FOEDUS: OLTP Engine for a Thousand Cores and NVRAM

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

ABSTRACT

Server hardware is about to drastically change. As typified by emerging hardware such as UC Berkeley's Firebox project and by Intel's Rack-Scale Architecture (RSA), next generation servers will have *thousands of cores*, *large DRAM*, and *huge NVRAM*. We analyze the characteristics of these machines and find that no existing database is appropriate. Hence, we are developing `FOEDUS`, an open-source, from-scratch database engine whose architecture is drastically different from traditional databases. It extends in-memory database technologies to further scale up and also allows transactions to efficiently manipulate data pages in both DRAM and NVRAM. We evaluate the performance of FOEDUS in a large NUMA machine (16 sockets and 240 physical cores) and find that FOEDUS achieves multiple orders of magnitude higher TPC-C throughput compared to H-Store with anti-caching.

1. DATABASES ON FUTURE SERVERS

Future server computers will be equipped with a large number of cores, large DRAM, and low-power, *non-volatile random-access memory* (`NVRAM`) with huge capacity. This disruptive difference in hardware also demands drastic changes to software, such as databases. Traditional OLTP databases do not scale up to a large number of cores. Recent advances in main-memory optimized databases have achieved far better scalability, but they cannot efficiently handle NVRAM as secondary storage. To fully exploit future hardware, we need a redesign of databases.

We are building `FOEDUS`[1] (Fast Optimistic Engine for Data Unification Services), an open-source[2], ground-up database engine for the next generation of server hardware. FOEDUS is a full-ACID database engine whose architecture is completely different from traditional databases. It is designed to scale up to a thousand cores and make the best use of DRAM and NVRAM, allowing a mix of write-intensive OLTP transactions and big-data OLAP queries.

FOEDUS employs a lightweight optimistic concurrency control (`OCC`) similar to those used in in-memory databases [30]. Unlike in-memory databases, however, FOEDUS maintains data pages in both NVRAM and DRAM and bridges the two in a unique manner for high scalability.

The key idea in FOEDUS is to maintain a *physically independent* but *logically equivalent* dual of each data page. The one side of the dual is a mutable `Volatile Page` in DRAM and the other side is an immutable `Snapshot Page` in NVRAM. FOEDUS constructs a set of snapshot pages from logical transaction logs, not from volatile pages, so that transaction execution and construction of snapshot pages are completely independent and run in parallel.

FOEDUS sequentially writes snapshot pages to NVRAM to maximize the I/O performance and endurance of NVRAM, similarly to LSM-Trees [23]. Unlike LSM-Trees, however, FOEDUS's `Stratified Snapshots` mirror each volatile page by a *single* snapshot page in a hierarchical fashion. When the volatile page is dropped to save DRAM, serializable transactions have to read only one snapshot page to retrieve the searched keys or non-existence thereof.

Another novel technique in this paper is the `Master-Tree` (a portmanteau of Masstree [22] and Foster B-tree [13]). It advances the state-of-the-art of OCC for simplicity and higher performance in future hardware, providing strong invariants to simplify OCC and reduce its aborts/retries.

These techniques together achieve excellent scalability. In our experiments on hundreds of CPU cores, we observed 100x higher throughput than H-Store [9], the state-of-the-art database, with anti-caching feature.

The key contributions in this paper are as follows:

- Analysis of next-generation server hardware trends and the resulting challenges for databases (§ 2-3).
- A new approach to scale beyond DRAM yet keep the performance of in-memory databases; *duality* of volatile pages and stratified snapshot pages (§ 4-7).
- A highly scalable and simple tree data structure for a thousand cores and NVRAM; *Master-Tree* (§ 8).
- OLTP scalability evaluation in an unprecedented scale; 240 physical cores in a single machine (§ 9).

2. NEXT GENERATION SERVERS

In this section, we characterize next-generation servers. Several universities and companies have recently started to design servers based on emerging hardware, such as `Firebox` [4], Intel's Rack-Scale Architecture (`RSA`), and Hewlett-Packard's `The Machine` [1].

These designs widely vary in some component. For instance, they differ in the technology to use for interconnect. However, they have some common features, described below.

[1] foedus: (Latin) contract, compact.
[2] http://github.com/hkimura/foedus

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author. Copyright is held by the author/owner.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2746480.

2.1 Thousands of CPU cores

Virtually all hardware vendors agree that many-core systems are the future. Typical servers in current data centers are equipped with up to one hundred CPU cores. Next-generation servers are expected to have thousands of CPU cores for an even higher density of computational power [35].

Non-uniform/coherent memory: However, with great (computational) power comes great responsibility. The cost of maintaining coherent memory-caches and other issues limit the number of CPU cores that can be placed in a uniform memory-access region. Most many-core servers today have two to eight CPU sockets that are connected to each other via QPI, which exhibits latency and bandwidth limitations as the number of sockets increases. Future machines will have further non-uniformity or potentially cache-incoherence in memory-accesses [4].

2.2 Large DRAM and Huge NVRAM

Over decades, the capacity of DRAM has increased exponentially. Future servers will have a large amount of main memory; hundreds of TB or more. Nevertheless, it is also a widely accepted observation that DRAM is becoming increasingly expensive to scale to smaller feature sizes for the scale beyond [3].

NVRAM: Major hardware vendors are developing innovative memory devices to overcome the limitations of DRAM.

Some of the front runners among them include *Phase-Change Memory* (`PCM`) [18], `STT-MRAM`, and `Memristors` [29]. In addition to lowering the cost per bit (compared to DRAM), the retention time of these technologies is also many years, making them *non-volatile*. With the widening performance gap between main memory and storage, the next big leap in system performance can be achieved by using the emerging NVRAM technologies as the primary data store [7].

Performance: NVRAM is expected to perform orders of magnitude faster than state-of-the-art non-volatile devices, such as SSD. However, the expected bandwidth and latency of NVRAM has a wide variation across vendors and even within each vendor because producing a mass-marketed storage device involves a wide range of challenges. The currently dominant expectation for the near future is that NVRAM will have higher latency than DRAM. For example, a current PCM product has 5 to 30 μs read latency and 100 μs write latency [17].

Endurance and thermal issues: A key limitation of emerging NVRAM technologies is their limited endurance. Depending on the type of cell (single level vs. multi level cell) and material used, NVRAM endurance is many orders of magnitude lower than DRAM. For example, PCM's endurance vary from 10^6 to 10^8 writes, and Memristor has an endurance of 10^{11} writes [19], whereas DRAM has an endurance of $> 10^{15}$ writes. In addition, technologies such as PCM and Memristor are also thermally sensitive: PCM retention time goes down with increase in temperature [6] and Memristor suffers from increased sneak current with temperature [28]. Prior work on NVRAM have shown that simple wear-leveling and wear-out tolerance techniques can alleviate the endurance problem and provide more than five years of server lifetime [26, 34].

2.3 Desiderata for Databases

These characteristics pose several requirements to databases running on future servers.

SOC/NUMA-aware memory accesses: Software in future servers must take into account that memory-access costs will be highly non-uniform. System-on-chip (`SOC`) [4] is one direction that can deal with even cache-incoherent architectures. Be it incoherent or not, DBMS must carefully place data so that most accesses to DRAM and NVRAM are *local*. In this paper, we interchangeably use the words SOC/NUMA-node and *SOC-friendly/NUMA-aware* to mean the requirement, targeting both SOC and NUMA systems.

Avoid contentious communications: The massive number of cores demands avoiding contentious communications as much as possible. For instance, many databases employ LSN-based concurrency control, which fundamentally requires at least one atomic `compare-and-swap` (`CAS`) in a single address (tail LSN) from every thread. We observed via experiments that even one contentious atomic `CAS` per transaction severely limits the performance when there are hundreds of cores, which is consistent with observations in prior work [30]. Databases inherently have synchronous communications, but *all* of them should be non-contentious.

NVRAM for *big* data, DRAM for *hot* data: A database must make use of NVRAM for big-data that do not fit in DRAM. However, DRAM still has an advantage in latency. Frequently-accessed data must reside in DRAM while cold data are moved in/out to NVRAM. When data are written to NVRAM, it is desirable that writes are converted into a small number of sequential writes so that the performance and the endurance of NVRAM are maximized.

3. RELATED WORK

Individual desiderata above are not new. We discuss key challenges to address *all* of them by reviewing prior work in the database literature categorized into three types; *traditional*, *in-memory*, and *hybrid*.

3.1 Traditional Databases

Traditional databases are optimized for disks. They can use NVRAM as faster secondary storage device, but they are significantly slower and less scalable compared to more recent alternatives in many-core servers [9]. Our empirical evaluation observes that even a well-optimized disk-based database does not benefit from replacing SSDs with NVRAM due to bottlenecks other than I/O, which is the key issue in traditional databases.

Shadow Paging: One noteworthy approach in traditional databases is *Shadow Paging*, which uses copy-on-write to modify each data page and atomically switches to the new versions during transaction commit [33]. Although Shadowing can avoid random in-place updates, it causes severe contention when the transaction runs in `serializable` isolation level and consists of multiple data accesses. Hence, its use in real world is limited to some filesystem that needs only local consistency or non-serializable database.

3.2 In-memory Databases

In-memory databases perform significantly faster and scale much better in many-core environments [10, 30], but they cannot utilize NVRAM.

SILO/Masstree: Nevertheless, recent efforts in in-memory databases have devised several breakthroughs to scale up to a large number of cores equipped with large NUMA memories. FOEDUS inherits the key enablers from these efforts: lightweight *optimistic concurrency control* (`OCC`) protocol in SILO [30] and *cache-sensitive* data structures, such as `Masstree` [22]. Because these are essential pieces of FOEDUS, we recap SILO's OCC protocol and Masstree in

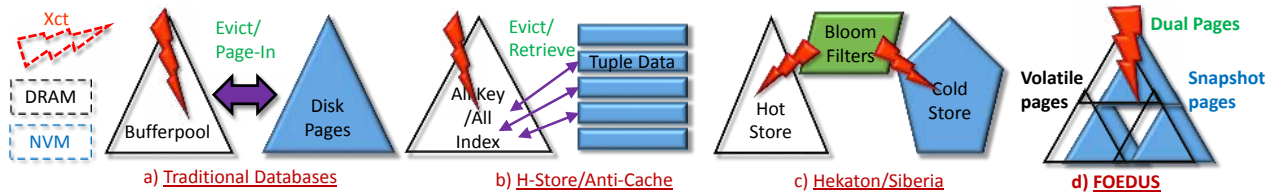

Figure 1: Various architectures to go beyond DRAM. a) xcts go through bufferpool that pages in/out, b) only non-key data are evicted, c) global bloom filters tell what is evicted, d) dual of volatile/snapshot pages.

later sections along with the design of FOEDUS.

