PLP: Page Latch-free Shared-everything OLTP

Ippokratis Pandis*
IBM Almaden Research Center

Pınar Tözün
École Polytechnique Fédérale de Lausanne

Ryan Johnson
University of Toronto

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne

ABSTRACT

Scaling the performance of shared-everything transaction processing systems to highly-parallel multicore hardware remains a challenge for database system designers. Recent proposals alleviate locking and logging bottlenecks in the system, leaving page latching as the next potential problem. To tackle the page latching problem, we propose physiological partitioning (PLP). The PLP design applies logical-only partitioning, maintaining the desired properties of shared-everything designs, and introduces a multi-rooted B+Tree index structure (MRBTree) which enables the partitioning of the accesses at the physical page level. Logical partitioning and MRBTrees together ensure that all accesses to a given index page come from a single thread and, hence, can be entirely latch-free; an extended design makes heap page accesses thread-private as well. Eliminating page latching allows us to simplify key code paths in the system such as B+Tree operations leading to more efficient and maintainable code. Profiling a prototype PLP system running on different multicore machines shows that it acquires 85% and 68% fewer contentious critical sections, respectively, than an optimized conventional design and one based on logical-only partitioning. PLP also improves performance up to 40% and 18%, respectively, over the existing systems.

1. INTRODUCTION

Due to concerns over power draw and heat dissipation, processor vendors can no longer rely on rising clock frequencies or increasingly aggressive micro-architectural techniques to boost performance. Instead, they focus on parallelism by placing many independent processing cores in each chip. The resulting multicore designs require software to expose enough execution parallelism in order to exploit the abundant and rapidly growing hardware parallelism. However, this is not an easy task, especially given the high degree of hardware resource sharing common to multicore designs.

On-line transaction processing (OLTP) is a particularly complex data management application that needs to perform efficiently in modern hardware. It has been shown that conventional shared-every-thing OLTP systems may face significant scalability problems in highly parallel hardware [13]. There is increasing evidence that one source of scalability problems arises from the conventional transaction-oriented assignment of work policy, which assigns each transaction to a thread [23]. The transaction, along with the physical arrangement of records within the data pages, determines what resources (e.g. records and pages) each thread will access. The random nature of transaction processing requests leads to unpredictable data accesses [24, 23] that complicate resource sharing and concurrency control.

Such unpredictability favors pessimistic systems which clutter the transaction's execution path with many lock and latch acquisitions to protect the consistency of the data. These critical sections often lead to *contention* which limits scalability [13] and in the best case imposes a significant penalty to single-thread performance [8]. In addition, the performance of shared-everything systems is vulnerable to *page false sharing* effects where hot but unrelated records happen to reside on the same page. Careful tuning is often needed to detect and resolve such issues, for example by padding problematic records to spread them out.

Following a different approach, shared-nothing systems deploy many independent database instances which collectively serve the workload [25, 6]. In shared-nothing designs the contention for shared data resources can be explicitly tuned (the database administrator determines the number of processors assigned to each instance), potentially leading to superior performance as long as inter-instance communication can be minimized. The H-Store system takes this approach to the extreme, with single-threaded database instances that eliminate critical sections altogether [26]. However, shared-nothing systems physically partition the data and deliver poor performance when the workload triggers distributed transactions [9, 4] or when skew causes load imbalance [4]. Repartitioning to rebalance load requires the system to physically move and reorganize all affected data. These weaknesses become especially problematic as partitions become smaller and more numerous in response to the multicore trend.

1.1 Multi-rooted B+Trees

To alleviate the difficulties imposed by page latching and repartitioning, we propose a new physical access method, a type of multi-rooted B+Tree called *MRBTree*. The root of each sub-tree in this structure corresponds to a logical partition of the data, and the mapping of key ranges to sub-tree

*Work done while author was at Carnegie Mellon University.

Proceedings of the VLDB Endowment, Vol. 4, No. 10

roots forms a durable part of the index's metadata. Partition sizes are non-uniform, making the tree robust against skewed access patterns, and repartitioning is cheap because it involves very little data movement.

When deployed in a conventional shared-everything system, the MRBTree eliminates latch contention at the index root; each partition is assigned to one thread, with partitions sized to balance load. Partitioning also reduces the expected tree height by at least one (partitions containing hot data can be very small). Thanks to the tree's fast repartitioning capabilities, the system can respond quickly to changing access patterns. Further, the MRBTree can also potentially benefit systems which use shared-nothing parallelism in a shared-memory environment (e.g. possibly H-Store [26]).

1.2 Physiological partitioning

Recent work proposes logical-only partitioning [23] to address problems with conventional execution while avoiding the weaknesses of shared-nothing approaches. Logical-only partitioning assigns each partition to one thread; the latter manages the data locally without the overheads of centralized locking. However, purely logical partitioning does not prevent conflicts due to false sharing, nor does it address the overhead and complexity of page latching protocols.

Ideally, we would like a system with the best properties of both shared-everything and shared-nothing designs: a centralized data store which sidesteps the challenges of moving data during (re)partitioning, and a partitioning scheme which eliminates contention and the need for page latches.

This paper presents *physiological partitioning (PLP)*, a transaction processing approach which partitions logically the physical data accesses. PLP achieves its goal by using the MRBTree access method to enhance logical-only partitioning and capture most types of physical data accesses as well. Under PLP, a partition manager assigns threads to sub-tree roots of MRBTrees and ensures that requests distributed to each thread reference only the corresponding sub-tree. As a result, threads can bypass the partition mapping and their accesses to the subtree are entirely latch-free (similar to shared-nothing systems). At the same time, the underlying MRBTree supports fast repartitioning and does not require distributed transactions when requests span partitions (like a shared-everything system). Finally, PLP can extend the partitioning down into the heap pages where non-clustered records are actually stored, eliminating another class of page latching.

1.3 Contributions and organization

The contributions of this paper are four-fold. (a) We provide a simple categorization of the communication patterns which identifies clearly latent scalability bottlenecks; (b) Using this categorization we identify page latching as a lurking performance and scalability bottleneck in modern transaction processing systems, with importance proportional to the available hardware parallelism. Further, we identify use cases where latching impacts shared-everything designs even today and show that our proposed multi-rooted B+Tree design alleviates this contention; (c) We design a shared-everything OLTP system based on physiological partitioning. The design eliminates the need for page latching during accesses to both index and heap pages. The page latch elimination makes PLP more scalable and less vulnerable to bad application design. (d) We evaluate a prototype implementation of PLP and show that it eliminates nearly all page latching in the system. As page latching constitutes the majority of centralized critical sections in the system, PLP acquires 85% and 68% fewer contentious critical sections per transaction than an optimized conventional design and a logical-only partitioned system, respectively, improving scalability and yielding up to 38% higher performance on multicore machines. We also identify remaining opportunities to eliminate the highly complex code paths that arise due to latching, improving performance and making code more maintainable.

The rest of the document is structured as follows. Section 2 analyzes the communication patterns in OLTP systems. Section 3 presents the PLP system design and Section 4 evaluates its performance. Finally, Section 5 presents related work and Section 6 concludes.

2. COMMUNICATION PATTERNS

Traditional transaction processing systems excel at providing high concurrency, or the ability to interleave multiple concurrent requests or transactions over limited hardware resources. However, as core counts increase exponentially, performance increasingly depends on execution parallelism, or the ability for multiple requests to make forward progress simultaneously in different execution contexts. Even the smallest of serializations on the software side therefore impact scalability and performance [10]. Unfortunately, recent studies show that high concurrency in transaction processing systems does not necessarily translate to sufficient execution parallelism [13, 14], due to the high degree of irregular and fine-grained communication they exhibit.

