Mitigating the Mismatch between the Coherence Protocol and Conflict Detection in Hardware Transactional Memory

Lihang Zhao[1], Lizhong Chen[2], and Jeffrey Draper[1]
[1]*Information Sciences Institute,* [2]*Ming Hsieh Department of Electrical Engineering*
University of Southern California
Email: {lihangzh, lizhongc}@usc.edu, draper@isi.edu

***Abstract*—Hardware Transactional Memory (HTM) usually piggybacks onto the cache coherence protocol to detect data access conflicts between transactions. We identify an intrinsic mismatch between the typical coherence scheme and transaction execution, which causes a sizable amount of unnecessary transaction aborts. This pathological behavior is called false aborting and increases the amount of wasted computation and on-chip communication. For the TM applications we studied, 41% of the transactional write requests incur false aborting. To combat false aborting, we propose *Predictive Unicast and Notification* (PUNO), a novel hardware mechanism to 1) replace the inefficient coherence multicast with a unicast scheme to prevent transactions from being disrupted unnecessarily and 2) restrain transaction polling through proactive notification. PUNO reduces transaction aborts by 61% and network traffic by 32% in workloads representative of future TM applications with a VLSI implementation area overhead of 0.41%.**

I. Introduction

Chip multiprocessor architectures are ubiquitous in today's high performance computing systems. 84.6% of the Top500 supercomputers use processors with six or more cores [1]. To exploit the massive thread-level parallelism available in chip multiprocessors, applications are divided into multiple parallel threads using a shared memory space programming model. Transactional Memory (TM) promises to increase the productivity in parallel programming by removing the burden of synchronizing shared memory accesses from the programmer. In particular, the Hardware Transactional Memory (HTM) approach implements hardware support for accelerated transaction execution. Extensive research in the past decade has paved the way for HTM to be implemented into commodity microprocessors [2], [3], [4]. As of June 2013, HTM-enabled microprocessors have been deployed in four of the Top10 supercomputers [1].

HTM simplifies synchronization by providing a simple construct: the transaction. A transaction is a sequence of memory accesses. Each transaction either executes in full or has no effect at all (i.e., atomicity), and cannot observe the partial memory updates of other concurrent transactions (i.e., isolation). A conflict occurs when multiple concurrent transactions access the same data and at least one access is a write [5]. Conflicts have catastrophic consequences on correctness. HTM typically implements contention management to detect and resolve conflicts. When a conflict between two transactions is detected, one of them needs to be stalled or aborted. The contention management scheme uses a certain formula to derive the priorities of the conflicting transactions. The transaction with lower priority is stopped to resolve the conflict. As the cache coherence protocol can detect data access conflicts, the majority of HTM designs [6], [7], [8] including commercial implementations (e.g., IBM System z [9]) piggyback onto the coherence protocol (typically directory-based) for conflict detection.

However, there is an intrinsic difference between the cache coherence scheme and transaction execution. The participating entities of cache coherence are processors with equal priority, whereas the participating entities of TM execution are transactions with unequal priorities. This difference results in a mismatch between the coherence scheme and conflict detection. In the coherence scheme, a GETX (request for exclusive access) is always forwarded exhaustively (multicast) to all the sharer nodes to invalidate their private data copy. As the HTM piggybacks onto the cache coherence protocol to forward the GETX from its requester transaction to all the sharer transactions, the sharers with higher priority than the requester will nack the request while other sharers with lower priority will acknowledge the request and abort themselves to avoid conflicts. However, if the request is nacked (i.e., the conflicts do not materialize), the aborted transactions on those low-priority sharers could have continued their execution. In other words, the aborting is unnecessary. This pathological aborting behavior is identified as *false aborting*, which wastes energy and degrades performance because 1) valid transaction computation is discarded needlessly and 2) multicasting transactional write requests to all the sharers generates superfluous on-chip communication. According to our study of a spectrum of TM workloads, 92% of the transaction aborts are caused by the transactional GETX requests and 41% of these requests incur false aborting.