Hekaton/Bw-Tree: SQL Server Hekaton [10] employs a commit verification protocol similar to SILO. One difference is that Hekaton uses the Bw-tree [20] for range-accessed indexes, which has a read-lock to prevent writers from accessing the tuple until the transaction that read the tuple commits. This protocol can reduce aborts, but instead even a read incurs an atomic write to shared memory. In other words, although both SILO and Hekaton employ OCC, SILO is *more* optimistic.

FOEDUS employs SILO-like OCC to minimize synchronous communications for reads because read operations happen more often than writes even in OLTP databases. Instead, the FOEDUS's Master-Tree ameliorates the issue of aborts.

Key Issues: In contrast with traditional databases, in-memory databases scale well but the entire data set must fit in DRAM. The key challenge arises when one tries to provide *both* scalability and ability to go beyond DRAM.

3.3 Hybrid Databases

An emerging **hybrid** approach combines in-memory and on-disk DBMSs to address the issues together. FOEDUS, H-Store with `anti-caching` [8, 9], and `Siberia` [12] fall into this category. Figure 1 summarizes these hybrid approaches for handling data sets larger than DRAM. Traditional databases use a bufferpool to handle disk-resident pages, which is tightly coupled with logging and locking/latching modules. All hybrid databases avoid this major source of contention and overhead so that they can keep the high performance of in-memory databases.

Anti-caching: H-Store with anti-caching keeps all essential data in DRAM and evicts only cold data to secondary storage so that a transaction can speculatively identify the set of records to retrieve from secondary storage in one pass. One limitation of this approach is that it has to keep all indexes and key attributes in memory. Another limitation is that it has to evict/retrieve/repack each tuple and maintain *per-tuple* metadata to track what is evicted, to where, and last access time. Hence, it has substantial overheads and memory footprints.

Siberia: Siberia uses Hekaton as the primary *hot-store* in DRAM while it moves infrequently used tuples to a separate *cold-store* on disk, tracking which tuple is in the cold-store with global Bloom Filters. An interesting benefit of this approach is that the two data stores can be structured differently (e.g., column-store for cold-store).

Although Siberia can move entire tuples to cold store, it still has `out-of-page` Bloom Filters. A system that maintains such global metadata out-of-page must either hold a huge number of metadata proportional to the size of the entire database and/or suffer from false positives.

Key Issues: Prior approaches have either *physical dependency* or *logical inequivalence* between corresponding *hot* and *cold* data.

Anti-cache, like traditional bufferpools, keeps two data physically dependent to each other, thus the tuple/page must be evicted/retrieved in a synchronized fashion. On the other hand, Siberia does not maintain logical equivalence in any granularity of data between hot-store and cold-store. Hence, to retrieve a piece of information, such as the value of a tuple or the existence of keys in a certain range, it needs to maintain a global data structure and lookup in it for each tuple to coordinate the two stores.

4. KEY IDEA: DUAL PAGES

This paper suggests an alternative approach for hybrid databases to bridge hot and cold data; `duality` of pages. FOEDUS stores all data in fixed-size pages and maintains all metadata `in-page` in a hierarchical fashion, using `dual page pointers`, depicted in Figure 2.

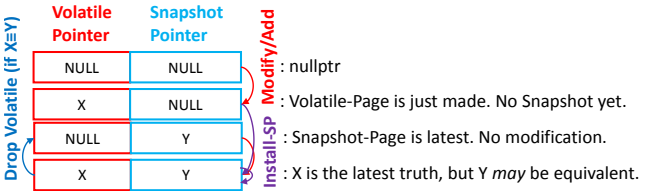

Figure 2: States/Transitions of Dual Page Pointers.

A dual page pointer points to a pair of logically equivalent pages, a mutable *volatile page* in DRAM for the latest version and an immutable *snapshot page* in NVRAM for previous version (which might be the latest if there was no modification).

The two pages are physically independent, thus a transaction that modifies the volatile page does not interfere with a process that updates the snapshot page and vice versa. Yet, the two pages are logically equivalent, thus we do not need additional information or synchronization to figure out the location of any data represented by the page.

When a volatile page exists, it is guaranteed to represent all data that exist in the snapshot page. When it is *null*, on the other hand, the corresponding snapshot page is guaranteed to represent all data that exist in the database.

No out-of-page information: FOEDUS maintains no out-of-page information, such as a separate memory region for record bodies [30], mapping tables [20], a central lock manager, etc. This invariant is essential for highly scalable data management where contentious communications are restricted to each page and all footprints are proportional only to the size of hot (DRAM-resident) data, not cold (NVRAM-resident) data. In an extreme-but-possible case where there are TBs of all-cold data, FOEDUS does not need any information in DRAM except a single dual pointer to the root page whereas anti-caching and Siberia both require huge amounts of metadata.

By storing all data purely in-page, FOEDUS also eliminates data compaction and migration, which often cause severe scalability issues for garbage collector (`GC`). FOEDUS

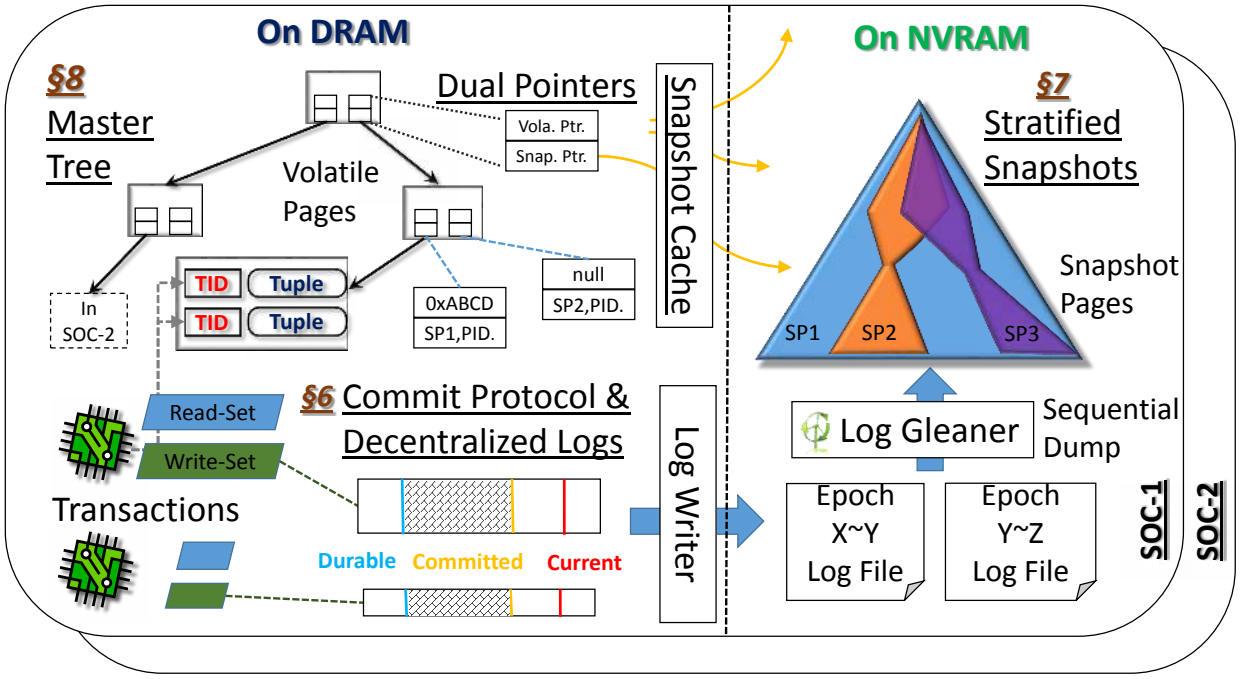

Figure 3: FOEDUS Architecture Overview. All page pointers are dual pointers. Volatile Pages in DRAM are physically independent but logically equivalent duals of Snapshot Pages in NVRAM, which are separately and concurrently constructed by Log Gleaner from logical transaction logs.

reclaims pages when they are no longer needed, but the reclaimed page can be immediately reused in another context without compaction or migration because we use a fixed-size page everywhere. This design also eliminates an additional CPU cache miss and potentially a remote SOC access because the record data are always in the page itself.

5. SYSTEM OVERVIEW OF FOEDUS

Figure 3 shows the architectural overview of FOEDUS. Conceptually, FOEDUS is a multi-version DBMS with lightweight OCC to coordinate transactions, which accesses two sets of data pages lazily synced via logical transaction logs.

Storages: *Storage* is the unit of data structure in FOEDUS. One table consists of one or more storages, such as a primary index and secondary indexes. FOEDUS maintains a tiny amount of static information for each storage, such as name, type, and pointer to the *root page*.

Dual Pointers: As described in previous section, most page pointers in FOEDUS are *dual* page pointers except a few temporary pointers, such as foster-twins described later. A transaction follows or atomically installs the volatile pointer to modify tuples in the page or its descendants. A transaction *might* follow the snapshot pointer when the volatile pointer is null (i.e., the snapshot page is the latest information) or it is not a `serializable` transaction. The main body of this paper focuses on the `serializable` execution. Appendix B discusses `SI` transactions in FOEDUS.

Page Pools and Data Sharing: We have two DRAM-resident page pools; one for volatile pages and another for caching snapshot pages. In traditional databases, all CPU cores would share all pages in such page pools. In FOEDUS, `snapshot cache` is completely SOC-local to eliminate inter-SOC communications, utilizing the fact that snapshot pages are immutable thus safe to replicate. Although the volatile pool inherently requires remote access to make transactions serializable, FOEDUS places a volatile page in a node that first modifies the page to exploit locality of access.

Commit Protocol and Logging: In order to guarantee ACID properties, each transaction in FOEDUS maintains read- and write-sets as well as log buffers. This bookkeeping information is completely thread-private to avoid contentions and inter-SOC communications. Section 6 explains the detailed protocols to verify and apply them when the transaction commits.

Stratified Snapshots: *Stratified Snapshots* are the central mechanism of FOEDUS to store snapshot pages in NVRAM and maintain the duality with volatile pages. It consists of an arbitrary number of snapshots, each of which is a complete, durable, and immutable image of the entire database as of a certain epoch (*snapshot-epoch*), which serves both transactional processing and crash recovery. Section 7 describes stratified snapshots in details.

Master-Trees: FOEDUS provides a few storage types, such as Heap and Hash-index. Its most general storage type is *Master-Tree*, a variant of B-trees designed for NVRAM and many-cores. Section 8 explains Master-Tree in details.

NVRAM Interface: FOEDUS manipulates NVRAM devices only via standard filesystem APIs. FOEDUS does not require vendor-specific APIs to support a wide range of NVRAMs, even including high-end flash devices.