Proposals to tackle overhead and scalability bottlenecks fall into two general categories: (1) reducing the degree of communication and contention within shared-everything systems, relying on efficient communication via shared caches to keep synchronization overheads low; and (2) taking a shared-nothing approach [25], relying on the low-latency of multicore hardware to keep overheads manageable in spite of the challenges which accompany distributed transactions and load balancing.

In this section we first categorize the types of communication that can occur in an OLTP system, and from this point of view we analyze the execution of a modern shared-everything system. Then, we revisit the debate between the shared-everything and shared-nothing approaches.

2.1 Types of communication

OLTP systems employ several different types of communication and synchronization. *Database locking* operates at the logical (application) level to enforce isolation and atomicity between transactions. *Page latching* operates at the physical (database page) level to enforce the consistency of the physical data stored on disk in the face of concurrent updates from multiple transactions. Finally, at the lowest levels, *critical sections* protect various code paths which must execute serially to protect the consistency of the system's internal state. Critical sections are traditionally protected by mutex locks, atomic instructions, etc. We note that locks and latches, which form a crucial part of the systems' internal state, are themselves protected by critical sections; analyzing the behavior of critical sections thus captures nearly all forms of communication in the DBMS.

Critical sections, in turn, fall into three categories depending on the nature of the contention they tend to trigger in the system. For example, pairs of threads which form

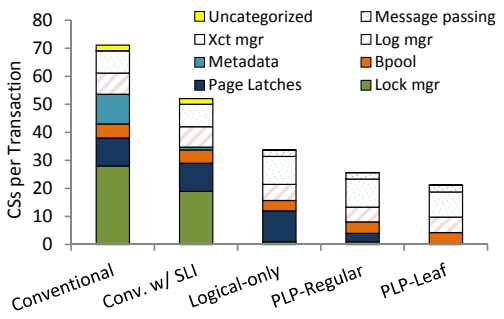


Figure 1: Breakdown of the critical sections when running the TATP OLTP benchmark.

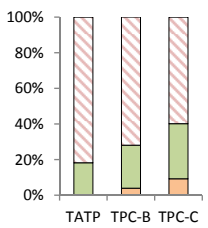


Figure 2: Page latch breakdown for various OLTP benchmarks.

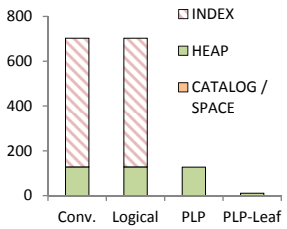


Figure 3: Page latches acquired by different designs in TATP.

producer-consumer pairs protect their communication with a critical section but cannot generate significant contention. We refer to these as *fixed-contention* critical sections because contention is independent of the underlying hardware and depends only on the (fixed) number of threads which communicate. At the other extreme, *unscalable* critical sections have the highly undesirable tendency to affect most threads in the system. As hardware parallelism increases the degree of contention also increases and inevitably grows into a bottleneck. Making these critical sections shorter or less frequent provides a little slack but does not fundamentally improve scalability. Finally, Moir et al. [21] introduce the notion of *composable* critical sections; those having the property that multiple threads can aggregate their operations. Composable critical sections are highly resistant to contention because threads take advantage of queuing delays to combine their requests and drop out of the queue. The critical section is thus self-regulating: adding more threads to the system gives more opportunity for threads to combine rather than competing directly for the critical section.

2.2 Communication patterns in OLTP

As the previous section hints, the real key to scalability lies in converting all unscalable communication to either the fixed or composable type, thus removing the potential for bottlenecks to arise. The three left-most bars of Figure 1 compare the number and types of critical sections executed by a conventional OLTP system and two others designed to reduce contention due to locking: Speculative Lock Inheritance [12] and data-oriented execution [23] (labeled as *SLI* and *Logical-only*, respectively). Each bar shows the number of critical sections entered during a mix of short transactions, categorized by the originating storage manager service (see Section 4.1). Locking and latching form a significant fraction of the total communication for the baseline system. SLI achieves a performance boost by sidestepping the most problematic critical sections associated with the lock manager, but fails to address the remaining (still-unscalable) communication in that category. Logical partitioning, in contrast, eliminates nearly all types of locking, replacing both contention and overhead of centralized communication with efficient, fixed communication via message passing.

With locking removed, latching remains by far the largest source of critical sections. There is no predefined limit to the number of threads which might attempt to access a given page simultaneously, so page latching represents an unscalable form of communication which should be either eliminated or converted to a scalable type. The remaining categories represent either fixed communication (e.g. transaction management), composable operations (e.g. logging [14]), or a minor fraction of the total unscalable component.

Examining page latching more closely, Figure 2 decomposes the page latches acquired by three popular OLTP benchmarks into the different types of database pages: metadata, index pages, and heap pages. The majority of page latches (60%-80%) reside in index structures. Heap page latches are another non-negligible component, accounting for nearly all remaining page latches.

2.3 Physical vs. logical partitioning

With the preceding characterization of communication patterns in mind, we now return to the question of logical partitioning (shared-everything) versus physical partitioning (shared-nothing). As its name suggests, logical partitioning eliminates unscalable communication at the logical level, namely database locking. However, it has little impact on the remaining communication, which arises in the physical layers and cannot be managed cleanly from the application level. Even when requests do not communicate at the application level, threads must acquire page latches and potentially perform other unscalable communication.

Shared-nothing systems [25, 6] are an appealing design, giving the designer explicit control over the number of threads per instance. Thus, the contention on each component of the system can be controlled or even eliminated. However, such designs give up too much by eliminating all communication within the engine. Even the composable and fixed types of critical sections, which do not threaten scalability become problematic. For example, logging is not amenable to distribution [14], and physically-partitioned systems either use a shared log [18] or eliminate it completely [26].

Perhaps the biggest challenge for shared-nothing systems arises with distributed transactions, due to requests accessing data from multiple physically distributed database instances. The scalable execution of distributed transactions has been an active field of research for the past three decades, with researchers from both academia and industry, persuasively arguing that they are fundamentally not scalable [2, 9]. Furthermore, the performance of shared-nothing systems is very sensitive to imbalances in load arising from skew in either data or requests while non-partition-aligned operations (such as non-clustered secondary indexes) may pose significant barriers to physical partitioning.

3. PHYSIOLOGICAL PARTITIONING

We have seen how both logically- and physically-partitioned designs offer desirable properties, but also suffer from weak-

nesses which threaten their scalability. In this work we therefore propose physiological partitioning (or PLP), a hybrid of the two approaches which combines the best properties of both. Like a physically-partitioned system the majority of physical data accesses occur in a single-threaded environment which obviate the need for page latching; like the logically-partitioned system, locking is distributed without resorting to distributed transactions and load balancing requires almost no data movement.

3.1 Design overview

Each transaction in a typical OLTP workload accesses a very small subset of records via indexes (sequential scans are prohibitively expensive). PLP therefore centers around the indexing structures of the database.

Figure 4 gives a high-level overview of a physiologically-partitioned system. We adapt the traditional B+Tree [1] (top left of Figure 4) for PLP by splitting it into multiple subtrees, each covering a contiguous subset of the key space (bottom of Figure 4). A *partitioning table* becomes the new *root* and maintains the partitioning as well as pointers to the corresponding subtrees. We call the resulting structure a multi-rooted B+Tree (MRBTree). The MRBTree partitions the data but unlike a horizontally-partitioned workload (top right of Figure 4), all subtrees belong to the same database file and can exchange pages easily; the partitioning, though durable, is dynamic and malleable rather than static. Implementation details about MRBTrees can be found in Section A of the Appendix.

