ATraPos: Adaptive Transaction Processing on Hardware Islands

Danica Porobic   Erietta Liarou   Pınar Tözün   Anastasia Ailamaki

École Polytechnique Fédérale de Lausanne
{firstname.lastname}@epfl.ch

***Abstract*—Nowadays, high-performance transaction processing applications increasingly run on multisocket multicore servers. Such architectures exhibit non-uniform memory access latency as well as non-uniform thread communication costs. Unfortunately, traditional shared-everything database management systems are designed for uniform inter-core communication speeds. This causes unpredictable access latencies in the critical path. While lack of data locality may be a minor nuisance on systems with fewer than 4 processors, it becomes a serious scalability limitation on larger systems due to accesses to centralized data structures.**

In this paper, we propose *ATraPos*, a storage manager design that is aware of the non-uniform access latencies of multisocket systems. ATraPos achieves good data locality by carefully partitioning the data as well as internal data structures (e.g., state information) to the available processors and by assigning threads to specific partitions. Furthermore, ATraPos dynamically adapts to the workload characteristics, i.e., when the workload changes, ATraPos detects the change and automatically revises the data partitioning and thread placement to fit the current access patterns and hardware topology.

We prototype ATraPos on top of an open-source storage manager Shore-MT and we present a detailed experimental analysis with both synthetic and standard (TPC-C and TATP) benchmarks. We show that ATraPos exhibits performance improvements of a factor ranging from 1.4 to 6.7x for a wide collection of transactional workloads. In addition, we show that the adaptive monitoring and partitioning scheme of ATraPos poses a negligible cost, while it allows the system to dynamically and gracefully adapt when the workload changes.

I. Introduction

Over the years, relational database vendors have optimized software for commodity servers: uniprocessors, shared memory multiprocessors, and multicores. The common denominator of these servers is the uniform communication speed between any pair of CPU cores. However, the latest processor design trends provide very different types of high performance servers. There, we have *groups* of cores that communicate very fast with cores that belong to the same group and several times slower with cores from other groups. We call such a group of fast communicating cores an *Island* [1]. Currently, an Island is represented by a processor socket but in the near future, with dozens of cores on the same socket, we expect that Islands will form within a chip [2].

The problem. Online Transaction Processing (OLTP) is one of the most important and demanding database applications used in large enterprise systems. Thus, providing the maximum possible performance is of significant importance. However, OLTP systems suffer from non-uniform communication between cores in modern hardware.

Relational database systems are typically designed either as shared-everything or shared-nothing architectures. In a shared-everything design, we have one database instance that uses all resources and manages all data. A shared-nothing design consists of a set of database instances, each holding a data partition, that collectively serve transactions. However, neither of these designs exploits the full potential of multisockets due to data sharing across sockets. Shared-everything design suffers from excessive communication and contention among threads [3][4][5], while shared-nothing systems suffer from the overhead of distributed transactions. With the new hardware at hand it is necessary to redesign existing software in order to extract the maximum performance.

Our approach. In this paper, we present *ATraPos*, a scalable shared-everything system that minimizes the impact of inter-socket communication in the critical path of transaction execution (i.e., the sequence of actions that determine the duration of the transaction). ATraPos relies on *precise data partitioning and placement* to maximize locality of data accesses and on *adaptive repartitioning* to maintain data locality even when the workload changes.

ATraPos first partitions the data *logically*, by allowing only specific threads to access each data item, and then *physically*, by partitioning tables and indices to map to the logical parts. It takes the data locality principle further by keeping the system state in hardware-aware data structures. These data structures are designed such as they require only socket-local data accesses in the critical path. Each thread is bound to a precisely chosen processor core that allows it to have a consistent view of the system state by accessing only a socket-local partition of the shared data structures that track the state.

ATraPos ensures good performance by choosing the appropriate partitioning scheme that maximizes resource utilization and balances the load. The choice is based on a cost model that takes into account a) static data dependencies, b) dynamic workload information, and c) the underlying hardware topology. ATraPos uses a lightweight monitoring mechanism that continuously captures the transaction behavior. When the workload changes, it adjusts the data partitioning and partition placement to guarantee high and predictable performance.

Contributions. The contributions of this work can be summarized as follows:

1) We show that scalable transaction processing systems for multisocket servers must avoid accessing any centralized data structure in the critical path. With more cores that commu-

nicate less uniformly, any such access eventually becomes a bottleneck.

2) We demonstrate that a shared-everything design can scale as well as a fine-grained shared-nothing design for perfectly partitionable workloads on multisocket multicores. To achieve that, we need to replace all centralized data structures with hardware-aware data structures that avoid inter-socket accesses in the critical path.

3) We present ATraPos, a storage manager design that is aware of the non-uniform access latencies of multisocket systems. On top of its hardware-aware internal structures, ATraPos adopts a lightweight monitoring and repartitioning mechanism that adapts the partitioning strategy upon workload changes. Through such techniques, ATraPos achieves good data locality by taking into account static and dynamic workload information as well as the hardware topology.

4) We show that ATraPos exhibits significant performance improvements, ranging from 1.4 to 6.7x, for a wide collection of transactional workloads. At the same time, the adaptive monitoring and partitioning scheme of ATraPos poses a negligible cost, while allowing the system to dynamically and gracefully adapt to workload and hardware changes.

Outline. The rest of this paper is structured as follows. Section II surveys the current hardware trends and related work. Section III motivates this work by identifying pitfalls of the existing designs on multisocket multicores. Section IV discusses how we can efficiently partition the shared data structures in order to minimize their negative impact on the scalability of the system. Section V presents our hardware and workload-aware adaptive data partitioning and partition placement method. Section VI presents our prototype implementation and experimental results on standard benchmarks TPC-C and TATP as well as on various microbenchmarks. Finally, Sections VII and VIII discuss future work and conclude the paper.

II. Related Work

Here, we discuss the necessary background and related work as well as how ATraPos enhances the state-of-the-art.

A. Multisocket Multicores

Uniprocessors have followed Moore's law for decades using higher frequencies and complex out-of-order execution techniques. Recently, due to thermal and power limitations, vendors turned to placing many simpler cores on the same chip to gain higher performance. Today's high-end servers have multiple multicore processors on the same board, which creates hardware Islands [1]. In a typical multisocket server, each socket represents an Island since all its cores communicate through shared last level cache. However, we can expect Islands to form even within one chip. For example, the Tilera family of multicore chips [6] has cores that are organized in the form of a mesh. In this case, communication latency depends on the number of hops between two cores, e.g., for the 36 core chip it ranges from 45 to 65 cycles.

