentially make distributed transactions much cheaper and particular system designs more appealing. In this section, we quantify the impact of network on the throughput of different distributed deployments across various systems, workloads and communication mechanisms.

5.4.1 Main Memory Optimized System

First, we quantify the impact of network on the main memory system by repeating the experiment from [Section 5.2.2](#) and [Section 5.2.3](#) with UNIX domain sockets that have higher cost as they require system calls (see [Figure 3.4](#) for the comparison).

Read-only case. [Figure 5.7](#) plots the results of the experiment with dark bars for the runs with UNIX domain sockets and white ones for the shared memory communication. For the experiment with the lightweight transactions, plotted on the left hand side of the figure, the choice of communication channel can reverse the relative performance of different deployments.

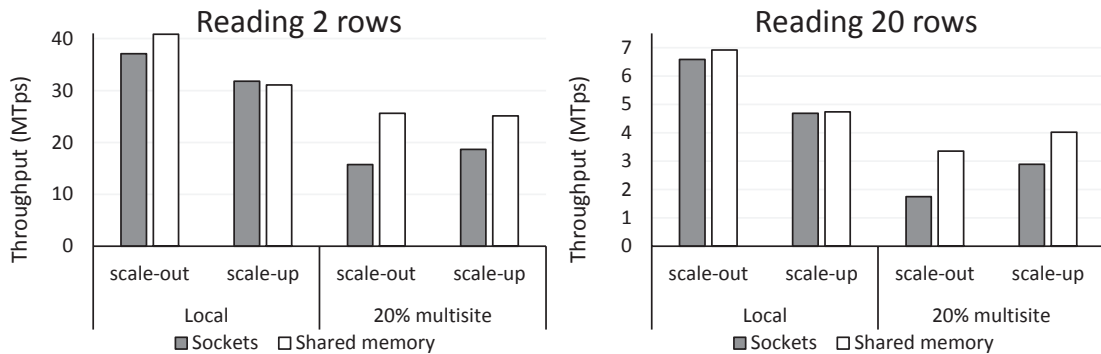


Figure 5.7: In a distributed deployment of the main memory system for read-only workload, communication overheads directly impact relative throughput of different configurations with higher costs favoring larger instances.

Namely, scale-out deployment has higher throughput due to locality of data accesses for the local transactions in both cases as well as with shared memory communication and multisite transactions. However, with higher overhead sockets, especially for the case of heavier transactions in scale-out deployment, the relative performance reverses and scale-up deployment performs better. For the heavier transactions, scale-up deployment has higher throughput for multisite transactions in both cases due to fewer instances that participate in a single transaction causing lower communication overheads.

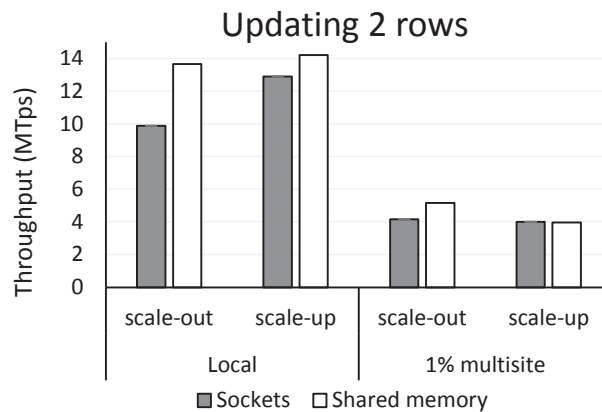


Figure 5.8: For update workloads running on a distributed deployment of the main memory system, increased contention leading to high abort rates causes performance drops for all communication mechanisms.

Update case. We plot the results of the update experiment in [Figure 5.8](#). In this case, the behavior of distributed deployments is completely different compared to the read-only case with a narrow gap between throughput for different communication mechanisms. For all combinations of deployment configuration and communication mechanisms, even small percentage of distributed transactions significantly affects contention. The increased contention leads to abort rates of 8% to 11.5% with slightly lower abort rates when using UNIX

domain sockets. Faster communication mechanism allows more concurrent transactions in the system due to lower overhead on the critical path, thus increasing the probability of conflicts on individual data items, and leading to higher abort rates. Without changing the distributed transaction coordination protocol to mitigate conflicts, faster communication will not have any impact on throughput of distributed transactions that contain updates.

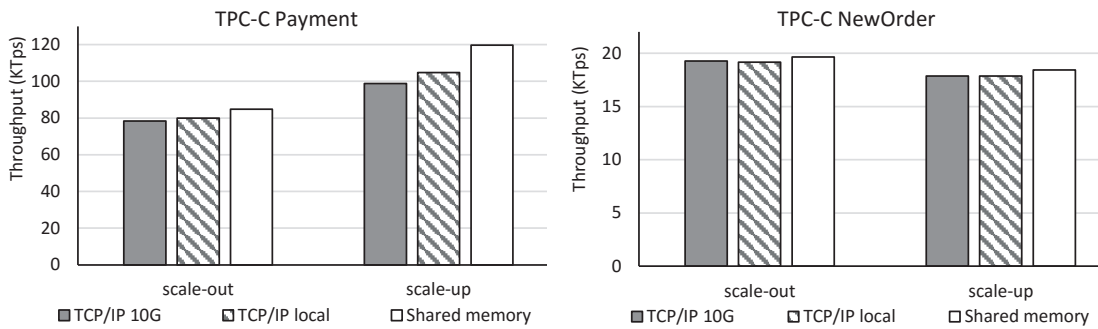


Figure 5.9: The impact of communication channel on different deployments of traditional system with TPC-C benchmark is pronounced for short Payment transactions on the scale-up deployment, while other work completely mask it for other scenarios.

5.4.2 TPC-C Transactions

For the traditional system, we first use TPC-C benchmark to isolate the impact of different components of the communication channel on the distributed transactions. We compare three communication mechanisms: 1) TCP/IP over Ethernet network, 2) TCP/IP in a single machine, and 3) shared memory communication in a single machine. The first case represents today’s mainstream option. The second case is the scenario with the fastest possible way of communication that still employs unmodified TCP/IP software stack. With the shared memory communication mechanism, we emulate the best RDMA scenario where accessing remote machine’s memory has the same latency as accessing local memory. We use a dataset with 12 warehouses (1.8GB) and compare *scale-out* (one per core) and *scale-up* (one per socket) deployments. For the first setting, we use two servers and deploy half of the instances (6 scale-out and 1 scale-up) on each server, while for the other two settings we deploy all instances on the same server.

