Abstract—In this report, we describe a number of Hardware Transactional Memory (HTM) designs and their basic mechanisms used for better programmability and higher performance than conventional synchronization techniques based on locking. We compare the systems considering their programming model, hardware design challenges, transactional dataset constraints and forward progress guarantees.

As high-speed networks emerge for the rack-scale systems [1], CPUs become a bottleneck and researchers start investigating optimistic concurrency control mechanisms to optimize different phases of the comprehensive distributed commit protocols [2]. To achieve high performance while maintaining strong data consistency in the system, we propose to leverage HTM design ideas to provide more efficient non-blocking operations on data objects stored in memory of different servers in a rack.

Keywords—architecture support, hardware transactional memory, synchronization

I. INTRODUCTION

To achieve high performance, modern microprocessors provide explicit hardware support for multithreading in the form of shared memory multiprocessors or multithreaded architectures [3]. Modern server workloads exhibit abundant thread-level parallelism [4] taking advantage of many-core and multi-socket systems.

Although multithreaded programs achieve higher performance, the speedup is usually sublinear. One of the reasons for such a behavior is the synchronization overhead. Synchronization is usually implemented with locks that guard critical sections. Locks enforce mutual exclusion in the critical section so that only a single thread at a time can enter. This can limit the application performance significantly or introduce deadlocks. Although a number of researchers investigated techniques for implementing lock-free data structures in software [5], experimental evidence shows that their performance is usually lower than the performance of their locking counterparts [6].

To unleash the full performance potential while keeping parallel programming as intuitive as possible, modern commercial processors provide hardware support for lock-free data structures [7], [8] that are more flexible than conventional hardware synchronization primitives, such as load-linked/store-conditional instructions [9]. This conceptual model, called Hardware Transactional Memory or HTM, offers programming constructs that provide transactional semantics for distinct code blocks. In particular, such atomic constructs can efficiently replace locks as the main synchronization mechanism and allow concurrency inside critical sections without losing correctness. Moreover, writing a complex multithreaded program with such non-blocking constructs is often easier than using fine-grain locks, while achieving comparable or higher performance.

In this report, we start with a description of the conventional synchronization approaches presenting programming ease and performance trade-offs in Chapter II. Then, we analyze a number of HTM proposals starting from the original work by Herlihy and Moss [10] that relies on explicit Instruction Set Architecture extensions and separate hardware structures that can be challenging for both programmers and hardware designers (Chapter III). The next paper presents the Speculative Lock Elision (SLE) [11] technique that dynamically removes unnecessary lock-induced serialization and enables highly concurrent multithreaded execution without undermining correctness (Chapter IV). Then, we describe the HW/SW co-design approach, taken by the LogTM system [12], that allows to overcome hardware-structure size limitations without significant augmentation of the core microarchitecture (Chapter V). We compare the designs in Chapter VI and propose using the HTM ideas and mechanisms to optimize comprehensive distributed commit protocols in Chapter VII. Chapter VIII concludes the paper.
II. TRADE-OFFS IN PARALLEL PROGRAMMING AND SYNCHRONIZATION

On modern highly parallel architectures such as chip multiprocessors, multithreaded programs can show significant speedups over their single-threaded counterparts. However, the speedups are commonly sublinear since only a part of the program can be parallelized. Synchronization blocks are one of the important serial fractions of the program that is difficult to parallelize. Application threads synchronize with critical sections that are software constructs that enforce mutually exclusive access among threads to shared data. Trivially satisfying serializability, programming with critical sections is appealing to the programmers due to its ease of reasoning about program correctness. Critical sections are commonly implemented using a software construct known as a lock. A lock is associated with shared data and determines whether the shared data is available.

