The VLDB Journal manuscript No.
(will be inserted by the editor)

Characterization of the Impact of Hardware Islands on OLTP

Danica Porobic · Ippokratis Pandis · Miguel Branco · Pınar Tözün · Anastasia Ailamaki ·

Abstract Modern hardware is abundantly parallel and increasingly heterogeneous. The numerous processing cores have non-uniform access latencies to the main memory and processor caches, which causes variability in the communication costs. Unfortunately, database systems mostly assume that all processing cores are the same and that microarchitecture differences are not significant enough to appear in critical database execution paths. As we demonstrate in this paper, however, non-uniform core topology does appear in the critical path and conventional database architectures achieve suboptimal and even worse, unpredictable performance.

We perform a detailed performance analysis of OLTP deployments in servers with multiple cores per CPU (*multicore*) and multiple CPUs per server (*multisocket*). We compare different database deployment strategies where we vary the number and size of independent database instances running on a single server, from a single shared-everything instance to fine-grained shared-nothing configurations. We quantify the impact of non-uniform hardware on various deployments by (a) examining how efficiently each deployment uses the available hardware resources and (b) measuring the impact of distributed transactions and skewed requests on different workloads. We show that no strategy is optimal for all cases and that the best choice depends on the combination of hardware topology and workload characteristics. Finally, we argue that transaction processing systems must be aware of the hardware topology in order to achieve predictably high performance.

Keywords Islands, Shared-everything, Shared-nothing, OLTP, Multisocket multicores, Non-uniform hardware topology

D. Porobic
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland
E-mail: danica.porobic@epfl.ch

I. Pandis
Amazon Web Services
Palo Alto, CA, USA
E-mail: ippo@amazon.com
Work done while author was affiliated with IBM

M. Branco
RAW Labs
Lausanne, VD, Switzerland
E-mail: miguel@raw-labs.com
Work done while author was affiliated with EPFL

P. Tözün
IBM Almaden Research Center
San Jose, CA, USA
E-mail: ptozun@us.ibm.com
Work done while author was affiliated with EPFL

A. Ailamaki
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne and
RAW Labs
Lausanne, VD, Switzerland
E-mail: anastasia.ailamaki@epfl.ch

1 Introduction

Online Transaction Processing (OLTP) is a multi-billion dollar industry [21] and one of the most important and demanding database applications. Innovations in OLTP continue to deserve significant attention, advocated by the recent emergence of appliances [46], startups[1], and research projects (e.g. [15,28,32,33,39,47,58,67]). 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 spend

[1] Such as VoltDB, MongoDB, MemSQL, NuoDB, and others.

significant effort in developing highly-optimized software releases, often with platform-specific optimizations.

Over the past decades, OLTP systems benefited greatly from improvements in the underlying hardware. Innovations in their software architecture have been plentiful but there is a clear benefit from processor evolution. Uniprocessors grew predictably faster with time, leading to better OLTP performance. Around 2005, when processor vendors hit the frequency-scaling wall, they started obtaining performance improvements by adding multiple processing cores to the same CPU chip, forming chip multiprocessors (multicore or CMP); and building servers with multiple CPU sockets of multicore processors (SMP of CMP).

Multisockets of multicores are highly parallel and characterized by heterogeneity 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 and in particular OLTP systems exploit multisockets.

As recent studies argue and this paper corroborates, traditional shared-everything OLTP systems underperform on modern hardware because of (a) excessive communication among various threads [9,22] and (b) contention on the shared data [48,58]. Practitioners report that even commercial shared-everything systems with support for non-uniform memory architectures (NUMA) are hard to tune for modern servers [25,14,70]. On the other hand, shared-nothing deployments [57] face the challenges of (a) higher execution costs when distributed transactions are required [12,16,24,50], even within a single node, particularly if the communication occurs between slower links (e.g., across CPU sockets); and (b) load imbalances due to skew [61].

The goal of this study is to 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. They are run on shared-nothing deployments of varying granularity as well as shared-everything deployments. We place particular emphasis on the impact of the percentage of multipartition transactions. The percentage of multipartition transactions depends both on the workload properties and on the quality of the partitioning scheme, however, finding a good partitioning scheme for complex workloads remains an open problem [16,49,66]. In conjunction with the granularity of instances in a shared-nothing deployment, the percentage of multipartition transactions determines the ratio of distributed transactions.

In this study, we use a state-of-the-art storage manager Shore-MT [28] for the majority of the experiments. We strengthen our conclusions using a state-of-the-art main memory optimized storage manager Silo [67]. Our experiments show that perfectly partitionable workloads, which require no distributed transactions, perform significantly better on fine-grained shared-nothing configurations. On the other hand, non-partitionable workloads favor coarse-grained configurations, due to the overhead of distributed transactions. None of the configurations, however, is optimal for all combinations of workload properties and hardware topologies. Additionally, we find that skewed accesses cause performance to drop significantly when using fine-grained shared-nothing configurations; this effect is less evident on coarser configurations and when using shared-everything deployments.

This paper extends our analysis of the impact of non-uniform hardware topology on transaction processing systems [53]. We enrich our experiments by using additional hardware and software platforms as well as more efficient communication mechanisms. In addition, we include experiments with standard benchmarks, TPC-B and TPC-C, and discuss the impact of the contention for hot data on the performance of different configurations. Our contributions are as follows:

- We demonstrate 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 processor sockets.
- Our experiments show that fine-grained shared-nothing deployments achieve significantly higher throughput than a shared-everything system when the workload is perfectly partitionable. By contrast, when the workload is not partitionable and/or exhibits skew, a shared-everything system has higher performance than a shared-nothing one. Therefore, there is no unique optimal deployment strategy for all workloads.
- We provide and validate the Islands performance model where we take 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 specific configuration optimal differ depending on the particular scenario, however, relative performance trends remain the same in all cases.

The rest of the document is structured as follows. Section 2 presents the background and related work, describing the two main database deployment approaches. Section 3 details the experimental methodology used throughout this study. Section 4 identifies modern hardware trends and discusses their implications on software design. Section 5 describes the impact of hardware topology and workload characteristics such as the percentage of multipartition transactions on the performance of the database systems and Sec-

tion 6 quantifies that impact. Section 7 expands the analysis, measuring the sensitivity to varying transaction size, number of processors, data access skew, and disk accesses. Section 8 discusses the impact of distributed transactions in the context of more complex workloads. Section 9 presents results in the context of Silo main-memory optimized system. Finally, Section 10 summarizes the findings and Section 11 concludes and discusses future work.

2 Background and related work

Shared-everything and shared-nothing database designs, described in the next two sections, are the most widely used approaches for OLTP deployments. In addition to these two extreme points of the design space, in this study we analyze a range of deployments that fall in between. Legacy multisocket machines, which gained popularity in the 1990s as symmetric multiprocessing (SMP) servers, had non-uniform memory access (NUMA) latencies. We discuss NUMA-specific optimizations to database and operating systems, as well as recent work on optimizations for multisockets, in Section 2.3.

2.1 Shared-everything Database Deployments

Within a database node, *shared-everything* is any deployment where a single database instance manages all the available resources. As database servers have long been designed to operate on machines with multiple processors, shared-everything deployments assume equally fast communication between all processors, since each thread needs to exchange data with all of its peers. Until recently, shared-everything was the most popular deployment strategy on a single node. All major commercial database systems adopt it.

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 transactions exhibit significant stalls during execution [2,7,22,62]; a result we corroborate in Section 6.2. It has also been shown that shared-everything systems have frequent shared read-write accesses [9,22], which are difficult to predict [56]. Modern 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 [27,28]. These characteristics make distributed memories (as those of multisockets), distributed caches (as those of multicores), and prefetchers ineffective. A lot of 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 [26,29,31,34]. Recent work suggests a departure from the traditional transaction-oriented execution model, to adopt a data-oriented execution model, circumventing the aforementioned properties - and flaws - of traditional shared-everything OLTP [35,47,48].

The large main memories available in modern servers have sparked a lot of interest in the main-memory optimized transaction processing designs for multisockets. Such systems have been marketed by major vendors for many years, including IBM solidDB [43] and Oracle TimesTen [37], however, only now are they becoming mainstream. Modern multicore optimized main-memory transaction processing systems, such as Hekaton, Silo, and Foedus, use multiversioned latch-free data structures and optimistic concurrency control mechanisms to achieve good scalability by reducing the number of critical sections and their duration [33,39,40,67]. 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 [71].

2.2 Shared-nothing Database Deployments

Shared-nothing deployments [57], based on fully independent (physically partitioned) database instances that collectively process the workload, are an increasingly appealing design even within a single node [32,55,58]. This is due to the scalability limitations of shared-everything systems, which suffer from contention when concurrent threads attempt to access shared resources [27].

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 executes transactions touching data on a single database instance. Systems such as H-Store [58] and HyPer [32] 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.

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 data such that every transaction touches a single instance. Whenever multiple instances must collectively process a request, shared-nothing databases require expensive distributed consensus protocols, such as two-phase commit, which many argue are inherently non-scalable [12,24]. Similarly, handling data and access skew is problematic [61].

The overhead of distributed transactions urged system designers to explore partitioning techniques that reduce the frequency of distributed transactions [16,49,54], and to explore alternative concurrency control mechanisms, such as speculative locking [30], multiversioning [10], optimistic concurrency control [36,39], and deterministic execution [60], 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 [68]. Eventual consistency eliminates the need for synchronous distributed transactions, but it makes programming transactional applications harder, with consistency checks left to the application layer. A promising approach for improving efficiency of distributed transactions is using semantic information about the workload to avoid unnecessary coordination [5].

