In-Memory Performance for Big Data

Goetz Graefe
HP Labs Palo Alto
goetz.graefe@hp.com

Haris Volos
HP Labs Palo Alto
haris.volos@hp.com

Hideaki Kimura
HP Labs Palo Alto
hideaki.kimura@hp.com

Harumi Kuno
HP Labs Palo Alto
harumi.kuno@hp.com

Joseph Tucek
HP Labs Palo Alto
joseph.tucek@hp.com

Mark Lillibridge
HP Labs Palo Alto
mark.lillibridge@hp.com

Alistair Veitch[*]
Google
alistair.veitch@gmail.com

ABSTRACT

When a working set fits into memory, the overhead imposed by the buffer pool renders traditional databases non-competitive with in-memory designs that sacrifice the benefits of a buffer pool. However, despite the large memory available with modern hardware, data skew, shifting workloads, and complex mixed workloads make it difficult to guarantee that a working set will fit in memory. Hence, some recent work has focused on enabling in-memory databases to protect performance when the working data set *almost* fits in memory. Contrary to those prior efforts, we enable buffer pool designs to match in-memory performance while supporting the "big data" workloads that continue to require secondary storage, thus providing the best of both worlds. We introduce here a novel buffer pool design that adapts pointer swizzling for references between system objects (as opposed to application objects), and uses it to practically eliminate buffer pool overheads for memory-resident data. Our implementation and experimental evaluation demonstrate that we achieve graceful performance degradation when the working set grows to exceed the buffer pool size, and graceful improvement when the working set shrinks towards and below the memory and buffer pool sizes.

1. INTRODUCTION

Database systems that use a buffer pool to hold in-memory copies of the working data enjoy a number of benefits. They can very efficiently manage working sets that far exceed available memory. They offer natural support for write-ahead logging. They are somewhat insulated from cache coherence issues. However, a buffer pool imposes a layer of indirection that incurs a significant performance cost. In particular, page identifiers are required to locate the page regardless of whether it is in memory or on disk. This in turn calls for a mapping between in-memory addresses of pages and their identifiers.

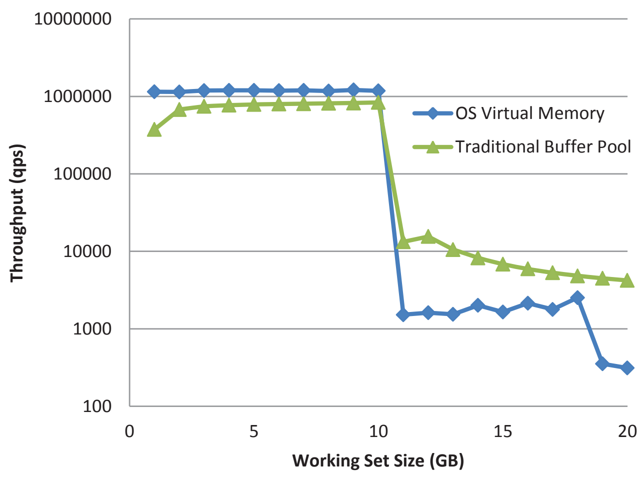


Figure 1: Index lookup performance with in-memory vs. traditional database.

Figure 1 illustrates the impact of the buffer pool's overheads, comparing the performance of an in-memory database using direct pointers between B-tree nodes (marked "Main Memory") and a database system using a traditional buffer pool with disk pages (marked "Traditional Buffer Pool"). The available memory is 10 GB; further details of this experiment are provided in Section 5. In the left half, the entire working set fits into the available physical memory and buffer pool. In the right half, the working set exceeds the available memory and spills to the disk. The in-memory database relies on the OS virtual memory mechanism to support working sets larger than the available physical memory. The in-memory database permits substantially faster index look-ups than the traditional buffer pool when the working set fits in physical memory. In fact, our experiments show

[*]Work done while at HP Labs.

Proceedings of the VLDB Endowment, Vol. 8, No. 1
Copyright 2014 VLDB Endowment 2150-8097/14/09.

that an in-memory database improves throughput by between 41% to 68% compared to one that uses a traditional buffer pool. This correlates well with earlier results that the buffer pool takes about 30% of the execution cycles in traditional transaction processing databases [14].

We borrow the technique of pointer swizzling and adapt it to the context of page-oriented data stores, selectively replacing identifiers of pages persisted in storage with direct references to the memory locations of the page images. Our techniques enable a data store that uses a buffer pool to achieve practically the same performance as an in-memory database. The research contributions here thus include:

1. Introduction of pointer swizzling in database buffer pool management.
2. Specification and analysis of swizzling and un-swizzling techniques for the specific case of B-tree indexes.
3. Victim selection (replacement policy) guiding eviction and un-swizzling of B-tree pages from the buffer pool.
4. Experimental proof that these simple techniques enable a database with a buffer pool to achieve in-memory database performance when the working set fits in memory, and furthermore achieve a continuum with graceful degradation as the working set size exceeds the memory size.

In the remainder of this paper, we first discuss prior efforts to improve the performance of in-memory database systems with regard to memory limitations as well as prior work in pointer swizzling (Section 2). We next describe the problem being solved (Section 3), and then present details of our techniques for swizzling and un-swizzling within a database buffer pool (Section 4). We evaluate the performance of our solution using an implementation (Section 5) and finally present our conclusions from the work to-date (Section 6).

2. RELATED WORK

This paper describes our efforts to adapt the known technique of pointer swizzling to the new context of making disk-based systems competitive with in-memory database systems. This section begins by discussing prior work that addresses the problem of how to preserve the performance of in-memory database systems when the working set does not quite fit in the available amount of memory. These prior systems do not consider how to match the performance of a traditional database when disk I/O is the bottleneck. Our design, on the other hand, enables a traditional database system that has been optimized for disk I/O to compete with in-memory systems when the working data set fits in-memory. Thus, the second half of this section discusses prior work on pointer swizzling, which is the technique we leverage to eliminate buffer pool overheads.

2.1 In-Memory Database Systems

A number of in-memory database systems, including Oracle's TimesTen, Microsoft's SQL Server Hekaton, CWI's MonetDB, SAP HANA, MIT's Silo, and VoltDB, avoid the overheads of disk-based systems by focusing on providing the best possible performance for in-memory workloads [5, 13, 28, 31, 38, 39]. However, both Oracle's and Microsoft's in-memory database systems force a database administrator to make explicit tuning decisions and requests, whereas our approach is entirely automatic and it adapts to cases of growing and shrinking working sets. As the raison d'etre of such systems is to provide superior performance for in-memory workloads, most of these do not support data sets that do not fit in memory.

For instance, Microsoft's SQL Server Hekaton stores raw memory addresses as pointers, eliminating the need for data pages and indirection by page IDs. However, once the working set size exceeds the available amount of memory, Hekaton simply stops inserting rows [1].

One possible solution is to rely on the VM layer to deal with any spills out of memory. For instance, some databases (such as Tokyo Cabinet) simply `mmap` their backing store [6]. This leads to several problems. The first is that the OS VM layer is particularly poor at making eviction decisions for transactional workloads (as we show later in our experiments). The second is that the operating system considers itself free to flush dirty pages opportunistically, without notifying the application. This can cause data integrity problems when a page dirtied by an in-flight transaction is written back without the matching log records.

Failure atomic msync [35] provides a mechanism by which 1) the kernel is prevented from lazily flushing pages, and 2) a modification to the `msync` call allows users to select a subset of pages to be written out atomically. Although this certainly helps, it is insufficient for a transaction processing system. First, failure atomic msync does not provide parallelism. From contact with the authors, failure atomic msync "only gives A and D . . . we provide no support for isolation. If you want safe, concurrent, database-style transactions, you need more" [34]. Second, as implemented, failure atomic msync hijacks the file system journaling mechanism, duplicating each page write. Unlike a traditional buffer pool based system where changing a single row causes a few bytes of log writes and a page-sized in-place update, a system based on failure atomic msync incurs a minimum of two page-sized writes, a significant performance penalty.