6. LOGGING AND COMMITTING

FOEDUS employs SILO's decentralized logging and optimistic committing protocols [30, 37] and extends it to snapshot pages stored in NVRAM. Section 6.1 gives a detailed review of the SILO's protocols while Section 6.2 explains FOEDUS's extension.

Algorithm 1: SILO precommit protocol [30]

Input: R: Read set, W: Write set, N: Node set

`/* Precommit-lock-phase */`
Sort W by unique order;
foreach $w \in W$ **do** Lock w;

Fences, get commit epoch;
`/* Precommit-verify-phase */`
foreach $r, observed \in R$ **do** **if** $r.tid \neq observed$ *and* $r \notin W$ **then** abort;

foreach $n, observed \in N$ **do** **if** $n.version \neq observed$ **then** abort;

Generate TID, apply W, and publish log;

Algorithm 2: FOEDUS precommit protocol

Input: R: Read set, W: Write set, P: Pointer set

`/* Precommit-lock-phase */`
while *until all locks are acquired* **do**
  foreach $w \in W$ **do if** *w.tid.is-moved()* **then** w.tid ← track-moved(w)

  Sort W by unique order;
  foreach $w \in W$ **do** Try lock w. If we fail and find that w.tid.is-moved(), release all locks and retry
end

Fences, get commit epoch;
`/* Precommit-verify-phase */`
foreach $r, observed \in R$ **do**
  if *r.tid.is-moved()* **then** r.tid ← track-moved(r)
  if $r.tid \neq observed$ *and* $r \notin W$ **then** abort;
end
foreach $p \in P$ **do** **if** $p.volatile\text{-}ptr \neq null$ **then** abort;

Generate TID, apply W, and publish log;

6.1 Logs and Read/Write Sets

Transactional Logging: Each worker thread (transaction) holds a circular, private log-buffer as shown in the left-bottom of Figure 3. The log buffer is written only by the worker thread and read by the log writer to write out to log files stored in NVRAM. Each log entry contains the ID of the storage, the key string, and the payload of the record inserted/modified/etc by the logical action.

The log buffer maintains a few markers that point to positions in the buffer 1) `durable`: upto where the log writer has written out to files, 2) `committed`: upto where the thread has completed pre-commit, and 3) `current`: where the thread is currently writing to. The job of a log writer is to dump out logs between `durable` and `committed`, then advance `durable`.

Read/Write Sets: Each transaction maintains `read-set` and `write-set`, which record the addresses of tuples the transaction accessed. The main differences from traditional databases are that 1) it only remembers the observed version number (Transaction-ID, or `TID`) of the tuple as of reading instead of taking a read-lock, and 2) it merely pushes the planned modification to the transaction's private log buffer and remembers the log position in the corresponding write set instead of immediately locking the tuple and applying the modification.

SILO's Commit Protocol: The core mechanism of our concurrency control lies in its `pre-commit` procedure, which concludes a transaction with verification for serializability but not for durability. We provide durability for a group of transactions by occasionally flushing transaction logs with `fsync` to log files in NVRAM for each `epoch`, a coarse-grained timestamp. In other words, pre-commit guarantees `ACI` out of `ACID`, whereas D is separately provided by *group-commit*. Algorithm 1 summarizes the original version of the pre-commit protocol proposed in SILO.

It first locks all records in the write set. SILO uses concurrency control that places an *in-page* lock mechanism (e.g., 8 bytes TID for each record) that can be locked and unlocked via atomic operations without a central lock manager. Placing lock mechanism in-page avoids high overheads and physical contention of central lock managers [27], and potentially scales orders of magnitude more in main-memory databases.

It then verifies the read set by checking the current version numbers *after* finalizing the epoch of the transaction, taking memory fences. If other transactions have not changed the TIDs since the reading, the transaction is guaranteed to be serializable. For example, *write-skews* are detected from the discrepancy between the observed TID and the current TID, which is either locked or updated by a concurrent transaction. The transaction then applies the planned changes in the private log buffer to the locked tuples, overwriting their TIDs with a newly generated TID of the transaction that is larger than all observed TIDs. The committed transaction logs are then *published* by moving `committed` to `current` so that log writers can write them to log files for durability.

Advancing Epochs: The *current* epoch and *durable* epoch of the system are periodically advanced by background threads that check the progress of each logger and worker thread with some interval (e.g., 20 ms). Pre-committed transactions are deemed as truly *committed* when the new durable epoch is same or larger than their TID's epoch. These decentralized logging/commit protocols based on coarse-grained epochs eliminates contentious communications in the traditional LSN-based databases.

6.2 Extension for NVRAM

Issues: SILO is a purely in-memory DBMS. It cannot handle the case where a data page is evicted from DRAM. On the other hand, FOEDUS might drop a volatile page when it is physically identical to the snapshot page. After that, a transaction only sees the read-only snapshot data page unless it has to modify the data page, in which case the transaction installs a volatile version of the page based on the latest snapshot page. However, this can violate serializability when other concurrent transactions have already read the snapshot pages.

Pointer Set: To detect the installation of new volatile pages, each transaction in FOEDUS maintains a `pointer set` in addition to read-set/write-set. Whenever a serializable transaction follows a pointer to a snapshot page because there was no volatile page, it adds the address of the volatile pointer to the pointer set so that it can verify it at the pre-commit phase and abort if there has been a change as shown in Algorithm 2.

The pointer set is analogous to the node-set (page-version set) in SILO. The difference is that the purpose of the SILO node-set is to validate page *contents*, whereas FOEDUS uses the pointer set to verify page *existence*. SILO cannot ver-

ify the existence of new volatile pages, which SILO did not need because it is a pure in-memory database. FOEDUS protects the contents of pages with a different mechanism, called foster-twins. We revisit FOEDUS's commit protocol with the foster-twin technique in Section 8.

FOEDUS's pointer set usually contains only a few entries for each transaction because we do not have to add pointer sets once we follow a snapshot pointer. By definition, everything under a snapshot page is stable. We have to verify only the first pointer we followed to jump from the volatile world to the snapshot world. There is no path from the snapshot world back to the volatile world. In an extreme (but not uncommon) case, we have only one pointer set for reading millions of records and several thousands of pages in a serializable OLAP transaction. In fact, we empirically observe that placing data in snapshot pages significantly (3x) speeds up the version verification during pre-commit.

7. STRATIFIED SNAPSHOTS

Stratified Snapshots are FOEDUS's main data repository placed in NVRAM. As the name suggests, stratified snapshots are layers of snapshots each of which is constructed by the *Log Gleaner* from log files. The log gleaner does not dump out the entire image of the database for each execution, which would be prohibitively expensive for large databases. Instead, it replaces only the modified parts of the database. LSM-tree [23] also writes out only changed data, but the difference is that log gleaner always outputs a snapshot that is a *single* image of the entire storage. Multiple snapshots together form a *stratified* snapshot where newer snapshots overwrite parts of older snapshots. Each snapshot contains a complete path for every record up to the epoch as of the snapshot. For example, the root page of a modified storage is always included in the snapshot, but in many cases the only change from the previous snapshot version is just one pointer to lower level. All other pointers to lower level point to previous snapshot's pages.

The benefit of this design is that a transaction has to read only one version of stratified snapshots to read a record or a range of records. This is essential in OLTP databases where checking an existence of key must be quick (e.g., inserting into a table that has primary key, or reading a range of keys as a more problematic case). LSM-Tree approaches would have to traverse several trees or maintain various Bloom Filters for serializability [12], whose footprints are proportional to the size of cold data, not hot data.

7.1 Log Gleaner Overview

The log gleaner occasionally (e.g., every 10 minutes) collects transactional logs in each SOC's log files and processes them in a map-reduce fashion, sequentially writing the snapshot pages to NVRAM. This guarantees that FOEDUS always, including the log files, writes to NVRAM in a sequential fashion without in-place-updates. This maximizes the I/O performance and endurance of NVRAM. The new snapshot pages then replace volatile pages in DRAM to reduce DRAM consumption.

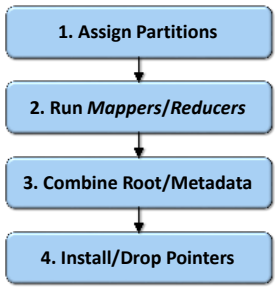

Figure 4: Log Gleaner

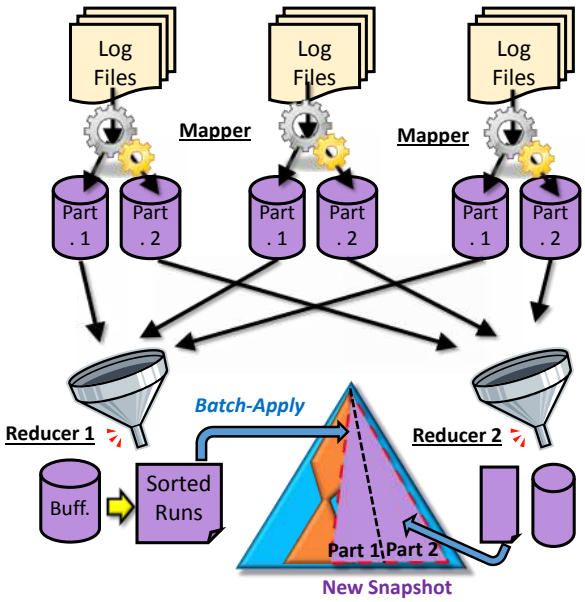

Figure 5: Log Mapper partitions logs. Log Reducer constructs snapshot pages in batches.

Each execution of log gleaner consists of the four stages shown in Figure 4, detailed below.

7.2 Assign Partitions

The first stage of log gleaner is to determine the partitioning keys and their assigned nodes (SOCs). To minimize inter-SOC communications, FOEDUS tries to store snapshot pages in a node that will most frequently use the page. For this purpose, FOEDUS maintains statistics in volatile pages to record which node has recently modified the page. FOEDUS uses this information to determine the boundary keys defining the partitions and their assigned nodes. This statistics is maintained without transactional guarantees to avoid unnecessary overheads.

7.3 Log Mappers and Reducers

The second stage of log gleaner is to run *mappers/reducers* in each node as Figure 5 illustrates.

Mappers: First, each mapper reads the log files containing log entries in the target epochs for the snapshot. Then, it buckets log entries by storages, buffering several log entries per storage, usually in MBs. Once a bucket for a storage becomes full, it sorts and partitions the logs in the bucket based on the boundary keys for the storage designed in the previous stage. The partitioned log entries are sent to each partition (reducer) per bucket. As far as the partitioning captures the locality, mappers send most logs to a reducer in the same node.