A lot of past work has focused on adapting databases for legacy multisocket systems. For instance, many commercial database systems provide configuration options to enable *NUMA support*, i.e., a set of optimizations for NUMA hardware. However, this setting is optimized for hardware where each individual chip contains a single core. With newer multisocket servers, enabling legacy *NUMA support* might lead to high CPU usage and degraded performance [7][8].

Adapting software systems to today's non-uniform hardware is an area of active research. Scalable synchronization structures typically rely on efficient inter-core communication using atomic operations. Since an atomic operation becomes much slower over inter-socket links, proposals for scalable NUMA-aware locks rely on hierarchically partitioned structures to maximize access locality [9][10]. On the system level, a recent study on the performance of garbage collectors on multisocket multicores analyzes synchronization patterns and systematically removes bottlenecks without completely redesigning the system [11]. We take inspiration from these works as we redesign our storage manager for multisockets. We identify the scalability bottlenecks in several steps since removing one bottleneck causes another unscalable data structure to surface as the next bottleneck. At each step we replace centralized data structures or locks with their hierarchically partitioned versions based on the socket boundaries.

Recent research efforts in NUMA-aware data management systems mostly focus on complex analytical processing and efficient evaluations of joins or aggregations [12][13][14] as these operations are expensive due to large data movement. Transactional workloads, however, pose a different set of challenges due to much less data movement and much more synchronization in the critical path compared to analytical workloads.

Two recent studies analyze the behavior of transaction processing systems on modern multisocket hardware [1][15]. The highlight of these studies is that traditional shared-everything systems suffer on multisockets due to various centralized communication points causing unpredictable access latencies in the critical path. On the other hand, distributed transactions hinder the performance of shared-nothing systems if the data partitioning is not done carefully and is not dynamic. In this paper, we corroborate the results of these studies but also present and evaluate a design that would take the best of both worlds. ATraPos achieves stable performance for non-partitionable workloads and workload changes as in a shared-everything design, while providing good data locality and low access latencies in the critical path as in a shared-nothing one.

B. Data partitioning strategies

Previous works on static or dynamic data partitioning mainly focus on shared-nothing environments and aim to minimize the number of distributed transactions. Schism proposes a graph-based partitioning and replication method for OLTP workloads. The graph is constructed from transaction access traces such that vertices represent tuples and edges connect the tuples used in the same transaction. The partitions are selected using the min-cut algorithm. A recent extension of Schism, Sword [16], proposes a different graph compression approach that allows incremental data movement between two partitioning solutions for different workloads. Another approach for automatic partitioning in shared-nothing OLTP systems is Horticulture [17], which utilizes large neighborhood search (LHS). It uses the database schema, the code of the stored procedures, the workload trace consisting of data items that were accessed, and timestamps. The output of the partitioning strategy is a set of decisions whether to range or hash partition a table or replicate it to all nodes. All these techniques generate good initial partitioning. However, they are not designed to monitor the workload changes at runtime and adapt to them dynamically.

On the other hand, one of the recent proposals for adaptive repartitioning algorithms targets physiologically partitioned shared-everything systems [18]. The load on each partition is monitored using histograms and work queues. Whenever a load imbalance exceeds the threshold, data is repartitioned. While this approach works great for adapting to skew on a single table, it does not take into account changes in the frequencies of different transactions, which causes the optimal number of partitions of certain tables to change.

Finally, none of the partitioning methods mentioned above takes into account the underlying non-uniform hardware topology. In addition to that, ATraPos is the first system that continuously adapts to the workload and hardware changes.

III. The Problem: Scaling OLTP on multisockets

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

We run our experiments on an 8-socket 10-core Intel Westmere server with 192 GB of RAM. We use the state-of-the-art multithreaded storage manager Shore-MT [19] and Intel's VTune Analyzer XE 2013 [20] for profiling.

A. *Design options*

Centralized shared-everything. We evaluate the traditional shared-everything configuration by running Shore-MT as one process using all available processor resources. In this case, all data structures accessed by transaction execution threads are centralized, e.g., the lock manager, the log, and the buffer pool. We enable the optimizations that are beneficial to the workloads we run, including speculative lock inheritance [21] and optimized logging using Aether [22].

Shared-nothing. We benchmark two shared-nothing configurations by running multiple instances of Shore-MT. All instances communicate using a thin distributed transaction execution layer implemented on top of the storage manager [1]. Specifically, we simulate the *extreme shared-nothing* architectures, such as H-Store [23], by running one instance of Shore-MT per processor core; each record and page are touched

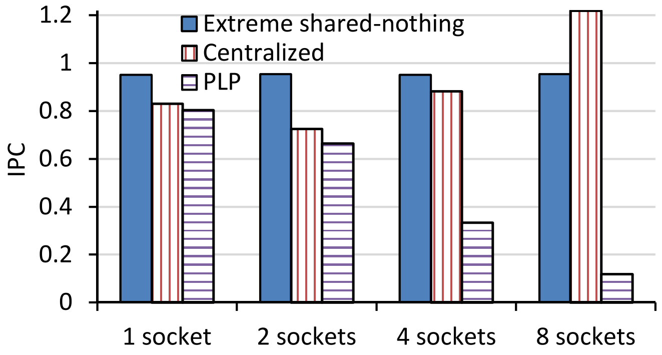

Fig. 1. Instructions retired per cycle.

by a single thread, while locking and latching are disabled for read-only workloads. For workloads that contain updates, we still need to use locking. We also test a *coarse shared-nothing* configuration, having one instance per processor socket, where locking and latching are enabled.

PLP. One of the main problems of centralized shared-everything systems on multicores is the contention in the lock manager. This problem can be eliminated by using physiological partitioning (PLP) [5][24]. PLP first logically partitions the data and assigns each partition to a separate thread. Transactions are decomposed into small actions, which are routed to the relevant threads. Each thread contains a local lock table that eliminates the need to access the centralized lock manager for the majority of locks that each transaction needs to acquire. Eliminating the lock manager bottleneck exposes the bottleneck of latching on database pages. PLP removes this bottleneck by using multi-rooted B-trees and seamlessly changing the record insert operation. Multi-rooted B-trees partition the original B-tree by having one root per each logical partition. All data pages are pointed by a single leaf page. Since subtree accesses are thread-local, both B-tree and data page accesses can be latch-free.