There are a small number of recent efforts that make VM-based in-memory databases performant when the working data set almost fits in memory. These systems address the problem of swapping by either explicitly moving some data records to virtual memory pages that the OS can then swap to disk more intelligently or else by explicitly compressing less-frequently accessed data records so as to reduce the space consumed, but also reducing access speed.

Stoica and Ailamaki profiled the performance of the state-of-the-art in-memory VoltDB database, and found that performance dropped precipitously when the working data set exceeded available memory [37]. Their work demonstrated that by intelligently controlling which tuples are swapped out by the operating system and which are kept in memory, it is possible to make much more effective use of available memory. To achieve this, two regions of memory are created – one hot and one cold. Pages in the hot region are pinned in memory so that the virtual memory manager will not evict them; pages in the cold region are not pinned. Offline log analysis identifies individual data records that are likely no longer needed, after which a special process explicitly moves those records from the hot area to the cold data area. For this offline analysis, they leveraged the work of Levandoski et al., who investigated how to efficiently identify hot and cold data in the context of a database system optimized for main memory [25]. After finding that maintaining access statistics incurred a 25% overhead, Levandoski et al. developed efficient algorithms by which they could analyze 1M log records in under a second.

DeBrabant et al. also consider how to evict individual cold tuples from memory to disk [4]. Evicted tuples are stored in an on-disk hash table (called a Block table) and tracked by a separate in-memory hash table. Because they assume that main-memory is the primary storage device (data is initially created directly in memory), they call their technique *anti-caching*. Like Stoica and Ailamaki, DeBrabant et al. implemented their technique using VoltDB. However, considering offline analysis to be a prohibitive expense yet wanting to make eviction decisions on a tuple-by-tuple basis, DeBrabrant et al. use a doubly-linked LRU chain of tuples to identify "hot" vs. "cold" tuples. In order to reduce the overhead of this tracking, they select only a fraction of their transactions to monitor at runtime.

Funke et al. [Funke+2012] compact memory-resident data, identifying and compressing immutable "frozen" data so as to dedicate as much memory as possible to the hot mutable working data set [7]. This distinction between mutable and immutable data enables a single main memory database system, HyPer, to support both online transactional processing (OLTP) and online analytical processing (OLAP).

In summary, all of these prior systems focus on how to enable in-memory databases to handle the case where the working data set does not quite fit in-memory; they do not consider how to match the performance of a traditional database when disk I/O is the bottleneck. We, on the other hand, enable a traditional database system that has been optimized for disk I/O to compete with in-memory systems when the working data set fits, or almost fits, in-memory.

2.2 Pointer Swizzling

Pointer swizzling refers to replacing a reference to a persistent unique object identifier with a direct reference to the in-memory image of that object. Following a swizzled pointer avoids the need to access a mapping between the identifiers and memory addresses of memory-resident objects [29]. Our approach, detailed in Section 4, eliminates the buffer pool overhead by swizzling a parent page's reference to a child page, replacing the persistent identifier with the in-memory address of the child page.

Some relational databases, notably in-memory databases, use in-memory pointers. Most relational databases separate their persistent storage and their volatile buffer pool. Those systems have pages point to one another by means of addresses in persistent storage, typically page identifiers on disk storage or other page-access devices. What is used here for the first time, as far as we know, is a dynamic translation between addresses for database containers, i.e., pages, by using a memory-optimized address system (virtual memory addresses) while navigating in memory and using a disk-address system (page identifiers) on persistent storage.

Pointer swizzling is well known in the context of application objects in specialized databases but we are not aware of any relational database management system employing it. E.g., Garcia-Molina et al. [8] anticipate the opportunity for pointer swizzling when they observe that "index structures are composed of blocks that usually have pointers within them. Thus, we need to study the management of pointers as blocks are moved between main and secondary memory." However, the book's discussion focuses on pointers between application objects, not storage containers such as pages. Similarly, White and DeWitt's [40] "QuickStore provides fast access to in-memory objects by allowing application programs to access objects via normal virtual memory pointers." But again, pointers between system objects such as frames in the buffer pool are not considered.

The general technique of pointer swizzling first emerged about 30 years ago [2, 19]. We are informed by the thoughtful and in-depth discussion and evaluation of pointer swizzling techniques by Kemper and Kossman [20, 21, 22]. They characterize pointer swizzling techniques according to two key design decisions: (1) when to swizzle references, balancing the cost of swizzling references that might never be actually used versus the performance benefits of having references swizzled in advance of being read, and (2) how to handle references to objects that are not resident in main memory, balancing the cost and complexity of maintaining reference relationship dependencies (e.g., in a reverse reference table) versus the cost of managing swizzled references to non-memory resident objects.

A significant, yet subtle, difference between our work and prior work in swizzling is that although prior work has considered swizzling at the granularity of B-tree pages and segments, the swizzling was only performed on object IDs (references between application objects). In contrast, we restrict swizzling to references between system objects — the in-memory images of pages in the buffer pool.

For example, Kemper and Kossman briefly mention swizzling pointers to pages [22]: "In some systems, address translation is carried out on a per-page basis. ... a page table records the in-memory address of every resident page." However, they consider this only in the context of object-oriented databases, as opposed to relational database management systems or key-value stores, i.e., for any storage layer. Moreover, their purpose is application performance (application logic running on top of some in-memory store or buffer pool) whereas our purpose is indexing performance. Finally, our system still employs indirection (via the descriptor data structures in the buffer pool) such that dirty bit and other temporary metadata can be supported, but this is indirection using in-memory addresses (pointers) rather than the persistent page identifiers that require lookup in the buffer pool's table of contents. Their notion of "indirect swizzling" is very different – the purpose is to permit evicting an object from memory without un-swizzling all pointers to this object. In other words, the design requires that some object remains in memory (and consumes memory) long after an object has been evicted due to memory contention. They suggest reference counting to control the lifetime of the indirection object as maintenance of a "reverse reference list" is too expensive.

3. PROBLEM DESCRIPTION

The context of our work is a transactional storage manager that adapts traditional tools to provide ACID guarantees on modern hardware. Data resides in B-tree structures that store all data items in their leaves, with separator keys in the upper B-tree levels serving only to guide searches [10]. A buffer pool holds in-memory copies of pages used by the working data set. Latching and locking respectively ensure mutual isolation of concurrent operations to physical data structures and logical database contents [9]. Write-ahead logging provides "all or nothing" failure atomicity, database consistency, and transactional durability [11, 27].

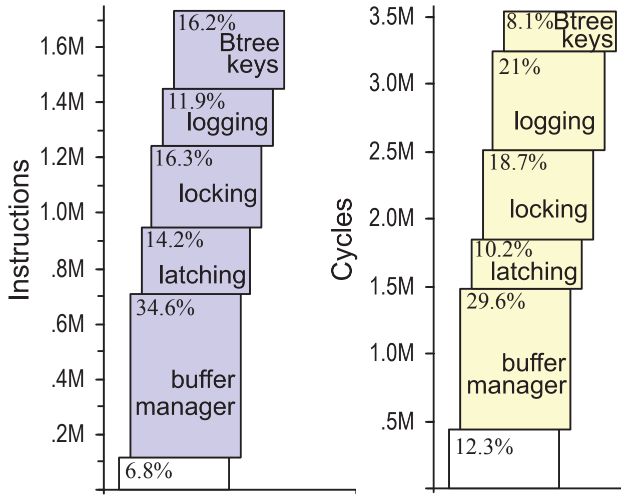


Figure 2: Module costs in transaction processing, copied from Harizopoulos et al. [14].