The mapper ships the log entries to the reducer's buffer in a three-step concurrent copying mechanism. It first reserves space to copy into the reducer's buffer by atomically modifying the state of the reducer's buffer. It then copies the entire bucket (usually MBs) into the reserved space by single write, which is much more efficient than issuing many writes especially in a remote node. This copying is the most expensive operation in this protocol, but it happens in parallel to copying from other mappers. Finally, the mapper atomically modifies the state of reducer's buffer to announce the completion of the copying.

Reducers: A reducer maintains two buffers, one for current batch and another for previous batch. A mapper writes to the current batch until it becomes full. When it becomes full, the reducer atomically swaps the current and previous batch and then waits until all mappers complete their copying. While mappers copy to the new current buffer, the reducer dumps the ex-current buffer to a file (*sorted runs*) after sorting them by storages, keys, and then serialization order (epoch and in-epoch ordinals).

Once all mappers are finished, each reducer does a merge-sort on the current buffer in memory, dumped sorted-runs, and previous snapshot pages if the key ranges overlap. In sum, this results in streams of logs sorted by storages, keys, and then serialization order, which can be efficiently applied.

Batching and Optimization: Separating the construction of snapshot pages from transaction execution enables several optimizations in mappers and reducers.

One optimization is `compaction` of log entries. In a few places where FOEDUS performs a batch-sort of logs, some log entries can be eliminated without affecting correctness. For example, repeated overwrites to one tuple in a storage can be safely represented by just one last update log for the tuple as far as the updated region (e.g., column) is the same. Deletion nullifies all previous inserts and modifications. Increment operations can be combined into one operation, too. This often happens in summary tables (e.g., `warehouse` in TPC-C), which compact several thousands log entries into just one before the mapper sends it out to reducers.

Another optimization in reducers is to `batch-apply` a set of logs into one data page. Reducers construct snapshot pages based on fully sorted inputs from log entries and previous snapshot pages. Hence, reducers simply append a large number of records in tight loops without any binary search or insert-in-between, which makes processing of the same amount of data substantially more efficient than in transaction executions. Also, some tasks are much easier to do using log records after a transaction has been committed than with live pages in DRAM during transaction processing. For example, it is very efficient for reducers to physically delete an empty page or a logically-deleted record as opposed to performing the same task during transaction processing. FOEDUS never physically deletes a record in volatile pages to simplify its concurrency control.

In Section 9, we observe that a single reducer instance can catch up with a huge influx of logs emit by several worker threads because of these optimizations.

7.4 Combining Root Pages and Metadata

When all mappers and reducers are done, the log gleaner collects root pages from reducers and combines them to a single new root page for each storage that had any modification. The log gleaner then writes out a new snapshot metadata file that contains updated metadata of all storages (e.g., page ID of their latest root pages in the stratified snapshots).

Combining root pages is a trivial job because the partitions are non-overlapping. The tricky case happens only when there is a skew; the choice of partitioning keys failed to capture the log distribution accurately. For example, there might be a case where one node receives few logs that result in a B-tree of two levels while another node receives many more logs that result in a B-tree of four levels. FOEDUS so far leaves the tree imbalanced in this case. It is possible to re-balance the tree, but it might result in unnecessary remote accesses for the node that has smaller sub-trees. In the above example, transactions on the first node can access local records in three reads (the combined root page and its sub-tree) while, if the re-balanced tree has five levels, they might have to read two more pages in higher levels that are likely in remote nodes.

7.5 Install/Drop Pointers

Once the metadata file and the snapshot files are written, the log gleaner installs new snapshot pointers and drops pointers to volatile pages that had no modifications during the execution of log gleaner. FOEDUS keeps volatile versions of frequently modified pages and their ascendants.

Installing Snapshot Pointers: The log gleaner installs snapshot pointers to *corresponding* dual pointers based on key ranges of the pages. A volatile page corresponds to a snapshot page if and only if the key ranges are exactly the same. Reducers try to minimize the cases of non-matching boundaries by *peeking* the key ranges of volatile pages while they construct snapshot pages, but it is possible that the volatile page had another page split because log gleaner runs concurrently to transactions. In such a case, the log gleaner leaves the snapshot pointer of the dual pointer to NULL, thus it cannot drop the volatile page until next execution.

Dropping Volatile Pointers: The last step of log gleaner drops volatile pointers to save DRAM. We currently pause transaction executions during this step by locking out new transactions. We emphasize that this does *not* mean the entire log gleaner is a *stop-the-world* operation. Dropping volatile pointers based on an already-constructed snapshot is an instantaneous in-memory operation that finishes in milliseconds. In the case of TPC-C experiments in Section 9, it takes only 70 milliseconds even after gleaning all data (initial data population). The log-gleaner spends the vast majority of time in mappers and reducers, which work fully in parallel to transaction executions. By pausing transaction executions while dropping volatile pointers, we can avoid expensive atomic operations for each pointer only at the cost of a sub-second transaction latency once in several minutes.

7.6 Snapshot Cache

Transactions read the snapshot pages via `snapshot cache` to avoid repeated I/O like traditional buffer pools.

However, because snapshot pages are immutable, this snapshot cache has several unique properties that distinguish it from typical buffer pools. First, snapshot cache is SOC-local. One snapshot page might be cached in more than one SOCs to avoid remote memory access. Such a *replication* for mutable pages in traditional databases would cause expensive synchronization. Second, when a thread requests a page that has already been buffered, it is acceptable if occasionally the page is re-read and a duplicate image of the page added to the buffer pool. This does not violate correctness, nor does it impact performance insofar as it occurs only occasionally. The occasional extra copies wastes only a negligible amount of DRAM, and the performance benefits FOEDUS gains by exploiting these relaxed requirements are significant. Snapshot cache does not use any locks or even atomic operations. It only needs a few memory fences and an epoch-based *grace-period* before reusing evicted pages.

7.7 Crash Recovery

In traditional databases, a crash recovery is a complex procedure because of the interaction between transaction logs and data pages, which may or may not have the modi-

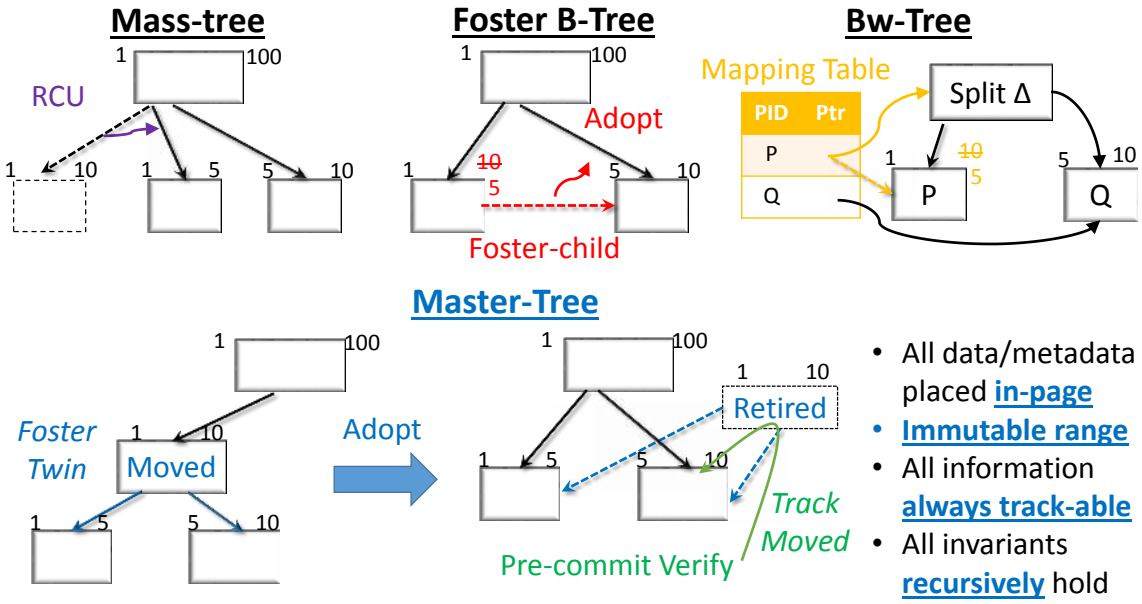

Figure 6: Master-Tree: Foster-Twin provides strong invariants to simplify OCC and reduce retries.

fications right before the crash. In fact, recent research [36] finds that most databases, including commercial databases, may fail to recover from power-failure, causing data loss or even inconsistent state.

The key principle of FOEDUS, physical separation between the volatile pages and snapshot pages, simplifies this problem. The only additional thing FOEDUS does after restart is to truncate transactional log files up to the place the previous execution durably flushed. As there is no volatile page after restart, FOEDUS then simply invokes a log gleaner just like a periodic execution, which is well optimized as described above. There is no code path specific to restart. This makes FOEDUS's crash recovery significantly more robust and efficient.

The interval between log gleaner executions (usually a few minutes) is also the upper limit of recovery time because log gleaning during restart cannot be slower than that of during normal executions as there are no live transactions to put pressure on CPU cores, its caches, and volatile pages to install/drop pointers from. Hence, stratified snapshots are also efficient checkpoints to quickly restart from.

8. MASTER-TREE

Master-Tree combines techniques in Masstree [22] and Foster B-Tree [13] to build B-trees appropriate for NVRAM and OCC. Not only combining the two, Master-Tree also has a key distinction that provides strong invariants to drastically simplify our OCC protocol and reduce aborts/retries. Section 8.1 briefly reviews the two prior work and describes how FOEDUS combines them. Section 8.2 then details our innovative *foster-twin* technique.

8.1 Basic Data Structures

Master-Tree leverages Masstree's cache-craftiness and employs a variant of its OCC protocol. In short, Masstree is a 64-bit B-trie where each layer is a B-tree optimized for 64-bit integer keys. Most key comparisons are done as efficient 64-bit integer comparisons with only a few cacheline fetches per page, digging down to next layers when keys are longer than 64-bit. When Masstree splits a full page, it does `RCU` (read-copy-update) to create a new version of the page and atomically switches the pointer as shown in Figure 6. A transaction in SILO guarantees serializability by taking node-set (page-version set) to the page and aborting if the page-version tells that the page has split.

Masstree is a pure main-memory data structure where a page stays there forever. To allow page-in/out, we extend it with Foster B-Tree techniques. The key issue to page-in/out in Masstree is that it has *multiple* incoming pointers per page, such as `next/prev/parent` pointer in addition to the pointer from parent pages. In a database with page in/out, such multiple incoming pointers cause many issues in concurrency control. Foster B-Tree solves it by the concept of *foster-child*, a tentative parent-child relationship that is de-linked when the real parent page *adopts* the foster-child. Master-tree guarantees a single incoming pointer per page with this approach except *retired* pages explained later.