B. *Perfectly partitionable workloads*

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

In Figure 1, we evaluate how well the above configurations use the available processor resources by measuring the number of retired instructions per cycle (IPC). Although we use a processor that can achieve up to 4 IPC, OLTP workloads can barely exceed 1. Low IPC is a general characteristic of OLTP [25] due to the large instruction footprints and unpredictable data accesses.

The shared-nothing architecture has constant IPC for all configurations. As we see in Figure 2, which shows the throughput of the three configurations as we increase the number of

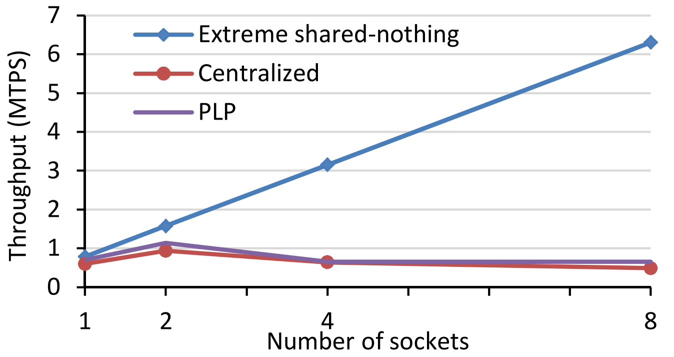

Fig. 2. Throughput of the shared-nothing, centralized, and PLP architectures.

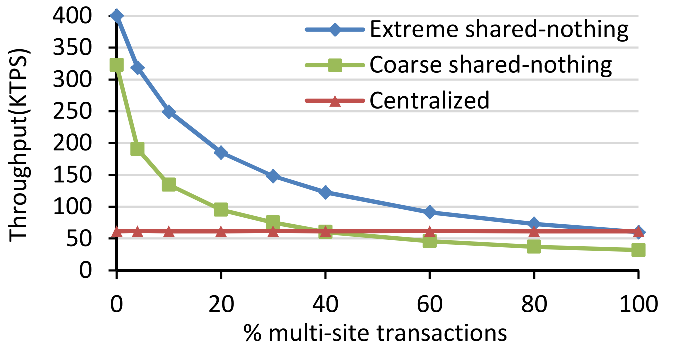

Fig. 3. Throughput of different configurations as percentage of multi-site transactions increases.

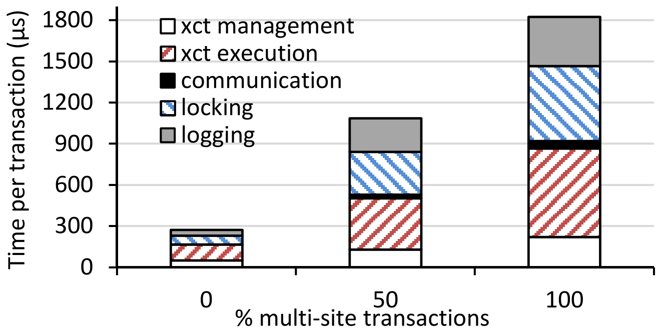

Fig. 4. Time breakdown for coarse shared-nothing configuration.

sockets, it scales linearly and has the best performance on this benchmark because the instances are completely independent from each other and do not communicate.

When we examine the traditional centralized architecture, we observe a slight decrease in IPC when we go from 1 to 2 sockets followed by an increase when we go to 4 and 8 sockets, where IPC exceeds 1.2. However, in these cases, high IPC is due to high cache hit rates while waiting to acquire contended locks. The time wasted on waiting is the reason why the throughput decreases with more sockets in Figure 2.

When we run the perfectly partitionable microbenchmark using PLP on more than one socket, we observe a performance degradation similar to the centralized configuration. However, the trends on the IPC graph are completely different. On the striped bars in Figure 1, we see large drops in IPC due to accesses to centralized data structures that are implemented using atomic compare-and-swap (CAS) instructions. While CAS instructions are executed efficiently on the same socket, they become very expensive across sockets, as they require accessing cache lines on remote processors.

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

C. Workloads That Are Less Amenable to Partitioning

While the shared-nothing architecture exhibits great performance on perfectly partitionable workloads, it suffers when the workload is not as partitionable. We illustrate this problem with a microbenchmark that has two types of transactions: 1) local transactions that update 10 rows chosen from the local site and 2) multi-site transactions that update 1 row chosen from the local site and the remaining 9 rows chosen uniformly from the whole dataset. Multi-site transactions whose rows belong to different instances run as distributed transactions. We run these transactions on the extreme shared-nothing configuration, the coarse shared-nothing configuration, and the traditional shared-everything configuration. In all cases, we use a dataset of 800K rows, equally divided between the participating instances.

In Figure 3, we plot the throughput when we vary the percentage of multi-site transactions from 0 to 100. We use shared memory communication channels, which are significantly faster than other communication mechanisms that involve the operating system, such as UNIX domain sockets and named pipes. However, we still observe a significant drop in the performance of partitioned systems as the percentage of multi-site transactions increases. The reason is that they must execute multi-site transactions as distributed transactions. In our implementation of distributed transactions we use the standard two phase commit protocol. There, transactions have to 1) hold locks until all the participating instances reach a decision (commit or abort), 2) log additional information for distributed transactions, and 3) store state information.

In Figure 4, we analyze the overheads of distributed update transactions by breaking down the execution time to different system components as we vary the percentage of multi-site transactions for the coarse shared-nothing configuration. The breakdowns are similar for the extreme shared-nothing configuration. As we increase the percentage of multi-site transactions, we see a significant increase in time spent in all components, especially in logging and locking.

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

D. Accessing remote memory

One significant advantage of shared-nothing configurations, where instances run within a single processor socket, is the ability to achieve perfect NUMA locality by allocating all memory in the local NUMA node. In this section, we quantify the impact of memory allocation on the performance. We run one Shore-MT instance per socket and change memory allocation policy using the Linux utility numactl [26]. We test the system in 3 modes: 1) each instance allocates memory in the local NUMA node, 2) all instances allocate memory in one NUMA node, and 3) every instance allocates memory in a different remote NUMA node.