The challenge is that when the working data set fits in memory, these traditional tools themselves become the new bottlenecks, accounting for about 80% of the CPU cycles used by a single transaction [14]. Figure 2 (copied from [14]) summarizes the execution costs of a TPC-C "new order" transaction in a traditional database system with the entire database in memory, expressed as a fraction of the total execution effort and calculated both in terms of instructions and cycles (time). A number of observations suggest themselves. For example, the relatively large number of instructions executed in the B-tree code compared to the relatively small time spent there indicates well-written and well-executed code with spatial and temporal locality, with the opposite effects in the logging code. However, it is readily apparent that the buffer manager is the most significant of the five bottlenecks shown, consuming about 30% of cycles and 35% of instructions. A naïve calculation indicates that total (or near-total) elimination of the buffer pool from the execution path should improve transaction throughput (performance, scalability) by a factor of about 1.5. Figure 1 suggests performance opportunities of the same magnitude. This is the principal motivation of our present work.

Note that Harizopoulos et al. advocates a complete redesign of the transactional storage manager, as opposed to improvements in any one of the big overheads, with the reasoning that eliminating any one of the overheads would improve performance only a little bit because the others would remain. We, on the other hand, are working to eliminate all of these overheads. In particular, this paper describes our effort to eliminate buffer pool overheads without limiting the resulting system to small databases that fit in memory.

3.1 Buffer pool and page identifiers

The buffer pool caches images of pages in memory while they are being read or modified. Each formatted database page has a unique persistent identifier which the storage manager can use to locate that page and bring it into the buffer pool if it is not already there. For example, when a B-tree parent page associates a given separator key with a child page, that reference is stored in persistent storage using the persistent page identifier.

Regardless of whether or not the working data set fits in available memory, the buffer pool is a useful tool for ensuring correctness and managing page consistency. For example, write-ahead logging requires that a modified database page must not be written (in place) until the modifications are logged on stable storage. The buffer pool allows page modifications to be made in memory as opposed to directly on storage. The buffer pool also enables check-pointing all dirty (modified but not persisted) pages to a backing store.

However, managing and protecting the mappings between the persistent page identifiers and in-memory page images can cause a significant amount of latch contention as the number of pages that can be stored in memory and the number of processes that will access those mappings increase. For example, consider a typical buffer manager that uses a hash table to implement the mapping. Simply traversing a parent page's pointer to a child page requires calculating the child page's hash key, protecting the mapping hash table, performing the hash table look-up, searching the hash bucket's linked list, etc. Furthermore, this operation is likely to incur CPU cache misses.

3.2 Virtual memory

Virtual memory might seem like a natural alternative approach, i.e., mapping the entire database into virtual memory and using each page's virtual memory address as a page identifier. Unmapping and closing the file would persist the database to the backing store. This would bypass the indirection of purely logical page identifiers and furthermore delegate the problem of controlling which pages to swap out of memory to hardware and the operating system in the case that the working data set grows larger than would fit in memory. Unfortunately, such an approach is unacceptable in terms of correctness and performance.

With respect to correctness, the problem is durability and control over writes to the backing store. Write-ahead logging requires that a modified database page must not be written until the modifications (relevant log records) have been logged on stable storage, and virtual memory might write a database page too early. Similarly, virtual memory may write a database page too late – e.g., in many implementations of database check-pointing, a checkpoint is not complete until all dirty pages have been written to the backing store. Finally, if the latency-optimized logging space is limited and requires periodic recycling, log records must not be recycled until all database changes have been persisted.

Some operating systems provide mechanisms to control physical I/O to memory-mapped files, e.g., POSIX `msync`, `mlock`, and elated system calls. Unfortunately, there are no mechanisms for asynchronous read-ahead and for writing multiple pages concurrently, i.e., multiple `msync` calls execute serially. As observed by [37], without extensive explicit control, the virtual memory manager may swap out hot data along with the cold. Our experiments in Section 5 demonstrate this performance penalty.

4. NEW TECHNIQUES

With the goal of eliminating the overhead of the buffer pool, we look to pointer swizzling, a technique used by object stores to avoid the cost of repeatedly translating between distinct address spaces when referencing in-memory objects [29, 41]. That is, we can remove the layer of indirection by translating page identifiers into memory addresses when a page is brought into the buffer pool.

For example, consider a conventional B-tree index that completely fits in memory, and a parent page that contains a pointer to a child page. The images of both pages are cached in memory as frames in the buffer pool. A search operation reads the contents of the parent page and finds the identifier of the child page. Locating the in-memory image of the child page requires a look up in a data structure that maps between page identifiers and in-memory page addresses. Furthermore, since pages will potentially move into and out of memory if the working set exceeds the buffer pool size, this data structure must be protected against concurrent accesses (e.g., using one or more latches).

Swizzling in the context of an object storage system swizzles references between application-level container objects, while our design applies swizzling to internal data structures (system objects) independent of any choices made for application-level objects and data structures. Swizzling page references, as opposed to arbitrary application object references, greatly limits the number of swizzled references that must be tracked and maintained, for example confining targets for swizzling to the interior nodes of a B-tree as opposed to potentially impacting every object in an object base.

Although confining swizzling to system objects simplifies problems such as how to keep track of which references have been swizzled so that they can be un-swizzled, it also raises new issues. In order to adapt the concept of swizzling for use within a database buffer pool, the database storage manager must coordinate swizzling, un-swizzling, and page eviction from the buffer pool.

4.1 Swizzling child pointers

Since most B-tree operations rely on root-to-leaf passes, parent-to-child pointers between B-tree pages are the most important pointers to swizzle (as those are the most frequently used). Ideally, the pointer to the root node, e.g., in the database catalogs, can also be swizzled, in particular if an in-memory cache holds the catalogs while the database is open and the index root page is in the buffer pool.

In a traditional database buffer pool, multiple cursors may read from the same page at the same time. A pin count, similar to a reference count, tracks the number of concurrent users of a buffer pool frame as well as swizzled parent-to-child pointers. A page in the buffer pool can be evicted and replaced only when its pin count is zero.

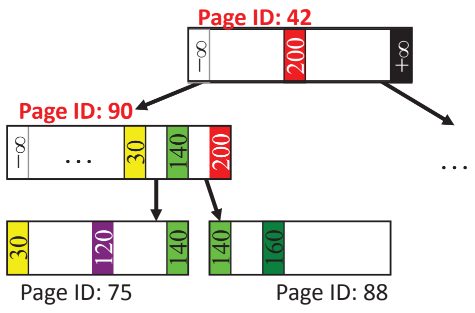


Figure 3: Parent-to-child pointers in a B-tree.

Figure 3 shows a simple B-tree with four nodes that we will use as a running example in the remainder of this section; we focus on the two nodes labeled in red – one representing the root node (page id 42) and the other representing a child node (page id 90). Page 90 is associated with the key range from $-\infty$ to 200.

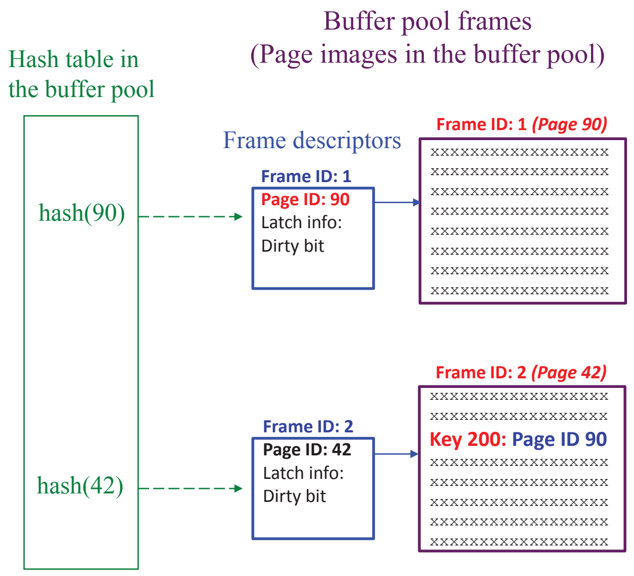


Figure 4: Parent-child relationship in a traditional buffer pool.