Although most of the architectures offer instructions for efficient implementation of locks, e.g. test-and-test-and-set [13] or spin locks with exponential back-off [14], lock usage can introduce a number of performance or correctness problems. First, locks introduce complex trade-offs between programmability and performance. Using multiple locks for a single data structure, e.g. one per data object field, can help performance but makes it difficult to guarantee that the program is deadlock-free. Alternatively, coarse-grain locks, e.g. one lock per entire data structure, usually help in writing correct deadlock-free code but can significantly limit performance when multiple threads contend to acquire a particular lock. Furthermore, even in the absence of contention, locks introduce considerable overhead [15]. Second, a number of problems arise if threads can be interrupted in the middle of a critical section guarded by a lock. For example, priority inversion when a higher priority thread cannot proceed due to the lock held by a preempted lower priority thread [10]. Implementing concurrent data structures in a lock-free manner avoids most of such problems, however they may perform worse when implemented entirely in software than their locking counterparts [6].

In their work [11], Rajwar et al. gives a number of examples when enforcing mutual exclusion when accessing shared data structures is dynamically unnecessary, i.e. when the probability of a conflict is low. For instance, multiple threads can update different fields of a shared object and these updates do not conflict most of the time so that locks are redundant in most of dynamic executions. Such examples motivate using optimistic techniques, like Optimistic Concurrency Control [16] instead of conservative ones based on locking.

A. Transactional memory

Using critical sections trivially solves the problem of access serialization to shared data structures but often introduces a number of performance and correctness issues when implemented with locking mechanisms. Historically, databases used transactions as an intuitive model for coordinating access to shared data [17]. A transaction comprises a number of properties such as failure-atomicity, meaning that a transaction must execute to completion or appear not to have executed at all; consistency, which requires transactions to follow a protocol that guarantees a consistent view of the data objects; isolation, i.e. a transaction’s execution must not affect the result of concurrently executing transactions; and durability, meaning that when a transaction is committed, its results cannot be undone. With these properties guaranteed, it’s easier for database developers to reason about correctness.

Transactional memory exploits similar properties as a database transaction in order to keep programming abstractions simple and allow higher performance by leveraging implicit concurrency better than the other systems that rely on locking. Each transaction is represented by a code block that performs a number of operations in a way that all the operations appear to execute atomically preserving the system properties. Each transaction operates on a set of memory locations that constitute the transactional dataset and can be divided into a read set and a write set. The read set of a transaction consists of the memory locations read by the transaction while the write set consists of the memory locations written by the transaction. However, the exact transactional semantics of a transactional system can vary. As for the implementations of the database transactions, the TM system design choices include optimistic versus pessimistic concurrency control, eager versus lazy version management and eager versus lazy conflict detection and resolution.

In this work, we consider hardware implementations of transactional memory, or Hardware Transactional Memory (HTM). Hardware support for transactional memory is desirable when building a high performance system that executes programs leveraging TM constructs. To offer hardware support for a TM, the system should include the following mechanisms:

- **Programming model** to begin and commit a transaction, i.e. make the results of the transaction visible to the rest of the system (besides this particular transaction). The programming interface can also include a method to validate the dataset of the transactions and abort the transaction explicitly if necessary.
- **Concurrency control** mechanism that is usually optimistic for HTMs so that additional mechanisms are necessary for conflict detection and resolution.
- **Version management** that restricts the access to the dataset updates of an uncommitted transaction as well as an architecture state recovery.
- **Conflict detection** mechanism that determines when and how the system detects conflicts between concurrent transactional and non-transactional accesses.
- **Conflict resolution** mechanism that determines at which point and how the detected conflicts are resolved. This mechanism can include a contention manager or a fallback mechanism to prevent a livelock and provide forward progress guarantees.