The emergence of multisocket multicore hardware adds further complexity to the on-going debate between shared-everything and shared-nothing OLTP designs. As Section 4 describes, multisocket multicores introduce an additional level into the memory hierarchy. Communication between processors is no longer uniform: cores that share caches communicate differently from cores in the same socket and other sockets.

2.3 Performance on Multisocket Multicores

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 multisocket servers, enabling NUMA support might lead to high CPU usage and degraded performance [14,70]. Similarly, modern operating systems offer better support for NUMA arhitectures, however, they do not improve application performance out-of-the-box. Tuning existing database systems to multisocket multicores is still a very challenging task [25, 69].

An alternative approach is taken by the Multimed project, which views the multisocket multicore system as a cluster of machines [55]. Multimed uses replication techniques and a middleware layer to split database instances into those that process read-only requests and those that process updates. The authors report higher performance than a single shared-everything instance. However, Multimed does not explicitly address NUMA-awareness and the work is motivated by the fact that the shared-everything system being used has inherent scalability limitations. In this paper, we use two scalable open-source shared-everything systems, Shore-MT [28] and Silo [67].

Table 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

Recent work that analyzes the impact of NUMA on data management systems uses analytical workloads. Zhang and Ré focus on statistical analytics and conclude that awareness of the core topology can improve performance by an order of magnitude compared to the state-of-the-art systems [72]. On the other hand, the majority of the proposals that target building NUMA-aware data management systems focus on removing memory bandwidth bottlenecks for analytical applications and specifically devising efficient join and sorting algorithms that minimize data movement [3,6,42,51]. However, OLTP workloads cannot saturate memory bandwidths and their main problem is ensuring efficient synchronization among threads [52].

Moreover, exploiting NUMA effects at the operating system level is an area of active research. Some operating system kernels such as the Mach [1] and exokernel [19], or, more recently, Barrelfish [8], 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 [11, 17,59]. Such schedulers have recently been adapted to task-oriented analytical database engines [20], however, they likely require extensive changes to a traditional database engine.

3 Experimental setup

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 of 6-core CPUs, and one with eight sockets of

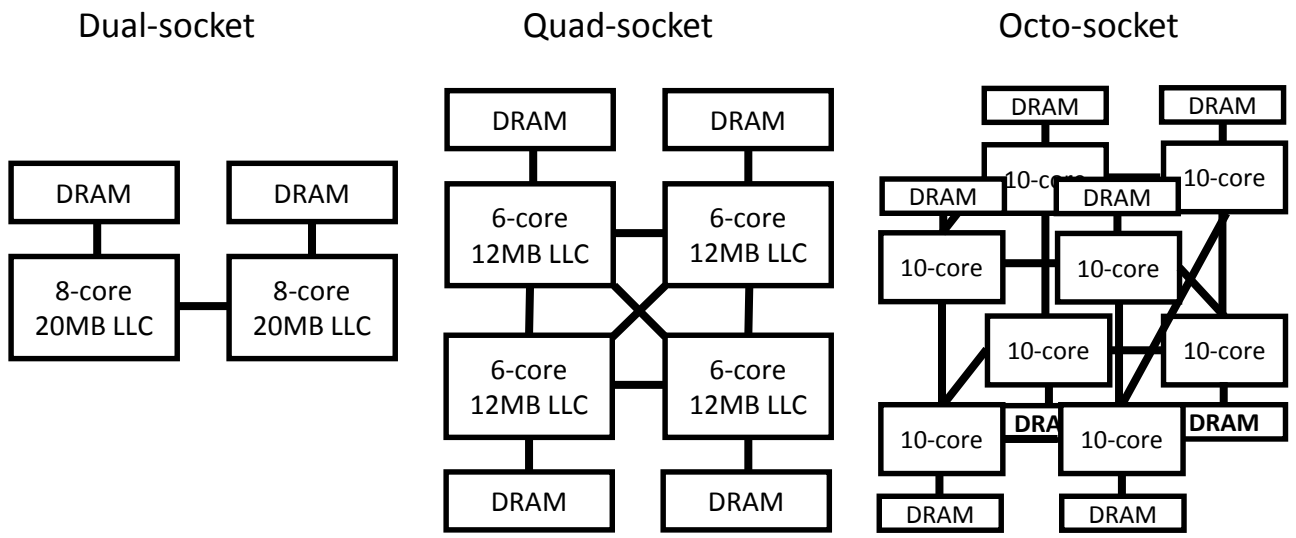

Fig. 1 Topology of the three machines used in the experiments.

10-core CPUs. The topology of these machines is depicted in Figure 1: 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 away [2]. The two socket machine is a typical representative of the multisockets used by the major cloud service providers such as Amazon Web Services [4]. 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 [45,46].

Hardware and tools. Table 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 7.4, we use two 146 GB 10kRPM SAS 2,5" HDDs in RAID-0.

We use Intel VTune Amplifier XE 2013 to collect basic micro-architectural and time-breakdown profiling results. VTune does hardware counter sampling, which is both accurate and light-weight. 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, 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.

[2] For more details see `http://www.supermicro.com/manuals/motherboard/7500/X8OBN-F.pdf`

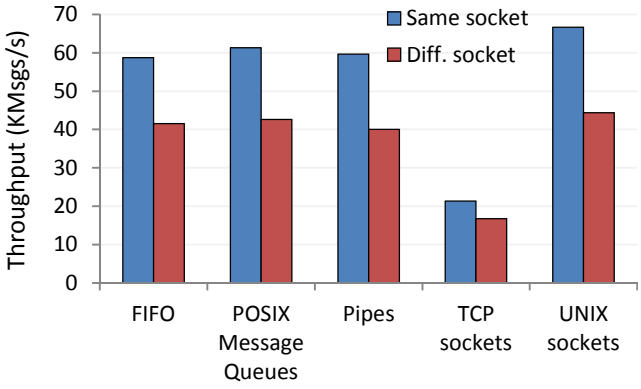

Fig. 2 Throughput of message exchanging (in thousands of messages exchanged per second) for a set of inter-process communication mechanisms. Unix domain sockets are the highest performing.

IPC mechanisms. The performance of any shared-nothing system heavily depends on the efficiency of its communication layer. Figure 2 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 facilities. Unix domain sockets achieve the highest performance and are used throughout the remaining evaluation. In Section 6.4 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.

3.1 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-MT [28] and Silo [67]. 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 [28]. Shore-MT is the improved version of the SHORE storage manager, originally developed as an object-relational data store [13]. 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 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 [58] with local and remote transaction parts known apriori. This allows coordinator and subordinate 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 [26] and Aether holistic logging [29]. 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.

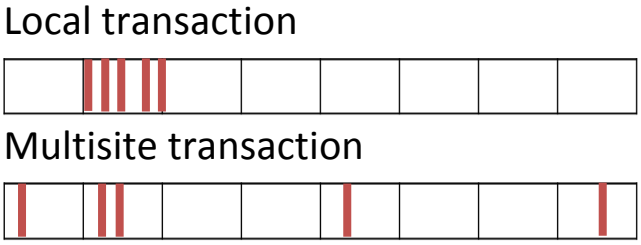

Fig. 3 Examples of microbenchmark transactions with N = 5 where the second partition is the local one.

3.2 Microbenchmark workload and experimental methodology

In our 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 assign 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 $\sim$ 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 (minimizing NUMA effects) 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 4.

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.

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 and define the two transaction types as follows:

- **Local transactions** perform their action (read or update) on the N rows located in the local site;
- **Multisite transactions** perform their action (read or update) on one row located in 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 wide range of workload types from the perfectly partitionable to the completely un-partitionable ones that access rows from many partitions requiring distributed transactions. For completeness, we also include well known industry standard TPC benchmarks, TPC-B and TPC-C, that also feature remote (multisite) transactions.

3.3 Standard workloads

TPC-B [63] 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 [64] benchmark models a transactional database of the wholesale supplier. Its schema

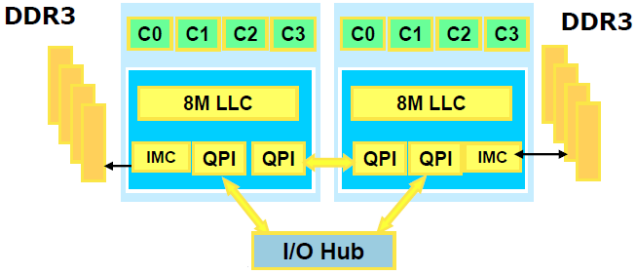

Fig. 4 Block diagram of a commodity multisocket multicore machine. Cores (C0 to C3) communicate either through a shared last-level cache (LLC), an interconnect across sockets (QPI) or main memory.

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 [58]. 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.

4 Hardware has Islands

Hardware has long departed from uniprocessors, which had predictable and uniform performance. Due to thermal and power limitations, vendors cannot improve the performance of processors by clocking them to higher frequency or by using more advanced techniques such as increased instruction width and extended out-of-order execution. Instead, vendors rely on 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

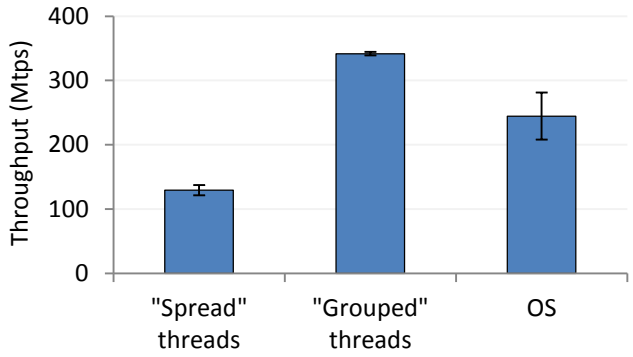

Fig. 5 Results of a counter benchmark where groups of 10 threads are incrementing a shared counter. Allocating threads and memory in a topology-aware manner provides the best performance and lower variability.