The left hand side of Figure 5.9 shows throughput for the Payment transaction with gray bars for TCP/IP over 2 machines, striped bars for TCP/IP on a local machine, and the white bars for shared memory communication channel. Faster communication increases the performance of Payment transactions for both configurations. The magnitude of the increase depends on the size of the instance and the workload type. However, faster communication does not change the relative performance: the scale-up configuration has higher throughput than the scale-out one. The right hand side of Figure 5.9 shows experiment for the NewOrder transactions. In this case, communication speed has a negligible impact on the performance

since NewOrder does many more operations per transaction than Payment and the cost of messaging is amortized.

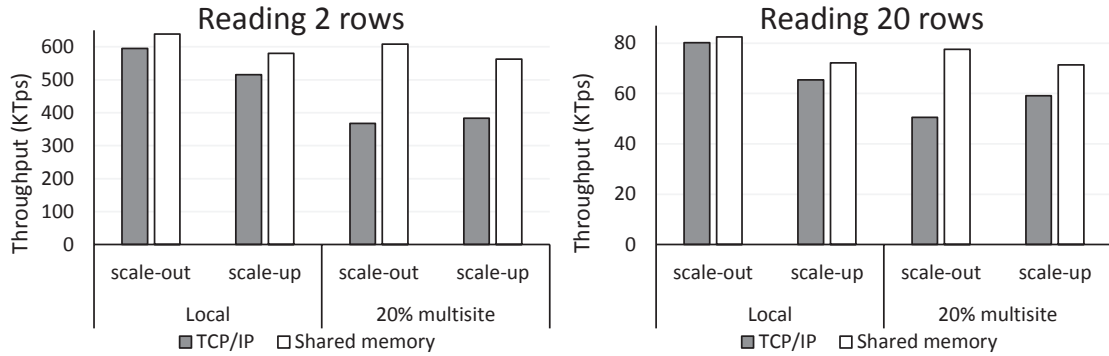


Figure 5.10: In the traditional system, communication costs directly impact the cost of read-only distributed transactions and can reverse the relative performance of different configurations.

5.4.3 Microbenchmarks

To better understand the impact of communication latency on the cost of distributed transaction depending on the type of operations, we run a series of experiments with microbenchmarks using TCP/IP and shared memory communication mechanisms. In all graphs, the dark bars show the case when we use TCP/IP for communication and the white bars represent shared memory. We use a single server and a dataset with 12 sites (120 000 rows) and compare *scale-out* (one per core) and *scale-up* (one per socket) deployments over 12 cores. We study read-only and update cases separately and repeat the experiments with only local transactions and with a mix containing 20% multisite transactions. Also, we repeat microbenchmarks for 2 and 20 rows to assess the impact of 1) the different percentage of multisite transactions that are executed as distributed transactions and 2) the different number of instances involved in the execution of a single distributed transaction.

Read-only case. Figure 5.10 plots the results of the experiment for the read-only transactions with the 2 rows case on the left hand side and 20 rows on the right. In all cases, the deployments that use shared memory communication have higher throughput than the ones using TCP/IP. Since the read-only transactions are short, higher static communication overheads in the TCP/IP case lead to noticeable difference in throughput for local only transactions. In order to fully exploit fast network, we need to avoid expensive system calls required for TCP/IP communication.

For both types of transactions, we observe that the communication channel has a significant impact on relative performance of two deployments, similarly to the main memory system. Namely, in the presence of multisite transactions, *scale-out* deployment has higher performance than the *scale-up* one for shared memory communication, while the situation is

reversed for TCP/IP. The impact of communication is higher for heavier transactions since every distributed transaction involves all instances which means scale-out configuration needs to exchange messages with 11 instances, compared to only one instance in the scale-up case. However, even in that case, fast network communication makes scale-out configurations faster than the scale-up one for all scenarios.

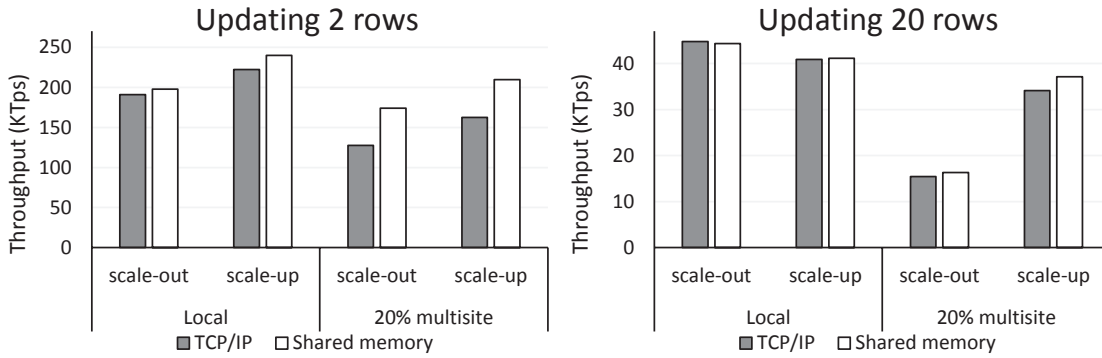


Figure 5.11: Constructive sharing among threads in larger instances in the distributed deployments of the traditional system mitigates the communication overhead for heavier update transaction and leads to higher performance compared to the smaller instances.

Update case. We plot the result of the update experiment in Figure 5.11 with lighter transactions on the left and heavier ones on the right. The impact of communication is less pronounced compared to the read-only case because distributed update transactions are significantly more expensive than their read-only counterparts. The difference comes from the ability to overlap logging and communication overheads. For example, transactions that update 2 rows generate less log, hence, they cannot overlap static communication overheads as effectively as the larger transactions. This effect is particularly evident in the presence of distributed transactions where the choice of communication mechanism has almost no effect on the throughput for the 20 row case.

5.4.4 Summary and Implications

The impact of network communication depends greatly on the type of operations performed by a transaction. For read-only transactions, communication has direct impact proportional to the number of instances involved in a transaction. For the traditional system, fast network makes fine-grained scale-out deployments preferable to the scale-up ones that incur thread synchronization overheads. With slower network, the choice of the best deployment configuration depends on the trade-off between communication overheads among instances and thread synchronization within an instance. When thread synchronization overheads are smaller, as is the case in main memory optimized system, scale-up instances are preferable to the scale-out ones.

On the other hand, the impact of network on the update workloads is much smaller due to other factors that dominate in the cost of distributed update transactions. In the traditional

system, the communication can be overlapped with logging or other processing. Furthermore, constructive sharing among threads makes scale-up instances preferable for many update workloads. For the main memory system, the increased contention in the presence of distributed transactions has a decisive impact on the performance. In this situation, instead of improving performance, faster network merely increases abort rates.