With the MRBTree in place, the system assigns each subtree to a single thread, guaranteeing exclusive access for latch-free execution. A *partition manager* layer controls all partition tables and makes assignments to threads. The threads in PLP do not reference partition tables during normal processing, which might otherwise become a bottleneck. Instead, the partition manager ensures that all work given to a thread involves only data it owns. The partition manager breaks transactions into directed graphs, passing each node to the appropriate thread and assembling the results into complete transactions. Repartitioning occurs at a higher level in the partition manager and therefore can be latch-free as well; the partition manager simply quiesces affected threads until the process completes.

All indexes in the system –primary, secondary, clustered, non-clustered– can be implemented as MRBTrees; data are stored directly in clustered indexes, or in tightly integrated heap file pages referenced by record ID (RID). When the system can infer partitions from secondary (non-clustered) index columns, the partition's thread manages them directly. The remaining (non-partition aligned) secondary indexes are accessed as in the conventional system, but each leaf entry records the associated fields used for the partitioning so that the result of each probe can be passed to its partition's owning thread for further processing.

3.2 Benefits of physiological partitioning

The database system must address two challenges as it assigns work to threads and threads to data. Within a partition, multi-threaded access leads to the overheads that come with unscalable critical sections. Between partitions, uneven access patterns reduce performance by leaving some partitions oversubscribed (with long response times) and others underutilized (leaving hardware idle). Physiological partitioning avoids both problems by assigning only one thread to any given piece of physical data, and by allowing frequent, lightweight repartitioning in response to changing load.

Table 1: Repartitioning costs when splitting a partition with 466 MB data in half (U: Updates, D: Deletes, I: Inserts).

	Records Moved	Primary Index Entries Moved	Primary Index #Pages Read	Primary Index #Pointer Updates	Primary Index Changes	Secondary Index Changes
PLP-Regular	-	8KB	-	7	-	-
PLP-Leaf	8.3KB	8KB	1	7	85 U	85 U
PLP-Partition	233MB	8KB	14365	7	2.44M U	2.44M U
Shared-Nothing	233MB	-	14365	-	2.44M I + 2.44M D	2.44M I + 2.44M D
PLP (Clustered)	8.3KB	5.3KB	-	7	-	85 U
Shared-Nothing (Clustered)	233MB	-	-	-	2.44M I + 2.44M D	2.44M I + 2.44M D

3.2.1 Load balancing and repartitioning

In database workloads, uniform data distributions and access patterns are the exception rather than the rule. Data values often follow long-tailed distributions, and accesses to those data are similarly non-uniform (e.g. the *slashdot effect*). These asymmetries necessitate partitioning for both capacity and load balancing reasons. Conventional horizontal partitioning requires splitting a data set –usually in an application-visible way– across multiple database files. In this way, applications determine statically which partition holds a given piece of data, avoiding the need to touch other partitions. The disadvantage of such schemes is that repartitioning data requires moving it physically from one file to another, with the corresponding index maintenance as well as logging and I/O overheads. Data movement can be performed lazily to move the cost partly off the critical path, but it cannot be avoided. The high cost of repartitioning favors a careful up-front partitioning which is updated only when the performance loss due to non-optimal tuning outweighs the pain of a data reorganization [4].

In contrast, modifying a physiological partitioning is a lightweight operation because even large adjustments update only a small amount of metadata on the physically monolithic database file. Repartitioning is also transparent at the application level because the system can tune itself automatically in response to the observed workload without harming performance in the process. In addition, the hottest data tends to reside in smaller index structures, leading to fewer page accesses and reduced access times.

Table 1 gives an example of the repartitioning costs for different systems (see Section 3.3 for the different PLP designs) based on the cost model described in Section C of the Appendix. In this example, a partition which contains 433MB of 100-byte data records in a heap file is to be split in half. We assume that there is a primary index of height 3 with 170 32-byte entries on each page. The first four rows of the table assume there is a unique non-clustered primary index and a secondary index in the system, whereas for the last two rows there is a unique clustered primary index and a secondary index. The Shared-Nothing system does not perform logging but instead maintains replicas to deal with crashes, and the cost given on the table is just for one replica. Finally, for the PLP designs the number of moved records represents the worst case scenario.

The PLP variations, excepting PLP-Partition, move very few records compared with the Shared-Nothing system, as Table 1 shows. In the worst case, PLP-Partition moves the same number of records as a Shared-Nothing system. For the clustered index case, the PLP system is less costly than the Shared-Nothing system, both in terms of record

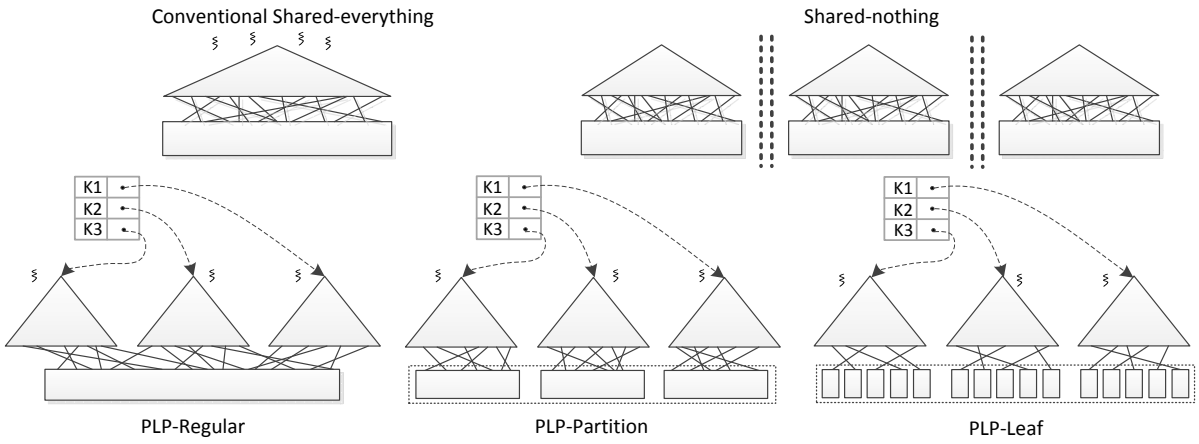


Figure 4: The conventional shared-everything and shared-nothing designs and the PLP variations.

movement and index maintenance. When we calculate the corresponding costs for a larger heap file with a B+tree of height 4, the repartitioning cost for the Shared-Nothing system (and PLP-Partition) becomes prohibitive. We conclude that the PLP-Regular and PLP-Leaf designs have an advantage over Shared-Nothing systems during load-balancing.

In addition, agile load-balancing gradually migrates hot records to small partitions. The subtrees will have fewer tree levels, and cheaper probes, for those partitions.

3.2.2 *Exclusive physical page accesses*

Under physiological partitioning, each partition is permanently locked for exclusive physical access by a single thread, which then handles all the requests for that partition. This allows the system to avoid several sources of overhead, as described in the following paragraphs.

Latching contention and overhead. Though page latching is inexpensive compared with acquiring a database lock, the sheer number of page latches acquired imposes some overhead and can serialize B+Tree operations as transactions crab down the tree during a probe. The problem becomes more acute when the lower levels of the tree do not fit in memory, because a thread which fetches a tree node from disk holds a latch on the node's parent until the I/O completes, preventing access to 80-100 other siblings which may well be memory-resident. Section 4.4 evaluates a case where latching becomes expensive for B+Tree operations and how PLP can eliminate this problem by allowing latch-free accesses on index pages.