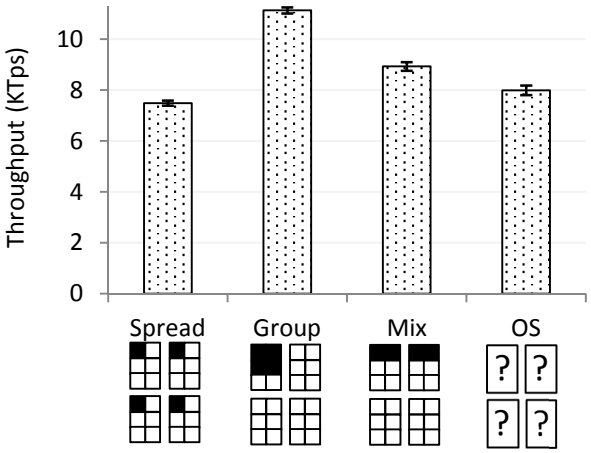

Fig. 6 Throughput of the system when varying placement of 4 worker threads. Running the `TPCC-Payment` workload with all cores on the same socket achieves 20-30% higher performance than other configurations.

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 and are expected to remain popular in the future. Figure 4 shows a simplified block diagram of a typical machine that has two sockets with quad-core CPUs (adapted from [41]). Communication between the numerous cores happens through different mechanisms. For example, cores in the same socket share a common cache, while cores located in different sockets communicate via the interconnect (called *QPI* for Intel processors). Cores may also communicate through the main memory if the data is not currently cached. The result is that the inter-core communication is variable: communication in multicores is more efficient than in multisockets, which communicate over a slower, power-hungry, and often bandwidth-limited interconnect.

Hence, there are two main trends in modern 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.

4.1 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 multisockets with different topologies [18].

We illustrate the impact of non-uniform topology on the efficiency of synchronization among threads with a simple microbenchmark. Figure 5 plots the throughput of a program running on a machine that has 8 CPUs with 10 cores each (the "Octo-socket" machine of Table 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 [8,17], 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 very promising line of research [20].

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 1). Figure 6 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 ob-

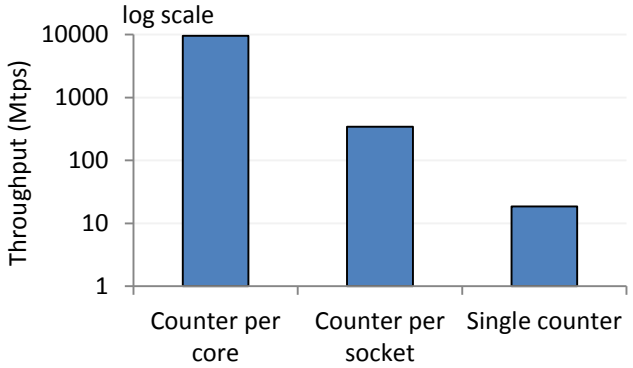

Fig. 7 Results of a counter benchmark where we always use 80 threads and change the number of counters they increment. Improving locality of communication improves the performance by an order of magnitude.

servations of Figure 5: 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., [7,22]).

4.2 Abundant Hardware Parallelism

Another major trend is the abundant hardware parallelism available in modern database servers. Higher hardware parallelism potentially causes additional contention in multi-socket multicore systems, as a higher number of cores compete for shared data accesses. Figure 7 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 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.

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 [64] Payment transaction, where all the requests are local and, hence, the workload is perfectly partitionable on `Warehouses`. We plot the results on Figure 8. The fine-grained shared-nothing configuration outperforms shared-everything by 4.5x, due

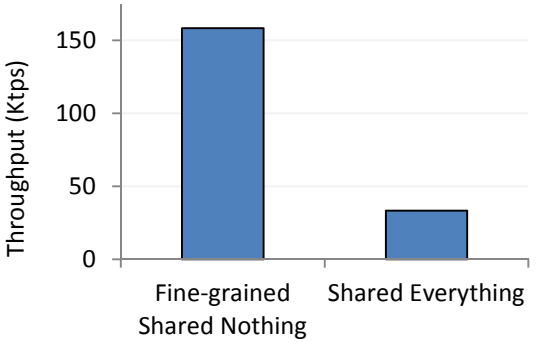

Fig. 8 Running the `TPC-C` benchmark with only local transactions. Fine-grained shared-nothing is 4.5x faster than shared-everything.

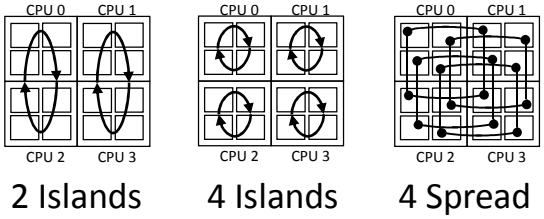

Fig. 9 Different shared-nothing configurations on a four-socket four-core machine.

in large part to contention on the `Warehouse` table in the shared-everything case.

In summary, modern hardware poses new challenges to software systems. Contention and topology have a significant impact on performance and 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. The next section argues in favor of topology-aware OLTP deployments that adapt to those hardware islands.

5 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 9 illustrates three different shared-nothing configurations. The two left-most configurations, labeled *"2 Islands"* and *"4 Islands"*, dedicate different number of cores per instance, but, for the given size, 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 more heterogeneous the design space over the possible shared-nothing configurations increases, and it is harder to determine the optimal deployment.

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 protocols (such as two-phase commit) need 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." [3]

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 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., [16,49,54]), then the more data a database contains the more likely for a transaction to be local.

[3] Explaining, among other reasons, the high compensation for skilled database administrators.

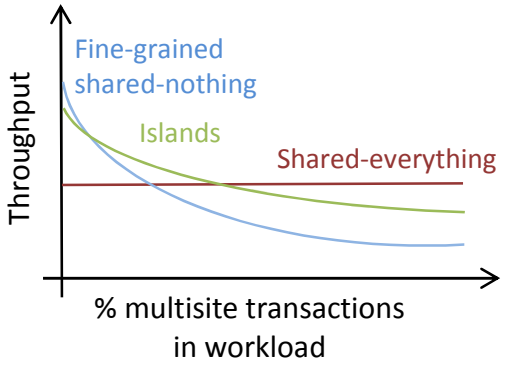

Fig. 10 Performance of various deployment configurations as the percentage of multisite transactions increases.

Given the previous reasoning one could argue that an optimal shared-nothing configuration consists of a few coarse-grained database instances. This would be a naive assumption as it ignores the effects of hardware parallelism and variable communication costs. For example, if we consider the 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 contenting 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, $T_{islands} > T_{naive}$.

In this paper we characterize the behavior of *OLTP Islands*, which are hardware topology-aware shared-nothing deployments. Figure 10 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 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 e.g. skew.

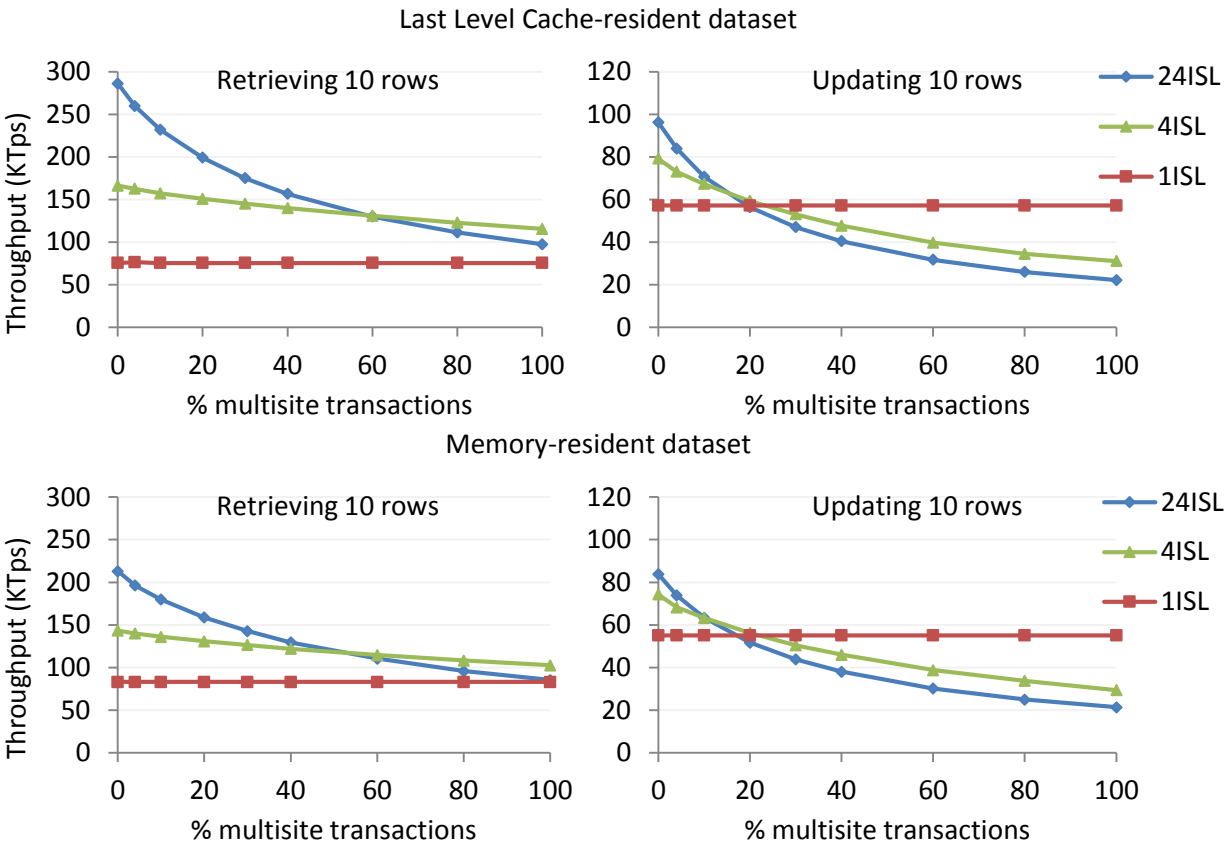

Fig. 11 Performance as the number of distributed transactions increases on cache and memory resident datasets. While shared-everything remains stable, performance of share-nothing configurations decreases. Smaller instances benefit a lot from cache-resident datasets, while smaller dataset incurs more data movement than the larger one due to poorer locality.