Figure 4 shows in-memory page images corresponding to the pages with ID's 42 and 90, realized using a traditional buffer pool. The buffer pool has a limited number of frames in which to store in-memory images of pages. Each frame has a descriptor that contains metadata about the page frame. A hash table maps from page identifiers to frame identifiers (e.g., offsets into arrays of frames and frame descriptors). In a traditional implementation, a page-to-page pointer is realized using the target page's identifier.

Figure 5 is a flow chart that sketches the process of searching a given in-memory page image (the contents of the buffer pool frame ID 1 from our example) for a search key, finding a parent-to-child pointer (pointing to page 90), and then using the child page's identifier to retrieve its in-memory page image. Following the process shown in Figure 5, the traversal from the root node (page ID 42) to the node with page ID 90 requires a hash calculation and traversal of a linked list representing a hash bucket to reach a descriptor for a buffer frame, which has a direct pointer to the buffer frame (and its contents).

In contrast, Figure 6 sketches the process of searching a page in an in-memory database that does not have a buffer pool. Because all pages are in-memory, traversing parent-child relationship can be done without indirection.

When a pointer from a B-tree parent page to a child page is swizzled, we modify the in-memory page image of the parent page, replacing the Page ID of the child page with a pointer to the buffer pool frame holding the in-memory image of the child page. This direct pointer can be realized using a virtual memory address, but in our reference implementation (and the illustration) we simply use the identifier of the buffer pool frame. For example, Figure 7 shows the

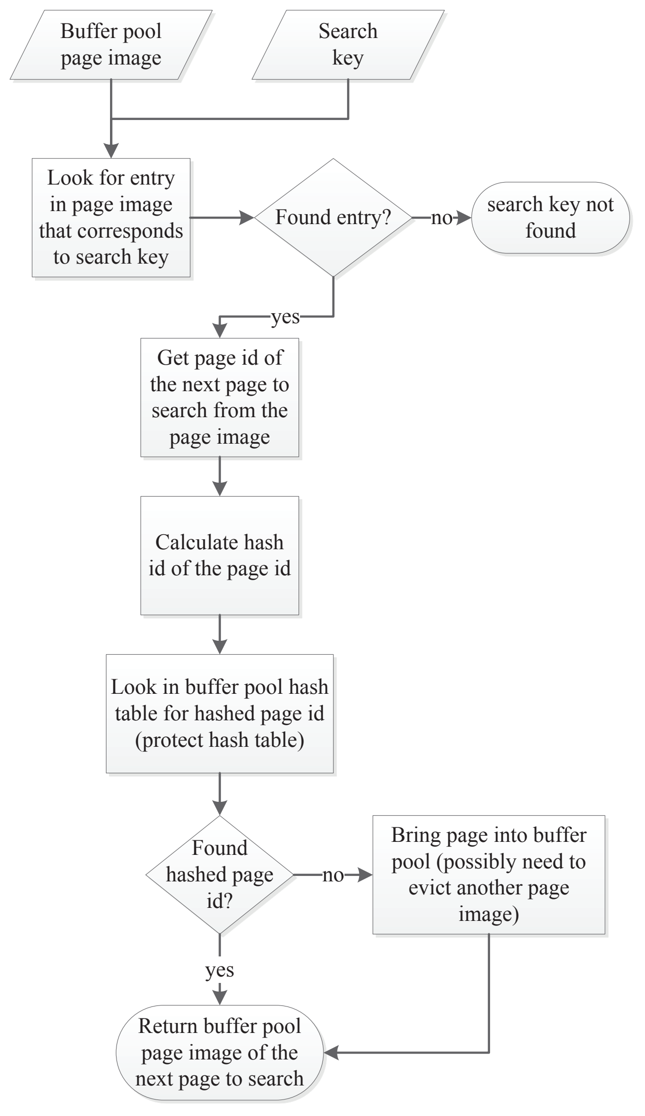

Buffer pool page image
Search key
Look for entry in page image that corresponds to search key
Found entry?
no
search key not found
yes
Get page id of the next page to search from the page image
Calculate hash id of the page id
Look in buffer pool hash table for hashed page id (protect hash table)
Found hashed page id?
no
Bring page into buffer pool (possibly need to evict another page image)
yes
Return buffer pool page image of the next page to search

Figure 5: Following a page pointer in a traditional buffer pool.

contents of the page image in the buffer frame with id 2 after the reference to page 90 has been swizzled.

Figure 8, in contrast to Figure 5 and Figure 6, sketches the process of traversing a swizzled pointer. In our design, pointer swizzling in the buffer pool is a side effect of normal B-tree operations and as such is transparent to applications. For example, when a root-to-leaf pass encounters a fault in the buffer pool, it loads the missing page and swizzles the pointer to the page. When a root-to-leaf pass encounters a hit in the buffer pool with an un-swizzled pointer, i.e., a traditional child reference using a page identifier, the pointer is swizzled to speed up future root-to-leaf passes within the pertinent key range.

In our current design, pointers are swizzled one at a time. Within a single parent page, some pointers may be swizzled while some are not; it is possible that some child pointers are swizzled while other child pages are not even present in the buffer pool. An earlier design that swizzled either

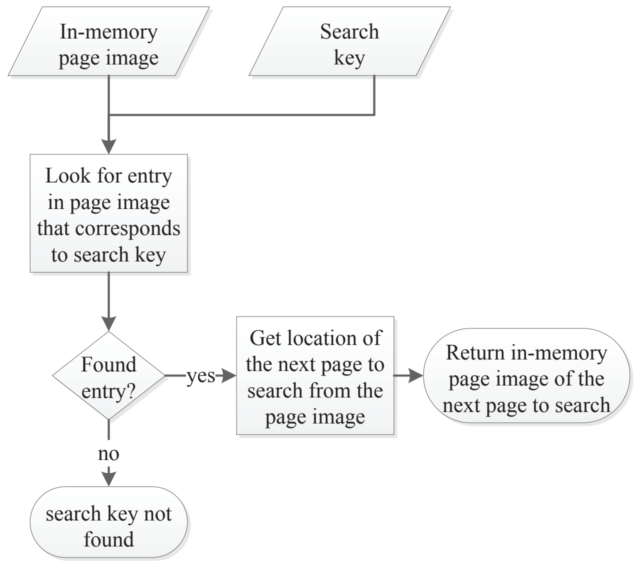

In-memory page image
Search key
Look for entry in page image that corresponds to search key
Found entry?
yes
Get location of the next page to search from the page image
Return in-memory page image of the next page to search
no
search key not found

Figure 6: Following a page pointer in an in-memory database.

all or none of the child pointers in a parent page simplified some bookkeeping and encouraged larger I/O operations (assuming good clustering of child pages) but either prevented speeding up search in specific key ranges or incurred excessive space requirements in the buffer pool.

By swizzling during root-to-leaf traversals, swizzling proceeds from the root towards the (active) leaves. Un-swizzling proceeds in the opposite direction. Since a node's parent is at least as active as the node itself, swizzled pointers occur only in parent nodes that are themselves swizzled, i.e., the pointer from grandparent to parent node is swizzled.