5.5 A Step Toward Rack-scale OLTP

We expect emerging high-performance OLTP applications to be deployed on the rack-scale hardware platforms that consist of low power multicore nodes with large main memories connected using low-latency interconnect. This motivated us to investigate whether the state-of-the-art main-memory optimized designs can be used as building blocks for rack-scale OLTP designs, how different multisojects and rack-scale systems are with respect to distributed transactions, and quantify the impact of communication channel performance. In this section we summarize our findings, discuss implications, and provide outlook toward future rack-scale OLTP designs.

Study implications. In order to fully utilize a rack-scale system, OLTP systems need to scale up and out simultaneously by scaling up within a node and scaling out single node configurations across the cluster. Scaling up within a node requires choosing the optimal instance granularity and the thread placement that maximizes locality. To scale across the cluster, a system needs to take into account the cluster topology and the workload properties. In general, for easily partitionable workloads that require no or very few distributed transactions, scale-out deployments are preferable as they achieve perfect locality. On the other hand, for workloads that are not easily partitionable, larger instances are better as they limit communication overheads and can potentially exploit constructive sharing among threads. For the read-only workloads, the distributed transaction overheads are directly proportional to the communication costs, while updates incur additional overheads that can significantly increase contention, especially in main memory optimized systems.

The impact of communication mechanism depends on many factors. For the read-only workloads, the fast network directly reduces communication overheads and improves performance. For the main memory optimized systems, the scale-up deployments are preferable to the scale-out ones regardless of the speed of the network. For the traditional systems, however, the fast network makes fine-grained scale-out deployments preferable to the scale-up ones that incur thread synchronization overheads. On the other hand, the fast network has much less impact on the distributed update transactions, as other overheads dominate the cost: logging for the traditional system and increased contention for the main memory one.

Concurrency control. The current state-of-the-art main memory scale-up designs rely on multiversioning and optimistic concurrency control mechanisms to achieve good performance [80, 81, 93, 109, 157, 173]. As they do not employ partitioning, they achieve good performance by minimizing the number and the duration of critical sections. The main issue

preventing them from efficiently scaling across multiple machines is their reliance on the assumption that transactions are very short: both in the terms of the number of items accessed and in their duration. While distributed transaction execution does not increase the number of data items accessed in a transaction, it has similar effects to long running reads in terms of effectively blocking concurrent short updates by introducing delays in the validation phase.

Distributed coordination. On the other hand, modern transaction coordination protocols have focused on datacenter deployments [12, 86, 149]. The drawback of these approaches is that they assume long latencies between nodes in the distributed deployment, thus allowing them to execute complex coordination protocols. We argue here that while such protocols use asynchronous communication and require fewer messages, they are not lighter than the classic 2PC in a rack scale environment. Techniques like deterministic transaction execution are a promising direction. However, increased latency due to the execution of conflict-free batches and the requirement that full read and write sets are known at the start of a transaction limit their applicability [131, 149].

Challenges. Inadequacies of the state-of-the-art concurrency control and coordination protocols stem from scale-up and scale-out design requirements respectively. On the one hand, concurrency control protocols for main-memory-optimized scale-up designs need to minimize the duration of any critical section so as not to introduce any scalability bottlenecks. This makes them sensitive to delays introduced in the critical path of transaction execution. On the other hand, coordination protocols aim to minimize the number of messages between nodes in the distributed system as communication latencies dominate all other delays in the system. However, this allows them to add significant local processing overhead that is prohibitive for lean main-memory-optimized systems.

Opportunities. In order to design protocols for rack-scale systems, concurrency control protocols need to become resilient against communication delays and the coordination protocols need to become more lightweight to capture the best of both worlds. One approach for making concurrency control protocols more amenable to distributed execution is using techniques such as controlled lock violation to shorten commit processing by tracking dependencies [50]. This optimistic approach may lead to a chain of aborts. However, such behavior is restricted to the situations where there are many read/write conflicts on the hot data. A complementary set of techniques rely on application semantics to enable phase reconciliation and knowing transaction write-set a priori to increase concurrency [44, 108]. Similar ideas that rely on application semantics to relax coordination requirements in distributed deployments have shown good results in datacenter deployments [13, 134]. We believe that the judicious use of semantic information from the application enables design of resilient concurrency control and lightweight coordination protocols required for efficient rack-scale OLTP designs. Two recent proposals leverage RDMA and modern hardware, namely non-volatile RAM and hardware transactional memory, to achieve good scalability for easily partitionable workloads, such

5.5. A Step Toward Rack-scale OLTP

as TPC-C, on clusters with fast networks [42, 163]. They present a good step toward designing efficient systems for arbitrary transaction processing workloads on rack-scale hardware platforms.

6 The Big Picture

Modern hardware platforms are getting more complex with non-uniformity at various levels of the system architecture without clear distinction among the levels. In order to efficiently utilize such hierarchical systems for transaction processing, we need to fundamentally redesign our software with focus on locality of communication and explicit awareness of the underlying hardware. Transactions typically access a few data items, often creating hotspots, and different transaction types can have very different data access patterns [152]. Therefore, the software needs to be agile and continuously adapt its configuration to the workload and underlying hardware topology to serve the workload with maximum efficiency [3].

This chapter summarizes the contributions of this thesis and discusses possible directions of future work.

6.1 What We Did

Motivated by the emergence of Hardware Islands in modern servers, we conducted a detailed study across a range of deployment configurations of different transaction processing systems, a number of multsocket servers, and a variety of workloads. We concluded that no single optimal system deployment configuration exists: the best configuration depends on the hardware topology and the workload. For example, the shared-nothing is twice as fast as shared-everything deployment configuration for the perfectly partitionable workloads, while situation is completely opposite for the non-partitionable workloads and workloads that exhibit heavy skew. Island-sized shared-nothing configurations fall between the two extremes. We proposed a straight-forward performance model based on the deployment configuration and the percentage of multipartition transactions in the workload and validated it against both traditional and main-memory optimized system designs. The fundamental takeaway is that more partitionable workloads favor the finer-grained deployment configurations, while less partitionable achieve better throughput on the coarser-grained configurations since the cost of synchronization within an instance is lower than the cost of coordination across instances.

To address the challenge of high reconfiguration cost in the presence of changing workload characteristics that require different optimal deployment configurations, we proposed ATraPos. ATraPos extends a scalable logically partitioned shared-everything system to Islands using automatic partitioning of the system state and dynamically assigning worker threads to specific partitions. In this way, it removes all intersocket accesses from the critical path of transaction execution for perfectly partitionable workloads. For other workloads, we rely on finding a good partitioning and placement scheme that balances the load across partitions and minimizes the synchronization overheads across Islands. Finally, to ensure robust performance in the presence of shifting workload patterns, we use a lightweight monitoring mechanism to detect and quick repartitioning mechanism to adapt to any change.