6 Impact of multisite transactions

In this section we analyze in depth the impact of good partitioning scheme on performance of different configurations. If a good partitioning scheme exists for a particular workload, the resulting percentage of multisite transaction will be low 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):

- **Fine-grained shared-nothing** (labeled 24ISL) is a deployment configuration where data is divided to 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 eliminates 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 communication mechanism and run experiments using 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 re-

place 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.

6.1 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 [30] or to determine an initial optimal partitioning strategy [16,49,54]. Our experiment, shown in Figure 11, 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 coarse-grained (4ISL) shared-nothing configurations have high performance for the workloads that contain only local transactions. The performance improvement compared to shared-everything is especially high for the read-only transactions and fine-grained configurations that run in 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 transactions 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 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 11, 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 that 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 dataset 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 ($\sim$ 600 MB) and plot throughput on the lower half of Figure 11. 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 6.3).

6.2 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 12 (top)) and memory-resident datasets (Figure 12 (bottom)).

The leftmost graph of the top row in Figure 12, 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 12 (top)). This is due to the presence of—expensive—last-level cache (LLC) misses (shown in the right-most graph in Figure 12 (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 12 (second from the right in the top row) that plots the ratio of cycles the system is accessing shared data to all cycles.

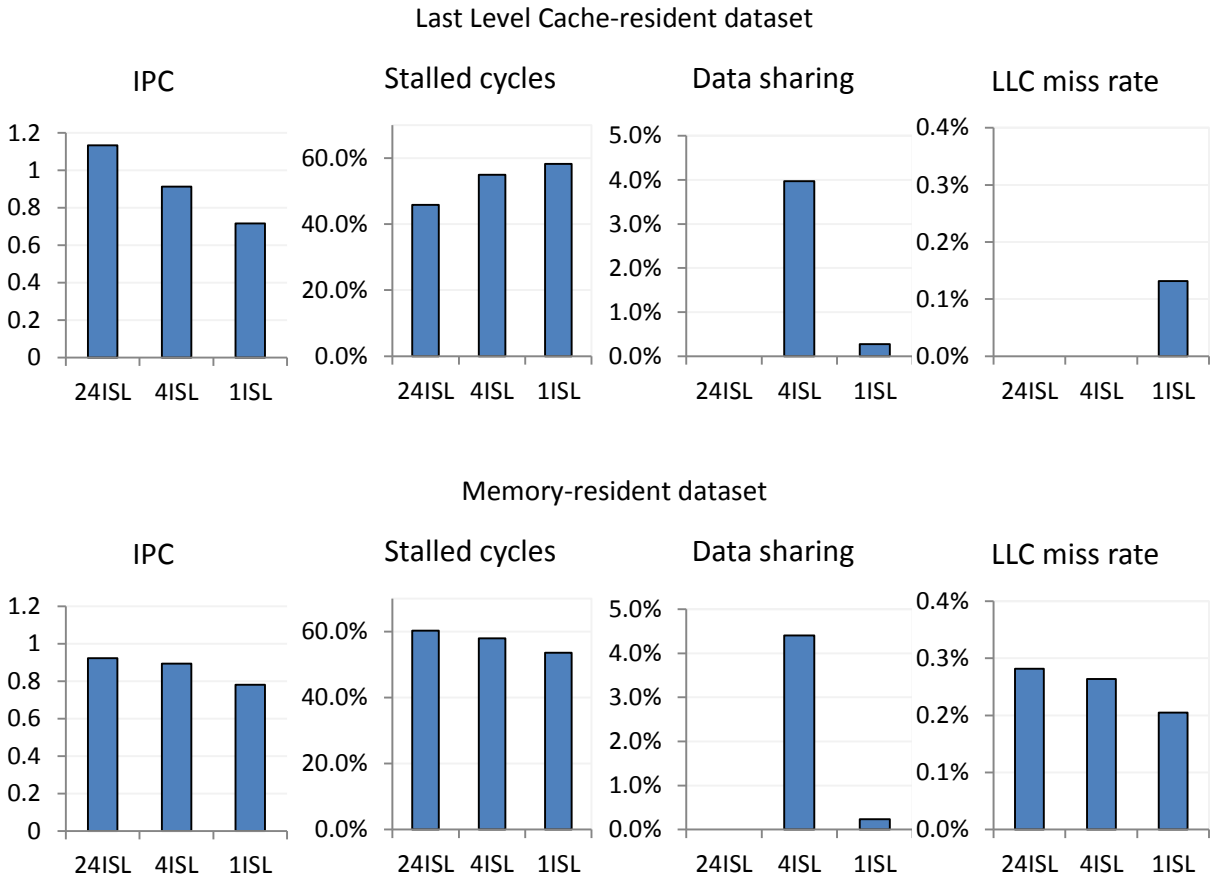

Fig. 12 Microarchitectural data for different deployments and datasets: smaller instances benefit a lot from locality in the workload.

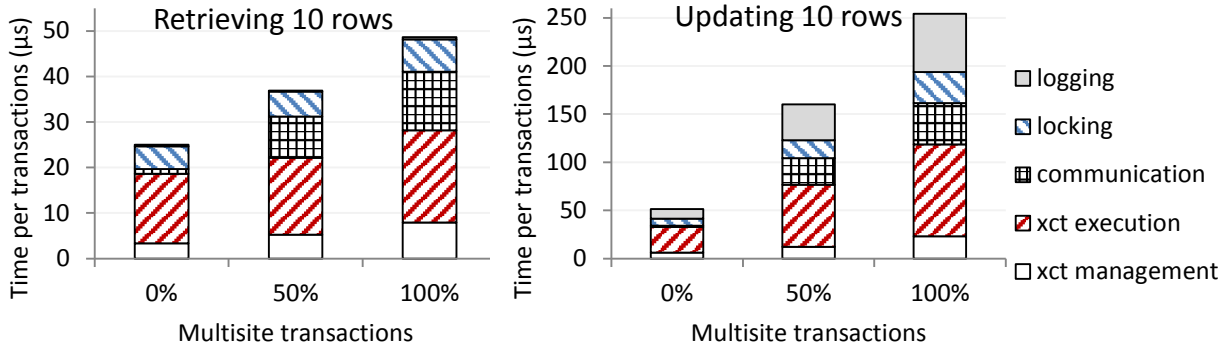

Fig. 13 Time breakdown for a transaction that retrieves (left) or updates (right) 10 rows and uses unix domain sockets for communication. The cost of communication dominates in the cost of distributed transaction in the read-only case, while in the update case overheads are divided between communication and additional logging.

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 12. 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.

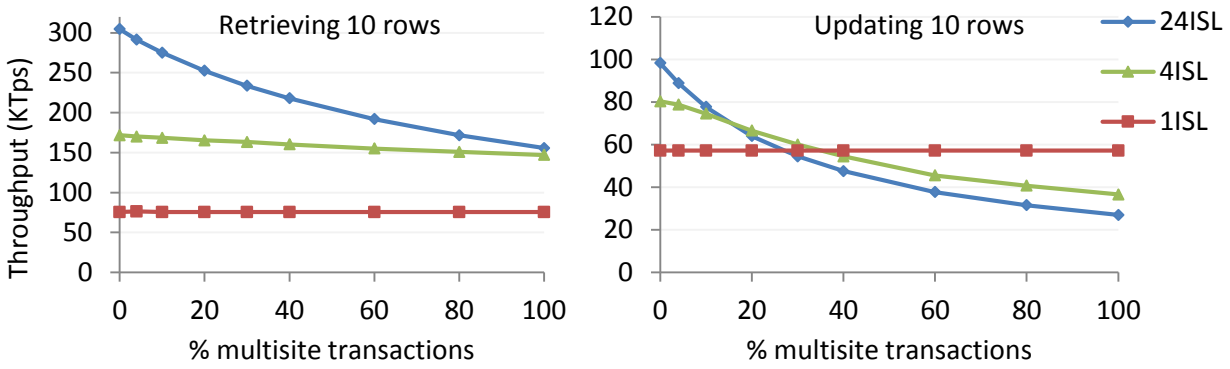

Fig. 14 Performance as the percentage of multisite transactions increases using shared memory communication channel. Read-only distributed transactions benefit from faster communication much more than the update ones.

6.3 Profiling

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 13 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.

6.4 Impact of the communication channel

Although unix domain sockets are the fastest messaging mechanism provided by the operating system (Section 3), they still cause large communication overheads when executing distributed transactions. 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 6.1 with a small dataset and plot the throughput in Figure 14. We observe that the performance trends of various configurations are the same as in the case of unix domain socket communication channels (Figure 11 (top)). However, 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 13) 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 6.3 with the shared memory communication channel and show the results in Figure 15. 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.