4.2 Buffer pool eviction

Pagewise-eager swizzling in object-oriented databases scans through a page and swizzles all the references to objects contained in the page at page-fault time [15, 22]. However, handling replacement in the buffer pool posed a major challenge, requiring expensive tracking of swizzled object references using data structures such as reverse reference lists [22], persisted reverse pointers [21], swizzle tables [26], and indirect swizzling descriptors, etc. [22].

Our design simplifies the challenge of how to handle swizzled references when objects are evicted from memory. When the buffer pool needs to evict a page, our implementation uses a fairly standard implementation of generalized clock counter [30, 36].

Swizzling pins the affected pages and thus protects them from replacement by the clock mechanism. Thus, another mechanism is needed that un-swizzles pages when the clock mechanism cannot find possible replacement victims. Our current design sweeps B-tree structures in the buffer pool using depth-first search. Each sweep resumes where the last one ended, retained using a key value, and frees some pages. This design requires detailed knowledge of the B-tree data structure within the buffer pool manager or appropriate callbacks into the B-tree module. Pages with no recent usage as indicated by the g-clock counter are un-swizzled unless the page itself is a branch page containing swizzled parent-to-

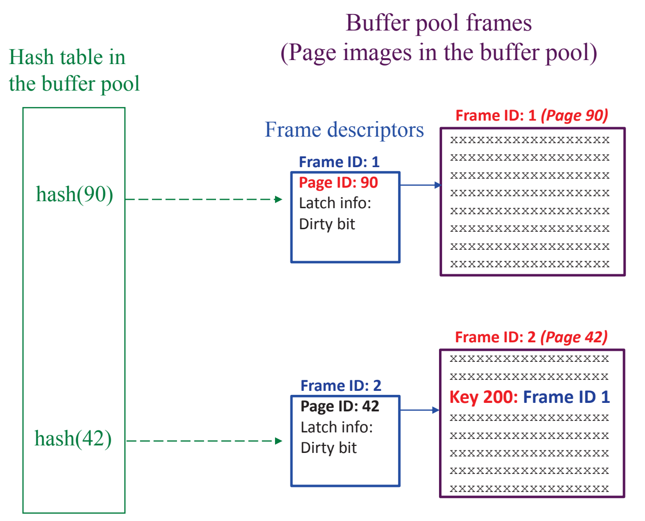


Figure 7: Swizzled parent-child relationship in a buffer pool.

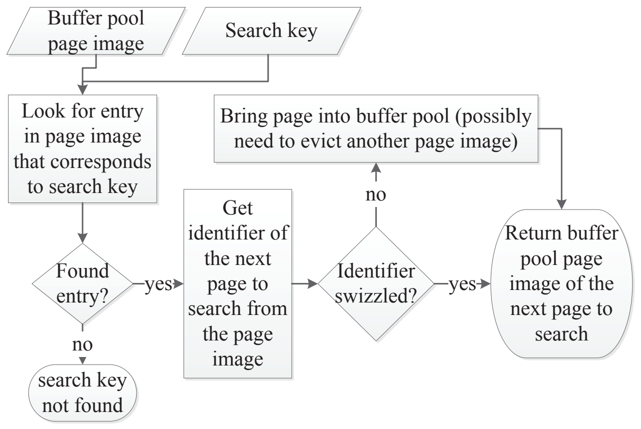

Buffer pool page image
Search key
Look for entry in page image that corresponds to search key
Bring page into buffer pool (possibly need to evict another page image)
Found entry?
yes
no
search key not found
Get identifier of the next page to search from the page image
Identifier swizzled?
no
yes
Return buffer pool page image of the next page to search

Figure 8: Swizzled pointer traversal.

child pointers. Thus, just as swizzling proceeds root-to-leaf, un-swizzling proceeds leaf-to-root.

Current work strives for a cleaner separation of buffer pool and B-tree modules; future work may produce a tighter integration of the two mechanisms, i.e., g-clock for eviction and hierarchical sweep for un-swizzling.

4.3 Child-to-parent pointers

Child-to-parent pointers are generally considered a bad idea due to the complexity and effort required when splitting non-leaf nodes. Thus, to the best of our knowledge, no database system has used persistent parent pointers since one cited by Küspert [23] in 1985.

While this is true for disk-based databases, in-memory databases require reconsideration, as does a database with swizzled pointers in the buffer pool. Specifically, the child-to-parent pointers can reside in and link descriptors in the buffer pool rather than page images. Moreover, if child-to-parent pointers exist only where the matching parent-to-child pointer is swizzled, maintenance while splitting a non-leaf node is quite efficient.

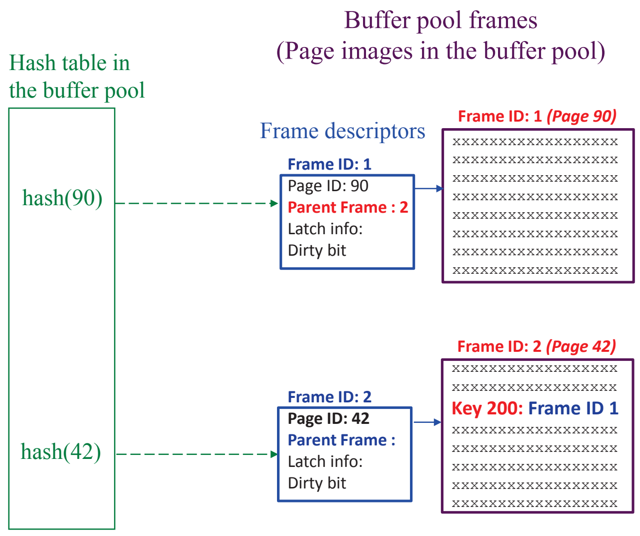


Figure 9: Child-to-parent pointers in the frame descriptors.

Figure 9 sketches a child-to-parent pointer added to the frame descriptors from Figure 7. The child-to-parent pointer speeds un-swizzling when a page is chosen for replacement (eviction) in the buffer pool. In that case, it is necessary to determine efficiently whether that page is a B-tree node and whether the parent node holds a swizzled pointer to the evicted page. For that reason alone, it seems that a parent pointer may be a good idea.

Note that the code used for the performance evaluation in Section 5 does not include child-to-parent pointers. Nonetheless, as seen in Figure 10, it performs similarly to both the traditional B-tree as well as the one in which all pointers are virtual memory addresses, even when the workload spills from memory and the swizzling solution needs to pay the overhead of unswizzling.

5. EVALUATION

We have implemented a prototype of the swizzling and unswizzling of page identifiers in the context of the Shore-MT experimental database system [3, 16, 32, 33]. Our hypothesis is that swizzling B-tree page identifiers alleviates the performance overhead imposed by the buffer pool and allows a traditional database to match the performance of an in-memory system that has no buffer pool. In this section we present experiments that compare the performance of three system configurations: a baseline configuration with a traditional buffer pool, an in-memory database (without a buffer pool), and a buffer pool that swizzles page identifiers. Our experiments consider (1) whether swizzling eliminates the performance overhead imposed by a buffer pool, compared to an in-memory database; (2) whether performance degrades gracefully as the working data set exceeds the size of memory, and (3) whether the swizzling approach can adapt to a drifting workload without requiring offload log analysis or explicit tracking of record-accesses.

5.1 Prototype Implementation

Aside from the swizzling techniques in the buffer pool module, we also applied several modern optimizations for many-cores to make our prototype a more appropriate test bed to evaluate the effect of swizzling. Overall, we observed that the following optimizations were highly effective.

- Foster B-tree [12] to make B-tree operations and the latching module more efficient and to ensure that every node in the B-tree has at most a single incoming pointer at all times (which simplified book-keeping for unswizzling).
- Consolidation Array [17] to speed-up the logging module.
- Read-After-Write style lock manager [18] to improve the scalability of the locking module.