To evaluate the design of the three systems, we consider in this report, we compare them considering a number of factors that describe the efficiency of aforementioned HTM mechanisms. First, from the perspective of the application developer, programmability, which considers the simplicity of the programming interface and its efficiency when writing a
correct program with TM constructs, and progress guarantees, which can be offered by the system, or the application developer that can potentially make the system less programmable. Second, from the perspective of the HTM hardware designers, how much augmentation is required to support the HTM mechanisms in hardware, e.g. complex modifications to the CPU core’s critical path microarchitecture is strongly undesirable as the circuits on the critical path are optimized to tightly fit in a fixed cycles budget. Further we refer to this factor as HW challenges and separately evaluate the maximum supported size of a transaction’s read and write sets, as Dataset size, implied by the limited hardware buffers used by different HTM mechanisms. These 4 factors allows us to qualitatively compare different HTM systems.

III. HARDWARE TRANSACTIONAL MEMORY BY HERLIHY AND MOSS

Herlihy and Moss [10] coined the term Hardware Transactional Memory and proposed the first programming model and hardware design for it. The authors proposed a system inspired by load-linked/store-conditional instructions [9], which guarantees that a store-conditional instruction succeeds only in the absence of a conflicting access to the data read by the load-linked instruction. However, the HTM transactions can operate with a larger number of memory locations as a dataset. This approach enables better performance and scalability than its locking counterparts, although the authors’ approach is quite complicated for a regular programmer. The programmers should know and be able to reason about the system behavior when writing code using transactional and non-transactional accesses. For example, the programmers need to estimate the size of the datasets, in cache blocks, of the transactions they use to compare it against the maximum dataset allowed by the hardware. Also, the application developers have to provide a software handler that will perform the architecture state recovery and contention management in the presence of conflicts.

Programming model. In Herlihy and Moss’ design, the hardware exposed explicit Instruction Set Architecture support for programming with serializable transactions. This includes transactional load and store, commit and abort instructions as well as a validate instruction. The beginning of a transaction is the execution of the first transactional memory access, while the transaction attempts to commit executing a commit instruction. The validate instruction checks if a conflict occurred since the beginning of the transaction. Validate instruction execution between the last transactional read and the first transactional write allows the system to guarantee the consistency of the transaction’s read set (opacity property [18]). Although explicit transactional memory accesses offer flexibility, e.g. to reduce a transaction’s dataset size by letting transactional and non-transactional accesses to intermix, this places a substantial burden on the application developer complicating the development process of a correct multithreaded program.

Version management. To track the state of the transactionally accessed memory blocks, Herlihy and Moss propose to use a separate transactional cache (further referred to as TCache) to store both old and updated versions of the cache blocks before the transaction attempts to commit (such an approach is called lazy version management). TCache is a fully-associative cache that is exclusive w.r.t. the core’s private data cache, i.e. each cache block can reside either in the regular data cache or in the TCache. The first transactional access to a memory location has to allocate two entries inside the TCache for bookkeeping the transactional updates and the old version of the block. Another modification to the core’s critical path is the logic to check the tags of the TCache in parallel with the tags of the data cache. As CPU designers put much effort into optimizing the critical path of the core, e.g. to fit the L1 cache lookup stage into 1 or 2 cycles, such HW augmentation can be challenging. To make the design more efficient, the authors propose to augment the TCache logic to work as a victim cache when the processor does not execute a transaction. The TCache’s size determines the number of the cache blocks that can be accessed within a transaction. If the transaction’s dataset exceeds the size of the TCache or in case of an interrupt or another system event, the transaction aborts. In case of an abort, the speculative state in the TCache is automatically discarded and the control is passed to the software handler that is specified by the developer. This handler is responsible for the register state recovery and contention management, e.g. exponential back-off.

Conflict detection. To track the conflicts with the transactional or non-transactional memory accesses w.r.t. to the transaction’s read and write sets, the authors propose to augment the snoopy bus protocol. The changes include a set of duplicate bus messages for transactional memory accesses, so that the receiving processor’s cache controller can respond to the conflicting transactional request with a BUSY signal. Having received a BUSY signal, the conflicting processor’s controller sets a bit meaning that the transaction will eventually abort. In case the conflicting core execute non-transactional memory access, the receiving core’s cache controller has to trigger such a bit to abort its transaction.