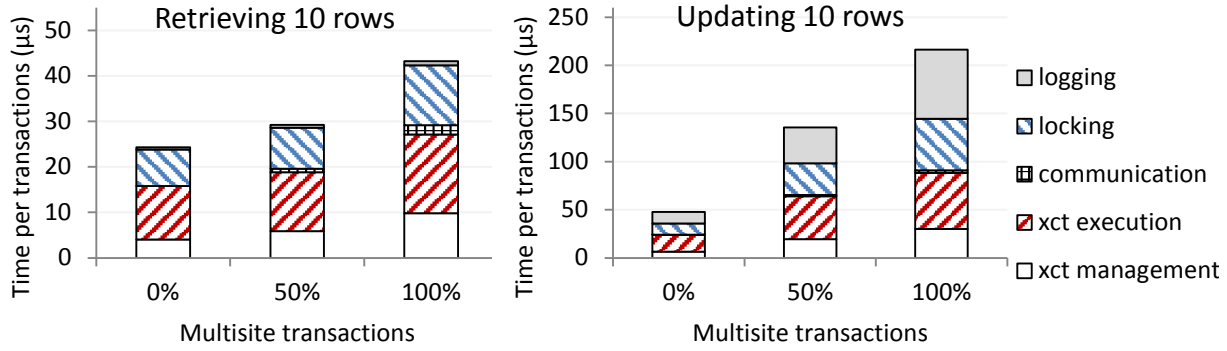

Fig. 15 Time breakdown for a transaction that retrieves (left) or updates (right) 10 rows and uses shared memory channels for communication. Lower cost of communication decrease significantly decrease the costs in the read-only case, while other costs increase in the update costs as they cannot be overlapped with communication anymore.

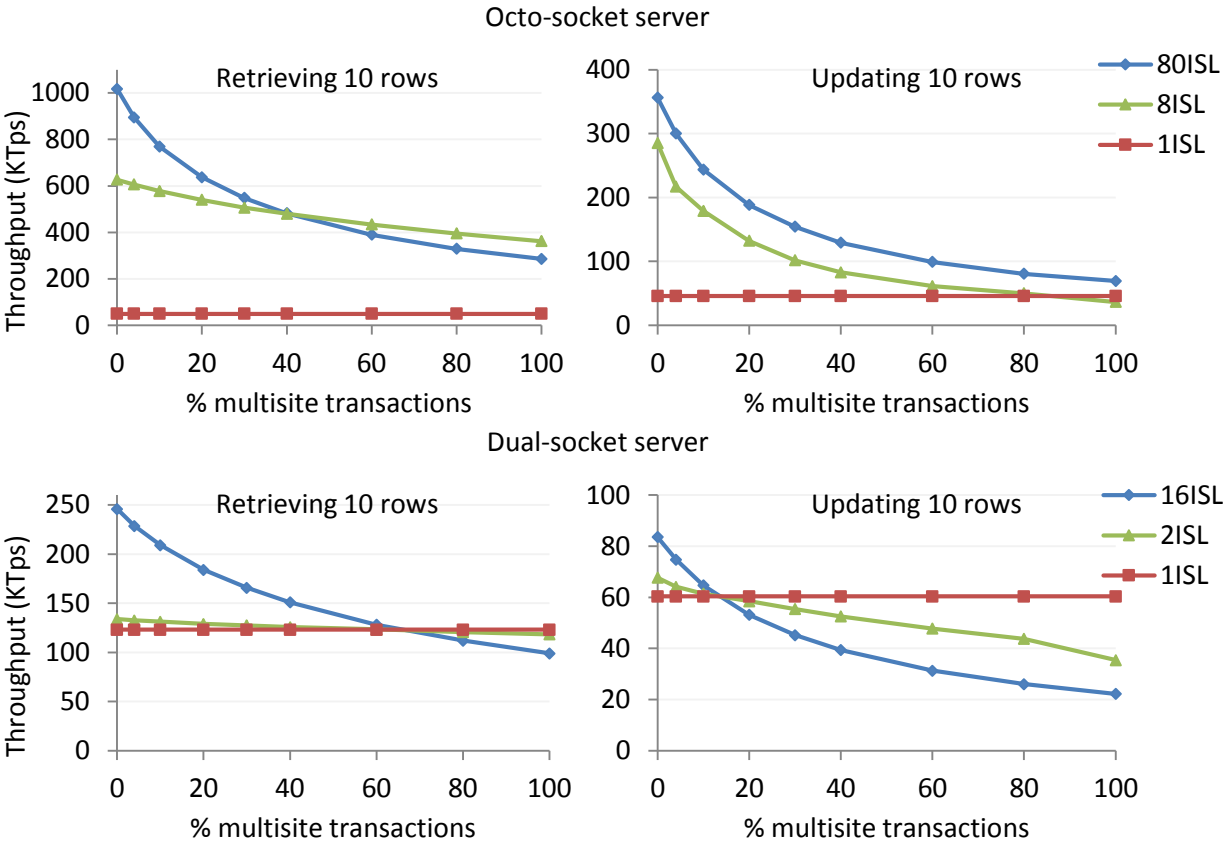

Fig. 16 Performance as the number of distributed transactions increases on dual and octo socket servers. Trends are common across machines, however, hardware topology determines relative performance of different configurations.

6.5 Different topologies

The number of islands is one of the most important factors that determine 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 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 16 (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 the total of 800,000 rows. Each of the eight processors on this machines has ten cores, hence we have 80 instances in the fine-grained (labeled 80ISL) and 8 instances in the coarse-grained shared-nothing deployment (labeled 8ISL). Similarly to the smaller, quad-socket, server used in the experiment in Section 6.1, throughput of the shared-everything system is constant irrespective of the percentage of multisite transactions. How-

ever, 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 7.2. 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 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.

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 16. This server has two eight core processors, so we deploy fine-grained shared-nothing configuration with 16 instances and the coarse-grained 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 performance 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 system benefits from optimized logging to offer consistently good performance for update transactions.

7 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.

7.1 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 smaller 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.

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

Figure 17 (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 [23].

The costs of multisite read-only transactions (Figure 17 right) show the opposite trend from 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 singe transaction. This trend flattens out once all instances are involved in 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 are so high that for large transactions, it has worse performance than all shared-nothing configurations which execute distributed transactions.

7.1.2 Update Case: Additional Logging Overhead Is Significant

The left graph of Figure 18 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

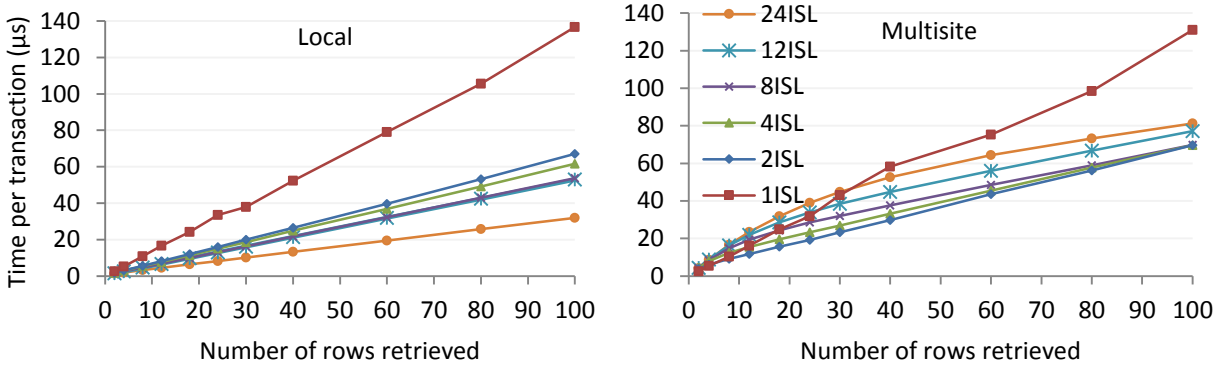

Fig. 17 Cost of local and multisite transactions in the read-only microbenchmark. For multisite transactions, communication costs rise until all instances are involved in every transaction.

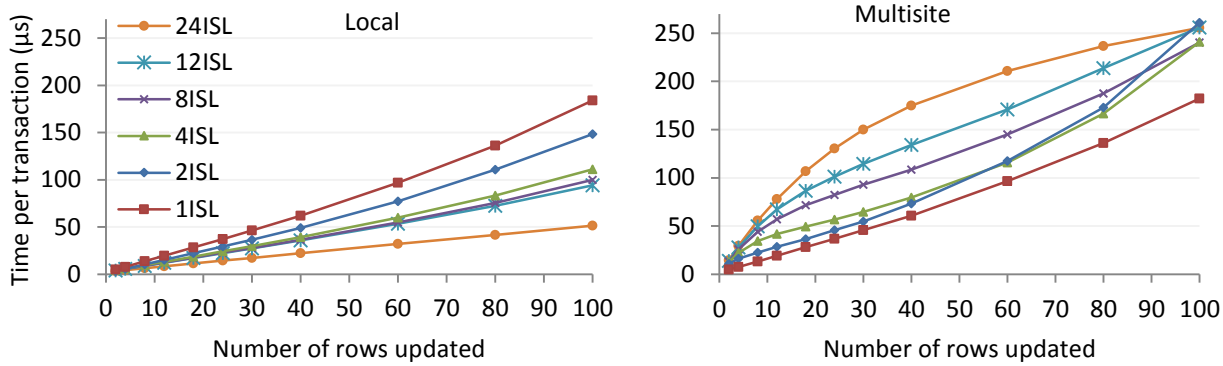

Fig. 18 Cost of local and multisite transactions in the update microbenchmark. Shared-everything can take advantage of consolidated logging that is especially significant for multisite transactions.