For the second and third optimizations, we thank the authors of the original papers for their generous guidance in applying the techniques in our code base. We also emphasize that our experiments confirm their observations in the aforementioned papers, reproducing significant speed-ups in a different code base. The throughput of our prototype in the standard TPC-C benchmark has more than doubled with these techniques.

5.2 Experimental set-up

The experiments were performed on a 4-socket Intel Xeon X7542 NUMA machine running at 2.7 GHz with 24 cores, and equipped with 256 GB of RAM. We used two RAID-10 10K rpm disk drives for storing the data files. We ran CentOS 5.5 with a 2.6.18-194 64-bit Linux kernel. All configurations were compiled using GCC 4.1.2, and used the same compile options, experimental codes, and parameters. Unless otherwise noted, we use a 10 GB buffer pool configured to use direct I/O (`O_DIRECT`) when reading from and writing to the disk to prevent the OS file system cache from caching disk pages. This is the standard method to prevent OS caching and buffering used even by commercial databases such as Oracle and DB2. Our database has a size of 100 GB (i.e., 10 times the buffer pool size). It consists of $1.73x10^9$ records, with each record having a key size of 20 bytes and value size of 20 bytes. For the main memory configuration we restrict the database size to the buffer pool size (10 GB). In order to isolate the effects of the buffer pool implementation (or lack of a buffer pool for the in-memory case), we use the same base implementation of the database; that is, the lock manager, logger, query engine, etc. are all identical across configurations. We do not evaluate the benefits of various data layouts (e.g. we do not consider T-trees [24]), and so all configurations use Foster B-trees with an 8 kilobyte page size.

5.3 Buffer pool performance

We first evaluate the impact of swizzling on the overhead imposed by the buffer pool through a set of experiments based on micro-benchmarks. We compare pointer swizzling in the buffer pool against two baselines, i.e., an in-memory database using direct pointers between B-tree nodes and a traditional buffer pool. As our microbenchmark study focuses solely on buffer pool performance, we disable other modules, including logging and transactional locking, in order to isolate the effects of swizzling on buffer pool performance. We discuss overall database system performance later in Section 5.4.

5.3.1 Query performance

Our first set of experiments evaluate the impact of swizzling in the context of a read-only workload. To this end, we compare the performance of an in-memory database that uses direct pointers between B-tree nodes to a database system that uses a traditional buffer pool.

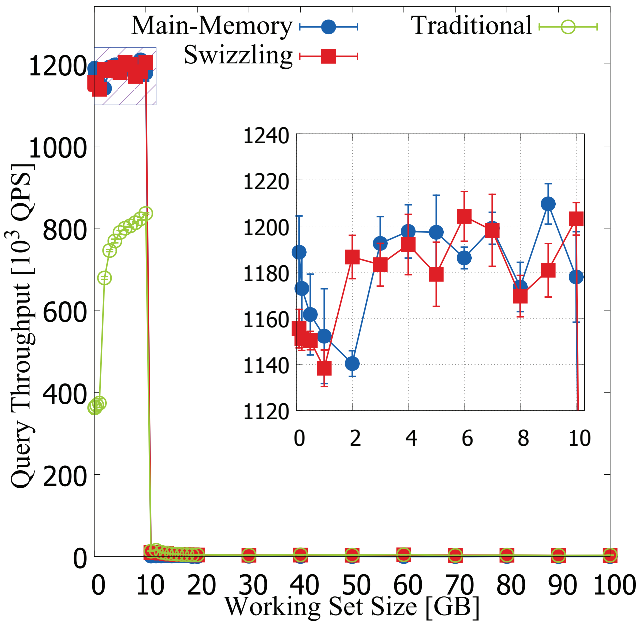


Figure 10: Read-only query throughput: Main-Memory (no bufferpool) vs Swizzling Bufferpool vs Traditional Bufferpool. Magnified sub-figure (inset) focuses on Main-memory vs Swizzling perfomance for an in-memory working set.

Figure 10 shows the performance of read-only queries, with pointer swizzling in the buffer pool compared against two baselines, i.e., an in-memory database and a traditional buffer pool. Error bars are one standard error. The inset sub-figure of Figure 10 magnifies the upper left-hand portion of the main graph so as to make it easier to compare the performance of the Main-Memory and Swizzling systems when the working set fits in memory. This figure is based upon the same data as Figure 1 with the addition of a buffer pool with pointer swizzling. Each data point represents at least five runs of a complete experiment with fixed buffer pool size (10 GB) and fixed working set size. 24 threads search the index for randomly chosen key values. The performance metric shows the number of look-ups completed per second.

These results confirm the principal goal and expectation of our design: when the working set is smaller than the buffer pool, query performance of a database with a buffer pool that supports pointer swizzling is comparable (practically equal) to that of an in-memory database that enjoys the performance advantage of in-memory pointers but is restricted to data sets smaller than memory. As a secondary effect, the performance of the database with the traditional buffer pool can be seen to improve significantly as the working set size increases to fill available memory (at the left-most edge of Figure 10) because threads work on a larger set of pages and hence suffer less latch contention. The traditional buffer pool is more sensitive to contention due to additional use of atomic instructions to pin and unpin pages.

The inset sub-figure in Figure 10 zooms into the range where the working set size is less than 10 GB — that is, where in-memory perfomance is feasible. The inset of Figure 10 provides a close-up view of the purple shaded square in the upper left of the main figure, where the Swizzling and Main-Memory curves overlap. As the inset figure illustrates, the performance difference between an in-memory database that uses only in-memory pointers and one that employs swizzling in an otherwise traditional buffer pool is very small. The differences are around 1-3% and quite comparable to the experimental standard error, indicated with vertical bars. At the left edge of the diagram, the working sets even fit into the CPU caches. Due to multiple processors and CPU caches, however, the additional performance gain is minimal.

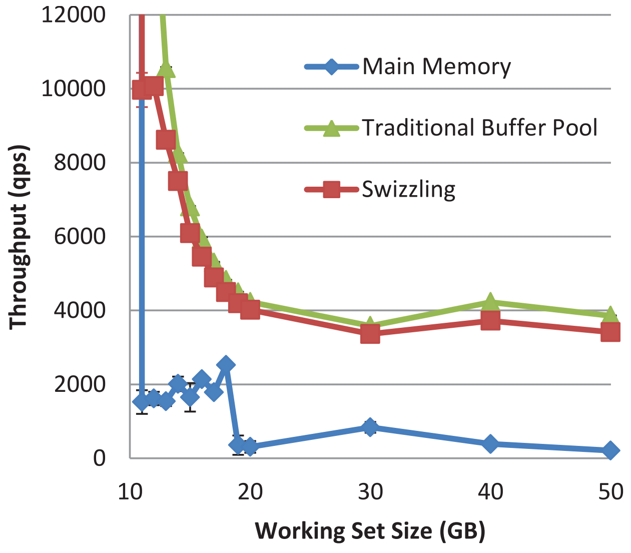


Figure 11: Graceful degradation

Figure 11 also zooms into the data of Figure 10, specifically the range of working set sizes just larger than the available buffer pool. In all three systems, even a small fraction of buffer faults and the required disk I/O cause performance to degrade significantly compared to in-memory operation. More interesting here is that the performance of the system with a traditional buffer pool and the one that employs swizzling deteriorate quite similarly as they use similar page replacement policies. The buffer pool with swizzling performs slightly worse than the traditional buffer pool as it has to un-swizzle the pointer to a victim page before evicting the page. In other words, the buffer pool with pointer swizzling does not introduce any disadvantages in operating regions outside its strengths. In contrast to the buffer pool designs that degrade gracefully, the main-memory design suffers a sudden performance drop when the OS virtual memory mechanism starts swapping pages to disk. This happens because the OS is agnostic with regard to the relative importance of inner-node pages versus leaf-node pages.