Future high performance hardware platforms will have hundreds and thousands of processor cores in a single system organized in a hierarchy of Islands. We generalized the characterization of the impact on Islands by analyzing the trade-offs involved in the deployment of different OLTP system configurations on commodity clusters. We show that different configurations are optimal for different combinations of workload characteristics, multsocket topologies, and network communication properties. This finding emphasizes that scaling out requires both Island and inter-Island awareness to efficiently utilize emerging rack-scale hardware platforms, even with faster interconnects and widespread use of RDMA blurring the lines between different machines. In such environment, Island-awareness will remain relevant to the synchronization within a coherence domain while inter-Island-awareness will be required for coordination between these domains.

6.2 Impact

This thesis is a first step toward fully understanding the impact of non-uniform hardware on the performance of software systems. By examining the behavior of different transaction processing designs across a number of dimensions, we show that the hardware-awareness is critical in order to achieve predictable high performance. We use the concept of *Hardware Islands* to understand non-uniformity in the horizontal dimension and enable more structured modeling of hardware-awareness. In practice, our deployment rules of thumb can be used to optimize the deployment configurations of different OLTP systems for various workload types and hardware platforms. The lightweight monitoring and repartitioning technique we developed in ATraPos is applicable to any existing distributed transaction processing architecture to enable adaptivity to changing workload characteristics. Finally, many recent scale up transaction processing designs use Island-aware data structures that focus on the locality of communication in addition to having as few critical sections as possible.

In the longer term, with clusters of many nodes containing few multicore chips with large main memories replacing high-end many socket servers, the deployment rules of thumb we identify in this thesis would significantly reduce synchronization and communication costs of transaction execution. With workloads becoming more diverse and dynamic, yet running on

partitioned infrastructure, lightweight monitoring and adaptive partitioning and placement scheme we devise will be applicable in a wide range of scenarios where traditional offline schemes are impractical.

6.3 Looking Ahead

ATraPos assumes that the transaction execution plans are static, however, different transaction execution plans are likely to be optimal for different combinations of transaction types. One interesting direction of future work would be to adapt the execution plans holistically with the partitioning and placement scheme based on the data dependencies. In this way, one can decrease contention on the hot data that would otherwise increase the length of the critical path of transaction execution [170]. Moving hot data accesses to the end of a transaction has been a common optimization in enterprise applications.

In this thesis we focus on the workloads comprising short transactions. However, many real worlds applications combine short transaction with longer running ones, in real time business analytics scenarios. Contention on the hot data items between the short update transactions and long running scans is one of the main challenges. One way to address this issue is to separate transactional and analytical processing by running analytics on the snapshots of the database, as HyPer does [78]. Another way, commonly used by commercial data analytics designs, is to apply updates to a delta store and periodically merge them with the primary data storage. It would be interesting to apply data oriented execution to these types of workloads and extend adaptive partitioning and placement mechanism with memory bandwidth as an additional input parameter to the cost model.

Finally, transaction processing designs are suboptimal for the upcoming rack-scale hardware platforms. Inadequacies of the state-of-the-art concurrency control and coordination protocols stem from the scale-up and scale-out requirements respectively. On the one hand, concurrency control protocols for main-memory-optimized scale-up designs need to minimize the duration of any critical section in order to prevent any potential scalability bottlenecks. This makes them sensitive to delays introduced in the critical path of transaction execution. On the other hand, coordination protocols aim to minimize the number of messages among the nodes in the distributed system as communication latencies dominate all other delays in the system. This allows them, however, to add significant local processing overhead, that is prohibitive for lean main-memory-optimized systems. To that end, concurrency control protocols need to become resilient to the communication delays, while the coordination protocols need to become lighter to capture the best of both worlds. Judicious use of semantic information from the application with the focus on locality on every node is a promising way toward efficient rack-scale OLTP designs.

Bibliography

- [1] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-phase commit: does it make sense? In *ICPDS*, pages 182–192, 1998. [2.3](#)
- [2] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Summer*, pages 93–112, 1986. [2.2.3](#)
- [3] Anastasia Ailamaki. Databases and Hardware: The Beginning and Sequel of a Beautiful Friendship. *PVLDB*, 8(12):2058–2061, 2015. [6](#)
- [4] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999. [2.1](#)
- [5] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *PVLDB*, 5(10):1064–1075, 2012. [2.2.3](#)
- [6] Gustavo Alonso. Rackscale computing - the things that matter. In *WRSC*, 2014. [2.3](#)
- [7] Gustavo Alonso, Donald Kossmann, and Timothy Roscoe. Swissbox: An Architecture for Data Processing Appliances. In *CIDR*, volume 11, pages 32–37, 2011. [2.3](#)
- [8] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *SC*, pages 188–197, 1992. [2.3](#)
- [9] Amazon. EC2 instance types, June 2015. Available at <https://aws.amazon.com/ec2/instance-types/>. [3.2](#)
- [10] Amazon. Aurora, March 2016. Available at <https://aws.amazon.com/rds/aurora/>. [1](#)
- [11] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. Transaction Processing on Confidential Data using Cipherbase. In *ICDE*, 2015. [2.2.4](#)

Bibliography

- [12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, pages 27–38, 2014. [2.3](#), [5.5](#)
- [13] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185 – 196, 2015. [2.3](#), [5.5](#)
- [14] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsü. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1), 2014. [2.2.3](#)
- [15] Luiz André Barroso, Kouros Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *ISCA*, pages 3–14, 1998. [2.1](#), [2.2.1](#)
- [16] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015. [2.3](#)
- [17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. The multi-kernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29 – 44, 2009. [2.2.1](#), [2.2.3](#)
- [18] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *MICRO*, pages 319–330, 2004. [1.3](#)
- [19] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM TODS*, 8(4):465–483, 1983. [1.3](#)
- [20] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It’s Time for a Redesign. *PVLDB*, 9(7), 2016. [2.3](#)
- [21] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, Don McCauley, Pat Morrow, Donald W Nelson, Daniel Pantuso, et al. Die stacking (3d) microarchitecture. In *MICRO*, pages 469–479, 2006. [2.2.4](#)
- [22] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *PACT*, pages 557–558, 2010. [2.2.3](#)
- [23] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, pages 7–7, 2000. [1.3](#), [2.3](#)
- [24] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *PPoPP*, pages 157–166, 2013. [2.2.3](#)
- [25] Mark Callaghan. Incremental vs Rewrite from Scratch - a Biased Guide to Building a Web-scale DBMS. In *CloudDM*, 2015. [1](#)