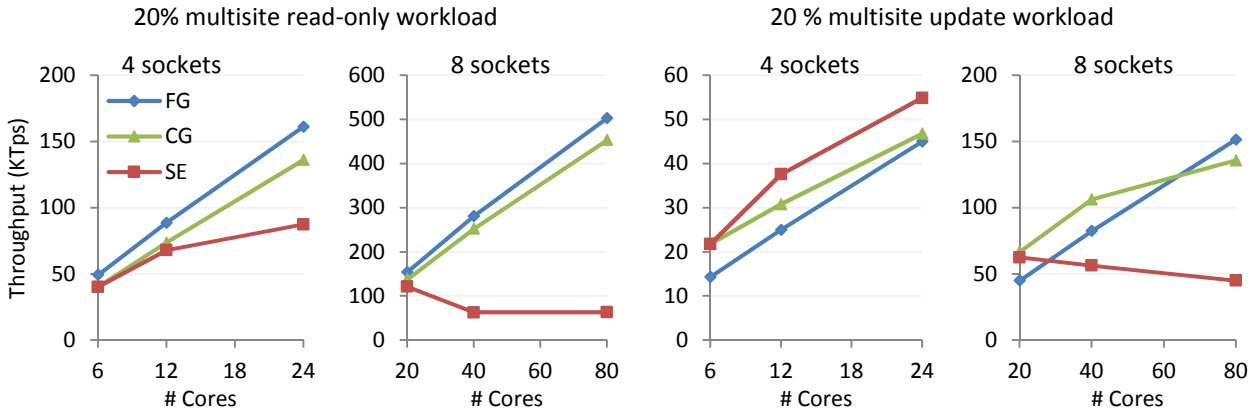

Fig. 19 Performance of alternative configurations as the hardware parallelism increases. Coarser-grained shared-nothing provides an adequate compromise between performance and predictability.

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 18, right) are significantly more expensive than their local counterparts. This is due to the overhead associated 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 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.

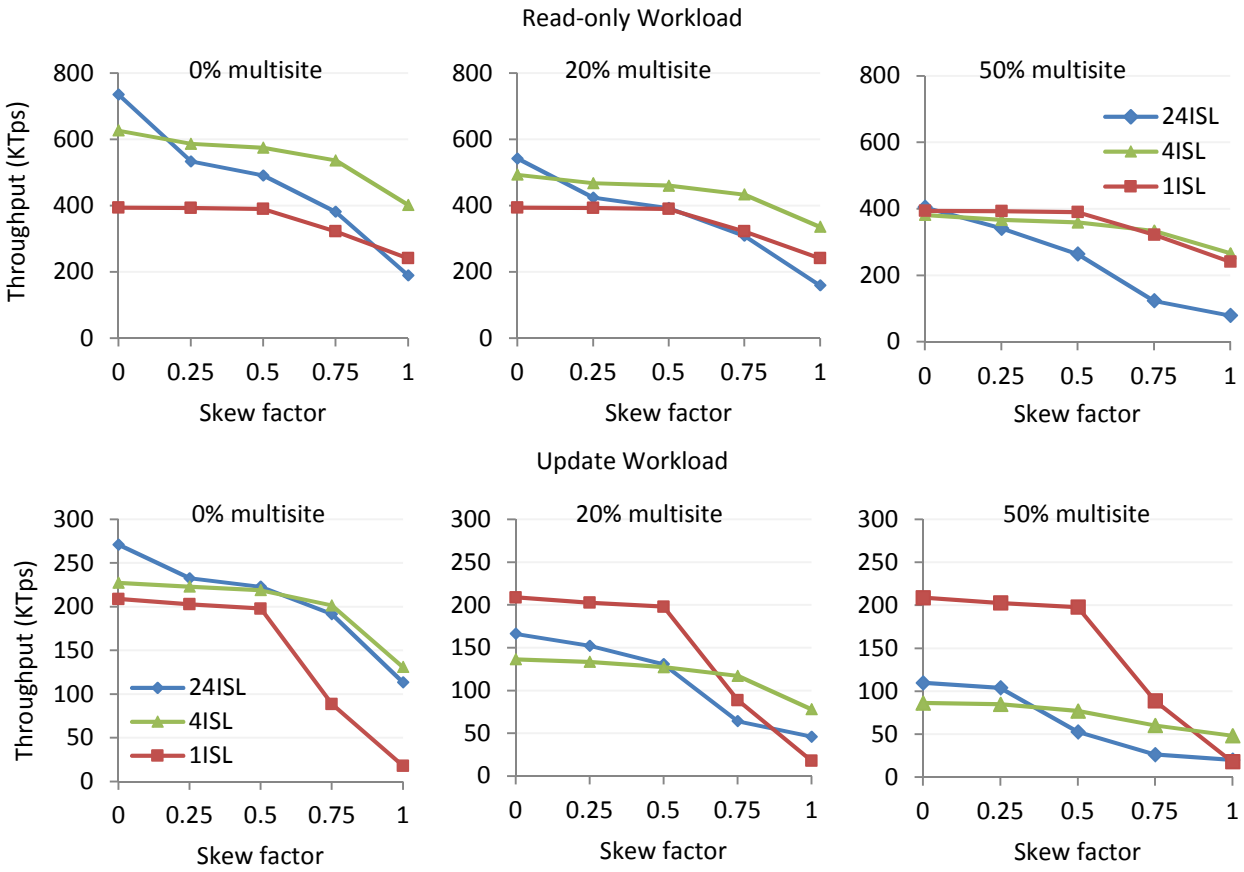

Fig. 20 Performance of read-only (top) and update (bottom) workloads with skewed accesses. As skew increases, shared-everything suffers from increased contention, while fine-grained shared-nothing suffers from a highly-loaded instance that slows others. Coarse-grained shared-nothing configuration cope better with a highly loaded instances, due to multiple internal threads.

7.2 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 19, 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 machines.

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. A higher QPI/IMC ratio means that 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 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 lock and latch managers [47,48]. ATraPos is a system that uses NUMA-friendly data structures and data-oriented execution to minimize inter-socket synchronization in the critical path of transaction execution [52]. It relies on dynamic workload and hardware topology-aware partitioning and placement

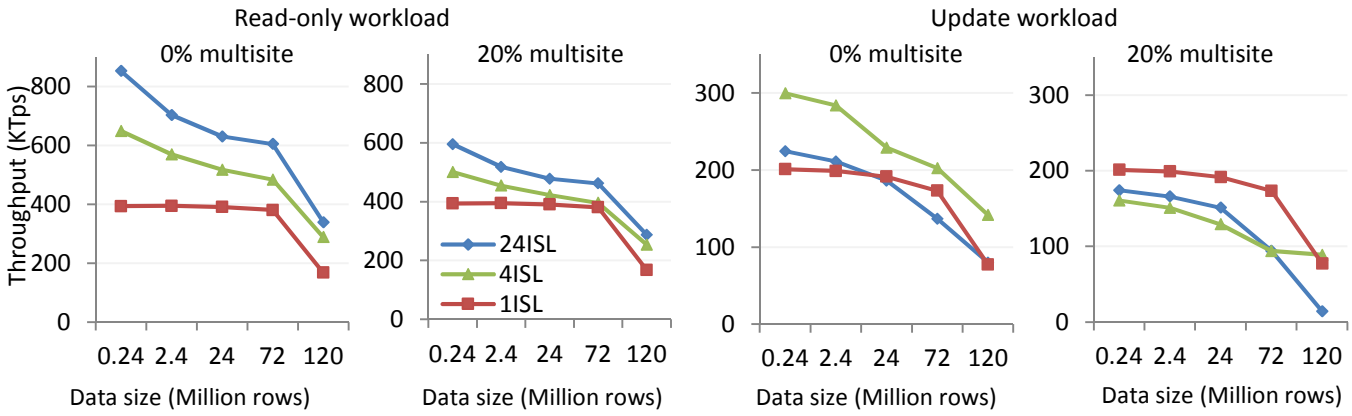

Fig. 21 Performance of the various configurations on workloads, as we gradually increase the database size from almost cache-resident to I/O-resident.

mechanisms to achieve balanced load and maximize locality of accesses.

7.3 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 20. The figures show the throughput for varying percentages of multisite transactions. We employ similar optimizations as described in 7.1.1 and 7.1.2.

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 performance of short read-write transactions [29]. In coarser-grained islands, 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.

Overall, coarse-grained shared-nothing configurations exhibit good performance in the presence of skewed requests, as they suffer less from increased contention and are more resistant to load imbalances.

7.4 Increasing Database Size

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 21, 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 decreases 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.

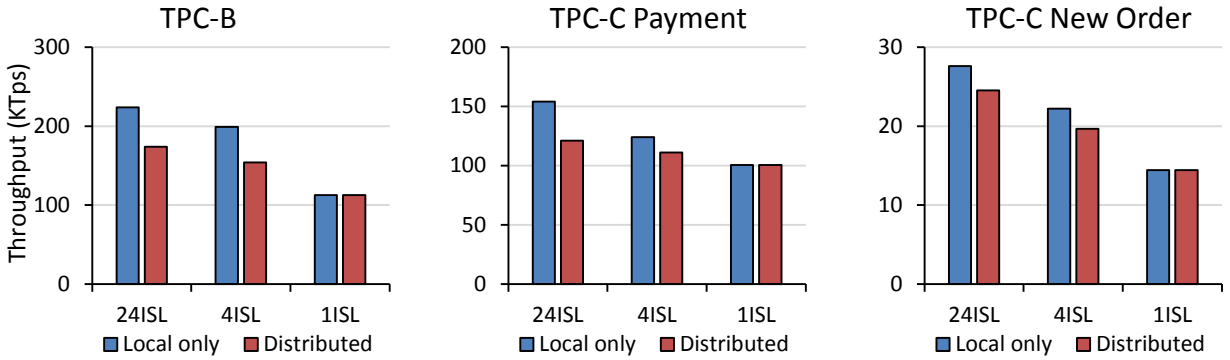

Fig. 22 Performance of different transactions in TPC-B and TPC-C benchmarks for their local only and standard-benchmark settings. Distributed transactions are more expensive than their local counterparts and they have higher impact on the finer-grained configurations.