Master-tree also extensively uses *system transactions* for various physical operations. For example, inserting a new record usually consists of a system transaction that *physically* inserts a logically-deleted record of the key with sufficient body length and a user transaction that *logically* flips the deleted flag and installs the record. It is worth noting that system transactions are especially useful when used with logical logging, not physiological logging. Because a system transaction does nothing logically, it does not have to write out any log nor touch log manager, which was one drawback of inserting with system transactions [13]. A system transaction in FOEDUS merely takes read-set/write-set and follows the same commit protocol as usual transactions.

Master-tree makes several contributions in extending prior work for FOEDUS. However, this paper focuses on a key invention: the foster-twins.

8.2 Foster-Twins

Page Split Issue: As described earlier, SILO uses an in-page locking mechanism based on `TID`. One issue of this protocol is that an in-page lock state (TID) can cause false aborts due to page splits or requires per-tuple GC.

When a page splits, we must keep at least the lock states

of all existing records in the old page because the concurrent transactions are pointing to the physical addresses in the old page. This means we cannot move any records to make room for new insertions, defeating the purpose of split. One trivial alternative is to abort such transactions (they can at least see that something happened by seeing updated TID in the old page or checking the split-counter used in SILO), but this causes the transaction to abort even when the tuple in the read-set was not modified. For an extreme but not uncommon case, even a single-threaded execution might have unnecessary aborts. The transaction inserts many pseudo-deleted records by system transactions, adding the records to write-set to logically insert at the pre-commit phase. Once it triggers splits of the page, the transaction has to abort at pre-commit even though there is no concurrent transaction. Another trivial alternative is to re-search all keys at pre-commit from the root of the tree, but this is significantly more expensive and sometimes no better than aborting and retrying the whole transaction.

Yet another trivial solution is to have pointers rather than data in pages. The pointers point to dynamically-sized tuple data in centrally allocated memory-pool, for example `delta-records` in Bw-trees [20]. This avoids the problem of a lock state disappearing at split, but instead requires per-tuple GC. When these tuples are not needed any longer, we have to reclaim memory for them. However, per-tuple GC is significantly more costly and complicated compared to per-page GC for fixed-size data pages. Per-tuple GC inherently causes defragmentation due to various sizes of tuples and thus must do compaction like general GCs in managed languages, which is notoriously expensive. Further, the pointer approach requires one CPU cache miss for accessing each tuple because the tuple data are not physically placed in a data page.

Foster-Twins: To solve the problem, we introduce a foster-child variation that we call *foster-twins*. When a page splits, we mark the TIDs of all records in the page as *moved* and also create two foster children; or foster-twins. Foster-twins consist of minor (or left) foster child and major (right) foster child. The minor foster child is responsible for the first half of key regions after split while the major foster child is responsible for the second half. In other words, the major foster child is the foster child in a Foster B-Tree while the minor foster child is a fresh-new mirror of the old page (but after compaction). At the beginning of the split, we mark the old page as *moved*, meaning that the page must no longer have any modifications. In the next tree traversal, the parent page of the old page finds it and adopts the major foster child like a Foster B-Tree, but also it modifies the pointer to the old page to the minor foster child, marking the old page *retired*. Retired pages are reclaimed based on epochs to guarantee that no transactions are referencing them as of reclaiming. Like a Foster B-Tree, our Master-tree has only one incoming pointer per page, thus there is no other reference to the retired page **except** concurrent transactions that took the address of the TIDs as read-set and write-set. The concurrent transactions, during their pre-commit phase, become aware of the moved-mark on the records and track the re-located records in foster-minor or foster-major children.

Commit Protocol: Now, let us revisit Algorithm 2 in the earlier section. The difference from Algorithm 1 is that we locate the new location of TID as we see the moved bit, using the foster-twin chain. In case of frequent splits, the foster-twin might form a binary tree of an arbitrary depth (although rarely more than one depth), hence this tracking might recurse. We do this tracking *without* locking to avoid deadlocks. We then sort them by address and take locks. However, this might cause staleness; concurrent transactions might have now split the page again, moving the TIDs. In that case, we release all locks and retry the locking protocol.

Benefits: The main benefit of foster-twins is that every page has a stable key-region **for its entire life**. Regardless of splits, moves, or retirement, a page *is* a valid page pointing to precisely the same set of records via foster-twins. Thus, even if concurrent transactions see moved or even retired pages, they do not have to retry from the root of the tree (which [22] and [30] does). This property drastically simplifies our OCC algorithm. Especially in interior pages, we never have to do the hand-over-hand verification protocol nor the split-counter protocol in the original Masstree. A tree search simply reads a *probably-correct* page pointer and follows it without even memory fences. It just checks the key-range, an immutable metadata of the page, and only *locally* retries in the page if it does not match.

When a transaction contains cursors or miss-search, FOEDUS verifies the number of physical tuples in the border page to avoid phantoms. However, unlike the page-version verification in SILO, FOEDUS does not need to check any information in intermediate pages or anything other than the number of tuples, such as split-counters.

The simplification not only improves scalability by eliminating retries and fences but also makes Master-Tree a more *maintainable* non-blocking data structure. While non-blocking algorithm is the key to higher scalability in many-cores, a complex non-blocking algorithm that consists of various atomic operations and memory fences is error-prone and hard to implement, debug, test, and even reason about its correctness [31]. Even a non-blocking algorithm published in a well-thought paper and an implementation in a widely used library are sometimes found to contain a bug after a few years [5]. Making the algorithm trivially simple and robust thus has tremendous benefits for real database systems.

Cursors do a similar trick described in Appendix C. Finally, we point out that the idea of Foster-Twins also applies to other dynamic tree data structures.

9. EXPERIMENTS

This section provides an empirical evaluation of FOEDUS to verify its efficiency and scalability. We have two sets of experiments; in-memory and NVRAM. The in-memory experiments evaluate the performance of FOEDUS as an in-memory database while the NVRAM experiments evaluate the performance when a part of the data reside NVRAM.

9.1 Setup

We run our experiments on two servers detailed in Table 1. The two servers have identical CPUs and OS. The only difference is the number of CPU sockets; DragonHawk has 4 times more CPUs.

We have implemented FOEDUS as a general embedded database library written in C++. All experiments presented in this section use the standard TPC-C benchmark. In each evaluated system, we assign every worker thread to its own *home* warehouse as done in the systems we compare with [9, 25, 30]. This also means that the scale factor (the number of warehouses) is same as the number of threads. TPC-C

Table 1: Experimental Environments

Model	HP DL580	HP DragonHawk
Sockets	4	16
Cores (w/HT)	60 (120)	240 (480)
CPU	Intel Xeon E7-4890 v2 @ 2.80GHz	
CPU Cache	L2 256KB/core, L3 38MB/socket	
DRAM	3 TB	12 TB
OS (Kernel)	Fedora 19 (3.16.6)	RHEL 7.0z (3.16.6)

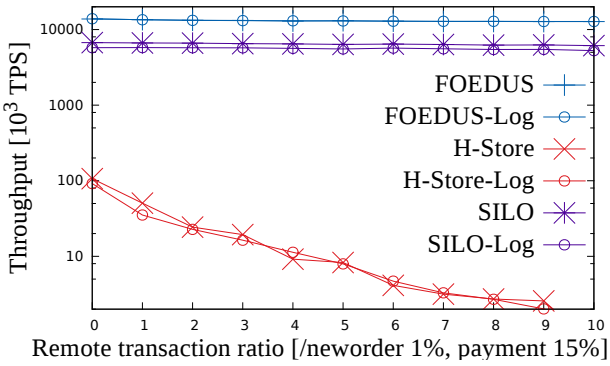

Figure 7: In-memory TPC-C with varied remote transaction fractions on DragonHawk. Remote-fraction=1 is the regular TPC-C. '-Log' writes out transactional logs to NVRAM (5 μs latency).

has two kinds of transactions that might access *remote* warehouses, `payment` and `neworder`. Some experiments vary the fraction of these transactions touching remote warehouses.

We retrieved the latest versions of the systems and compiled them by the latest compilers with highest optimization levels. We also carefully tuned each system for the environments, varying several parameters and choosing the best-performing ones. In fact, we observe throughput of most, but not all, systems evaluated in this section even better than the numbers in prior publications [14, 30, 32].

Above all, we configured `H-Store`, the academic branch of VoltDB, with help from the H-Store team. H-Store runs independent *sites*, each of which consists of a native execution engine written in `C++` and a `Java` VM that hosts users' stored procedures. We allocate 8-12 sites and set the number of partitions in each site to 8 in order to avoid the scalability issue of Java GC. All clients are non-blocking and configured to maximize H-Store's throughput.

All experiments use the `serializable` isolation level.

9.2 In-Memory Experiments

The first set of experiments places all data in DRAM to compare FOEDUS with the best performance of in-memory databases. We evaluate three systems; FOEDUS, H-Store, and SILO. In these experiments, we turn on/off transactional logging in each system. When we turn on transactional logging, we use our emulated NVRAM device described in next section with 5 μs latency. Both SILO and FOEDUS assign 12 worker threads per socket.

Figure 7 shows the results of experiments on DragonHawk. The x-axis is the fraction of each `orderline` stored in remote warehouses, which we call `remote-ratio` below. We also vary the fraction of remote payments accordingly (remote-ratio * 15%, up to 100%). Prior work [30] did an equivalent evaluation. We denote remoteness as a fraction of remote *orderlines*, not remote *transactions*. As each `neworder` transaction issues 5 to 15 line-items, `remote-ratio=1` corresponds to about 10% remote transactions.

In the regular TPC-C setting (`remote-ratio=1`), FOEDUS achieves 400 times higher throughput over H-Store. The throughput of H-Store significantly drops when some transactions touch remote warehouses because H-Store triggers distributed transactions with global locks for such transactions. In fact, more than 90% of transactions in H-Store abort with higher remote ratios. This makes H-Store perform even slower on DragonHawk except `remote-ratio=0`. On the other hand, FOEDUS and SILO suffer from only modest slowdowns because of their lightweight concurrency control and cache-conscious data structures, resulting in several orders of magnitude faster performance for larger remote ratios.

Table 2: In-memory TPC-C Scale-up (Remote=1). Total/per-core Throughput [kTPS]. Log ON.

Server	FOEDUS	SILO	H-Store
DL580	3659	1868	55.3
(per-core)	(61.0)	(31.1)	(0.92)
DragonHawk	13897	5757	35.2
(per-core)	(57.9)	(24.0)	(0.15)

FOEDUS also performs consistently faster than SILO. FOEDUS's throughput is twice that of SILO because Master-Tree eliminates most transaction aborts and all global search retries as well as hand-over-hand verification steps in interior pages. Table 2 compares the entire and per-core throughput of the systems on DL580 (60 cores) and DragonHawk (240 cores). Although both FOEDUS and SILO scale well, the relative performance difference modestly but consistently grows as the number of core grows. In all remote-ness settings, FOEDUS performs 2.4x faster than SILO on 240 cores and 2x faster on 60 cores. FOEDUS is less affected by transactional logging, too. Whether FOEDUS writes out logs to `tmpfs` or NVRAM, it only issues a few large sequential writes for each epoch, thus the latency has almost no impact.