False sharing of heap pages. One significant source of latch contention arises when multiple threads access unrelated records which happen to reside on the same physical database page. In a conventional system false sharing requires padding to force problematic database records to different pages. A PLP design that allows latch-free heap page accesses achieves the same effect automatically (without the need of expensive tuning) as it splits hot pages across multiple partitions. Section 4.4 evaluates this case as well.

Serialization of structural modification operations (SMOs). ARIES/KVL indexes [19] allow only one SMO (such as a leaf split) to occur at a time, serializing all other accesses until the SMO completes. Partitioning the tree physically with MRBTrees eases the problem by distributing SMOs across subtrees (whose roots are fixed) without having to apply more complicated protocols [20, 11]. The benefits of parallel SMOs are apparent in the case of insert-heavy workloads which we evaluate in Section B of the Appendix.

Code complexity. Finally, with all latching eliminated, the code paths which handle contention and failure cases can be eliminated as well, simplifying the code significantly. For example, a huge source of complexity in traditional B+Trees arises due to the sophisticated protocols which maintain consistency during SMO in spite of concurrent probes from other threads. The simpler code not only is more efficient but also it is easier to maintain. In this paper, we did not attempt the code refactoring required to exploit these opportunities, and the performance results we report are therefore conservative. We note that B+Tree probes are the most expensive remaining component of the PLP system.

3.3 Heap page accesses

In PLP a heap file scan is distributed to the partition-owning threads and performed in parallel. Large heap file scans reduce concurrency of OLTP applications and PLP has little to offer. Still, heap page management opens up an additional design option, since we can extend the partitioning of the accesses at the heap pages. That is, when records reside in a heap file rather than in the MRBTree leaf pages, PLP can ensure that accesses to pages are partitioned in the same way as index pages. There are three options on how to place and access records in the heap pages, depicted in Figure 4: (1) keep the existing heap page design (*PLP-Regular*); (2) each heap page keeps records of only one logical partition (*PLP-Partition*); and (3) each heap page is pointed by only one leaf page of the primary MRBTree (*PLP-Leaf*).

PLP-Regular simply keeps the existing heap page operations. Without any modification, the heap pages still need to be latched because they can be accessed by different threads in parallel. This may be acceptable because heap page accesses are not the biggest fraction of the total page accesses in OLTP (as low as 30%, according to Figure 2). Thus, there is room for significant improvement even if we ignore them. However, allowing heap pages to span partitions prevents the system from responding automatically to false sharing or other sources of heap page contention.

In PLP-Partition and PLP-Leaf the MRBTree and heap operations are modified so that heap page accesses are partitioned as well. The difference between the two is that in the former a heap page can be pointed by many leaf pages as long as they belong to the same partition, while in the latter a heap page is pointed by only one leaf page.

Both variations provide latch-free heap page accesses, but they suffer some disadvantages. Forcing a heap page to contain records that belong to a specific partition causes fragmentation. In the worst case, each leaf has room for one more entry than fits in the heap page, resulting in nearly double the space requirement (Section D of the Appendix measures this cost). Further, in PLP-Leaf every leaf split must also split one or more heap pages, increasing the overhead of record insertion (deletions are simple because a leaf may point to many heap pages). On the other hand, PLP-Partition by allowing multiple leaf pages from a partition to share a heap page, forces the system to reorganize potentially significant numbers of heap pages with every repartitioning. Significant reorganization costs go against the philosophy of physiological partitioning, so we favor PLP-Leaf.

The two extensions impose one additional piece of complexity: During record insertion, the system must identify the correct MRBTree entry before selecting a heap page for the record. Because, the storage management layer is completely unaware of the partitioning strategy (by design), it must make callbacks into the upper layers of the system to identify an appropriate heap page for each insertion.

Similarly, a partition split may split heap pages as well, invalidating the record IDs of migrated records. The storage manager therefore exposes another callback so the metadata management layer can update indexes and other structures which reference the stale RIDs. We note that when PLP-Leaf splits leaf pages during record insertion, the same kinds of record relocations arise and use the same callbacks.

4. EVALUATION

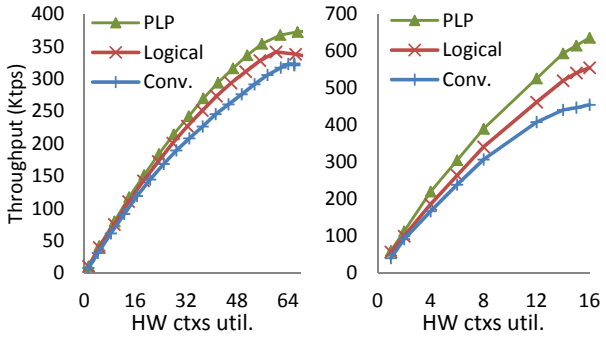
Our evaluation consists of four parts. First, Section 4.2 quantifies how the different designs impact page latching and critical section frequency. Section 4.3 evaluates the performance impact of those changes. Section 4.4 examines how PLP reduces latch contention on index and heap pages, and its performance impact. Finally, Section 4.5 shows the effect of repartitioning on performance.

4.1 Experimental setup

We consider five different systems: (a) The *conventional* system employs Speculative Lock Inheritance [12] to reduce the contention in the lock manager; (b) *Logical-only* is a data-oriented transaction processing prototype that applies logical-only partitioning; (c) *PLP or PLP-Regular* prototypes the basic PLP design and accesses the MRBTree index pages without latching; (d) *PLP-Partition* extends the PLP system, so that one logical partition "owns" each heap page, allowing latch-free both index and heap page accesses; finally, (e) *PLP-Leaf* assigns heap pages to leaves of the primary MRBTree index, also allowing latch-free index and heap page accesses.

To ensure reasonable comparisons, all the prototypes are built on top of the same version of the Shore-MT storage manager [13], incorporate the logging optimizations of [14], and share the same driver code.

All experiments were performed on two machines: an x86_64 box running Red Hat Linux 5 (four sockets, quad-core AMD Opteron 8356 processors, 2.4GHz clock) and a Sun UltraSPARC T5220 server running Solaris 10 (one socket, 64-core Sun Niagara II, 1.4GHz clock). The high degree of hardware parallelism on these systems makes them good indicators of the challenges all platforms will face as multi-sockets host more and more cores per socket. Due to unavailability of a suitably fast I/O sub-system, all the experiments are with memory-resident databases, but the relative behavior of the systems will be similar with larger databases.



(a) Sun Niagara II (b) 4-socket Quad x86_64

Figure 5: Throughput of the `GetSubscriberData` transaction in two multicore machines. PLP's benefit increases with higher throughput.

4.2 Page latches and critical sections

First we measure how PLP reduces the number of page latch acquisitions in the system. Figure 3 shows the number and type of page latches acquired by the conventional and logically-partitioned systems, plus two versions of the PLP system: PLP-Regular, which does not change heap page accesses and PLP-Leaf, which allows latch-free heap accesses. Each system executes the same number of transactions from the TATP benchmark. PLP-Regular reduces the amount of page latching per transaction by more than 80%; PLP-Leaf reduces the total further to roughly 1% of the initial page latching (the remaining latches are associated with metadata and free space management).