We introduce *Predictive Unicast and Notification* (PUNO), a novel hardware mechanism to mitigate false aborting. In PUNO, upon transaction conflict, the directory attempts to unicast (instead of multicast) the conflicting request to the very highest priority sharer transaction for conflict resolution while still maintaining correctness. As other concurrent sharers are not disrupted, false aborting can be avoided. To further reduce false aborting, the receiving transaction of the unicast request proactively notifies the nacked requester with the time when the requested cacheline will be available. Therefore, the requester can backoff before the data is ready, thereby limiting false aborting due to myopic polling to the sharers. PUNO does not require re-engineering the coherence protocol. Evaluations using full system simulation show that PUNO reduces transaction aborting by 61% on average (up to 89%) in a set of high contention benchmarks representative of future TM workloads. Due to the reduction of wasted transaction execution, the on-chip network traffic is reduced by 32% (up to 67%), and the execution time is reduced by 12%. These improvements are achieved with a meager 0.41% VLSI implementation area

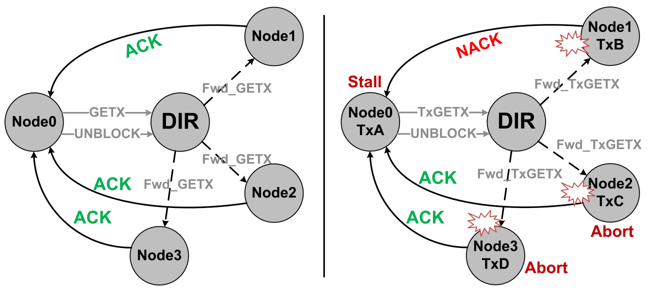

Figure 1. Comparison between cache coherence and conflict detection. (a) coherence protocol handling a GETX request; (b) conflict detection mechanism handling the GETX request. Explosion marks indicate transaction conflicts.

overhead. The contributions of this paper are three-fold:

- We identify an intrinsic mismatch between the coherence protocol and HTM that leads to pathological transaction aborting behavior. The finding reveals new optimization opportunities for HTM-enabled CMPs.
- We propose PUNO, a novel mechanism to suppress the transaction aborting by replacing the disrupting and inefficient multicast of transactional requests with unicast and notification.
- We evaluate PUNO with full system simulations to demonstrate its capability to improve execution efficiency and performance with a marginal area overhead.

The rest of this paper is organized as follows. In Section 2, we discuss cache coherence and conflict detection in HTM and highlight the problem of false aborting. Section 3 describes the implementation details of PUNO. Experimental setup and results are presented in Section 4. Section 5 summarizes related work and Section 6 concludes this paper.

II. Background and Motivation

This section describes the basics of the directory protocol and conflict detection in HTM. Then, we discuss the gravity of the disruptive false aborting, which motivates this work.

A. Directory Coherence

CMPs usually use a directory coherence protocol to provide a shared memory space. Each directory entry tracks the coherence state and the location of all the cached copies for a memory block. The directory is typically distributed among all the nodes by mapping each memory block to its home node [10]. The home node is responsible for maintaining directory entries and servicing coherence requests to its memory blocks. Upon a GETS (request for shared access), the requesting node is added to the sharer list in the directory entry. Upon a GETX, the home node forwards the request to all the sharers for invalidation. Figure 1(a) depicts how a GETX is serviced in the MESI (Modified, Exclusive, Shared, Invalidate) protocol. As shown, Node0 has a local miss and sends a GETX request to the home node directory, which forwards the request to all the three sharers as recorded in the directory entry. The sharers always invalidate their private copy and acknowledge the request, as all the nodes are of identical priority in the cache coherence protocol. After receiving all the responses, the requester sends an UNBLOCK message to conclude the request.

B. Conflict Detection in HTM

In general, conflict detection can be eager or lazy. The eager approach detects conflicts progressively as transactions load and store, whereas the lazy approach postpones detection to the commit time. This work targets eager conflict detection which can be more energy efficient as conflicts are detected early to minimize discarded work. When a transaction is executing, the load address (store address) is added into the transaction's read set (write set). Upon receiving a request from another node, the transaction checks the request against its read and write sets to detect a conflict that violates the "single-writer, multi-reader" invariant. Conflicts are resolved by stalling or aborting one or more conflicting transactions. HTM designs implement a conflict resolution policy to decide which transaction(s) should be stopped and which transaction(s) can continue executing. Essentially, such a policy prioritizes some transactions over others. So, without loss of generality, the following discussion assumes transactions have priorities. In particular, the time-based policy [11] assigns a timestamp to each transaction. The timestamp is attached to all the inter-transaction communication (e.g., coherence messages). Older transactions are given higher priority in conflict resolution. The HTM of IBM BG/Q processor [26] adopts a similar policy.