-
- [26] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *SIGMOD*, pages 383–394, 1994. 3.2.1
- [27] George Chrysos. Intel Xeon Phi coprocessor (codename Knights Corner). In *Hot Chips*, 2012. 2.2.4
- [28] Kevin Closson. You buy a NUMA system, Oracle says disable NUMA! What gives?, 2009. Available at <http://kevinclosson.wordpress.com/2009/05/14/you-buy-a-numa-system-oracle-says-disable-numa-what-gives-part-ii/>. 2.2.3
- [29] CockroachDB. Available at <https://www.cockroachlabs.com/>. 1
- [30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s Globally-Distributed Database. In *OSDI*, pages 261–264, 2012. 1
- [31] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and S Zdonik. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015. 2.3
- [32] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010. 1.3, 2.1, 3.1, 3.3.1, 3.7, 4.6
- [33] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *ACM TODS*, 38(1):5:1–5:45, 2013. 2.1
- [34] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ASPLOS*, pages 381–394, 2013. 2.2.1, 2.2.3
- [35] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP*, pages 33–48, 2013. 2.2.1, 2.2.3, 2.2.4
- [36] Bhavya K Daya, Chia-Hsin Owen Chen, Sivaraman Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P Chandrakasan, and Li-Shiuan Peh. Scorpio: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In *ISCA*, pages 25–36, 2014. 2.2.4

Bibliography

- [37] Yigit Demir, Yan Pan, Seukwoo Song, Nikos Hardavellas, John Kim, and Gokhan Memik. Galaxy: a high-performance energy-efficient multi-chip architecture using photonic interconnects. In *SC*, pages 303–312, 2014. [2.2.4](#)
- [38] A Dhodapkar, G Lauterbach, S Li, D Mallick, J Bauman, S Kanthadai, T Kuzuhara, GSM Xu, and C Zhang. SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips. In *HotChips*, 2011. [1.2](#), [2.3](#)
- [39] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *PPoPP*, pages 247–256, 2012. [2.2.3](#)
- [40] Diego Didona, Nuno Diegues, Rachid Guerraoui, Anne-Marie Kermarrec, Ricardo Neves, and Paolo Romano. ProteusTM: Abstraction Meets Performance in Transactional Memory. In *ASPLOS*, 2016. [2.2.1](#)
- [41] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *NSDI*, pages 401–414, 2014. [2.3](#)
- [42] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015. [5.5](#)
- [43] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995. [2.2.3](#)
- [44] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015. [5.5](#)
- [45] Philip W Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009. [2.3](#)
- [46] Holger Fröning. Data movement options in accelerated clusters. In *Euro-Par*, pages 418–422, 2013. [1.2](#)
- [47] Hector Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM TODS*, 8(2):186–213, 1983. [2.3](#)
- [48] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. Deployment of Query Plans on Multicores. *PVLDB*, 8(3):233 – 244, 2014. [2.2.1](#), [2.2.3](#)
- [49] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *ASPLOS*, pages 229–240, 2013. [2.2.3](#)
- [50] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. Controlled lock violation. In *SIGMOD*, pages 85–96, 2013. [5.5](#)

-
- [51] Colleen Graham, Bhavish Sood, Hideaki Horiuchi, and Dan Sommer. Market Share: Database Management System Software, Worldwide, 2009. Available at <http://www.gartner.com/DisplayDocument?id=1044912>. 1
- [52] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. 2.1
- [53] Erik Hagersten and Michael Koster. WildFire: A scalable path for SMPs. In *HPCA*, pages 172–181, 1999. 1.2, 2.3
- [54] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, pages 184–195, 2009. 1.3, 2.1, 2.2.1
- [55] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, 2011. 2.2.4
- [56] Stavros Harizopoulos, Daniel J. Abadi, Sam Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008. 3.4.1
- [57] Tim Harris. Hardware Trends: Challenges and Opportunities in Distributed Computing. *ACM SIGACT News*, 46(2):89–95, 2015. 2.3
- [58] Pat Helland. Life beyond Distributed Transactions: an Apostate’s Opinion. In *CIDR*, pages 132–141, 2007. 1.3, 2.3
- [59] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Foundations and Trends (R) in Databases*, 1(2), 2007. 2.1
- [60] HP. Running Microsoft SQL Server 2014 on HP Integrity Superdome X - Reference Configuration Guide, 2015. Available at <http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-8846ENW>. 1.2, 2.2.3
- [61] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md Wasi-ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. High-performance design of HBase with RDMA over Infiniband. In *IPDPS*, pages 774–785, 2012. 2.3
- [62] IEEE. Ieee 802.1qbb: Priority-based flow control, 2011. 1.2, 2.3
- [63] InfiniBand Trade Association. Infiniband architecture specification: Release 1.0, 2000. 1.2
- [64] Intel. Intel VTune Amplifier XE performance profiler, . Available at <http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/>. 3.2, 4.1

Bibliography

- [65] Intel. Intel Xeon Processor E5 v3 Family Uncore Performance Monitoring, . Available at <http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html>. 2.2.4
- [66] Intel. Intel performance counter monitor, . Available at <http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization>. 3.4.2, 4.1.4
- [67] Intel. Disrupting the Data Center to Create the Digital Services Economy, 2014. Available at <https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy>. 2.2.4
- [68] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High Performance RDMA-based Design of HDFS over Infiniband. In *SC*, pages 1–12, 2012. 2.3
- [69] Ryan Johnson and Ippokratis Pandis. The bionic DBMS is coming, but what will it look like? In *CIDR*, 2013. 2.2.4
- [70] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009. 2.1, 3.2.1, 4.1.1, 5.3.1
- [71] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009. 1, 1.3, 2.1, 3.2.1, 4.1, 4.5, 5.3.1
- [72] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3:681–692, 2010. 2.1, 3.2.1, 3.4.1, 3.4.3, 4.1.1, 5.3.1
- [73] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Eliminating unscalable communication in transaction processing. *Vldb Journal*, 23(1):1–23, 2014. 1.3, 2.1
- [74] Evan Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010. 1.3, 2.1, 3.3.1
- [75] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An Appliance for Big Data Analytics. pages 1–13, 2015. 2.3
- [76] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A Scalable Lock Manager for Multicores. In *SIGMOD*, pages 73–84, 2013. 2.1
- [77] Anuj Kalia, Michael Kaminsky, and David Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014. 2.3