5.3.2 *Insertion performance*

Figure 12 evaluates insertion performance for a B-tree index. The entire index fits into memory throughout the entire experiment. In the experiment, 24 threads add 50 million records to an initial 10 million records, with key values chosen at random and in random order.

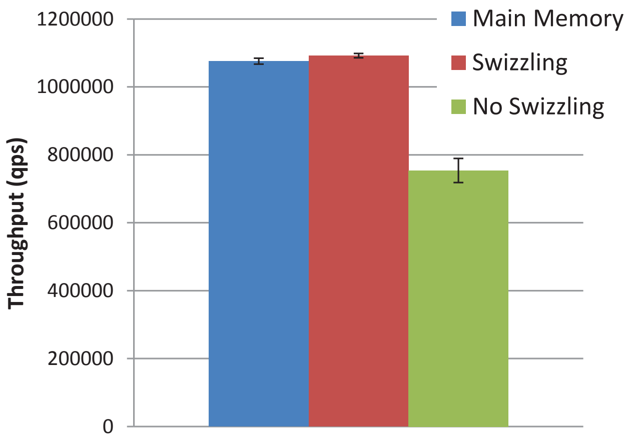


Figure 12: Insertion performance.

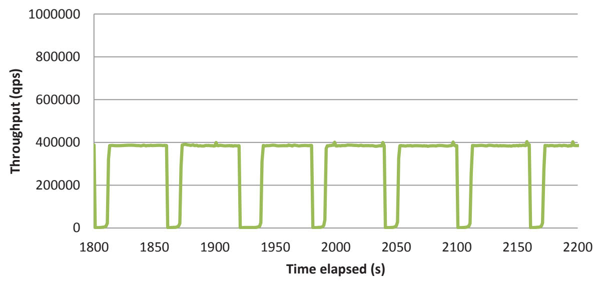
The performance differences follow the same pattern as in prior experiments using read-only index queries. The number of operations per second is smaller compared to the read-only experiments because of extra time spent in index maintenance, including leaf insertions and tree re-balancing. As shown in the figure, the performance of swizzling is equivalent to that of an in-memory database, and both out-perform a traditional buffer pool with page identifiers serving as child pointers.

5.3.3 *Drifting working set*

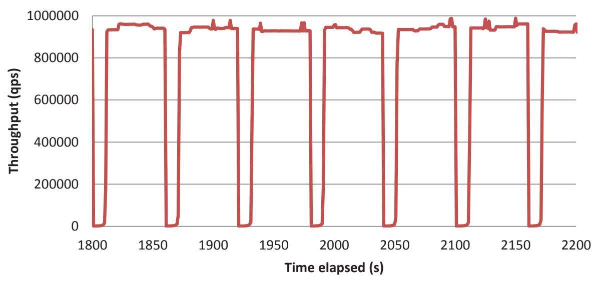
An additional experiment focuses on changes in the working set, i.e., the set of hot database pages changes over time. In the following, the working set is defined by a key range within a B-tree index chosen for a working set size of 0.1 GB throughout. Every 60 seconds, 25% of the key range (and thus of the current working set) is dropped and an equal key range (and working set size) is added. Thus, for a short transition period, performance is expected to drop. Ideally, the buffer pool loads the new 25% of the working set and then again exhibits performance comparable to an in-memory database. This experiment models scenarios where people want to read the latest data such as user status updates. As the interesting behavior happens when evicting pages from the buffer pool, we perform this experiment using a smaller buffer pool size of 1 GB to reduce the buffer-pool warm up time and to artificially create high pressure on the buffer pool.

Figure 13(a) shows the effects in a traditional buffer pool. Performance is steady except for the brief transition periods. After 1,920 seconds, the experiment has touched 1 GB of data and the buffer pool must start evicting pages; that is why the transition period at that time is slightly longer than other transition periods.

Figure 13(b) shows the corresponding performance, but for a buffer pool that employs pointer swizzling. As expected from the experiments reported above, when the working data set is in memory, throughput with swizzling is approximately twice as high as with the traditional system shown



(a) Traditional buffer pool



(b) Buffer pool with swizzling

Figure 13: Drifting working set

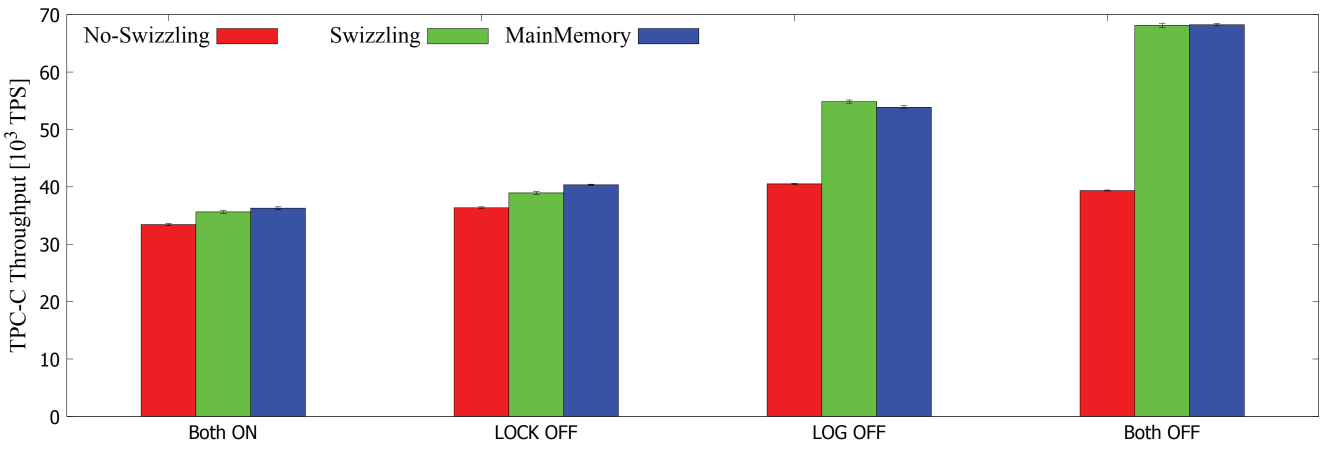


Figure 14: TPC-C Benchmark Result. 100 Warehouses, 12 Clients, warmed-up buffer pool larger than data set. Logging and Locking Modules are turned ON/OFF to analyze the contribution of Swizzling improvement in the entire database engine.

in Figure 13(a), and the length of the transition periods are equivalent between the traditional and swizzling systems.

5.4 TPC-C Benchmark

The last experiment runs the standard TPC-C benchmark [1]. Figure 14 compares the throughput of our database engine 1) with a traditional buffer pool (*No-Swizzling*), 2) with a swizzling buffer pool (*Swizzling*), and 3) without a buffer pool (*MainMemory*). As TPC-C is a write-intensive workload, we also turn on and off the locking module and the logging module in order to isolate the performance impact of swizzling in buffer pool compared to the logging and locking bottlenecks.

When both locking and logging modules are on, swizzling improves throughput over the traditional buffer pool about 10%. When both modules are off, on the other hand, swizzling improves throughput as much as 50%. The result is consistent with earlier observations (Figure 2) that the buffer pool is *one* of the bottlenecks in databases. A significant improvement in one module does not necessarily result in an equally significant improvement in overall performance; the removal of one bottleneck can expose new bottlenecks.