Conflict resolution. Although the conflicts are detected eagerly using the cache coherence mechanism, the conflict resolution is performed at the commit time or when the validate instruction is executed allowing wasted work of the transaction that continues its execution while it will eventually has to abort.

Although the design of the original HTM is far from optimal in terms of programmability, challenges for the hardware designers and progress guarantees, the design outperforms locking counterparts and shows better scalability [10] as it allows to operate with various data structures in a lock-free manner.

IV. SLE: LEVERAGING CPU SPECULATIVE STRUCTURES

Explicit ISA extension, as the one proposed by Herlihy and Moss [10], requires compiler support or more efficient higher level abstractions for the average programmer to be able to write correct multithreaded programs. Furthermore, explicit interfaces may limit the reuse of legacy software libraries inside HTM transactions. To solve the backwards compatibility
problem, Rajwar et al [11] proposed a hardware technique, called Speculative Lock Elision or SLE, to dynamically remove unnecessary lock/unlock operations to achieve higher performance by allowing concurrent execution of multiple threads inside the critical sections while preserving correctness of a program.

**Programming model.** With a hardware supporting SLE, programmers can continue using simple coarse-grain lock abstractions. According to the experimental evidence [19], SLE-based HTM shows higher performance when using few coarse-grain locks instead of multiple fine-grain locks. To offer backwards-compatibility to locks, SLE hardware use a hardware predictor to predict a silent-pair of stores and, then, leverages conventional speculative execution structures to speculatively elides the stores. The silent-pairs can be predicted with high confidence as these stores write to the same memory location and after the execution of both stores the value in the memory does not change. The instruction stream between the load to the same memory location (this load plays a role of the transaction’s beginning) preceding the first store of a silent pair and the last store of the pair (acting as the transaction’s commit verb) appear to execute atomically. The load and store operations belonging to this interval can be considered as members of the transaction’s read set and write set correspondingly, and hardware is responsible to detect any conflicting memory accesses from the other cores.

**Version management.** As in the previous HTM we considered, SLE exploits lazy version management, i.e. keeps the speculative state invisible before the transaction successfully commits. To reduce the design’s complexity, SLE reuses speculative mechanisms that CPU leverages for instruction-level speculation, e.g. speculative execution ahead of branch instruction resolution. For register state management, SLE propose to use an architecture register file checkpoint (only a single one is necessary), taken before the execution of the first instruction of the transaction, or the re-order buffer mechanism is used (further, we consider that the checkpoint mechanism is used). During transaction’s execution, the cache blocks in the transaction’s write set are buffered in the core’s write-buffer. If the transaction manages to commit, the write-buffer is marked as having the latest state (this involves setting a single bit), so that all the updates become visible atomically, and can be lazily drained into the cache.

**Conflict detection.** is performed eagerly using a conventional cache coherence protocol. As mentioned above, the write set of a transaction is buffered in the write-buffer. To track the dataset’s cache blocks, the core’s private data cache is augmented so that each cache block has an additional bit, access bit, that is flash-cleared when the transaction commits or aborts. The SLE implies that the conventional cache controller can be augmented to check if external requests conflict with the cache blocks with the their access bits set and trigger a misspeculation. Thus, the maximum read set size is constrained by the private cache’s size and associativity while the maximum write set size is limited by the write-buffer size. As in the Herlihy and Moss’ proposal, SLE aborts transactions in case a system event occurs or dataset, either read or write, exceeds buffering capabilities, however in case of transaction’s overflow, the speculative state is not discarded immediately but lock acquire is attempted (the memory location of the lock is in the transaction’s read set). If it is not possible, i.e. the lock has been acquired by another core, the transaction aborts.