-
- [78] Alfons Kemper and Thomas Neumann. HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011. [1](#), [1.3](#), [2.1](#), [6.3](#)
- [79] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012. [2.3](#)
- [80] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast Memory-Optimized Database System for Heterogeneous Workloads. *SIGMOD*, 2016. [2.1](#), [5.2.4](#), [5.5](#)
- [81] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*, pages 691–706, 2015. [1](#), [2.1](#), [5.5](#)
- [82] Hideaki Kimura, Goetz Graefe, and Harumi Kuno. Efficient Locking Techniques for Databases on Modern Hardware. In *ADMS*, 2012. [2.1](#)
- [83] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *ADMS*, pages 74–85, 2014. [1.2](#), [2.1](#), [2.3](#)
- [84] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*, pages 468–479, 2013. [2.2.4](#)
- [85] Pranay Koka, Michael O McCracken, Herb Schwetman, Xuezhe Zheng, Ron Ho, and Ashok V Krishnamoorthy. Silicon-photonic network architectures for scalable, power-efficient multi-chip systems. In *ISCA*, pages 117–128, 2010. [2.2.4](#)
- [86] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Eurosys*, pages 113–126, 2013. [2.3](#), [5.5](#)
- [87] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981. [1.3](#)
- [88] George Kurian, Jason E Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C Kimerling, and Anant Agarwal. Atac: a 1000-core cache-coherent processor with on-chip optical network. In *PACT*, pages 477–488, 2010. [2.2.4](#)
- [89] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013. [2.1](#)
- [90] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwillig. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011. [1](#), [1.3](#), [2.1](#)

Bibliography

- [91] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, pages 580–591, 2014. 2.1
- [92] Daniel Lenoski and James Laudon. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, pages 241–251, 1997. 1.2, 2.3
- [93] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013. 2.1, 5.5
- [94] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. NUMA-aware Algorithms: The Case of Data Shuffling. In *CIDR*, 2013. 2.2.3
- [95] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013. 2.1
- [96] Linux. numactl - linux man page. Available at <http://linux.die.net/man/8/numactl>. 4.1.4
- [97] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-Out Processors. In *ISCA*, pages 500–511, 2012. 2.2.4
- [98] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, pages 183–196, 2012. 3.6
- [99] Fraser McArthur and Samuel Poon. Leverage data partitioning for scalability and high performance on Linux. Available at <http://www.ibm.com/developerworks/data/library/techarticle/dm-0601poon>. 3.7
- [100] MemSQL. Available at <https://www.mongodb.org/>. 1
- [101] Microsoft. Analytics Platform System, June 2015. Available at <http://www.microsoft.com/en-us/server-cloud/products/analytics-platform-system>. 2.3, 3.2
- [102] Microsoft SQL Server 2014. Cross-Database Queries, 2015. Available at <http://msdn.microsoft.com/en-us/library/dn584627.aspx>. 2.3
- [103] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX*, pages 103–114, 2013. 2.3
- [104] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *TODS*, 11(4):378–396, 1986. 2.1, 2.3
- [105] MongoDB. Available at <http://www.memsql.com/>. 1
- [106] Héctor Montaner, Federico Silla, Holger Froning, and José Duato. MEMSCALE: A Scalable Environment for Databases. In *HPCC*, pages 339–346, 2011. 2.3

-
- [107] Dheemanth Nagaraj and Chris Gianos. Intel Xeon Processor D: The First Xeon Processor Optimized for Dense Solutions. In *HotChips*, 2015. 2.3
- [108] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. *OSDI*, pages 511–524, 2014. 5.5
- [109] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, pages 677–689, 2015. 5.5
- [110] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark (TATP), 2009. Available at <http://tatpbenchmark.sourceforge.net/>. 4.5.1
- [111] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-Out NUMA. In *ASPLOS*, 2014. 2.3
- [112] NuoDB. Available at <http://www.nuodb.com/>. 1
- [113] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, pages 29–41, 2011. 2.3
- [114] Oracle. Project RAPID. Available at https://labs.oracle.com/pls/apex/f?p=labs:49:::::P49_PROJECT_ID:14. 2.3
- [115] Oracle. Oracle’s Sun Server X4-8 System Architecture, 2014. Available at <http://www.oracle.com/technetwork/server-storage/sun-x86/documentation/x4-8-system-architecture-2210935.pdf>. 2.3
- [116] Oracle Corp. Exadata Database Machine, June 2015. Available at <https://www.oracle.com/engineered-systems/exadata/database-machine-x4-8/features.html>. 1, 2.3, 3.2
- [117] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015. 2.3
- [118] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-Oriented Transaction Execution. *PVLDB*, 3(1):928–939, 2010. 1, 1.3, 2.1, 3.4.2, 4.1.1, 4.3.1
- [119] Ippokratis Pandis, Pınar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011. 1.3, 2.1, 3.4.2, 4.1.1
- [120] Ruud van der Pas. Getting OpenMP Up to Speed. *SC OpenMP Booth*, 2015. 2.3

Bibliography

- [121] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, 2011. [1.3](#), [2.1](#)
- [122] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012. [2.1](#), [3.1](#), [3.3.1](#), [3.7](#), [4.6](#)
- [123] Orestis Polychroniou and Kenneth A. Ross. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD*, pages 755–766, 2014. [2.2.3](#)
- [124] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. Track join: distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494, 2014. [2.3](#)
- [125] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *ICDE*, 2014. [2.2.3](#)
- [126] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. *ADMS*, 2013. [2.2.3](#)
- [127] 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 *PVLDB*, volume 8, pages 1442–1453, 2015. [2.2.3](#), [4.1.4](#)
- [128] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, pages 13–24, 2014. [2.2.4](#)
- [129] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441, 2013. [2.1](#), [3.1](#), [3.3.1](#)
- [130] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003. [2.1](#)
- [131] Kun Ren, Alexander Thomson, and Daniel J Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, pages 821–832, 2014. [5.5](#)
- [132] Wolf Rodiger, Tobias Muhlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014. [2.3](#)