Nonetheless, this result demonstrates both that the significance of the buffer pool bottleneck as well as that swizzling techniques virtually eliminate the buffer pool bottleneck. The first is evidenced by the fact that the throughput of *No-Swizzling* did *not* significantly improve by turning off both locking and logging modules. Even supposing perfectly scalable locking and logging modules, a database using a buffer pool without swizzling would not achieve many-core performance competitive with a database without a buffer pool. The second is evidenced by the fact that in all cases, performance of a database with a swizzling buffer pool is statistically equivalent to that of an in-memory database.

Finally, Figure 14 motivates our eventual goal to overhaul *all* modules for the many-core era. Our current, ongoing, work is to significantly reduce the locking and logging bottlenecks with drastically different architectures optimized for many cores, such as [38]. We anticipate that as we make the locking and the logging modules as scalable and efficient as swizzling makes the buffer pool, overall throughput will dramatically increase; the potential improvement can be seen in the rightmost bars of Figure 14.

6. SUMMARY AND CONCLUSIONS

In summary, the buffer pool in a traditional database management system enables efficient data access and data manipulation for databases larger than memory as well as transactional updates with write-ahead logging. However,

[1]http://www.tpc.org/tpcc/

when the working set fits in memory, the performance overhead of a traditional buffer pool is so large that it motivates in-memory databases, whose principal value is that they do not use a buffer pool.

Some recent work has addressed in-memory database systems with working sets slightly larger than memory. These approaches have relied on virtual memory, compression, and offline analysis of hot and cold data items in order to improve the ability of the working data set to fit in memory. In contrast, our pointer swizzling approach simply eliminates most buffer pool overheads.

Unlike prior work on pointer swizzling between application objects, with difficulties arising from shared objects (multiple pointers to the same object) and from shared containers (multiple objects per database page), the proposed approach swizzles pointers between page frames in the buffer pool, i.e., between a finite number of system objects.

Our prototype implementation adds metadata to buffer pool frame descriptors without impacting actual data page contents, which simplifies the task of un-swizzling. An index structure with only a single (incoming) pointer per node simplifies both swizzling a pointer to an in-memory (virtual memory) address and un-swizzling a pointer, and also greatly simplifies the mechanisms involved in selecting a page for replacement in the buffer pool.

An experimental evaluation of this prototype demonstrates that swizzling parent-to-child pointers between the buffer pool's page frames practically eliminates the buffer pool bottleneck. Experiments with shifting working sets also show that un-swizzling overheads (tracking page usage, etc.) are minimal. Further experiments illustrate graceful performance degradation when the working set size grows and exceeds memory as well as quick performance improvements when the working set shrinks below memory size.

Judicious swizzling and un-swizzling of pointers within the buffer pool enables the performance of in-memory databases for memory-resident data, even when the total data set is much larger than the available buffer pool. In those cases where modern hardware with a large enough memory encompasses the application's entire active working set, database performance matches that of special-purpose in-memory databases. In other cases, graceful degradation and graceful improvement enable fluid transitions between in-memory mode and traditional buffer pool operation. For big data and working sets far exceeding memory capacity (which may well become the common case rather than an exception), the design enables scalability with the storage costs of traditional disk drives. Graceful degradation and fluid transitions between those cases ensure optimal performance throughout.

In conclusion, the proposed design turns the contradiction between in-memory computing and big data into a continuum with competitive performance across the entire spectrum, eliminating the need to divide memory between an in-memory database and a buffer pool for a persistent database. While our research context is in databases, the design applies directly to key-value stores and it should apply with little change to file systems and their allocation trees (indirection pages) and directory trees. Thus, we hope that turning a contradiction into a continuum, with competitive performance throughout, will prove useful for many future storage systems.

Acknowledgments

We thank with deepest appreciation all the developers and researchers of the Shore-MT team at EPFL, CMU and UW-Madison for making the Shore-MT code-base[2] available. Our development was greatly aided by their clean and efficient code as well as its extensive and insightful comments. We particularly thank Ryan Johnson and Ippokratis Pandis for their advice and suggestions regarding our implementation of the Consolidation Array they described in [17] and which was used in order to improve our logging module, as well as Hyungsoo Jung and the other authors of [18] for their clarification and advice with regard to our implementation of the Read-After-Write style lock manager that they proposed in their paper.

7. REFERENCES

[1] T. Anderson. Microsoft SQL Server 14 man: 'Nothing stops a Hekaton transaction'. `http://www.theregister.co.uk/2013/06/03/microsoft_sql_server_14_teched/`, 2013.

[2] M. P. Atkinson, K. Chisholm, W. P. Cockshott, and R. Marshall. Algorithms for a Persistent Heap. *Softw., Pract. Exper.*, 13(3):259–271, 1983.

[3] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *SIGMOD*, pages 383–394, 1994.

[4] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB*, 6(14):1942–1953, 2013.

[5] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. SIGMOD, 2013.

[6] FAL Labs. Tokyo Cabinet: a modern implementation of DBM. `http://fallabs.com/tokyocabinet/`.

[7] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP &amp; OLAP Databases. *PVLDB*, 5(11):1424–1435, 2012.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database system implementation*, volume 654. Prentice Hall Upper Saddle River, NJ, 2000.

[9] G. Graefe. A Survey of B-tree Locking Techniques. *ACM TODS*, 35(2):16:1–16:26, 2010.

[10] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[11] G. Graefe. A Survey of B-tree Logging and Recovery Techniques. *ACM TODS*, 37(1):1:1–1:35, 2012.

[12] G. Graefe, H. Kimura, and H. Kuno. Foster B-Trees. *ACM Transactions on Database Systems (TODS)*, 2012.

[13] SAP HANA. `http://www.saphana.com/`.

[14] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, 2008.

[2]`http://diaswww.epfl.ch/shore-mt/`

[15] A. L. Hosking and J. E. B. Moss. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *OOPSLA*, pages 288–303, 1993.

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

[17] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment*, 3(1-2):681–692, 2010.

[18] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84. ACM, 2013.

[19] T. Kaehler and G. Krasner. LOOM: Large Object-Oriented Memory for Smalltalk-80 Systems. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 298–307. Kaufmann, San Mateo, CA, 1990.

[20] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. In *ICDE*, pages 155–162, 1993.

[21] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *VLDB*, pages 427–438, 1994.

[22] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB J.*, 4(3):519–566, 1995.

[23] K. Küspert. *Fehlererkennung und Fehlerbehandlung in Speicherungsstrukturen von Datenbanksystemen*. Informatik-Fachberichte. Springer-Verlag, 1985.

[24] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, VLDB '86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[25] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[26] M. L. McAuliffe and M. H. Solomon. A trace-based simulation of pointer swizzling techniques. In *ICDE*, pages 52–61, 1995.

[27] C. Mohan. Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In *ICDE*, pages 324–331, 1995.

[28] MonetDB. `http://www.monetdb.org/`.

[29] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Software Eng.*, 18(8):657–673, 1992.

[30] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *SIGMETRICS Perform. Eval. Rev.*, 20(1):35–46, June 1992.

[31] Oracle TimesTen In-Memory Database. `http://www.oracle.com/technetwork/products/timesten/overview/index.html`.

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

[33] I. Pandis, P. Tozun, R. Johnson, and A. Ailamaki. PLP: Page latch-free shared-everything OLTP. *PVLDB*, 2011.

[34] S. Park. Personal Communication, 2013.

[35] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys '13*, 2013.

[36] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM TODS*, 3(3):223–247, Sept. 1978.

[37] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*, page 7, 2013.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[39] VoltDB. `http://www.voltdb.com`.

[40] S. J. White and D. J. DeWitt. *QuickStore: A high performance mapped object store*, volume 23. ACM, 1994.

[41] P. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Computer Architecture News*, pages 364–377, 1992.