These results verify the scalability of FOEDUS on a large number of cores with highly non-uniform memory accesses.

9.3 NVRAM Experiments

The second set of experiments places the data files and transactional log files in an emulated NVRAM device. We again evaluate three systems, but this time we do not evaluate SILO because it is a pure in-memory database. Instead, we compare FOEDUS with H-Store with anti-caching (75% eviction threshold) and a variant of `Shore-MT` [14, 16, 25, 32].

9.3.1 NVRAM Emulator

As discussed in Section 2, the latency of future NVRAM devices widely varies. We therefore developed a simple emulator to emulate NVRAM devices with an arbitrary latency. Our emulator leverages the fact that all three systems we evaluate use standard filesystem APIs to access and manipulate log and data files. It emulates NVRAM using DRAM and by extending a Linux memory file system (i.e., `tmpfs`) to inject software-created delays to each file read and write I/O operation. It creates delays using a software spin loop that uses the `x86 RDTSCP` instruction to read the processor timestamp counter and spin until the counter reaches the intended delay. It also allocates and places data in emulated NVRAM in a NUMA-aware fashion. Our NVRAM emula-

tor is completely transparent and requires no source code changes to applications as long as they access NVRAM via the filesystem API.

Compared to previous hardware-based approaches for emulating NVRAM [11], our emulation approach requires no special hardware or firmware, and most importantly it is not limited to single-socket systems. Thus, our approach enables us to evaluate larger scale systems comprising thousands of cores and huge NVRAM.

9.3.2 Transaction Throughput

Figure 8 shows TPC-C throughput of the three systems in DragonHawk with various NVRAM latencies that cover all expectations and types of NVRAM (100 ns to 50 μs).

FOEDUS: Because most of its writes are large and sequential, `FOEDUS` is nearly completely unimpacted by the higher NVRAM latency. Even though the size of snapshot-cache was only 50% of the entire data size, the page-based caching effectively reduces the number of read accesses to NVRAM.

FOEDUS-NoSC: To confirm the effects of the snapshot-cache, `FOEDUS NoSC` always reads snapshot pages from NVRAM. As the result shows, the throughput visibly deteriorates around 10 μs and hits major I/O bottlenecks around 30 μs. In other words, when the snapshot-cache is too small or its cache replacement algorithm (so far a simple `CLOCK`) does not capture the access pattern, the performance of FOEDUS highly depends on whether the read latency of the NVRAM device is $\leq$10 μs or not. Another interesting result is that the throughput of `FOEDUS NoSC` is lower than `FOEDUS` even when the latency is negligibly low (e.g., 100 ns). This is because snapshot-cache also avoids the overhead to invoke filesystem API and copying the page (4 KB) itself, which is expensive especially when the snapshot page exists in another SOC. Therefore, SOC-local snapshot-cache is beneficial regardless of the NVRAM device's latency.

H-Store: On the other hand, anti-caching in H-Store incurs high overheads due to its need to maintain the status of eviction for each tuple and frequent I/Os to NVRAM. Java stored procedures with query optimization help developers but have overheads, too. Furthermore, anti-caching requires an additional JNI call when the transaction retrieves evicted tuples (*dependencies*). As a result, FOEDUS's throughput is more than 100 times higher than that of H-Store.

Note: The current implementation of anti-caching in H-Store has some unstability when a transaction retrieves a tuple from anti-cache in remote partitions. Hence, this experiment runs H-Store without remote transactions (remote=0), which is guaranteed to only improve H-Store's performance (according to Figure 7, about 2x) rather than degrade.

Shore-MT: Shore-MT keeps about the same performance as H-Store in all latencies. Its throughput drops only when NVRAM latency is beyond the expected range. We observed 10% slow down with 0.1 ms latency, 70% with 0.5 ms, and 150% with 1 ms. However, even an SSD device in the current market has less than 0.1 ms latency. In other words, Shore-MT benefits from replacing HDDs with SSDs but not from replacing SSDs with NVRAMs because its disk-based architecture incurs high overheads that overshadow the differences in μs order. In fact, we did not observe performance differences by varying the size of its bufferpool. The figure shows the throughput of Shore-MT whose bufferpool size is 75% of the data file, but 50% bufferpool size had almost identical throughput in all realistic latency settings

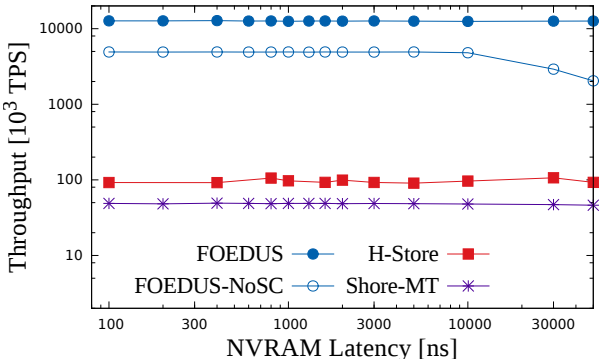

Figure 8: NVRAM TPC-C with varied NVRAM latency on DragonHawk, Remote=1 except H-Store (Remote=0). Transactional Logging on.

for NVRAM. Shore-MT does not scale up to the large number of cores, either. Due to contentions in locking and logging modules, we had to limit the number of worker threads to 26 to get its peak performance, which is consistent with the numbers [32] observes.

9.3.3 Log Gleaner Throughput

Next, we measure the performance of the log gleaner. Although the log gleaner runs concurrently to transactions without any interference, the log gleaner must catch up with the rate at which transactions emit logs. Otherwise, too many volatile pages must be kept in-memory, eventually causing transactions to pause until the log gleaner catches up. The same applies to log writers. If loggers cannot write out logs to files fast enough, the circular log buffers of workers become full and cause paused transaction executions.

Table 3: FOEDUS Log Gleaner Throughput.

Module	Throughput $\pm$`stdev` [10^6 logs/sec]
Mapper	3.39 ± 0.13 per instance
Reducer	3.89 ± 0.25 per instance
Logger	12.2 ± 1.3 per instance

Table 3 shows the throughput of each module in the TPC-C experiments on NVRAM. Assuming 50k transactions per core per second, which emit about half a million log entries per core per second, this result indicates that 1 or 2 log mappers and 1 log reducer can catch up with 8 worker threads. For example, in the environments we used in the experiments which have 15 cores per CPU, we assign 12 worker threads, 2 log mappers, and 1 log reducer per socket.

One logger in FOEDUS keeps up with 24 worker threads. Despite the similar design in transactional logging, SILO reports a different balance; 1 logger thread can handle up to only 7 worker threads [30]. The improvement is due to FOEDUS's design to directly write out worker buffers to log files. A logger in FOEDUS does not have its own buffer aside from a small 4 KB buffer to align boundary cases for direct I/O. On the other hand, SILO copies logs from worker buffers to loggers buffers, doubling the overhead.

These results on NVRAM verify that 1) FOEDUS achieves a high performance on NVRAM equivalent or better than in-memory systems, and that 2) FOEDUS's logging and snapshotting framework can handle a large influx of transaction logs with a sizable number of cores per socket.

9.4 Where the time goes

This section analyzes the current bottlenecks in FOEDUS. Figure 9 shows the results of CPU profiling in DragonHawk during the TPC-C executions in the in-memory setting and the NVRAM setting with a latency of 5 μs. In this figure, `Lock-Acquire` contains the *wait* time of conflicting locks.

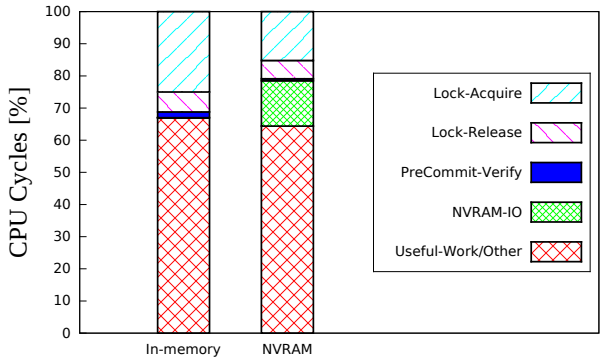

Figure 9: CPU Profile of FOEDUS during TPC-C.

Unlike the famous analysis in Shore [15], the vast majority of CPU cycles are spent in actual transactional processing in both settings. This is a significant improvement especially considering that the prior analysis was about overheads on single-threaded execution. FOEDUS effectively keeps locking and logging contentions small even on the unprecedentedly large number of cores. It is exactly one of the goals to completely separate transactional modifications in volatile pages from log writers and the log gleaner. They can thus construct snapshot pages in parallel to transaction executions without any races or frequent flushes.

The CPU profile also shows an interesting benefit of the stratified snapshots. Although the NVRAM execution adds the cost to access NVRAM (`NVRAM-IO`), it reduces lock contention (25.5% $\rightarrow$ 15.2%) and the overheads of verification (1.8% $\rightarrow$ 0.6%). As Section 7 discusses, snapshot pages are immutable. Once the transaction followed a snapshot pointer, it never has to check record TIDs, page versions, or page locks except for the single pointer to jump from the volatile world to the snapshot world. This property dramatically reduces the number of read-sets to verify in the pre-commit phase and speeds up reading static pages. This is especially beneficial for OLAP workloads, discussed next.

9.5 OLAP Experiments

TPC-C is an insert-heavy OLTP workload. In order to evaluate the efficiency of FOEDUS for OLAP workloads, we also ran an altered TPC-C that executes analysis-style queries on orders of magnitude larger data. We found that FOEDUS achieves orders of magnitude larger throughput compared to H-Store. Further, we observed an interesting aspect of the dual page architecture; FOEDUS performs **faster**, not slower, if it drops all volatile pages (hot-data) from DRAM and has only snapshot pages (cold-data). This happens because verification costs are more significant in OLAP settings and snapshot pages are more efficient to read than equivalent volatile pages thanks to its immutability and density. Appendix A describes this experiment in details.

9.6 Discussions

Finally, we discuss a few issues that are not explicitly visible in the performance numbers.

First, we observed and also confirmed with the original authors that some of the bottlenecks in H-Store are attributed to the `Java` runtime. H-Store/VoltDB allows users to write their transaction in Java stored procedures that communicate with the native execution engine via JNI. This causes expensive *cross-boundary* costs as well as the need for JavaVM's GC. Although garbage collection could be expensive in any language, Java GC is especially known for its poor scalability for small and frequent transactions like OLTP on large NUMA machines. On the other hand, FOEDUS supports only `C/C++` interface, which does not have this issue but is not as productive as Java in terms of developers' productivity. Providing both developer productivity and system efficiency requires further research.