-
- [133] Wolf Rodiger, Tobias Muhlbauer, Alfons Kemper, and Thomas Neumann. High-speed Query Processing over High-speed Networks. *PVLDB*, 9(4):228–239, 2015. 2.3
- [134] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, pages 1311–1326, 2015. 5.5
- [135] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It’s Time for Low Latency. In *HotOS’13*, pages 11–11, 2011. 2.3
- [136] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize When You can Distribute? In *EuroSys*, pages 17–30, 2011. 1.3, 2.1
- [137] SAP. Hana appliance. Available at <http://hana.sap.com/deployment/on-premise.html>. 1.2, 2.3
- [138] Daniel Schall and Theo Harder. Dynamic physiological partitioning on a shared-nothing database cluster. In *ICDE*, pages 1095–1106, 2015. 2.1, 4.6
- [139] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *PVLDB*, 7(12):1035–1046, 2014. 2.1
- [140] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*, 2016. 4.1.2
- [141] Ram Sivaramakrishnan and Sumti Jairath. Next Generation SPARC Processor Cache Hierarchy. In *HotChips*, 2014. 2.2.4
- [142] Stephen Somogyi, Thomas F. Wenisch, Nikos Hardavellas, Jangwoo Kim, Anastasia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *WMPI*, pages 37–45, 2004. 1.3
- [143] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9:4–9, 1986. 1.3, 2.1
- [144] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it’s time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007. 1, 1.3, 2.1, 3.2.1, 3.2.3, 4.3.1, 5.1.1, 5.1.3
- [145] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *USENIX*, pages 347–353, 2012. 2.3
- [146] 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. 8(3):245–256, 2014. 2.1

Bibliography

- [147] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011. 2.2.3
- [148] Alexander Thomson and Daniel Abadi. The case for determinism in database systems. *PVLDB*, 3, 2010. 2.3
- [149] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database systems. In *SIGMOD*, pages 1–12, 2012. 2.1, 2.3, 5.5
- [150] Christian Tinnefeld, Donald Kossmann, Martin Grund, Joos-Hendrik Boese, Frank Renkes, Vishal Sikka, and Hasso Plattner. Elastic online analytical processing on RAM-Cloud. In *EDBT*, pages 454–464, 2013. 2.3
- [151] Pınar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. *VLDB J.*, 22(2):151–175, 2013. 1.3, 2.1
- [152] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: Analyzing TPC’s OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored. In *EDBT*, pages 17–28, 2013. 2.1, 3.7, 4.1.2, 6
- [153] TPC. TPC benchmark B standard specification, revision 2.0, 1994. Available at <http://www.tpc.org/tpcb>. 3.2.3
- [154] TPC. TPC benchmark C standard specification, revision 5.11, 2010. Available at <http://www.tpc.org/tpcc>. 2.2.2, 3.2.3, 4.5.1, 5.1.3
- [155] TPC. TPC benchmark E standard specification, revision 1.12.0, 2010. Available at <http://www.tpc.org/tpce>. 3.7
- [156] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *APSys*, pages 17:1–17:5, 2011. 1.2, 2.3
- [157] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, pages 18–32, 2013. 1, 2.1, 3.2.1, 3.6, 5.5
- [158] Basant Vinaik and Rahoul Puri. Oracle’s Sonoma Processor: Advanced Low-cost SPARC Processor for Enterprise Workloads. In *HotChips*, 2015. 2.3
- [159] Werner Vogels. Eventually consistent. *CACM*, 52:40–44, 2009. 1, 1.3, 2.3
- [160] VoltDB. Available at <https://voltdb.com/>. 1

-
- [161] Mehul Wagle, Daniel Booss, and Ivan Schreter. NUMA-Aware Memory Management with In-Memory Databases. In *TPCTC*, 2015. 2.2.3
- [162] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 49–70, 1983. 2.3
- [163] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, 2015. 5.5
- [164] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27:15–31, 2007. 2.2.4
- [165] Michael Wilson. Disabling NUMA parameter, 2011. Available at <http://www.michaelwilsondba.info/2011/05/disabling-numa-parameter.html>. 2.2.3
- [166] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *PVLDB*, 7(11):963–974, 2014. 2.2.4
- [167] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. In *ISCA*, pages 249–260, 2013. 2.2.4
- [168] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *ASPLOS*, pages 255–268, 2014. 2.2.4
- [169] William A Wulf and C Gordon Bell. C. mmp: a multi-mini-processor. 2.3
- [170] Cong Yan and Alvin Cheung. Leveraging lock contention to improve OLTP application performance. *PVLDB*, 9(5):444–455, 2016. 6.3
- [171] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB*, 8(3):209–220, 2014. 2.1
- [172] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*, 2016. 2.1
- [173] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB*, 9(6):504–515, 2016. 5.5
- [174] Ce Zhang and Christopher Ré. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12):1283–1294, 2014. 2.2.3
- [175] Charles Zhang. Mars: A 64-core ARMv8 Processor. In *HotChips*, 2015. 2.2.4

Danica Porobić

EPFL IC IIF DIAS, BC 241, Station 14, 1015 Lausanne, Switzerland

Office phone: +41216931429 Cell phone: +41793996369

e-mail: danica.porobic@epfl.ch

website: danica.azurewebsites.net

Research interests:

Efficient data management systems, transaction processing architectures for modern hardware

Education:

- 2010-2016
 - École Polytechnique Fédérale de Lausanne, *PhD in Computer Science*
 - Thesis topic: High Performance Transaction Processing on Non-Uniform Hardware Topologies
 - Advisor: Prof. Anastasia Ailamaki
- 2008-2010
 - University of Novi Sad, Faculty of Sciences, *MSc in Informatics*
- 2004-2008
 - University of Novi Sad, Faculty of Sciences, *BSc in Informatics*
 - Best student of the Faculty of Sciences
 - Best student in the field of study, University of Novi Sad

Publications:

- D. Porobic, P. Tözün, R. Appuswamy, A. Ailamaki: “More Than A Network: Distributed OLTP on Clusters of Hardware Islands”, DaMon 2016
- U. Sirin, P. Tözün, D. Porobic, A. Ailamaki: “Microarchitectural Analysis of In-memory OLTP”, SIGMOD 2016
- D. Porobic, I. Pandis, M. Branco, P. Tözün, A. Ailamaki: “Characterization of the Impact of Hardware Islands on OLTP”, VLDB Journal 2015
- D. Cervini, D. Porobic, P. Tözün, A. Ailamaki: “Applying HTM to an OLTP System: No Free Lunch”, DaMon 2015
- A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, I. Psaroudakis: “How to Stop Under-Utilization and Love Multicores” (*tutorial*), ICDE 2015
- D. Porobic: “Smoothing non-uniform communication latencies for OLTP” (abstract), CIDR 2015
- I. Psaroudakis, T. Kissinger, D. Porobic, T. Ilsche, E. Liarou, P. Tözün, A. Ailamaki, W. Lehner: “Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries”, DaMon 2014
- A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, I. Psaroudakis: “How to Stop Under-Utilization and Love Multicores” (*tutorial*), SIGMOD 2014
- D. Porobic, E. Liarou, P. Tözün, A. Ailamaki: “ATraPos: Adaptive Transaction Processing on Hardware Islands”, ICDE 2014