The two right bars of Figure 1 compare total critical section entries of PLP vs. the conventional and logically-partitioned systems. The two PLP variants eliminate the vast majority of lock- and latch-related critical sections, leaving only metadata and space management latching as a small fraction of the critical sections. Transaction management, the largest remaining component, mostly employs fixed-contention communication to serialize threads which attempt to modify the transaction object's state. Similarly, the buffer pool-related critical sections are mostly due to the communication between cleaner threads, which again do not impact scalability. Overall, PLP-Leaf acquires 85% and 65% fewer contentious critical sections than the conventional and logically-partitioned systems respectively.

4.3 Scalability and performance

Having established that the PLP designs effectively reduce the page latch acquisitions and critical sections, we next measure their impact on performance and overall system scalability. The two graphs of Figure 5 show the through-

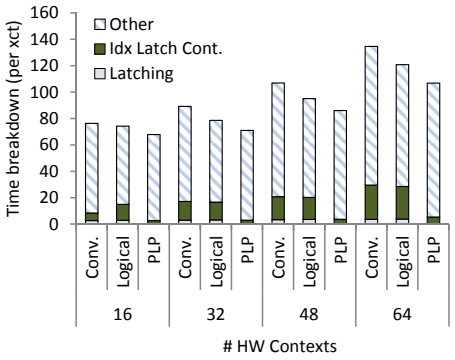


Figure 6: Time breakdown per transaction in an insert/delete-heavy benchmark.

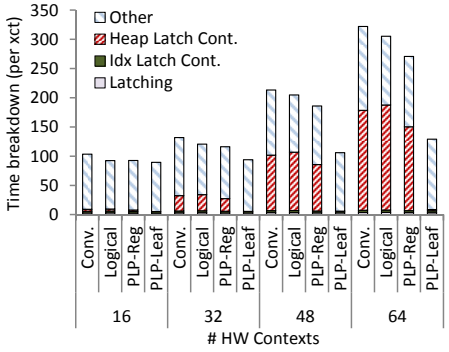


Figure 7: Time breakdown per transaction in the TPC-B benchmark with false sharing on heap pages.

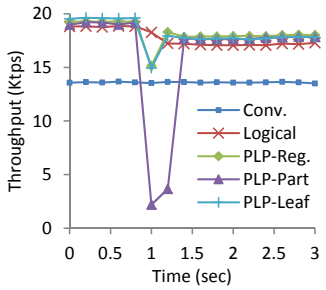


Figure 8: Throughput during repartitioning.

put of the three main designs under comparison as we increase hardware utilization of the two multicore machines. The workload consists of clients that repeatedly submit the `TATP-GetSubscriberData` transaction, which is read-only and ideally should impose no contention whatsoever.

PLP shows superior scalability, as evidenced by the widening performance gap with the other two systems as utilization increases. As we can see, logical partitioning in the 4-socket Quad x86_64 system delivers a 22% speedup over the baseline case. PLP delivers an additional 18%, or nearly 40% over the baseline. The corresponding improvements in the more parallel but overall slower Sun Niagara machine are 6% and 16%. A significant fraction of the speedup comes from the MRBTree probes, which are effectively one level shallower, since threads bypass the "root" partition table node during normal operation.

4.4 Reducing latch contention

Since PLP designs eliminate the latch acquisitions for index and heap pages, they also eliminate the time spent waiting to acquire these latches.

Figure 6 shows the impact in the transaction execution time as PLP eliminates the contention on index page latches. The graph gives the time breakdown per transaction for the different designs as an increasing number of threads run an insert/delete-heavy workload on the TATP database. In this benchmark, each transaction makes an insertion or a deletion to the `CallFwd` table, causing page splits and contention for the index pages that lead to the records being inserted/deleted. As Figure 6 shows, the conventional and the logically-partitioned systems experience contention on the index page latches. They both spend 15-20% of their time waiting, while the PLP prototype eliminates the contention achieving proportional performance improvements.

Figure 7 gives the time breakdown per transaction when we run the `TPC-B` benchmark. In this experiment we do not pad records to force them onto different pages. Transactions often wait for others because the record(s) they update happen to reside on latched heap pages. The conventional, logical, and basic PLP design all suffer from this false sharing of heap pages. At high utilization this contention wastes more than 50% of execution time. On the other hand, PLP-Leaf is immune, reducing response time by 13-60% and achieving proportional performance improvement.

4.5 Tolerance to repartitioning

Finally, we evaluate how repartitioning affects the performance of the different designs. Figure 8 shows the throughput for the conventional, the logically-partitioned, and the three PLP designs when we run a microbenchmark where 2 clients repeatedly submit a transaction which probes for the account balance of a `Subscriber` in the `TATP` database. One second into the measurement, requests change from being uniformly distributed to having 50% of the requests access only 10% of the database. To rebalance load, the partitioning-based systems must move 40MB (out of 50MB) from the hot partition to the cold one, splitting requests evenly between the hot (10MB) and cold (90MB) partitions.

The performance of the conventional system remains mostly unaffected because it does not partition the data. Logical-partitioning quickly adjusts its routing tables and also maintains nearly the same performance. Repartitioning is more expensive for the PLP designs because they have to perform physical operations. In PLP-Regular, very few index entries are updated, leading to a small dip in throughput during repartitioning. PLP-Leaf suffers an equally small dip. PLP-Partition suffers a much larger dip while it reorganizes a large number of heap pages as explained in Section 3.3. We note that differences in performance during repartitioning will increase with partition sizes.

5. RELATED WORK

The complexity and overheads of database management systems are well-known. For example, [8] shows that, even in a single-threaded OLTP system, logging, locking, latching, and bufferpool accesses contribute roughly equal overheads and together account for the majority of machine instructions executed during a transaction. Our previous work shows that these overheads become scalability burdens in multicore hardware [13]. PLP eliminates entire categories of serializations, along with the corresponding bottlenecks.

In the shared-everything arena, recent proposals for speculative lock inheritance [12] and data-oriented transaction execution [23] minimize the need for interaction with a centralized lock manager. Where speculative lock inheritance allows the system to spread lock operations across multiple transactions to reduce contention, data-oriented systems replace the central lock manager with thread-local lock management. Reducing lock contention with data-oriented execution is also studied for data-streams' operators [5] by mak-

ing threads delegate the work on some data to the thread that already holds the lock for that data and move to the next operation in their queues.

Other proposals tackle the weakness posed by the centralized log manager, with [14] presenting a scalable log buffer and [3] exploiting flash technology to reduce logging latencies. These proposals show even seemingly-pervasive forms of communication can be reduced or sidestepped to great effect. However, none of them addresses physical data accesses involving page latching and the buffer pool, the other two major overheads in the system, which PLP eliminates.

In clustered databases, shared-cache shared-disk designs [16] allow database instances to share their buffer pools and avoid accesses to the shared-disk. However, they do not handle the physical latch contention while accessing the pages from the shared-cache.

As discussed previously, shared-nothing [25, 6, 26] systems have an appealing design that eliminates critical sections altogether. However, they struggle both pro-actively to reduce the need to execute distributed transactions through efficient partitioning [4] as well as re-actively to reduce overheads when distributed transactions cannot be avoided [15]. On the other hand, PLP, in addition to eliminating a big portion of the unscalable critical sections, offers a less costly way of load balancing and communication for distributed transactions since partitions share the same memory space.

Alternatives to traditional B+Tree concurrency control protocol are studied to allow multiple SMOs at the same time [20, 11]. The MRBTree index structure provides an alternative to these techniques, allowing concurrent SMOs with less code complexity. However, these techniques could be implemented alongside with MRBTrees to achieve concurrency within a partition, should that be desirable for a conventional system. Several earlier works propose B+Trees having multiple roots to reduce contention due to locking [22, 7], and Lee et al. [17] also propose a partitioned B+tree design to improve the online reorganization of records in a shared-nothing system. However, again none of these proposals targets physical latch contention in the system.