**Conflict resolution.** The conflicts are resolved eagerly using the passive policy. Having received a conflicting request, the receiving core triggers misspeculation so that the architecture register is restored with the previously taken checkpoint, the access bit is flash-cleared for all the cache blocks, the speculative stores in the write-buffer are discarded and the control is transferred to the beginning of the transaction block. The microarchitecture can be extended to retry lock elision for a number of times. Once this threshold is achieved, the transaction aborts and retries acquiring the lock conservatively.

To keep a coarse locking programming model, offer backwards-compatibility and allow concurrency inside critical sections, SLE lets the hardware predict and elide lock/unlock operations speculatively keeping the interface exposed to the application developer the same as in the case when the coarse locks are used. Such an approach allows to reuse speculation structures and mechanisms of the modern out-of-order CPUs with minimal hardware modifications. However, the transaction’s dataset size is constraint by the sizes of the write-buffer and the private data cache. Finally, SLE can guarantee system progress as it can fall back into locking after a number of lock-free attempts. Provided that the original program was deadlock-free, the system’s forward progress is guaranteed.

V. **LogTM: SW/HW Co-Design**

As in the SLE proposal, Moore et al [12] describe a system that aims to avoid comprehensive hardware modifications, although the authors suggested using direct memory updates maintaining software-managed undo-logs. LogTM designers assumed that transactions commit much more frequently than abort and optimized their system for faster commits while aborts are processed by the system software. Such an approach allowed LogTM transactions to have larger datasets limited by the capacity of the directory and reduce the abort rate of the transactions while providing deadlock freedom guarantees.

**Programming model.** LogTM exposes simple methods to the application developer that include functions to begin, commit and abort a transaction. The system, e.g. thread packages, is responsible for initializing per-thread undo-logs and register a handler for conflict resolution. On the other hand, LogTM transaction logs are allocated in the same user-level virtual space that makes both speculative and non-speculative state visible to the debugging tools automatically that can simplify the development and debugging process.

**Version management.** To make commits faster, LogTM uses eager version management that implies direct memory updates while the old memory state is stored in the software-managed undo-logs. To support logging, the CPU architecture state is extended with a number of registers to keep the transaction-related state including pointers to the base and the end of the log. Each store in the transaction, except the stores accessing the cache blocks accessed before in the same transaction, cause a write back of the original cache block to
the in-memory log without invoking any software. To avoid a substantial increase in cache write bandwidth, the authors propose to add an additional write-buffer for log entries that will prevent transactions with a small dataset to write the log entries back to memory. With transactional updates already in place, transaction commit includes only flash-clearing the access bits in the cache and resetting the log pointer to its beginning. The architecture state is saved/restore in the same way as in the SLE design.

Conflict detection. The conflicts are detected eagerly as in the aforementioned works, however the design supports 2 separate modes of conflict detection for a normal and overflowed cases. In the normal mode, the dataset membership is tracked with 2 extra per-cache-block bits, R and W, for the members of the read and write set respectively that are used by the cache controllers to detect conflicts with external requests forwarded by the directory. In the overflowed mode, LogTM permits false positives by letting the receiving cache controller to trigger a conflict for any external requests forwarded by the directory (filtering is performed by the directory). To allow transactions to proceed even after their dataset overflows the core’s buffering capacity, the authors propose using an extended directory MOESI cache coherence protocol. To allow eviction of the dataset’s cache blocks, the protocol supports sticky M-state. If the block is evicted in M-state, the core uses transactional write-back to transition the cache block’s state to "sticky-M@coreId" state in the directory. While eviction of a block in the S state is silent so that the directory keeps the core in the sharer list, eviction of the block in O or E states is performed with a write-back to notify the directory so that false conflicts can be filtered at the directory level. The sticky states are cleaned lazily with an explicit CLEAN messages from the cores that were previously overflowed.