We use a microbenchmark that reads 100 rows chosen

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

TABLE I
THROUGHPUT (IN TRANSACTIONS PER SECOND) FOR VARIOUS MEMORY ALLOCATION POLICIES.

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

To explore the causes of these performance drops for different configurations, we use the Intel's Performance Counter Monitoring tool [27] to examine the interconnect utilization. We measure that the ratio of interconnect (QPI) to memory controller (IMC) data traffic is 0.01 for the local case, in contrast to 1.36 for the central case, and 1.49 for the remote case. Total utilization of all QPI links for accessing memory and maintaining cache coherence increases from 13Gb/s for local node allocation to 21 Gb/s and 22 Gb/s, respectively. Even in the case where all instances allocate memory on a single node, QPI links are lightly utilized with the most used link being utilized at 14%.

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

IV. SCALING UP THE STORAGE MANAGER

As we show in the previous section, state-of-the-art techniques that are scalable for multicores are not sufficient for multisockets. This is caused by the bottleneck of accessing the centralized data structures in the critical path, i.e., the list of active transactions and various mutexes. Sharing data among threads that run on different sockets is expensive due to the cost of cache coherence and high latency of accesses to cache lines on remote sockets. ATraPos solves this problem by partitioning these structures among sockets to increase the locality of accesses. This section details our general approach to hardware-aware data structures.

Most centralized data structures in a typical storage manager are used for maintaining the global system state and are protected by read/write locks. Typically a transaction acquires a lock in read mode for a short period of time in order to change state, e.g, a transaction acquires volume read lock during the initialization phase. This is a fairly inexpensive operation on a single chip, but becomes increasingly expensive when we need to update data that is located on a remote chip or in memory. These locks are never acquired in write mode in the critical path of transaction execution. They are only used in write mode by threads performing background tasks, e.g., checkpointing, to ensure that no transaction changes state during this operation.

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

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

Thread binding. In ATraPos we exploit information about the underlying hardware to further improve scalability and performance. On top of data partitioning to ensure locality, we bind threads to specific processor cores and cache information about their socket. This ensures that each thread always accesses the same partition of any NUMA-aware data structure to guarantee correctness. For example, each transaction is removed from the list of active transactions by the same thread that added it, which ensures that both operations are performed on the same partition. Each partition is always local to the socket where the thread is running on.

Proof of concept. In Figure 5, we repeat the experiment of Figure 2 and include ATraPos as well as the coarse shared-

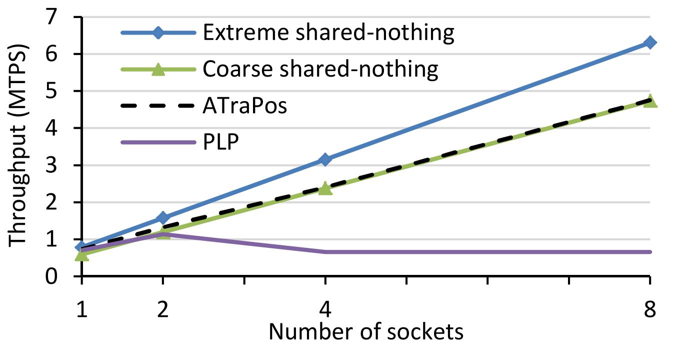

Fig. 5. Throughput of a perfectly partitionable workload.

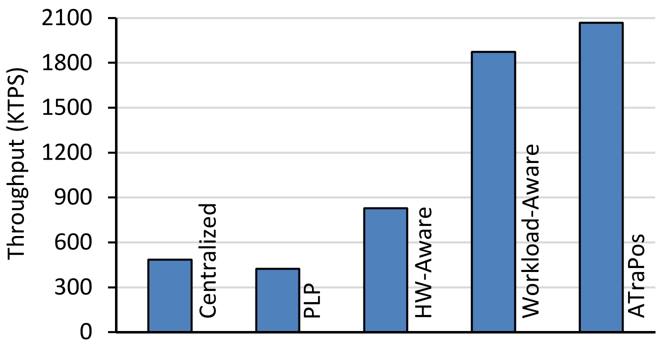

Fig. 6. Throughput of a simple transaction with varying partitioning and placement strategies.

nothing configuration discussed in Section III. Since we remove expensive accesses to the centralized data structures from the critical path, ATraPos can scale over multiple sockets and make full use of the fact that the workload is perfectly partitionable. In this case, we use the *naïve partitioning scheme* where a table is range partitioned with one partition per core. ATraPos matches the performance of the coarse grained shared-nothing configuration that has perfect locality because it runs one instance per socket. Both of these architectures scale similarly to the extreme shared-nothing architecture.

V. Workload and hardware-aware partitioning

Making the storage manager scalable for simple perfectly partitionable workloads is only one part of the problem. In this case, ATraPos scales linearly since each worker thread operates independently on its own data partition. For more complex workloads, however, we need to partition and place the data on cores in a way that reduces the inter-socket data exchange as much as possible.

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

A. Factors influencing transaction processing

There are a number of factors that we have to consider when choosing a partitioning scheme for an OLTP workload. Typically, the database schema is fixed and known a priori. In addition, most or all transactions fall into one of the predefined transaction classes expressed as parameterized stored procedures [23]. Furthermore, the input parameters of a transaction point to all data items this transaction is going to access (with the exception of the items accessed through the secondary indices). ATraPos exploits all this knowledge about the workload (static and dynamic information about recently executed transactions) and the underlying hardware topology to efficiently choose a good partitioning scheme.

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

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

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

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

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

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

Figure 6 shows the throughput on various configurations. We use the centralized shared-everything and the PLP designs as baselines. We compare them against the naïve partitioning scheme from Section IV and the ATraPos model using the criteria discussed above.