6. CONCLUSIONS

Unlike conventional systems, which either embrace fully shared-everything or shared-nothing philosophies, physiological partitioning takes the best features of both to produce a hybrid system that operates nearly latch- and lock-free, while still retaining the convenience of a common underlying storage pool and log. We achieve this result with a new multi-rooted B+tree structure and careful assignment of threads to data, which allows easy repartitioning and naturally adapts to hot-spots accelerating their accesses. As the hardware of database servers becomes even more parallel and non-uniform the benefits of PLP will only increase.

Acknowledgments

We cordially thank Ioannis Alagiannis and the other members of the DIAS laboratory for their support. This work was partially supported by a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

7. REFERENCES

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *SIGFIDET*, 107–141, 1970.

[2] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, 7–7, 2000.

[3] S. Chen. FlashLogging: exploiting flash devices for synchronous logging performance. In *SIGMOD*, 73–86, 2009.

[4] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010.

[5] S. Das, S. Antony, D. Agrawal, and A. El Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *PVLDB*, 2:217–228, 2009.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDE*, 2(1):44–62, 1990.

[7] G. Graefe. Sorting and indexing with partitioned B-trees. In *CIDR*, 1–13, 2003.

[8] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 981–992, 2008.

[9] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 132–141, 2007.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33 –38, 2008.

[11] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. B-tree concurrency control and recovery in page-server database systems. *ACM TODS*, 31:82–132, 2006.

[12] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 24–35, 2009.

[14] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3:681–692, 2010.

[15] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 603–614, 2010.

[16] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton, and S. Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In *VLDB*, 683–686, 2001.

[17] M.-L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *SIGMOD*, 225–236, 2000.

[18] D. Lomet, R. Anderson, T. K. Rengarajan, and P. Spiro. How the Rdb/VMS data sharing system became fast. Technical Report CRL-92-4, DEC, 1992.

[19] C. Mohan. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *VLDB*, 392–405, 1990.

[20] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *SIGMOD*, 371–380, 1992.

[21] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*, 253–262, 2005.

[22] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8:199–221, 2000.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[24] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In *WMPI*, 37–45, 2004.

[25] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9:4–9, 1986.

[26] 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*, 1150–1160, 2007.

APPENDIX

In this appendix, we first present in detail the design of MRBTrees, which is centerpiece to the PLP design, and show how beneficial the MRBTrees can be even for conventional systems. Next, we give a cost model for the repartitioning analysis and evaluate the fragmentation costs of the three PLP variations. Finally, we discuss weaknesses of the design and its potential in future hardware architectures.

A. MULTI-ROOTED B+TREE

At Section 3 we outlined the operation of the MRBTree at a high level. In this section we examine the implementation details and important design considerations, including the layout of the partition table and the tree operations.

A.1 Partition table design

The "root" of an MRBTree is a partition table which identifies the disjoint subsets of the key range which are assigned to each sub-tree as well as a pointer to the root of each tree. Because the routing information is cached in memory as a ranges map by the partition manager, the on-disk layout favors simplicity rather than optimal access performance. We therefore employ a standard slotted page format which stores key/root pairs. If the partitioning information cannot fit on a single page (for example, if the number of partitions is large or the keys are very long) the routing page is extended as a linked list of routing pages. In our experiments we have never encountered the need to extend the routing page, however, as several dozen mappings fit easily in 8KB, even assuming rather large keys.

A.2 Inserting and deleting records

Record insertion (deletion) takes place as in regular B+trees. When the key to insert (delete) is given, the ranges map routes it to the sub-tree that corresponds to the key range the key belongs to and the insert (delete) operation is performed as in a regular B+tree in that sub-tree. The other sub-trees, ranges map, and the routing page do not get affected by the insert (delete) operation at all.

The main difference of an MRBTree index during an insert (delete) operation becomes apparent when the insert (delete) causes a structure modification in the B+tree. For example, a B+tree page split during an insert may result in a new entry insertion at the root node of the tree. Due to this, only one structure modification is allowed for a B+tree index at a time in traditional ARIES/KVL concurrency control protocol [19]. In the MRBTree index the routing page contains the partitioning information which does not depend on the actual values that are in the index. Thus, it does not get affected from a structure modification in one of the sub-trees, which allows having multiple structure modifications in parallel for the index.

A.3 Structural modifications

The MRBTree inherits node split/merge capability, with a few changes, from the standard ARIES/KVL B+Tree [19]. In addition, it adds two new operations, which we call slice and meld, which allow entire sub-trees to migrate between partitions for repartitioning purposes.

A.3.1 Melding sub-trees

When a sub-tree is accessed less frequently than others leading to load imbalance, it can be merged with other sub-trees. The simplest way to achieve this is to merge a sub-tree with the sub-tree that comes before it. In such a merge operation, there are three cases to consider; (1) when two sub-trees have the same height, (2) when the sub-tree with lower key values (T_l) has a higher level than the other sub-tree, and (3) when the sub-tree with higher key values (T_h) has a higher level than the other sub-tree.

When the two sub-trees to be merged have the same height, the entries of T_h's root are appended at the end of the entries of T_l's root. Since the entries of the root page have information about the pointers to the interior nodes, copying the entries of the root page is sufficient for this merge operation. In this case the complexity of the merge operation only depends on the number of entries in the root page of T_h. If the number of entries destined to the new root exceeds the page capacity, an SMO happens and a new root page is created, the same way a page split happens after a record insert.

When T_l is taller than T_h, T_l is traversed down to one level higher than the level of T_h. Then an entry is inserted at the right-most node of this level which points to T_h and has the key value equal to the starting key of the key range of T_h. Therefore, the complexity of the merge operation depends on the height difference between the two trees in this case.

When T_h is taller, the merge operation is very similar to the second case and the complexity is the same. T_h is traversed down to one level higher than the level of T_l and instead of the right-most node, the left-most node gets the entry which points to T_l and has the key value equal to the starting key of the key range of T_l.

After the delete operation, the partition table is updated according to the new key range and its corresponding sub-tree root page id.

A.3.2 Slicing sub-trees

A sub-tree can be split if it contains a hot key range. The slot for the starting key of the frequently accessed key range is searched in the sub-tree by traversing the tree pages from root to leaves. This slot can contain the B+tree entry which is greater than or equal to the start key. Once the slot is found, all the entries to the right of this slot in the slot's page are moved to newly created B+tree pages in a bottom-up manner like in a regular B+tree page split. The pages to the right of the slot's page do not need to be moved because the entries on the new pages will have pointers to them. Therefore, the complexity of the sub-tree split depends on the number of entries that come after the slot on slot's page and the height of the B+tree.

After the sub-tree splits, a new entry to the partition table is inserted with the new key range and the newly created sub-tree root page id.

A.4 Page cleaning

Page cleaning cannot be performed naively in PLP designs. Conventionally there is a set of page cleaning threads in the system that are triggered when the system needs to clean dirty pages (for example, when it needs to truncate log entries). Those threads may access arbitrary pages in the buffer pool, which breaks the invariant of PLP where a single thread can access a page at each point of time.

To handle the problem of page cleaning in PLP each thread does the page cleaning for its logical partition. Each logical partition has an additional input queue which is for system requests, and the page cleaning requests go to that queue.