Second, there is another approach to bring the performance of purely in-memory databases to NVRAM. One can run a purely in-memory database, such as SILO, on a `virtual memory` backed by NVRAM. This approach does not require any code change in the database, but has a few fundamental limitations. The most important limitation is that it cannot recover from a system crash without redoing all transactional logs. A modification to a virtual memory right before the crash might or might not be made durable. Unless the NVRAM device and the OS provide additional guarantees, such as *write-order-dependency* and *flush-on-failure*, the database needs an extra mechanism to handle recovery. Further, the OS must wisely choose memory regions to swap out and modify mapping tables (e.g., hash-table) in a thread-safe fashion just like database bufferpools, which poses challenges on scalability and complexity. In fact, [14] observes that virtual memory performs poorly as a bufferpool mechanism. One reason is that an OS has no application-domain knowledge. Nevertheless, it is still an interesting direction to overcome these limitations without completely changing the architecture of existing DBMSs for the sake of software compatibility.

10. CONCLUSIONS

The advent of NVRAM and thousand CPU cores demands new database management systems. Neither disk-based nor in-memory databases will scale. We presented the design of FOEDUS, our from-scratch OLTP engine that scales up to a thousand cores and fully exploits NVRAM. Our experiments on a 240-core server revealed that FOEDUS performs orders of magnitude faster than state-of-the-art databases and is substantially more resilient to contention.

There are several topics in FOEDUS we will further investigate. Although the performance results on the 240 core server imply that FOEDUS will not peak out its performance at least for high-hundreds of cores, we will run experiments on a thousand cores as soon as possible. We plan to improve the resilience to even higher contention, the main drawback of OCC, by autonomously combining pessimistic approaches depending on workloads.

Another issue we plan to work on is to improve the automatic partitioning algorithm in the log gleaner. The current algorithm simply checks which node has modified the page recently. This sometimes misses the best partitioning, for example a page that is occasionally *modified* in node-1 but very frequently *read* in node-2. However, we must avoid requiring each read operation to write something to a global place, even just statistics. We have a preliminary idea on efficient collection of statistics for partitioning and will evaluate its accuracy and overheads.

APPENDIX

A. OLAP EXPERIMENTS

Section 9 used the standard TPC-C to evaluate the performance of FOEDUS, which is an insert-heavy OLTP workload. This section evaluates the efficiency of FOEDUS for OLAP workloads, using an altered TPC-C workload as a micro-benchmark. The OLAP workload uses `order-status`, a read-only transaction in TPC-C that consists of three cursor accesses; 1) retrieving customer-ID from customer name, 2) retrieving last order-ID from customer-ID, and 3) retrieving line items of the order.

We also increased the average number of tuples this query reads. Each order in TPC-C contains 5 (`MIN_OL_CNT`) to 15 (`MAX_OL_CNT`) line items. Analytic queries usually read hundreds or thousands of tuples. We thus varied `MAX_OL_CNT` from 15 to 500. The initial size of the database is increased accordingly, turning it to an OLAP-style database of 0.5 TB.

Throughput: Figure 10 shows the throughputs of the OLAP workload. In this experiment, we disabled H-Store's anti-caching to evaluate its best performance, assuming that DRAM is larger than the data set. The workload has no distributed transaction, either. We tested two configurations of FOEDUS to evaluate different aspects of its architecture. `FOEDUS-Hot` keeps all volatile pages, thus no snapshot pages on NVRAM are accessed. `FOEDUS-Cold`, on the other hand, drops all volatile pages, thus the transactions read snapshot pages from NVRAM (5 μs latency) with a large snapshot cache.

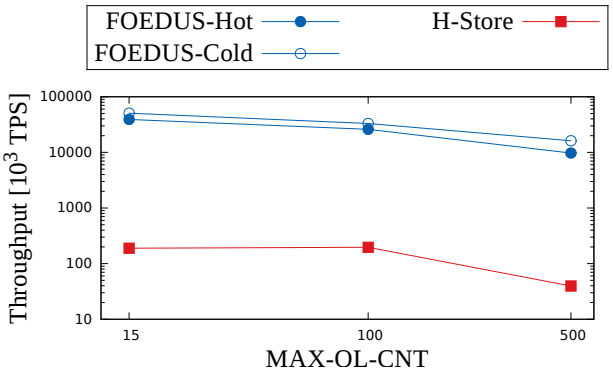

Figure 10: OLAP Workload (`order-status` with many more records). FOEDUS-Hot keeps all volatile pages while -Cold has only snapshot pages.

FOEDUS achieved orders of magnitude higher throughputs than H-Store. This is not a surprising result because H-Store is not designed for OLAP workloads. A far more interesting observation is that `FOEDUS-Cold` is **faster** than `FOEDUS-Hot` for about 30-70% in this cursor-heavy OLAP workload. We consistently observed the speed-up unless the snapshot cache is small or the workload consists of a large number of random seeks (e.g., `stock-level`, which is dominated by uniformly random lookups on `stock` table). This is surprising because the general assumption in hybrid databases is that cold-data are slower to manipulate. To analyze how this happens, we took CPU profile of FOEDUS.

Profile: Figure 11 shows the CPU profile in this experiment. A cold execution adds the I/O cost on NVRAM, but there are a few costs that are substantially lower instead.

First, as reported in the NVRAM experiment, reading

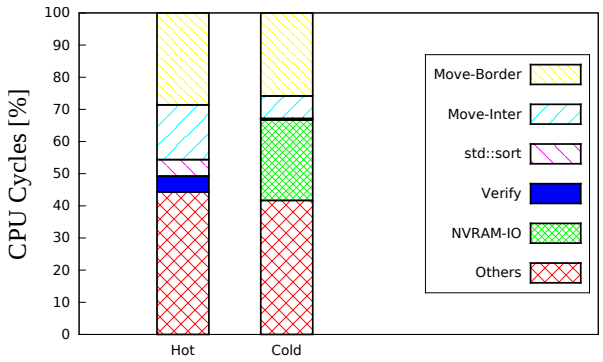

Figure 11: CPU Profile of FOEDUS during a cursor-heavy OLAP workload on 0.5 TB data.

from snapshot pages reduces the verification cost (5.0% → 0.5%), which is much more significant in this OLAP setting.

Second, the snapshot versions of Master-Tree's border pages are always fully sorted. Hence, a cold execution skips the sorting step to scan a border page. Appendix C explains how cursors work in Master-Trees.

Third, stratified snapshots contain more dense intermediate pages than the volatile pages. Fill factor is almost 100% and it does not have foster twins. There are simply fewer intermediate pages to read.

Finally, the snapshot cache serves as a *local* replica, thus reading from it causes only NUMA-local accesses. On the other hand, mutable volatile pages cannot be replicated easily. Especially on a larger database (`MAX_OL_CNT=500`), this causes a significant speedup (70%) due to memory bandwidth limitation across sockets.

In sum, the complete separation of volatile and snapshot pages has a few additional performance benefits for OLAP workloads. These observations verify the design of dual pages, whose main goal is to scale beyond the size of DRAM and also efficiently process big-data OLAP applications.

B. NON-SERIALIZABLE TRANSACTIONS

FOEDUS provides a few weaker isolation levels for use-cases where the user prefers even higher efficiency and scalability at the cost of non-serializable results. This appendix section discusses a popular isolation level, *snapshot isolation* (`SI`).

B.1 Snapshot Isolation (SI)

`SI` guarantees that all reads in the transaction will see a consistent and complete image of the database as of *some* time (time-stamp). FOEDUS's `SI` requires that the time-stamp of an `SI` transaction in FOEDUS must be the epoch of the latest (or older) snapshot. Unlike a database that assigns a transaction's time-stamp at its beginning (e.g., Oracle), the granularity of timestamps in FOEDUS is much coarser.

B.2 How FOEDUS Executes SI

The execution of `SI` transactions in FOEDUS is fairly simple because a stratified snapshot by itself is a complete, consistent, and immutable image of the entire database as of a single point of time (snapshot epoch). For reads, a transaction merely follows the root snapshot pointer for all storages as of the latest snapshot. It does not have to remember read

sets nor pointer sets. For writes, it does the same as serializable execution.

FOEDUS's multi-versioning is based on pages. Unlike tuple-based multi-versioning databases where a new key insertion might be physically modifying the page, reading and traversing immutable snapshot pages in FOEDUS does not need any additional logic (e.g., skipping a tuple with a newer time-stamp) even for cursor accesses. The transaction merely reads locally-cached snapshot pages and processes them as if the system is single-threaded. Further, because all data as of the exact epoch reside in the page, reading each tuple does not need to access *UNDO logs*, which is the major source of I/O overheads and CPU cache misses in traditional MVCC databases. In fact, FOEDUS does not have any UNDO logs because FOEDUS applies the changes to data only after the transaction is committed.

The drawback, however, is the coarser granularity of time-stamps, which is more likely to cause *write skews*. The reason for us to choose this design of `SI` is that, because FOEDUS provides a highly scalable and efficient serializable execution, a user application of FOEDUS would choose `SI` probably because of a significantly high priority on performance rather than recency of reads, or full serializability. Hence, the drastically simple and efficient protocol above would best serve the usecases.

B.3 Instant Recovery for SI

As described in Section 7.7, FOEDUS runs the log gleaner after restart, taking up to a few minutes when the previous execution did not run log gleaner during the shutdown, probably because of a system crash. When the user needs to run only read-only SI transactions, FOEDUS can even skip this step because stratified snapshots are immediately queriable databases by themselves unlike traditional checkpoints. This is a restricted form of *instant recovery* [24] that provides additional fault tolerance at no cost.

C. CURSORS IN MASTER-TREE

Master-Tree provides a cursor interface for range access, which is the *raison d'être* of B-trees. A cursor in Master-Tree is designed with the same principles as point-queries discussed in the main body, namely no global retries or hand-over-hand verifications.

Algorithm 3 summarizes how Master-Tree scans records forward. The key idea is again the foster-twin and the stable key-region of pages.

In intermediate pages, the cursor remembers up to which key it scanned so far (`last`). The next page to follow is correct *if and only if* the page's low fence key is equal to the key. If it does not match, there was some reorganization in the page, but it is guaranteed that the separator still exists in the page. Thus, it locally re-locates the position rather than aborting the transaction or globally retrying from the root. Observe that the algorithm is correct even if the intermediate page is now moved or retired. The cursor eventually reaches the exactly same set of border pages without locks or hand-over-hand verifications.