The naïve partitioning scheme (*HW-aware*) creates one partition of each table per processor core. As both tables have the same number of rows and we use range partitioning,

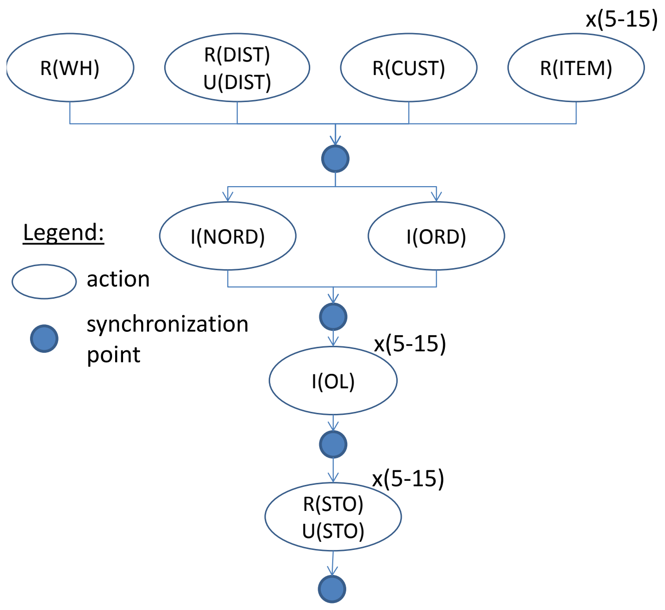

Fig. 7. Transaction flow graph for the TPC-C NewOrder transaction.

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

Complex Transaction Example. In this example, we briefly illustrate a more complex scenario, i.e., the `NewOrder` transaction in the TPC-C benchmark and explain the challenges in choosing good partitioning and placement scheme. This transaction models the ordering for 5 to 15 items from one warehouse and Figure 7 depicts its execution plan.

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

Conclusions from examples. From the previous two examples, we can conclude that using the naïve partitioning scheme is not enough; both workload and hardware-awareness of the partitioning mechanism are important for achieving high performance. Our system uses the data-oriented transaction execution model [5] where each worker thread operates on a single partition of a specific table. Using well-known partitioning schemes for TATP and TPC-C workloads (which are practically identical to the naïve partitioning in our system) causes severe overloading.

The next section presents a model that guides ATraPos in choosing good partitioning and placement scheme for complex workloads.

B. Cost model

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

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

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

where $RU(c)$ is the utilization of a particular core c and $RU_{avg} = \frac{\sum_c RU(c)}{N}$ is the average utilization for all N cores. We compute the utilization of one core c as:

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

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

We compute the transaction synchronization overhead $TS(S,W)$ for the workload trace W and the partitioning and placement scheme S as $TS(S,W) = \sum_{T \in W} Sync(T)$, where $Sync(T)$ is the synchronization cost of a single transaction T. We express this cost with the following formula:

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

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

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

Algorithm 1 Choose Partitioning

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

Algorithm 2 Choose Placement

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

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

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

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

C. Search strategy

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

In the first step, we use information about the current load for sub-partitions of existing partitions to choose a new partitioning scheme. As shown in Algorithm 1, we group sub-partitions into new partitions that balance the resource utilization according to our cost model. We initially assign one new partition per core in a greedy fashion: we first estimate the target average utilization and keep adding sub-partitions until we exceed that load. Then, move to the next core. Next, we iteratively try to improve the assignment by choosing a new partition placed on a core with the highest under-utilization, moving a sub-partition of the same table to that partition, and recomputing the utilization metric. If an under-utilized core contains the only partition of a table, we place a sub-partition of another table on that core to improve overall utilization. If the global utilization balance improves, we use this solution as the current best case and restart the search. We conclude the search when we cannot improve the overall utilization of the scheme by moving sub-partitions to under-utilized cores.

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

D. Monitoring and adaptive behavior

While the hardware topology and the static workload characteristics are inferred beforehand, the dynamic properties are captured at runtime. Our goal is to capture all the required information we use in our cost model in a lightweight manner.

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

Monitoring data structures. Since both the number of tuples in a table and the number of transactions that arrive in a time period vary greatly across different transactional workloads, the space overhead of the tracing structure should not depend on the dynamic characteristics of the workload. We use two thread-local arrays per partition: a) one that stores the cost of all actions executed by a specific sub-partition, and b) one that keeps the number of synchronization points executed for each local sub-partition. We initialize arrays based on the number of sub-partitions upon a new partition creation. In our experiments we use 10 sub-partitions per partition as it offers a good trade-off between the size of the arrays and the number of repartitioning operations needed to adapt to even the most drastic changes in the workload.

Detecting changes. ATraPos uses the lightweight monitoring mechanism described above to be able to adapt to any change in the workload. When the system starts up for the first time, it has no information about the dynamic aspects of the workload so it sets up the partitions using the naïve partitioning scheme described in Section IV. ATraPos continuously monitors the workload using the array-based approach described above. It periodically aggregates the trace information using the monitoring thread and decides the optimal partitioning and placement scheme according to the cost model. Since changes in the workload may happen during different time intervals, ATraPos uses an adaptive approach where it tunes the time interval length based on the frequency of the workload fluctuations. When the workload is stable for a long time it increases the intervals, while upon having frequent workload changes it shortens them. ATraPos starts from a 1 second interval and monitors the throughput. If the throughput is within 10% of the average of the previous 5 measurements it doubles the monitoring interval. After each monitoring interval, it checks if the throughput difference has exceeded the threshold; if it has, it evaluates the model, otherwise it increases the monitoring interval. If the result of the evaluation is the decision to repartition, ATraPos resets the monitoring interval to 1 second.

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

VI. Experimental evaluation

In this section, we discuss a detailed experimental evaluation using both microbenchmarks and standard benchmarks such as TPC-C and TATP. We designed and implemented ATraPos on top of Shore-MT [19]. We show that ATraPos manages to better exploit hardware resources compared to the state-of-the-art, providing a significant performance boost even when the workload changes.

A. Experimental setup

Our experimental platform is a server with 8 Intel Xeon E7-L8867 processors connected in a twisted cube topology. Each processor has 10 cores with private L1 (32KB each for data and instructions) and L2 (256KB) caches, as well as 30MB of shared L3 cache. Our system has 192GB of RAM and we use memory mapped disks for both data and log files. All experiments run on Red Hat Enterprise Linux 6.4 (kernel 2.6.32) and we compile using GCC 4.4.7 with maximum optimizations.