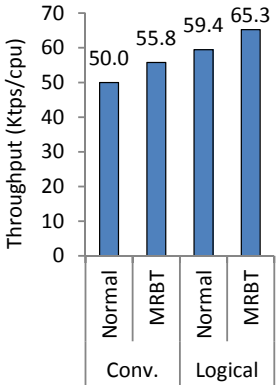


Figure 9: Performance of a conventional and a logically-partitioned system in TATP.

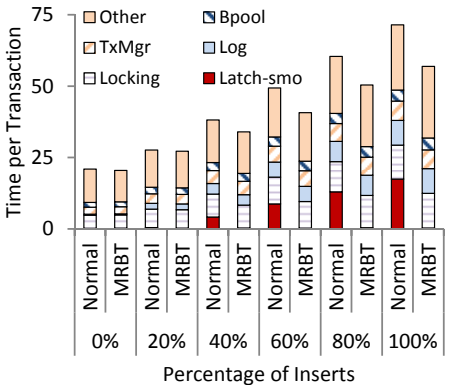


Figure 10: Time-breakdown of conventional transactions when parallel SMOs are allowed with MRBTrees.

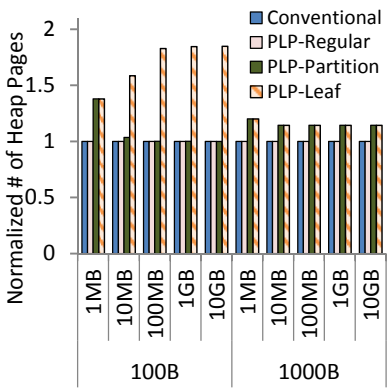


Figure 11: Space overhead of the PLP variations.

The system queue has higher priority than the queue of completed actions. Their execution won't be delayed by more than the execution time of one action (typically very short). In addition, because page cleaning is a read-only operation, the thread can continue to work (and even re-dirty pages) during the write-back I/O.

B. MRBTREES IN CONVENTIONAL OLTP

The MRBTree can improve performance even in the case of conventional systems in two ways. First, since it effectively reduces the height of the index by one level, each index probe accesses one fewer node and hence it is faster. In addition, any possible delay due to contention on the root index page is also reduced roughly proportionally with the number of sub-trees. Figure 9 highlights the difference in the peak performance of the conventional and the logically-partitioned system when they run with and without MRBTrees available. The workload is the TATP benchmark. The improvement in performance is in the order of 10%.

Second, each subtree can have a structure modification operation (SMO) in flight, whereas a traditional B+Tree can have one SMO in flight at any time. Hence, in workloads with high entry insertion (deletion) rates the MRBTree improves performance by parallelizing the SMOs. Figure 10 shows the time breakdown of the conventional system with and without MRBTrees as we run a microbenchmark which consists of either a record probe or insert, and we increase the percentage of inserts. Without the MRBTree, as the insertion rate increases the system spends an increasing amount of time blocked waiting for SMOs to complete, which is not the case when MRBTrees are used. In this case using MRBTrees improves performance by up to 25%.

C. REPARTITIONING ANALYSIS

In this section we model the cost of repartitioning for a Shared-Nothing system and the different PLP designs. The model calculates the number of records and index entries that have to be moved, the number of update/insert/delete operations on the indexes, the number of pointer updates on index pages and the routing page, and the remaining number of read operations that have to be performed.

For a subtree to be partitioned we define; h as the level of the tree, n as the number of entries in a B+tree node, m_i as the number of entries to be moved from the B+tree at level i, and M as the number of records that has to be moved. The read operations during a key value search in the B+tree is omitted since it is the same for all the systems (a binary search at each level from root to leaf).

C.1 Non-Clustered Indexes

In this section we assume that we have a unique non-clustered primary and a secondary index for a table and the data are partitioned based on the primary index key values.

In PLP-Regular, only some index entries have to be moved. Let's say there are m_1 entries that are greater than or equal to the starting key of the new partition on the leaf page that the slot for this starting key is found. All needs to be done is to move these m_1 entries on that leaf page to a newly created B+tree page and this procedure has to be repeated as the tree is traversed from this leaf page to the root. It is not necessary to move any entry from the pages that keep the key values greater than the ones in the leaf page containing the starting key. Setting the previous/next pointers of the pages at the boundaries of the old and new partitions is sufficient. Finally, a new entry to the routing page should be added for the new partition. The overall cost is given in the first row of Table 2.

The cost model in Table 2 for PLP-Regular describes the worst case scenario for the system. If the starting key of the new partition is in one of the interior tree pages, there is no need to move any entries from the pages that are below this page because the moved entries from the interior page already have pointers to their corresponding child pages; resulting in fewer reads, updates, and moved entries.

In PLP-Leaf, the index structure related modification cost is the same as the one in the PLP-Regular case, but we also have to move the records that belong to the new partition now. The records that belong to the new partition are the records that are pointed by the m_1 leaf page entries that has to be moved to the new partition's subtree. Thus, in the worst case m_1 records have to be moved. Since this is a non-clustered index, we have to scan these m_1 entries to get the RIDs of the records to be moved so that we can spot which heap pages they reside. In the end, the new RIDs for the moved records have to be updated in the secondary index. Based on these, the cost for the PLP-Leaf design is given in the second row of Table 2.

Table 2: Repartitioning cost model.

System	#Records Moved (M)	Primary Index					Secondary Index
		#Entries Moved	#Reads	#Pages Read	#Pointer Updates	Changes	Changes
PLP-Regular	-	$\sum_{k=1}^{h} m_k$	-	-	$2 \times h + 1$	-	-
PLP-Leaf	m_1	$\sum_{k=1}^{h} m_k$	M	1	$2 \times h + 1$	M *updates*	M *updates*
PLP-Partition	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	$\sum_{k=1}^{h} m_k$	M	$1 + \frac{M-m_1}{n}$	$2 \times h + 1$	M *updates*	M *updates*
Shared-Nothing	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	-	M	$1 + \frac{M-m_1}{n}$	-	M *inserts* M *deletes*	M *inserts* M *deletes*
PLP (Clustered)	m_1	$\sum_{k=2}^{h} m_k$	-	-	$2 \times h + 1$	-	M *updates*
Shared Nothing (Clustered)	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	-	-	-	-	M *inserts* M *deletes*	M *inserts* M *deletes*

The cost model for PLP-Leaf, again, illustrates the worst case scenario. If the starting key of the new partition is found in one of the interior nodes, no record movement has to be done since there will be no leaf page splits and the constraint of having all heap pages pointed by only one leaf page is preserved. Moreover, even if the key is found on the leaf page, we might not have to move all the records that are specified by the model above. If the records on a heap page is only pointed by the new partition now, then these records can stay on that heap page.

One disadvantage of PLP-Leaf is that during a B+tree leaf split some records might have to be moved. This also adds the additional cost described for the PLP-Leaf repartitioning to a regular B+tree leaf split.

In PLP-Partition, the index structure related modification cost is the same as the one in the PLP-Regular case. However, there is a risk of moving many records because in the worst case we have to move all the records that belong to the new partition. This number is equal to the number of entries that are on the leaf pages of the new B+tree. These entries have to be scanned to get the RIDs of the records to be moved as in PLP-Leaf case, and the new RIDs have to be updated in the secondary index after the record movement. The cost model for PLP-Partition is given in the third row of Table 2.

In a Shared-Nothing system, the cost for the record movement takes place as in the worst case of the PLP-Partition design. In addition, instead of entry movements from the B+tree, for each record to be moved an insert (for the new partition) and a delete (for the old partition) operation have to be performed in both primary and secondary index structures. Moreover, this procedure has to be repeated for all the replicas of a partition. Therefore, the repartitioning cost for one replica in a Shared-Nothing system is given as in the fourth row of Table 2.