- D. Porobic, I. Pandis, M. Branco, P. Tozun, A. Ailamaki: "OLTP on Hardware Islands", In Proceedings VLDB, Volume 5, 2012.
- I. Pandis, P. Tozun, M. Branco, D. Karampinas, D. Porobic, R. Johnson, A. Ailamaki: "A Data-oriented Transaction Execution Engine and Supporting Tools" (*demo*), SIGMOD 2011 **Best Demonstration Award**
- B. Uzelac, N. Matkovic, D. Porobic: "Testing Spatial Methods in SQL Server 11.0", DBTest 2011
- M. Kallay, D. Porobic: "Buffer with Arcs on a Round Earth", GiScience 2010
- D. Porobić: "Efficient distance computation between sets of planar geometric objects" (in Serbian), Telfor 2009
- B. Dresevic, B. Bjelajac, D. Porobic: "Non-Language-Specific Raw Ink Segmentation", Microsoft Defensive Publication, ip.com, 2008

Awards/Honors:

- Best Demonstration Award, ACM SIGMOD Conference 2011
- „Aleksandar Saša Popović“ University award for exceptional graduation thesis in the field of Computer Science, Novi Sad 2009
- University and faculty award for excellent results during the undergraduate studies
- University and faculty award for excellent results in the first, third and fourth year of studies
- EFG Eurobank Scholarship for best university students in Serbia, 2007
- Bronze medal at International Olympiad in Informatics, Athens, Greece, 2004
- Bronze medal at Balkan Olympiad in Informatics, Plovdiv, Bulgaria, 2004
- Silver medal at Balkan Olympiad in Informatics, Iasi, Romania, 2003

Fellowships:

- 11/2011-05/2014 – Oracle Labs Fellowship
- 09/2010-08/2014 – Dositeja Fellowship from the Serbian Government for students studying abroad
- 09/2010-09/2011 – EPFL I&C School Fellowship
- 01/2005-09/2008 - Serbian Ministry of Science Fellowship
- 01/2004-12/2004 - Serbia Tomorrow Fellowship
- 10/2003-12/2003 - Serbian Ministry of Science and Environment Fellowship
- 10/2002-09/2003 - Serbian Foundation of Scientific Youth Development Fellowship

Professional experience:

- September 2010-present - École Polytechnique Fédérale de Lausanne
 - Doctoral Assistant in the Data-Intensive Application and Systems Laboratory
 - Working on scaling up transaction processing systems on non-uniform hardware
- July-October 2013 – Oracle Labs, Belmont, CA, USA
 - Research Intern
 - Worked on the RAPID project
- February 2008-July 2010 – Microsoft Development Center Serbia, Belgrade
 - Software Development Engineer in SQL Engine Team

- Worked on the core geometry algorithms of the SQL Spatial Library
- August-October 2007 – Microsoft Corporation, Redmond, WA, USA
 - Software Development Engineer in Test Intern with Windows Serviceability division
 - Worked on improving the test coverage of the networking stack in Windows Vista
- July-October 2006 - Microsoft Development Center Serbia, Belgrade
 - Software Development Engineer Intern with the Tablet PC team
 - Worked on non-language specific tools for training of handwriting recognizer engines for tablet PC

Conference presentations and invited talks:

- “How to Stop Under-Utilization and Love Multicores” invited tutorial at COMAD 2016
- “Smoothing non-uniform communication latencies for OLTP” at Dagstuhl Seminar on “Rack-Scale Computing” 2015
- “Toward Rack-Scale OLTP” at HPTS 2015
- “Toward Rack-Scale OLTP” at EcoCloud Annual Event 2015
- “Applying HTM to an OLTP System: No Free Lunch” at DaMon 2015
- “How to Stop Under-Utilization and Love Multicores” at ICDE 2015
- “OLTP for modern hardware: scalability and instruction locality” at ETH/Oracle Labs workshop 2015
- “Smoothing non-uniform communication latencies for OLTP” at CIDR 2015
- “Adaptive Transaction Processing on Hardware Islands” at University of Novi Sad, 2014
- “How to Stop Under-Utilization and Love Multicores” at SIGMOD 2014
- “Adaptive Transaction Processing on Hardware Islands” at ICDE 2014
- “HPTS on Islands” at HPTS 2013
- “OLTP on Hardware Islands” at VLDB 2012

Professional activities:

- Program committees: SIGMOD 2016 (Demo)
- Reviewer: SIGMOD Record, ComSIS

Teaching experience:

- Spring 2016: Guest Lecturer, CS-322 Introduction to Database Systems
- Fall 2015: Teaching Assistant, Analyse I
- Spring 2015: Teaching Assistant and Guest Lecturer, CS-322 Introduction to Database Systems
- Fall 2014: Teaching Assistant, CS-450 Advanced Algorithms
- Spring 2014: Teaching Assistant, CS-322 Introduction to Database Systems
- Spring 2013: Teaching Assistant, CS-322 Introduction to Database Systems
- Fall 2012: Teaching Assistant, CS-522 Principles of Computer Systems
- Spring 2012: Teaching Assistant, CS-322 Introduction to Database Systems
- Spring 2011: Teaching Assistant, CS-422 Advanced Databases

Other information:

- Member of the Scientific Committee for the 1st Junior Balkan Olympiad in Informatics, Belgrade, 2007
- Leader of the Serbian team at the 14th Balkan Olympiad in Informatics, Ohrid, Macedonia, 2008
- Member of the Scientific Committee of Serbian Informatics Competitions, 2004-2010
- Presenter at the Festival of Science, Novi Sad, 2009
- Member of the organizing committee of Bubble Cup international student programming competition in 2008 and 2009
- Professional memberships: ACM since 2005, IEEE since 2006

Patents:

- “Buffer construction with geodetic circular arcs”, M. Kallay, D. Porobic, US Patent No 8669983 (March 2014)
- “Workload-aware data placement in read-only memory of a highly distributed system”, D. Porobic, S. Idicula, S. Petride, pending

Languages:

- Native: Serbian
- Fluent: English
- Basic: French

References:

- Prof. Anastasia Ailamaki
Full Professor, École Polytechnique Fédérale de Lausanne, Switzerland
Email: anastasia.ailamaki@epfl.ch
- Prof. Wolfgang Lehner
Full Professor, Technische Universität Dresden, Germany
Email: wolfgang.lehner@tu-dresden.de
- Dr. Garret Swart
Architect, Oracle Corporation, USA
Email: garret.swart@oracle.com