We use microbenchmarks and the standard OLTP benchmarks TATP [28] and TPC-C [29]. The TATP benchmark models a mobile phone provider. Its schema contains 4 tables that are perfectly partitionable on the `SubscriberID` attribute. TATP uses a set of 7 transactions of 3 different classes. It contains read-only transactions that access only a single table (e.g., `GetSubData`), read-only transactions that access multiple tables (e.g., `GetNewDest`), and update transactions (e.g., `UpdLocation`). In all experiments with TATP, we use

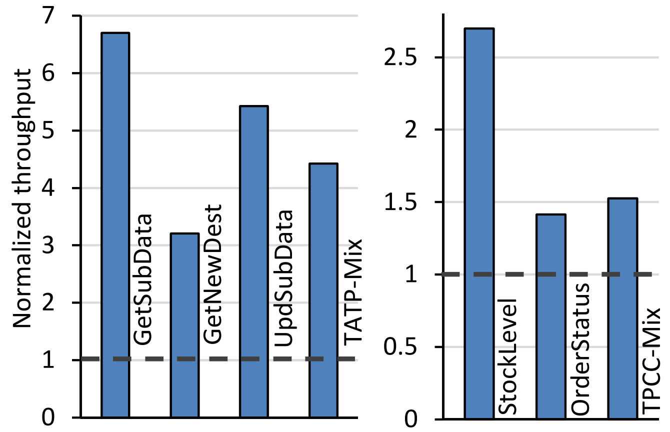

Fig. 8. Improving throughput on standard benchmarks (TATP and TPC-C) with ATraPos: Normalized performance over the state-of-the-art ($y = ATraPos/PLP$).

a dataset with 800K subscribers (1.8GB). The more complex TPC-C benchmark models a wholesale supplier. There, we have 9 tables and 5 different transactions. In contrast to TATP, all TPC-C transactions require data from 3 or more tables. We use the TPC-C dataset with scaling factor 80 (13GB) in all experiments.

B. Improving throughput on standard benchmarks with ATraPos

In our first experiment we demonstrate the significant performance boost that ATraPos brings on the standard benchmarks TATP and TPC-C. The performance metric used is throughput, i.e., how many transactions the system executes per second. We compare ATraPos against the state-of-the-art, PLP, which assigns one partition of each table per processor core.

The graph on the left-hand side of Figure 8 shows the behavior of ATraPos on the TATP benchmark. The y-axis depicts the throughput of ATraPos normalized over the throughput of PLP, i.e., $y = Throughput(ATraPos)/Throughput(PLP)$. In this way, the y-axis represents the throughput improvement achieved by ATraPos; 1 for no-improvement. We show results both for individual transaction types as well as the standard TATP transaction mix (denoted as `TATP-Mix`). Although the `GetSubData` transaction is perfectly partitionable and both PLP and ATraPos place one partition of the `Subscriber` table per core, ATraPos achieves 6.7x improvement due to NUMA-aware data structures. For other transactions, ATraPos achieves significant throughput improvements due to good partitioning and placement scheme. For example, for the `GetNewDest` transaction, where we need to access data from two tables, ATraPos brings an improvement of 3.2x. The improvement rises to 5.4x and 4.4x for `UpdSubData` and `TATP-Mix`, respectively. The higher improvement in performance for update transactions mainly comes from the decreased contention on the log since the better partitioning scheme of ATraPos creates fewer partitions, hence less threads are competing for the log resources.

The graph on the right-hand side of Figure 8 depicts the throughput improvement on the TPC-C benchmark. We plot the normalized performance of ATraPos (over PLP) for the two read-only transactions of TPC-C as well as for

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

TABLE II
ATRAPOS MONITORING BRINGS NEGLIGIBLE OVERHEAD.

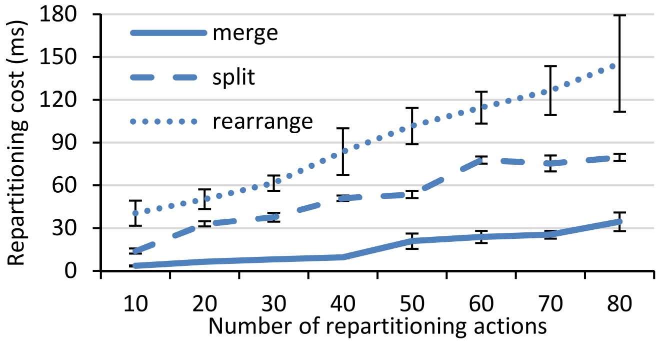

Fig. 9. Scalability of ATraPos repartitioning mechanism.

TPCC-Mix. We observe a large performance improvement of 2.7x for the heavyweight StockLevel transactions. For the lightweight OrderStatus transactions, improvement is 1.4x. This variation on performance comes from the fact that the StockLevel transaction benefits significantly from the NUMA-aware data structures while it performs a join requiring many data accesses. On the other hand, OrderStatus mainly benefits from the better data partitioning that creates balanced load across all cores in the system. Finally, the throughput of TPCC-Mix improves by 50%.

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

C. Monitoring and repartitioning cost

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

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

To quantify the repartitioning overhead, we use the following experiment. On a table of 800K rows and 10 integer attributes, we vary the number of repartitioning actions we trigger and measure the time it takes to complete each individual action. Figure 9 shows the results. For each case we show the average time of 10 repeated measurements with standard deviation. The merge operation combines two trees into one, the split divides one tree into two, and the rearrangement performs one

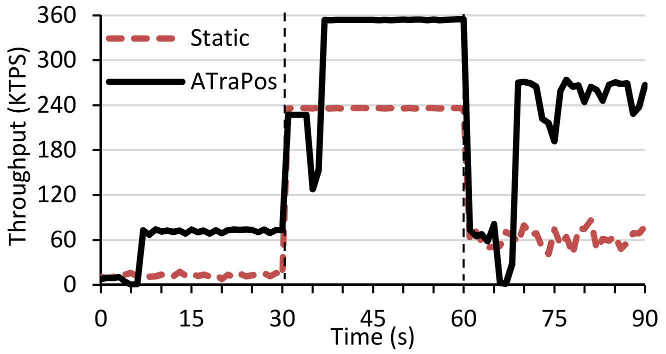

Fig. 10. Adapting to workload changes.

split and one merge. As we see in Figure 9, the cost of all repartitioning sequences increases linearly with the number of repartitioning actions needed. The merge operation is always cheaper compared to the split operation. This is because the latter performs more updates to the metadata. A rearrangement consists of one split and one merge. In this way, a sequence of rearrangements is hard to predict, because of the interference of splits and merges. In Figure 9, we observe the trend of slowly increasing costs as we increase the number of operations. However, even the costliest repartitioning scenario (i.e., 80 rearrangements in our 80-core system) completes in less than 200 milliseconds.