8 Standard workloads

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. Also, when the microbenchmark setting involved more than two instances or rows, there are more than two such instances. 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 6 and Section 7, 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 executed 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.

8.1 TPC-B

Figure 22 (left) 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

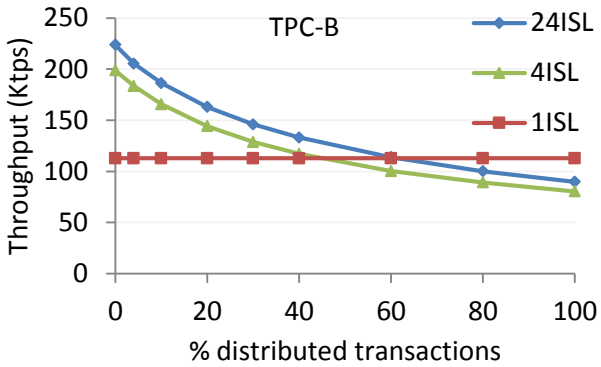

Fig. 23 Performance of different configurations as the percentage of distributed transactions increases for the TPC-B workload. Distributed transactions increase contention for hot data causing the drop in performance for shared-nothing configurations.

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 performance for the local only transactions due to less synchronization among the 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.

8.2 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 8.1, but we gradually increase the percentage of remote transactions from 0% to 100% and plot the throughput in Figure 23.

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.

8.3 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 are read-write, 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 transactions defined in the benchmark specification: 15% for the `Payment` transaction and 10% for the `NewOrder`.

Figure 22 (middle) plots the throughput for different configurations of the `Payment` workload, while Figure 22 (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

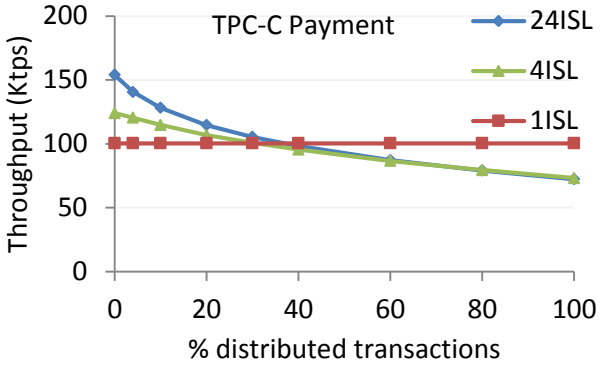

Fig. 24 Performance of different configurations as the percentage of distributed transactions increases for the TPC-C Payment transactions. Shared-everything configuration offers robust performance in the presence of remote transactions which cause throughput drops for partitioned systems.

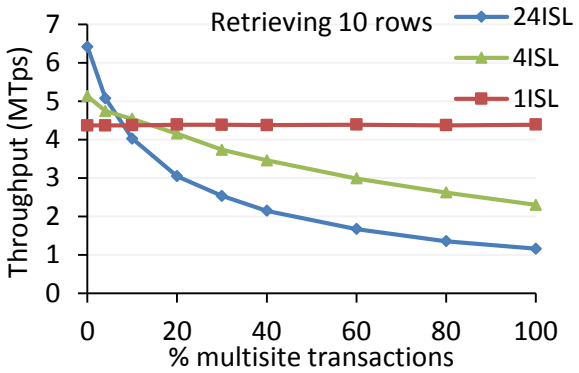

Fig. 25 Performance of different deployments of Silo as the number of multisite transactions increases. It shows the same trends as the deployments based on Shore-MT.

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).

8.4 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 24 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.

9 Main-memory optimized system

In this section we quantify the impact of hardware islands on the performance of different deployments of a modern

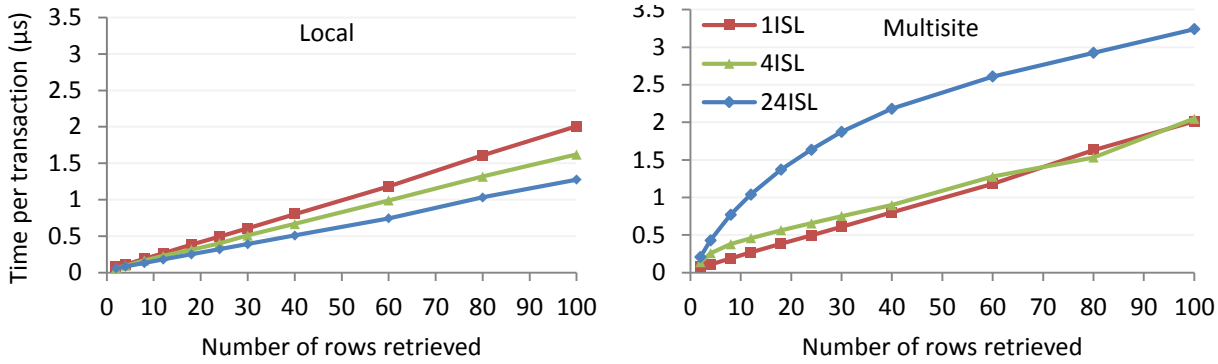

Fig. 27 Cost of local and multisite transactions in the read-only microbenchmark. The cost of multisite transactions rises until all instances participate in every transaction.

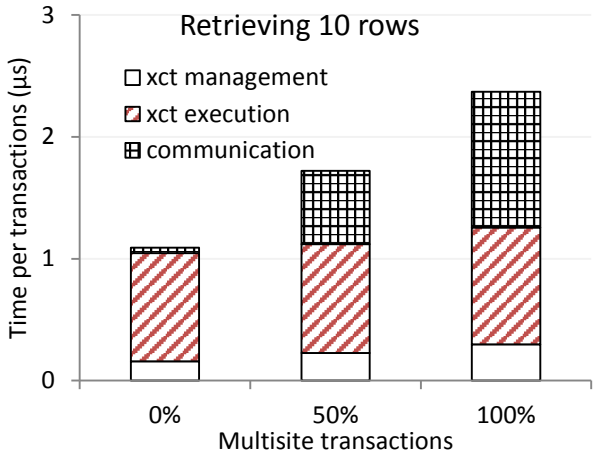

Fig. 26 Time breakdown for a transaction that retrieves 10 rows in 4ISL deployment. Communication costs determine overall cost of a transaction.

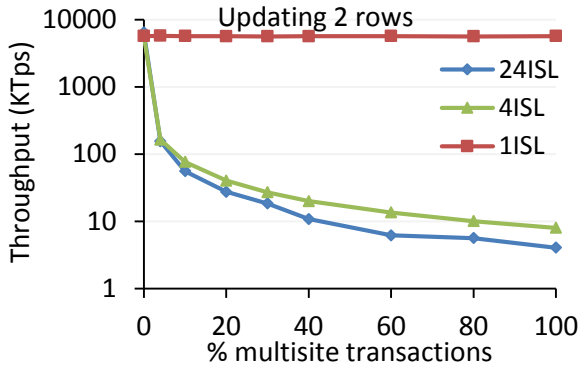

Fig. 28 Performance of different deployments of Silo as the percentage of multisite transactions increases. The trend is the same as with read-only transaction, however, update transactions are much more expensive.

main-memory optimized system. We use Silo [67] which is a multicore optimized system that utilizes cache-conscious multiversioned Mass-tree design [44] 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 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.

Figure 25 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.

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 26 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 com-

ponents remain the same. This trend shows that the communication costs are the dominant factor in the cost of the distributed transactions.

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 7.1.1. The left hand side of Figure 27 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 27, 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.

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 28. We use fewer rows because heavier transactions increase contention even further resulting in a very low throughput. Figure 28 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.

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 5, 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 distributed update transactions.

10 Summary and discussion

Modern multisocket 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 **topology-awareness improves OLTP performance under a variety of scenarios.** The OLTP Islands design, being hardware topology-aware, provides 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 hardware-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 8 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. Finally, many complex workloads, including TPC-E benchmark [65], are not easily partitionable as they contain multiple tree schemas and transactions that access data from many different tables [62]. In this case, even a very good partitioning scheme will generate many multisite transactions [16,49,66]. For such workloads that are not easily partitionable, shared-everything deployments are the best choice.

11 Conclusions and future work

High-end servers, used for OLTP applications, are nowadays designed as multisocket multicores. In contrast to previous generations of such servers that had uniform core-to-core latencies, multisockets have non-uniform topology. In this paper, we conduct a detailed analysis of the impact of hardware islands on the performance of different transaction processing system configurations. We show that the shared-nothing configuration is twice as fast as the shared-everything one for perfectly partitionable workloads, while situation is completely opposite for non-partitionable workloads and workloads that exhibit heavy skew. Island-sized shared-nothing configurations fall between the two extremes. Overall, there is no single optimal configuration: the best configuration depends on the hardware topology and the workload characteristics.

Future work will focus on determining the ideal size of each island automatically for the given hardware and workload. Moreover, in clustered databases, shared-cache shared-disk designs [38] allow database instances to share buffer pools, avoiding accesses to the shared-disk. Studying the performance of shared-disk deployments within a single multisocket multicore node is also part of our future plans. Scaling-out OLTP across multiple machines is an orthogonal problem, but the Islands concept would also be applicable in a distributed setting. Finally, investigating the effect of hardware Islands on more complex non-partitionable workloads (e.g., TPC-E) is still an open research problem.

Acknowledgements We would like to thank Eric Sedlar and Brian Gold for many insightful discussions and the members of the DIAS laboratory for their support throughout this work. This work is partially funded by Oracle Labs and by the Swiss National Science Foundation (Grant No. 200021-146407/1).