Conflict resolution. To reduce the abort rate of the transactions, LogTM forces the conflicting transaction to exponentially back-off and retry instead of aborting so that the receiving transaction can continue execution. To avoid deadlocks, LogTM uses TLR’s distributed timestamp method [19] so that all conflicting transactions can be strictly ordered by their age. To detect a potential deadlock, the conflicting transaction checks if the receiver transaction’s timestamp is earlier and the conflicting transaction has forced another earlier transaction to wait in the past. The mechanism is implemented using an additional bit in the cache controller. Although false positives are possible, such an approach allows to resolve most of the conflicts without aborting transactions, thus, avoiding wasted work and power consumption. Provided that potential deadlock is detected, hardware raises an exception that must be trapped by the handler previously registered by the system. If a transaction aborts, software walks the log in a LIFO order and resets the transactional CPU registers, e.g. log pointers while the architecture register file is recovered by the hardware from the previously taken checkpoint.

To simplify HTM hardware and allow large dataset transactions, LogTM exposes a simple programming interface to the applications developers and relies on hardware to detect and resolve the conflicts, if possible, while the software handles rare transactions aborts. With such an approach, LogTM does not require considerable modification to the core’s critical path microarchitecture, however private caches require a mechanism to flash-clear R and W bits on a transaction’s commit or abort event. Furthermore, LogTM allows transactions with a large dataset, limited only by the directory capacity, at the cost of possible false positives when detecting conflicts.

VI. Qualitative Comparison

As described in Section II, we compare the HTM designs w.r.t. the 4 factors: programmability, design challenges for the hardware, transaction dataset constraints and progress guarantees. Although explicit transactional memory access and manual dataset consistency validation, as in the proposal by Herlihy and Moss, allow to minimize the dataset size, such an approach requires care and sufficient knowledge from the application developer that makes writing a correct program quite a hard task. The SLE approach [11] allows to keep a well-understood coarse lock programming model that reduces programming effort. The LogTM [12] approach offers a simple programming interface to begin and commit a transaction that can be extended with other transactional constructs, such as nested transactions [20]. Also, the first two designs expect that the programmer considers the buffering capacity of the underlying HTM system so that she can avoid significant performance degradation in case of frequent aborts and fallbacks to software, e.g. locking. LogTM allows the developers to use transactions with large datasets (limited only by the directory capacity), however in the overflowed mode the system can suffer from false positives in the conflict detection mechanism.

Comparing the complexity of the hardware augmentation for each of the designs, we can qualitatively evaluate the changes applied to the critical path of the CPU core and the cache coherence protocol. The first paper introduces a separate exclusive transactional cache that requires cache block copying on transactional accesses and must be looked up in parallel with the conventional data cache that significantly impact the performance degradation in case of frequent aborts and fallbacks to software, e.g. locking. LogTM allows the developers to use transactions with large datasets (limited only by the directory capacity), however in the overflowed mode the system can suffer from false positives in the conflict detection mechanism.

Comparing the complexity of the hardware augmentation for each of the designs, we can qualitatively evaluate the changes applied to the critical path of the CPU core and the cache coherence protocol. The first paper introduces a separate exclusive transactional cache that requires cache block copying on transactional accesses and must be looked up in parallel with the conventional data cache that significantly impact the performance degradation in case of frequent aborts and fallbacks to software, e.g. locking. LogTM allows the developers to use transactions with large datasets (limited only by the directory capacity), however in the overflowed mode the system can suffer from false positives in the conflict detection mechanism.

Finally, no system, from the ones we considered, provides forward progress guarantees to arbitrary transactions but relies on software contention management or falls back to locking. However, if we assume that the programmer uses transactions whose execution time do not exceed the OS scheduling quantum or system timers periods and whose datasets fit into the core’s buffering capacity, SLE, LogTM and the Herlihy and Moss’ HTM, provided that the developer provided a correct software fallback, should be able to make forward progress.
Provided that the original program with locking synchronization is deadlock-free, the forward progress is guaranteed as the transactions can eventually fall back into locking. LogTM reduces transaction abort rate and is able to detect and avoid deadlocks, while repeating conflicts should be mitigated by the exponential back-off mechanism provided by the cache controllers.