HTM designs piggyback onto the directory protocols for conflict detection to minimize the added hardware complexity. Figure 1(b) depicts how a conflict is detected using the MESI protocol. The requester transaction TxA issues a GETX to the directory, which forwards the request to all the nodes currently sharing the cacheline. Depending on their relative priorities, the sharer transactions could respond with either a NACK (negative acknowledgement) if they have higher priority than the requester or an ACK if they have lower priority. As long as one of the responses is a NACK, TxA stalls. In what follows, the transaction that sends a NACK message is called a *nacker transaction* or *nacker*.

C. False Aborting

As discussed in Section 1, false aborting occurs when the exhaustive multicast of a transactional GETX request aborts several low priority sharer transactions before the request is eventually nacked by the high priority sharers. So any transaction aborts caused by the nacked GETX are unnecessary. We assess the gravity of false aborting by tracking the coherence requests from transactions in a set of high-contention benchmarks running on a representative HTM design (see Section 4.1 for experiment details). Figure 2 shows a breakdown of transactional write requests. It is observed that an average of 41% of those requests incur false aborting. Figure 3 shows the distribution of the number of transactions being aborted unnecessarily due to false aborting. For example, in Intruder, 5 transactions are aborted unnecessarily in 10% of the false aborting cases. The long trailing indicates that false aborting can severely disrupt transaction execution as it causes a considerable number of transactions being aborted unnecessarily. Thus, the potential energy and performance gain of reducing false aborting is substantial.

However, combatting false aborting is challenging. Mitigating false aborting using a conventional coherence pro-

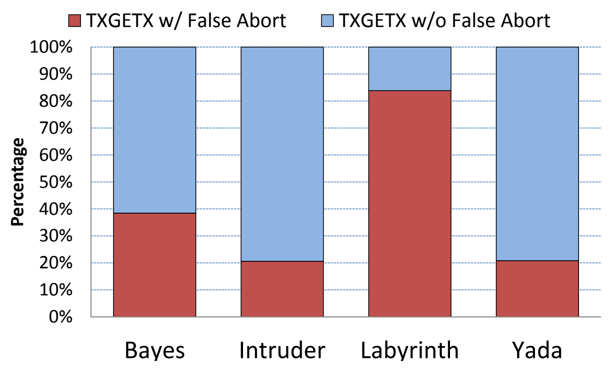

Figure 2. Percentages transactional GETX requests that trigger false aborts.

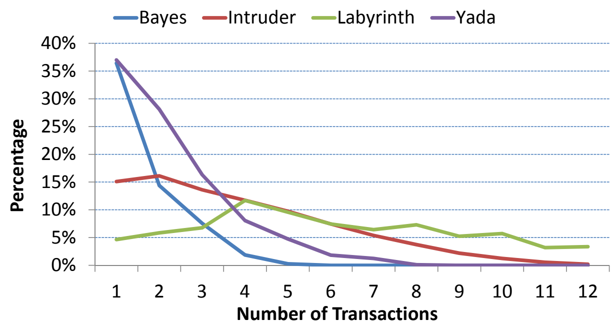

Figure 3. Distribution of the number of transactions being aborted unnecessarily due to false aborting.

tocol is difficult as it has no notion of transactions. Thus, GETX requests from transactions are always forwarded to all the sharers conservatively, even though the multicast disrupts transaction execution unnecessarily and incurs false aborting. Also, a TM-specific cache protocol is an impractical solution due to the exorbitant cost. Supposing such a TM-aware protocol does exist, it is still obscure how the protocol decides which sharers can be exempt from receiving the GETX requests without jeopardizing correctness.