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

D. Adaptive behavior of ATraPos

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

1) Workload characteristics: First, we test the behavior of ATraPos when the workload changes. We use TATP and every 30 seconds we switch to a different transaction type. Specifically, for the first 30 seconds we run only UpdSubData transactions; then for the next 30 seconds we run only GetNewDest transactions; and for the last 30 seconds we run the standard TATP-Mix. Figure 10 depicts the results.

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

2) Data skew: Figure 11 depicts the benefits of the adaptive ATraPos behavior when skew appears in the workload. In this experiment, we use the GetSubData transaction from the

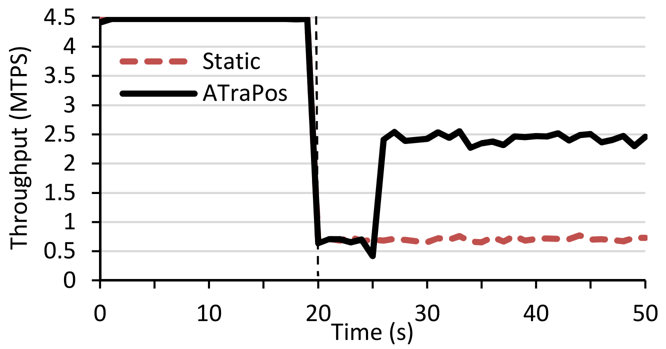

Fig. 11. Adapting to sudden workload skew.

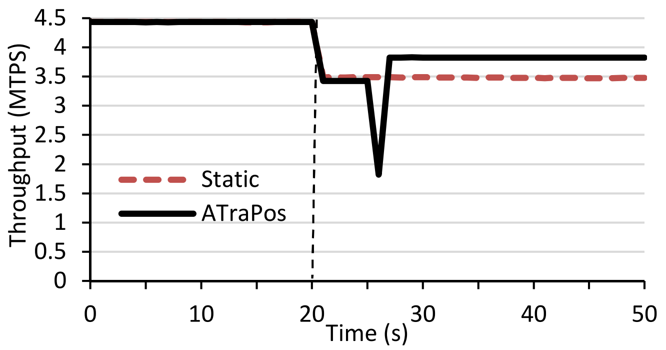

Fig. 12. Adapting to hardware failures.

TATP benchmark. This transaction initially chooses uniformly distributed values from the whole dataset. After 20 seconds, we introduce skew by specifying that 50% of the requests go to the 20% of the data. The heavy skew causes the throughput to drop by $\sim 80\%$. ATraPos quickly detects the change, optimizes for the new workload characteristics and it manages to achieve 3x better performance than the static system.

3) Underlying hardware topology: The next experiment demonstrates the ability of ATraPos to gracefully adapt to hardware changes. In this case, we test the behavior when a processor fails. We simulate the failure of a processor P by excluding all cores of P and leaving them idle. We use the `GetSubData` transaction from TATP since it is a very short transaction that is sensitive to the changes in the environment. Figure 12 shows that at the time of the simulated processor failure (one 10-core processor fails at the 20th second), the static system fails to optimally use the rest of the available hardware. It still uses a partitioning plan that assumes 80 processor cores are available. Therefore, it implicitly overloads 1 full processor (with 10 cores) that now needs to satisfy not only its own requests but also the requests that would normally go to the processor that failed. This causes a 22% drop in throughput. On the other hand, ATraPos detects the change in the underlying hardware topology and repartitions the data to create one partition for each of the 70 available cores. The optimized repartitioning removes the overloading effects and improves throughput by 11%.

4) Frequency of changes: In our last experiment, we demonstrate how ATraPos gracefully adapts to workload fluctuations. We test a dynamic scenario that consists of workloads `GetNewDest` and `TATP-Mix` from the TATP benchmark, denoted as A and B, respectively, in Figure 13. Workload A is active for the first 60 secs. ATraPos continuously monitors the throughput and as long as it remains stable, it relaxes its monitoring interval; during the first 60 secs the interval is 1 sec and it gradually becomes 8 sec (this is the upper bound). When the workload shifts to workload B at the 60th sec, ATraPos manages to identify the throughput degradation within 8 seconds. Then, it adjusts to the optimal partitioning scheme for workload B and it sets its monitoring interval back to 1 sec so it can be more alert until it realizes that the workload is stabilized; when this happens, it gradually increases the monitoring interval again. As Figure 13 depicts, when frequent workload fluctuations occur, ATraPos remains alert (keeping the monitoring interval low) and it quickly adapts to the changes. For example, in the last two workload shifts ATraPos adapts within about 2 seconds. Overall, ATraPos manages to continuously adapt and autonomously retune its monitoring setup to follow the workload fluctuations.

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

VII. Future Work

In this paper, we present the ATraPos workload and hardware-aware dynamic partitioning and placement mechanism that is designed on top of a physically partitioned shared-everything architecture. However, our techniques can also be applied to other transaction processing architectures, with the modifications we describe in the next two paragraphs.

Coarse-grained shared-nothing. We can apply the ATraPos cost model to the physically partitioned shared-nothing architecture with a few modifications. Since data is physically partitioned, the primary cost in the model is the cost of distributed transactions, as in previously proposed partitioning methods for the physically partitioned systems [30][17]. Similarly, the cost of repartitioning includes the cost of physical data movement from one instance to another. This cost is generally much higher than the repartitioning cost in the logically partitioned systems. The resource estimation part of the model can be used to determine sizes of individual instances in the system if amended with the cost model for the contention among different threads in larger instances.

Fine-grained shared-nothing. The ATraPos model can also be applied to fine-grained shared-nothing systems that are aware of the hardware topology. Such systems could detect a situation where all the participating instances of a distributed transaction are located on the same machine. Then they are able to switch to a more efficient communication channel, e.g., shared memory. In that case, the cost model could include information about the relative cost of two types of distributed transactions to choose the partitioning scheme that reduces the number of more expensive distributed transactions.