C.2 Clustered Indexes

Let's consider the case where we have a unique clustered primary index and a secondary index, and the data partitioning is done using the primary index key columns. In this setup, no heap file exists and the primary index keeps the actual data records rather than RIDs. Thus, the three PLP designs are equivalent.

In that case, the B+tree leaf page entry movement for PLP is the record movement, and the B+tree entry movement is only going to be from the B+tree levels greater than 1. The cost model is given in the fifth row of Table 2.

The cost model for the Shared-Nothing system is going to be similar to the non-clustered case. The only difference is that there is no need to scan the leaf page entries to get the RIDs of the records to be moved since the leaf pages have the actual records. Therefore, the cost model for a replica is given as it is in the last row of Table 2.

C.3 Moving fewer records

With some additional information we can actually move fewer data during a repartitioning operation with the increased cost of number of reads. For example, in PLP-Partition design instead of directly moving all the records that belongs to a new partition, we can scan all the B+tree leaf pages that is going to be split and collect information on all the records. With the information we collected on the records, we can compute whether a heap page has more records that belong to the old partition or the new partition. If the heap page has more records that belong to the new partition, moving the records that belong to the old partition might be more convenient in this case.

The number of reads during the scan of the leaf pages can easily become a bottleneck due to the number of I/O operations that has to be performed in a disk resident database. However, in an in-memory database or a system that uses flash storage devices, this bottleneck can be prevented [3] and the above mentioned technique can reduce the number of data that has to be moved during repartitioning. For a Shared-Nothing system, this technique cannot be applied because either the records that belong to the new partition or the old one has to be moved to somewhere else all together since the partitions do not share the same storage space. In our experiments, since we used an in-memory database, we performed the repartitioning with this technique.

D. FRAGMENTATION COST

The last two PLP variations, PLP-Partition and PLP-Leaf, create some fragmentation on the heap file since they change the regular heap file structure (Section 3.3). This section evaluates their fragmentation cost based on two things; the number of heap pages each design uses and the amount of time required to scan the whole heap pages.

Figure 11 shows how the ratio between the number of pages used in a PLP design and in the conventional system changes as we increase the database size. The x-axis shows the total size of the database when each record is 100B (left side of the graph) and 1000B (right side of the graph). The y-axis is the ratio between the number of pages used in each design and the conventional system. The conventional system has one partition where the PLP variations have 100 and 10 partitions for the cases where record size is 100B and 1000B, respectively. The heap page size is 8KB.

As expected, PLP-Regular does not create any fragmentation since it maintains the regular heap file format. For PLP-Partition, the amount of fragmentation becomes negligible as the database size increases for small records. How-

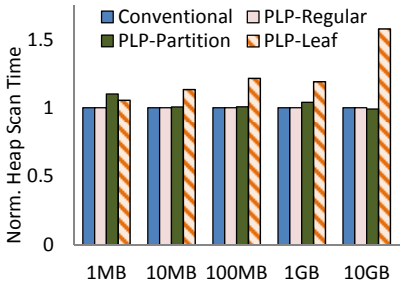


Figure 12: Overhead of PLP variations in file scan.

ever, PLP-Leaf uses up to 80% more heap pages than a Conventional system for the same case creating a visible fragmentation on the heap file. On the other hand, as we increase the record size, the fragmentation cost decreases because since each heap page is able to keep fewer records in this case the amount of empty space left on each heap page is reduced.

Figure 12 shows the time to scan the heap file for each PLP variation compared to the conventional system as we increase the size of the database. The setup is same as in Figure 11 when the record size is 100B. The size of the buffer pool is 4GB for each measurement. From Figure 12, the fragmentation cost of PLP-Leaf does not significantly increase the file scan time when there are no I/O operations performed (from 1MB to 1GB) because the total number of records that are scanned is the same. However, in the larger database case (10GB), PLP-Leaf increases the heap file scan time by 60% since there are more I/O requests.

Overall, among the PLP variations, only the PLP-Leaf design can introduce some significant fragmentation cost when a heap page can keep many database records. However, as the number of records a heap page can keep decreases, this cost becomes less significant.

E. WEAKNESSES

While we cannot find weaknesses in the MRBTree access method, the PLP design has some, most of them coming from its ancestor, data-oriented execution [23].

First of all, this system is designed for high performance transaction processing which imposes great pressure on the internal of the database storage layer. Thus, certain classes of applications may not benefit from it, or even get penalized. For example, for our evaluation we use the specialized `TATP` and `TPC-B` benchmarks instead of the more popular `TPC-C`. The reason for that is that our baseline systems (conventional and logically-partitioned) did not encounter any of the issues we try to address in `TPC-C` and there was very little room for improvement. Another example, are business intelligence applications with large file scans or joins. In such workloads PLP may penalize performance since it may require the transfer of large volumes of data between the participating threads. It is common practice, however, to employ dedicated database engines (usually column-stores) for processing such workloads.

Since PLP is using some kind of partitioning it is amenable to problems due to skew on the requests or data. As we have already argued, those problems are less troublesome compared with the case of shared-nothing systems since the MRBTrees provide the machinery for quick repartitioning. However, the fact remains that some threads may be assigned more work than they can handle becoming bottlenecks. We currently investigate techniques to rapidly detect and efficiently handle problems due to load imbalance.

PLP partitions each table using range-based partitioning to the keys of a specific subset of the columns of the table. The DBA, however, may have decided to build indexes (usually non-clustered secondary indexes) that do not contain the columns which PLP uses for the partitioning. We refer to such indexes as *non-partitioning aligned indexes* and they may become performance bottlenecks. In data-oriented execution and PLP we handle such accesses by appending each index leaf entry with the fields of the record that are needed for identifying the partition-owning thread. The non-partitioning aligned index is accessed as a conventional index, without avoiding any locking or latching, in order to retrieve the id of the record to be accessed in the heap file and then the access is passed to the appropriate thread. As a proactive measure, we have implemented tools that help the application developer and the DBA to avoid having workloads with very frequent such index accesses [30].

F. PLP AND FUTURE HARDWARE

Conventional OLTP is ill-suited to modern and upcoming hardware for at least three reasons; (a) The code of OLTP system is full of unscalable critical sections [13], (b) The access patterns are unpredictable [24] that even the most advanced prefetchers fail to detect [31], (c) The majority of the accesses are shared read-write and hence they underperform on caches with non-uniform access latency [27, 28].

As we have seen, PLP, combined with previous advances in logging, eliminates all three problems. The majority of unscalable critical sections are completely eliminated, access patterns are regularized by the thread assignments, and threads no longer share data to communicate, eliminating the shared R/W problem.

This regularity will become increasingly important as hardware continues to make more and more demands of the software. For example, it is almost inevitable that processor cache access latencies will be non-uniform [27, 29, 28]. Unfortunately, OLTP will only be able to utilize effectively these new architectures if it can eliminate the majority of accesses which are shared among multiple processors.

In short, by eliminating a large class of communication, PLP leaves OLTP engines much better-poised to take advantage of upcoming hardware, whatever form it may take.

G. REFERENCES

[27] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO*, 319–330, 2004.

[28] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 184–195, 2009.

[29] N. Hardavellas, I. Pandis, R. F. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, 79–87, 2007.

[30] I. Pandis, P. Tözün, M. Branco, D. Karampinas, D. Porobic, R. Johnson, and A. Ailamaki. A data-oriented transaction execution engine and supporting tools. In *SIGMOD*, 1237–1240, 2011.

[31] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA*, 69–80, 2009.