III. Implementing PUNO

A. The Basic Idea

The basic idea of PUNO is based on the following two important observations. First, the exhaustive multicast of transactional GETX request to the sharers is needless if the conflict caused by the request can be resolved by a sharer with higher priority than the requester. Second, the nacked requester transaction cannot proceed until the nacker sharer transaction finishes executing, as immediate retry of the request will still be rejected by the nacker. PUNO takes advantage of the two observations by 1) replacing the multicast with predictive unicast to the high priority sharer and 2) performing proactive notification to the nacked requester with regard to when to poll the sharers again.

Figure 4 compares PUNO with the conventional scheme. In the example, a cacheline is read-shared among three transactions (i.e., TxA, TxC and TxD). TxB wishes to write to the cacheline. TxB has a higher priority than TxC and TxD, but has a lower priority than TxA. In the conventional scheme (see Figure 4(a)), The GETX from TxB is forwarded by the directory to all the three sharers. The

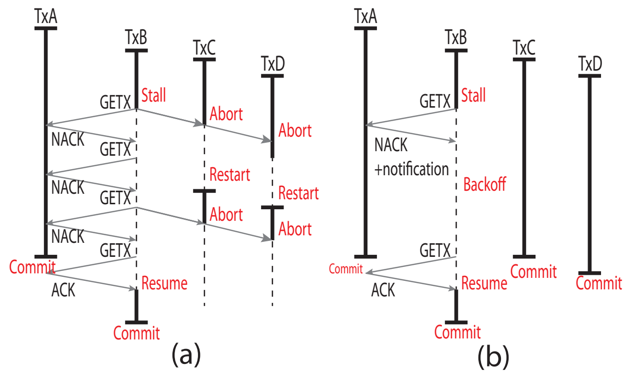

Figure 4. Comparison of transaction executions in a baseline system and PUNO.

request is nacked by TxA. However, it causes false aborting as TxC and TxD are aborted unnecessarily. TxB keeps polling the sharers and succeeds with the request when TxA finishes. The polling exacerbates false aborting as TxC and TxD are aborted several times. In contrast, in Figure 4(b), PUNO directs the directory to unicast the GETX request to TxA which is predicted with high confidence to nack the request. TxA nacks the request and notifies TxB with an estimation of its remaining running time. Consequently, TxB enters backoff and does not retry the request until TxA commits. PUNO reduces inter-transaction communication, and increases transaction throughput by allowing TxC and TxD to commit along with TxA.

While the basic idea is conceptually straightforward, the effectiveness of PUNO depends on accurate prediction of the unicast destination and a reliable scheme to derive a transaction's running time. In the subsequent discussion, corresponding mechanisms are described. Then, we discuss the protocol support. Finally, operation examples are provided.

B. Unicast Destination Prediction

To predict the unicast destination, each directory is augmented with hardware structures to track the priority of active transactions on each sharer node. As shown in Figure 5(a), each coherence controller is augmented with a Transaction Priority Buffer (P-Buffer), which has N entries to record the latest transaction priority on all N nodes on the CMP. The P-Buffer is updated constantly with the {host node, priority} pair retrieved from the incoming coherence requests as each request carries the host node and priority of the requesting transaction. Also, each directory entry is augmented with a UD (Unicast Destination) pointer, which is the id of the node that has the highest priority among all the nodes sharing that data block. The node id in the UD pointer is used to index into the P-Buffer to retrieve the transaction priority of the sharer. Updating a UD pointer is off the critical path after the directory services a request to the associated data block.

Predicting the node to which the GETX requests will be unicasted depends on the P-Buffer and UD pointer. Upon receiving such a request to a data block, the block's UD pointer is accessed in parallel with the directory entry. Then, the UD pointer is used to retrieve the sharer priority (Priority$_{sharer}$) from the P-Buffer. If Priority$_{sharer}$ is larger than Priority$_{requester}$ (obtained from the request), it is