In border pages, however, we have to check the possibility of *phantoms*. Hence, the transaction remembers the key count of the page as of initially opening the page in addition to per-tuple read-sets. Also, like Masstree [22], Master-Tree never changes the location of existing records to simplify concurrency control. It always appends a new record at the end of the page. Thus, keys are not sorted in border pages. To efficiently scan the page, a cursor calculates the order of keys in the page when it follows a pointer to the page. Bw-Tree has a similar step [20]. This sorting is skipped in snapshot pages because log gleaner always construct a page that is fully sorted.

Algorithm 3: Summary of Master-Tree's cursor scan

```
Procedure cursor_next()
  move_border(last route);
Procedure move_inter(route)
  pos = route.pos + 1;
  if pos > page.count then wind up and move;
  if page.fences[pos] ≠ route.last then relocate pos;
  next = page.pointers[pos];
  if next.low ≠ route.last then local retry;
  route.pos = pos, route.last = next.high;
  follow(next);
Procedure move_border(route)
  pos = ++route.pos;
  if pos ≥ page.count then wind up and move;
  if Points to next layer then follow(next layer);
  Optimistic-Read(page.records[route.order[pos]]);
Procedure follow(page)
  route ← new route(pos = 0, last = page.low);
  if page is border and volatile then
    if page is moved then follow(foster twins);
    /* Verified during precommit                */
    remember page.count;
    route.order = sort(page.keys);
  end
  move_border/inter(route);
```

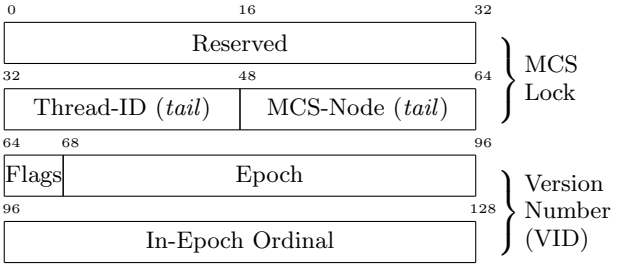

Figure 12: FOEDUS's TID Layout.

D. DETAILED DATA LAYOUTS

This appendix section provides a concrete layout of a few data structures explained in the main body. Although the main idea and mechanism will stay the same, the exact data layout might change over time. We thus recommend to check the latest source code of FOEDUS.

D.1 TID

`TID` (Transaction-ID) is the metadata placed next to every tuple. Our TID layout is slightly different from SILO to accommodate more information and to employ *MCS-locking* [21], which avoids remote-spinning for higher scalability. FOEDUS's TID consists of two parts, each of which is 64-bits as shown in Figure 12. The first part, `MCS-Lock`, contains the ID of the *tail-waiter* of the tuple and address of its lock-node. A transaction atomically swaps (e.g., `x86`

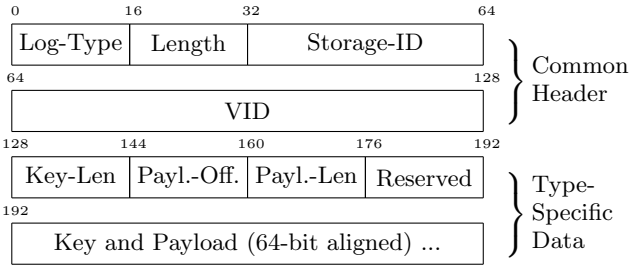

Figure 13: Example Log Record: Master-Tree Log.

`lock xchg`) this portion to install its lock request and spins locally on its own node as done in typical MCS-locking.

The second part, `VID`, is the version number protected by the lock. All logical status of the record are stored in this part, including `moved` flag, `deleted` flag, `next-layer` flag, and `being-written` flag that indicates whether the record is half-written. Most code in FOEDUS only deals with this part. For instance, an optimistic read just reads VID, take a `consume` or `acquire` fence, then read the tuple. Logical log records, described next, also contain only VIDs.

D.2 Log Record

Figure 13 exemplifies the logical log record in FOEDUS.

Unlike traditional LSN-based databases, it does not contain any physical information, such as page-ID and undo-chain. All log records are 64-bit aligned with appropriate paddings. In addition to performance benefits, such as efficiently slicing a 64-bit key for Master-Tree with aligned pointers, the alignment is essential to contiguously fill an arbitrary part of unused log regions with a special *filler* log for each Direct-IO operation.

D.3 Read-Write Set

Each transaction maintains an array of the following read- and write-sets.

```
struct Read {
  VID     observed_;
  StrID   storage_;
  void*   address_;
  Write*  related_;
};
```

```
struct Write {
  MCSNode locked_;
  StrID   storage_;
  void*   address_;
  Read*   related_;
  Log*    log_;
};
```

Figure 14: Read- and Write-Set Structure.

The `related_` field is set when an operation both reads and writes to a tuple, which is a very common case. The bi-directional chain is useful to avoid a few redundant work during the pre-commit phase, such as tracking moved records.

A write set additionally contains the MCS lock node of the transaction, which is needed to unlock. Both the log pointer and the MCS node always point to the transaction's own memory.

E. H-STORE CONFIGURATION

This appendix section provides the full details of the H-Store configuration for verifying and reproducing our performance comparison with H-Store.

Versions and Patches: Following recommendations from the H-Store team, we used the latest version of H-Store as of Dec 2014 with a patch provided by the H-Store team to partially fix the unstability issue of anti-caching mentioned in Section 9. The H-Store team also recommended us to disable remote transactions to measure the best performance of anti-caching, thus we disabled remote transactions in all experiments that use H-Store's anti-caching.

Table 4: H-Store General Configuration.

Parameter Name	Configuration
site.jvm_asserts	false
site.cpu_affinity	true
client.blocking	false
client.memory	20480
site.memory	30720
global.memory	2048
client.txnrate	10000
client.threads_per_host	[Adjusted to the throughput]
hosts	localhost:0:0-7;localhost:1:8-15;...

Table 5: H-Store Command Logging Configuration.

Parameter Name	Configuration
site.commandlog_enable	true
site.commandlog_dir	[The NVRAM filesystem]
site.commandlog_timeout	500

Table 6: H-Store Anti-caching Configuration.

Parameter Name	Configuration
evictable	HISTORY,CUSTOMER, ORDERS,ORDER_LINE
site.anticache_dir	[The NVRAM filesystem]
site.anticache_enable	true
site.anticache_threshold_mb	1000

General Configuration: In both in-memory and NVM experiments, we used the parameters listed in Table 4 to run the experiments.

Based on recommendations by the H-Store team, we ran the experiments with turning CPU affinity ON/OFF and observed that enabling CPU affinity gives a better performance. We always kept the JVM heap size within 32 GB so that the JVM can exploit `Compressed-Oops` [2] for its best performance. The JVM we used is 64-bit OpenJDK 24.51-b03 (1.7.0_51), Server VM.

For TPC-C benchmark, we modified the `tpcc.properties` file. The only changes from the default setting are the fraction of remote-transactions. We vary `neworder_multip` (`multip` stands for multi-partition) and `payment_multip` as described in Section 9. When remote=0, we also set `false` to `neworder_` and `payment_multip_mix` so that H-Store completely avoids the overhead of distributed transactions.

For the OLAP experiment in this appendix, we varied the transaction weights and `MAX_OL_CNT` in `TPCCConstants.java`. We also slightly modified the data loader to flush more frequently to handle an increased number of records.

Command Logging Configuration: In the NVM experiment and in-memory experiment with logging, we enabled H-Store's command-logging feature, placing the log files in our emulated NVRAM device as listed in Table 5.

Anti-caching Configuration: In the NVM experiment, we used H-Store's default implementation of anti-caching backed by `BerkeleyDB` on `history`, `customer`, `orders`, `orderline` with 1 GB eviction threshold (or 75%) per site as listed in Table 6.

Acknowledgments

We appreciate kind helps from the H-Store team, especially Joy Arulraj, Michael Giardino, and Andy Pavlo, for configuring, debugging, and tuning H-Store as well as suggestions to the draft of this paper. We thank Harumi Kuno and numerous colleagues in HP for editorial help, Haris Volos for building the NVRAM emulator, and William Kuszmaul for studying Cuckoo Hashing. We owe Paolo Faraboschi, Gary Gostin, and Naveen Muralimanohar for their insights on hardware characteristics of NVRAM and future servers. Davidlohr Bueso, Aswin Chandramouleeswaran, and Jason Low tuned the linux kernel for DragonHawk and gave us insightful inputs to make FOEDUS more scalable. We thank the DragonHawk hardware team to setup the system and the anonymous reviewers for insightful comments.

References

[1] `http://www8.hp.com/hpnext/posts/discover-day-two-future-now-machine-hp`.

[2] `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`.

[3] Process integration, devices, and structures. In *International Technology Roadmap for Semiconductors*, 2007.

[4] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers (keynote). In *FAST*, 2014.

[5] S. Burckhardt, R. Alur, and M. M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *SIGPLAN Notices*, 2007.

[6] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, et al. Phase change memory technology. *J. of Vacuum Science Technology B*, 28(2):223–262, 2010.

[7] D. Chakrabarti, H. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *SIGPLAN OOPSLA*, 2014.

[8] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. R. Dulloor. A prolegomenon on oltp database systems for non-volatile memory. In *ADMS*, 2014.

[9] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14), 2013.

[10] 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. In *SIGMOD*, 2013.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of EuroSys*, 2014.

[12] A. Eldawy, J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11), 2014.

[13] G. Graefe, H. Kimura, and H. Kuno. Foster b-trees. *TODS*, 37(3):17, 2012.

[14] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-memory performance for big data. *PVLDB*, 8(1), 2014.

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

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

[17] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: a study of caching and tiering approaches. In *FAST*, 2014.

[18] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Computer Architecture News*, 37(3), 2009.

[19] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo, et al. A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2-x bilayer structures. *Nature Materials*, 10(8), 2011.

[20] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

[21] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium*, 1994.

[22] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Eurosys*, 2012.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases.

[25] 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*, 2011.

[26] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of MICRO*, pages 14–23, 2009.

[27] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2), 2012.

[28] T. Sakata, K. Sakata, G. Höfer, and T. Horiuchi. Preparation of nbo2 single crystals by chemical transport reaction. *Journal of Crystal Growth*, 12(2):88–92, 1972.

[29] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453, 2008.

[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of SOSP*, 2013.

[31] J. van den Hooff. *Fast Bug Finding in Lock-Free Data Structures with CB-DPOR*. PhD thesis, MIT, 2014.

[32] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[33] G. Weikum and G. Vossen. Transactional information systems, 2002.

[34] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *HPCA*, 2011.

[35] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3), 2014.

[36] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *OSDI*, 2014.

[37] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *SOSP*, 2014.