References

1. Accetta, M.J., Baron, R.V., Bolosky, W.J., Golub, D.B., Rashid, R.F., Tevanian, A., Young, M.: Mach: A new kernel foundation for UNIX development. In: USENIX Summer, pp. 93–112 (1986)
2. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs on a modern processor: Where does time go? In: VLDB, pp. 266–277 (1999)
3. Albutiu, M.C., Kemper, A., Neumann, T.: Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. PVLDB **5**(10), 1064–1075 (2012)
4. Amazon: EC2 instance types (2015). Available at https://aws.amazon.com/ec2/instance-types/
5. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Coordination avoidance in database systems. PVLDB **8**(3), 185 – 196 (2015)
6. Balkesen, C., Alonso, G., Teubner, J., Ozsu, M.T.: Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. PVLDB **7**(1), 85–96 (2014)
7. Barroso, L.A., Gharachorloo, K., Bugnion, E.: Memory system characterization of commercial workloads. In: ISCA, pp. 3–14 (1998)
8. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems. In: SOSP, pp. 29 – 44 (2009)
9. Beckmann, B.M., Wood, D.A.: Managing Wire Delay in Large Chip-Multiprocessor Caches. In: MICRO, pp. 319–330 (2004)
10. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM TODS **8**(4), 465–483 (1983)
11. Blagodurov, S., Zhuravlev, S., Fedorova, A., Kamali, A.: A case for NUMA-aware contention management on multicore systems. In: PACT, pp. 557–558 (2010)
12. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC, pp. 7–7 (2000)
13. Carey, M.J., DeWitt, D.J., Franklin, M.J., Hall, N.E., McAuliffe, M.L., Naughton, J.F., Schuh, D.T., Solomon, M.H., Tan, C.K., Tsatalos, O.G., White, S.J., Zwilling, M.J.: Shoring up persistent applications. In: SIGMOD, pp. 383–394 (1994)
14. Closson, K.: You buy a NUMA system, Oracle says disable NUMA! What gives? (2009). See http://kevinclosson.wordpress.com/2009/05/14/you-buy-a-numa-system-oracle-says-disable-numa-what-gives-part-ii/
15. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's Globally-Distributed Database. In: OSDI, pp. 261–264 (2012)
16. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. PVLDB **3**, 48–57 (2010)
17. Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., Roth, M.: Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In: ASPLOS, pp. 381–394 (2013)
18. David, T., Guerraoui, R., Trigonakis, V.: Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In: SOSP, pp. 33–48 (2013)
19. Engler, D.R., Kaashoek, M.F., O'Toole Jr., J.: Exokernel: an operating system architecture for application-level resource management. In: SOSP, pp. 251–266 (1995)
20. Giceva, J., Alonso, G., Roscoe, T., Harris, T.: Deployment of Query Plans on Multicores. PVLDB **8**(3), 233 – 244 (2014)
21. Graham, C., Sood, B., Horiuchi, H., Sommer, D.: Market share: Database management system software, worldwide (2009). See http://www.gartner.com/DisplayDocument?id=1044912
22. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Reactive NUCA: near-optimal block placement and replication in distributed caches. In: ISCA, pp. 184–195 (2009)

23. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
24. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR, pp. 132–141 (2007)
25. 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
26. Johnson, R., Pandis, I., Ailamaki, A.: Improving OLTP scalability using speculative lock inheritance. PVLDB **2**(1), 479–489 (2009)
27. Johnson, R., Pandis, I., Ailamaki, A.: Eliminating unscalable communication in transaction processing. Vldb Journal **23**(1), 1–23 (2014)
28. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: EDBT, pp. 24–35 (2009)
29. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. PVLDB **3**, 681–692 (2010)
30. Jones, E., Abadi, D.J., Madden, S.: Low overhead concurrency control for partitioned main memory databases. In: SIGMOD, pp. 603–614 (2010)
31. Jung, H., Han, H., Fekete, A.D., Heiser, G., Yeom, H.Y.: A Scalable Lock Manager for Multicores. In: SIGMOD, pp. 73–84 (2013)
32. Kemper, A., Neumann, T.: HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206 (2011)
33. Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In: SIGMOD, pp. 691–706 (2015)
34. Kimura, H., Graefe, G., Kuno, H.: Efficient Locking Techniques for Databases on Modern Hardware. In: ADMS (2012)
35. Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D., Lehner, W.: ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In: ADMS, pp. 74–85 (2014)
36. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM TODS **6**(2), 213–226 (1981)
37. Lahiri, T., Neimat, M.A., Folkman, S.: Oracle TimesTen: An In-Memory Database for Enterprise Applications. IEEE Data Eng. Bull. **36**(2), 6–13 (2013)
38. Lahiri, T., Srihari, V., Chan, W., MacNaughton, N., Chandrasekaran, S.: Cache fusion: Extending shared-disk clusters with shared caches. In: VLDB, pp. 683–686 (2001)
39. Larson, P.A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. PVLDB **5**(4), 298–309 (2011)
40. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-Tree: A B-tree for new hardware platforms. In: ICDE, pp. 302–313 (2013)
41. Levinthal, D.: Performance analysis guide for Intel Core i7 and Intel Xeon 5500 processors (2009). Available at http://software.intel.com/sites/products/collateral/hpc/vtune/ performance_analysis_guide.pdf
42. Li, Y., Pandis, I., Mueller, R., Raman, V., Lohman, G.: NUMA-aware Algorithms: The Case of Data Shuffling. In: CIDR (2013)
43. Lindström, J., Raatikka, V., Ruuth, J., Soini, P., Vakkila, K.: IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. IEEE Data Eng. Bull. **36**(2), 14–20 (2013)
44. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Eurosys, pp. 183–196 (2012)
45. Microsoft: Analytics Platform System (2015). Available at http://www.microsoft.com/en-us/server-cloud/products/analytics-platform-system
46. Oracle Corp.: Exadata Database Machine (2015). Available at https://www.oracle.com/engineered-systems/exadata/database-machine-x4-8/features.html
47. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-Oriented Transaction Execution. PVLDB **3**(1), 928–939 (2010)
48. Pandis, I., Tözün, P., Johnson, R., Ailamaki, A.: PLP: page latch-free shared-everything OLTP. PVLDB **4**(10), 610–621 (2011)
49. Pavlo, A., Curino, C., Zdonik, S.: Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In: SIGMOD, pp. 61–72 (2012)
50. Pavlo, A., Jones, E.P.C., Zdonik, S.: On predictive modeling for optimizing transaction execution in parallel OLTP systems. PVLDB **5**(2), 85–96 (2011)
51. Polychroniou, O., Ross, K.A.: A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In: SIGMOD, pp. 755–766 (2014)
52. Porobic, D., Liarou, E., Tözün, P., Ailamaki, A.: ATraPos: Adaptive Transaction Processing on Hardware Islands. In: ICDE (2014)
53. Porobic, D., Pandis, I., Branco, M., Tözün, P., Ailamaki, A.: OLTP on Hardware Islands. PVLDB **5**(11), 1447–1458 (2012)
54. Quamar, A., Kumar, K.A., Deshpande, A.: Sword: scalable workload-aware data placement for transactional workloads. In: EDBT, pp. 430–441 (2013)
55. Salomie, T.I., Subasu, I.E., Giceva, J., Alonso, G.: Database engines on multicores, why parallelize when you can distribute? In: EuroSys, pp. 17–30 (2011)
56. Somogyi, S., Wenisch, T.F., Hardavellas, N., Kim, J., Ailamaki, A., Falsafi, B.: Memory coherence activity prediction in commercial workloads. In: WMPI, pp. 37–45 (2004)
57. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**, 4–9 (1986)
58. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: VLDB, pp. 1150–1160 (2007)
59. Tang, L., Mars, J., Vachharajani, N., Hundt, R., Soffa, M.L.: The impact of memory subsystem resource sharing on datacenter applications. In: ISCA, pp. 283–294 (2011)
60. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast distributed transactions for partitioned database systems. In: SIGMOD, pp. 1–12 (2012)
61. Tözün, P., Pandis, I., Johnson, R., Ailamaki, A.: Scalable and dynamically balanced shared-everything OLTP with physiological partitioning. VLDB J. **22**(2), 151–175 (2013)
62. Tözün, P., Pandis, I., Kaynak, C., Jevdjic, D., Ailamaki, A.: From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored. In: EDBT, pp. 17–28 (2013)
63. TPC: TPC benchmark B standard specification, revision 2.0 (1994). Available at http://www.tpc.org/tpcb
64. TPC: TPC benchmark C standard specification, revision 5.11 (2010). Available at http://www.tpc.org/tpcc
65. TPC: TPC benchmark E standard specification, revision 1.12.0 (2010). Available at http://www.tpc.org/tpce
66. Tran, K.Q., Naughton, J.F., Sundarmurthy, B., Tsirogiannis, D.: JECB: A join-extension, code-based approach to OLTP data partitioning. In: SIGMOD, pp. 39–50 (2014)
67. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy Transactions in Multicore In-memory Databases. In: SOSP, pp. 18–32 (2013)
68. Vogels, W.: Eventually consistent. Commun. ACM **52**, 40–44 (2009)
69. Wagle, M., Booss, D., Schreter, I.: NUMA-Aware Memory Management with In-Memory Databases. In: TPCTC (2015)
70. Wilson, M.: Disabling NUMA parameter (2011). Http://www.michaelwilsondba.info/2011/05/disabling-numa-parameter.html
71. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. PVLDB **8**(3), 209–220 (2014)
72. Zhang, C., Ré, C.: DimmWitted: A Study of Main-Memory Statistical Analytics. PVLDB **7**(12), 1283–1294 (2014)