VIII. Conclusions

In this paper, we analyze how non-uniform hardware topology influences transaction processing. We quantify the impact of hardware topology and show that ignoring it severely

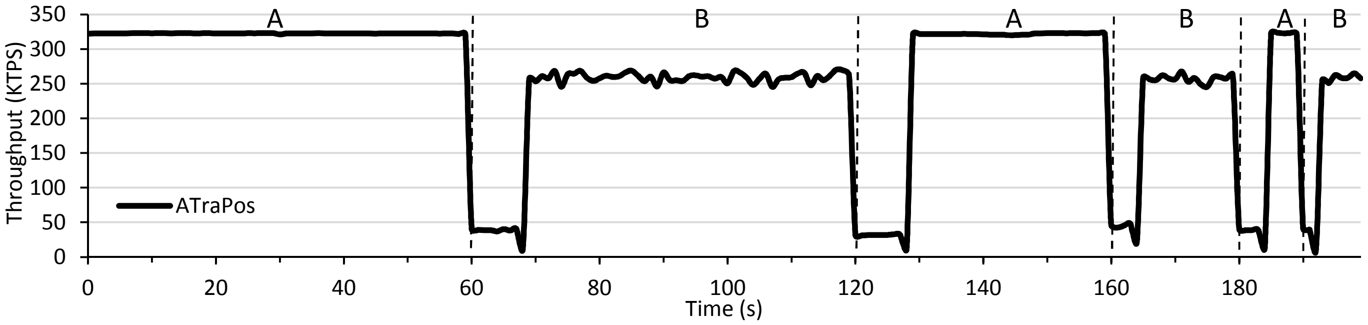

Fig. 13. Adapting to frequent changes.

limits system scalability. We identify the main shortcoming of the state-of-the-art shared-everything transaction processing systems on multisocket multicore servers as the existence of centralized data structures in the critical path.

We address this problem in ATraPos by systematically making all data structures accessed in the critical path hardware-aware. This allows us to achieve linear scalability for perfectly partitionable workloads. To address the workloads that are not perfectly partitionable, ATraPos includes a dynamic lightweight monitoring and repartitioning mechanism. Our partitioning mechanism takes into account static and dynamic workload characteristics as well as the hardware topology to choose a good partitioning and placement scheme for current workload. When workload or hardware characteristics change, it quickly adapts the current partitioning scheme to the new environment. In this way, ATraPos offers robust performance on a variety of dynamic transactional workloads on today's and upcoming non-uniform hardware platforms.

Acknowledgments

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. We also thank the reviewers for their constructive feedback. This work is partially funded by Oracle Labs and by the Swiss National Science Foundation (Grant No. 200021-146407/1).

References

[1] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki, "OLTP on Hardware Islands," *PVLDB*, vol. 5, no. 11, pp. 1447–1458, 2012.
[2] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *ISCA*, 2012, pp. 500–511.
[3] B. M. Beckmann and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *MICRO*, 2004, pp. 319–330.
[4] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009, pp. 184–195.
[5] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-Oriented Transaction Execution," *PVLDB*, vol. 3, no. 1, pp. 928–939, 2010.
[6] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. chang Miao, J. F. B. III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.
[7] K. Closson, "You buy a NUMA system, Oracle says disable NUMA! What gives?" 2009, http://kevinclosson.wordpress.com/2009/05/14/you-buy-a-numa-system-oracle-says-disable-numa-what-gives-part-ii/.
[8] M. Wilson, "Disabling NUMA parameter," 2011, http://www.michaelwilsondba.info/2011/05/disabling-numa-parameter.html.
[9] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit, "NUMA-aware reader-writer locks," in *PPoPP*, 2013, pp. 157–166.
[10] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: a general technique for designing NUMA locks," in *PPoPP*, 2012, pp. 247–256.
[11] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores," in *ASPLOS*, 2013, pp. 229–240.
[12] L. Bouganim, D. Florescu, and P. Valduriez, "Load Balancing for Parallel Query Execution on NUMA Multiprocessors," *Distributed and Parallel Databases*, vol. 7, pp. 99–121, 1999.
[13] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *PVLDB*, vol. 5, no. 10, pp. 1064–1075, 2012.
[14] Y. Li, G. Lohman, I. Pandis, R. Mueller, and V. Raman, "NUMA-aware algorithms: the case of data shuffling," in *CIDR*, 2013.
[15] T. Kiefer, B. Schlegel, and W. Lehner, "Experimental Evaluation of NUMA Effects on Database Management Systems," in *BTW*, 2013, pp. 185–204.
[16] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *EDBT*, 2013, pp. 430–441.
[17] A. Pavlo, C. Curino, and S. Zdonik, "Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems," in *SIGMOD*, 2012, pp. 61–72.
[18] P. Tözün, I. Pandis, R. Johnson, and A. Ailamaki, "Scalable and Dynamically Balanced Shared-Everything OLTP with Physiological Partitioning," *VLDB Journal*, vol. 22, no. 2, pp. 151–175, 2013.
[19] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: A Scalable Storage Manager for the Multicore Era," in *EDBT*, 2009, pp. 24–35.
[20] Intel, "Intel VTune Amplifier XE performance profiler," http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.
[21] R. Johnson, I. Pandis, and A. Ailamaki, "Improving OLTP Scalability Using Speculative Lock Inheritance," *PVLDB*, vol. 2, no. 1, pp. 479–489, 2009.
[22] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A Scalable Approach to Logging," *PVLDB*, vol. 3, no. 1, pp. 681–692, 2010.
[23] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *VLDB*, 2007, pp. 1150–1160.
[24] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki, "PLP: Page Latch-free Shared-everything OLTP," *PVLDB*, vol. 4, no. 10, pp. 610–621, 2011.
[25] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored," in *EDBT*, 2013, pp. 17–28.
[26] Linux, "numactl - linux man page," http://linux.die.net/man/8/numactl.
[27] Intel, "Intel performance counter monitor," http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization.
[28] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka, "Telecom application transaction processing benchmark (TATP)," 2009, http://tatpbenchmark.sourceforge.net/.
[29] TPC, "TPC benchmark C (OLTP) standard specification, revision 5.11," 2010, http://www.tpc.org/tpcc.
[30] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, pp. 48–57, 2010.