VII. Research Proposal

In this work, we considered a number of HTM designs that showed how performance and programmability can be improved with hardware support for optimistic synchronization. As technology evolves, the number of concurrently executed tasks in modern datacenter increases dramatically. Modern datacenters execute enormous amounts of real-time queries that operate on vast datasets that cannot be captured in memory of a single machine so that modern workloads have to run their threads on tens and hundreds of servers.

As for many-core processors, efficient synchronization is important for providing high quality of service that requires both high performance and consistency of the results delivered to the system user. Thus, system designers desire to maintain strong consistency model, e.g., ACID transactions, and enable high concurrency. Unfortunately, distributed protocols, e.g., 2-phase commit, are usually implemented entirely in software and use locking techniques to keep the data consistent. As new high-speed networks, such as RDMA [21], [22] or Scale-out NUMA [1] emerge, CPU overheads become significant when using conventional TCP/IP software stack. Hence, the researchers start investigating the implications of optimistic concurrency control for datacenter systems, such as FaRM [2].

Although implementing a full-blown HTM design for distributed transactions is challenging, atomic constructs can be useful when optimizing individual stages of distributed commit protocols. For example, FaRM [2] provides atomic read-only operations for transferring a single large data object that can span multiple cache blocks. To guarantee data object consistency, FaRM has to maintain and verify metadata along with the useful data that requires additional copying to separate useful data from the metadata before providing a response to the user query. Alternatively, this CPU overhead can be avoided if the network interface card, or Remote Memory Controller [1], can provide hardware support for detecting and efficiently resolving conflicts with read and write operations of the other transactions in the system. To evaluate the overhead introduced by maintaining data consistency in software, we used a cycle-accurate full-system simulator, Flexus [23], that we configured to run 2 FaRM nodes on top of soNUMA servers. Both nodes run a read-only microbenchmark on top of FaRM. According to our experiments, whole system performance can be improved by at least 10-20% for 2-4KB data objects transfers, if the hardware provided support for atomic remote operations.

The research proposed will go through the following steps:

- Performance breakdown evaluation for each phase of the FaRM’s distributed commit protocol.
- Evaluation of the roles of destination and source nodes in a conflict detection, abort and retry mechanisms and multiple object processing from the perspectives of the performance and end-to-end principle [24].

The ultimate goal of the proposed research is to expose an efficient set of hardware-based primitives to the distributed in-memory application as well design efficient concurrency support mechanisms, such as conflict detection and resolution. Based on Scale-out NUMA as a state-of-the-art rack-scale computing platform, the system should be able to scale across the whole rack and remove software-related overheads software, such as FaRM’s atomic remote read operations, atomic data objects locking and offer other more complex operations, e.g., scatter/gather of multiple objects, that can be efficiently provided by the extended soNUMA protocol.

VIII. Conclusion

In this work, we discussed three designs of Hardware Transactional Memory system each providing a different programming interface and a number of mechanisms for optimistic non-blocking concurrency control. Such hardware support allows to write deadlock-free multithreaded programs easier than using conventional fine-grain locks with higher or comparable performance. To make an HTM design feasible, researchers proposed to either reuse speculative structures of modern out-of-order cores or to use software support to avoid speculative state buffering in hardware structures. Unfortunately, the application programmer still should be able to consider hardware constraints, e.g., for transactional datasets size, to achieve high performance and ensure forward progress. However, some of the modern HTM systems, e.g., LogTM [12], allow transactions to operate on sufficiently large datasets at a cost of less efficient conflict detection.

As new high speed networks emerge, optimistic concurrency control mechanisms in datacenters are becoming a hot research topic. We propose to leverage insights and ideas from HTM designs to optimize different phases of comprehensive distributed protocols using atomic constructs for remote memory operations at rack-scale.

References