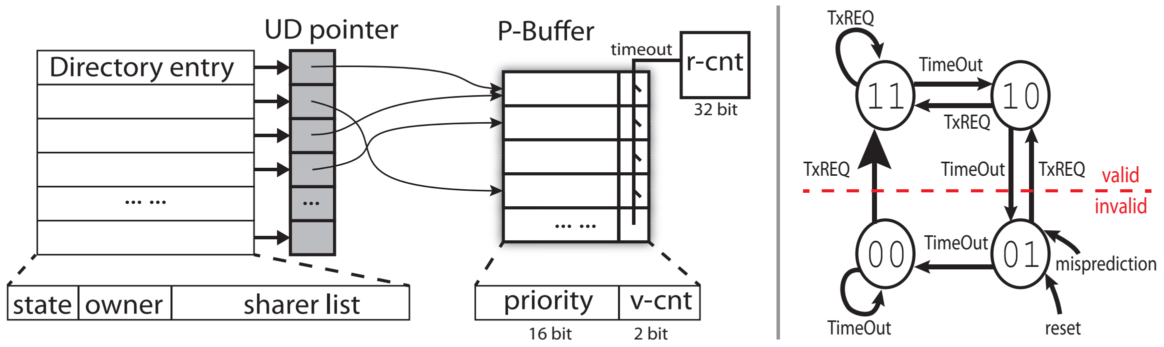

Figure 5. (a) Directory augmentation to support unicast destination prediction. Added hardware structures in bold rectangles. r-cnt: rollover counter; v-cnt: validity counter. (b) State transition of the validity counter.

predicted that the request will be nacked by that sharer. Therefore, that sharer is the unicast destination of the GETX request. Otherwise, if Priority$_{sharer}$ is smaller, the request is forwarded to all the sharers as normal.

An adaptive timeout mechanism is implemented to improve the accuracy of the unicast prediction as stale priorities in the P-Buffer can cause mispredictions. A priority in the P-Buffer becomes stale if the remote node begins executing a new transaction and the P-Buffer has not been updated with the new priority. The priority is updated when a request from the new transaction is received. The hardware support of the timeout mechanism is shown in Figure 5(a). The directory is augmented with one 32-bit *rollover counter* for the entire directory and 2-bit *validity counters*, one per P-Buffer entry. Upon overflow, the rollover counter generates a timeout signal to trigger the state transition of all the validity counters. The timeout period used by the rollover counter is determined dynamically based on the average transaction length obtained from a hardware mechanism (discussed in the subsequent section). The adaptivity to transaction characteristics enhances the timeout mechanism for workloads with a large variance in transaction length. Figure 5(b) depicts the state transition of the validity counter. When the rollover counter generates a timeout signal, all the non-zero validity counters are decremented by 1 so the validity of the associated priority is decreased. When a priority is updated, its validity counter is incremented. So, priorities that have not been updated for a long period of time have small validity counters whereas recently updated priorities have larger validity counters. Only those priorities with validity counters greater than 1 are used for unicast prediction. After updating the priority with 0 validity, the validity counter is incremented twice to allow a longer timeout period.

C. Handling Misprediction

A misprediction occurs when a GETX request is unicasted to a sharer transaction with a lower priority than the requester (i.e., predict incorrectly that the requestor has a lower priority). As the unicast message has a special bit set to one (coherence message extension is discussed shortly), the receiving sharer can detect a misprediction when the special bit is set and it has a lower priority than the requestor. Misprediction, if not handled properly, may cause a correctness problem. Consider the transaction execution in Figure 4 (b). Misprediction happens if TxB's GETX is unicasted

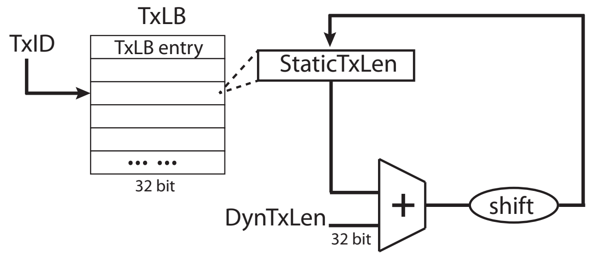

Figure 6. Structure of the transaction length buffer and computing logic.

to TxC instead of TxA. If TxC, which has a lower priority, acknowledges the request, TxB can write to the cacheline without the awareness of the other two sharers (i.e., TxA and TxD). Consequently, the "single-writer-multi-reader" invariant is violated. To guarantee correctness, misprediction is handled conservatively by letting the mispredicted sharer nack the request. So, the requester is forced to retry the request. To improve prediction accuracy, a misprediction feedback mechanism is devised. The mispredicted sharer (e.g., TxC in previous example) informs the requester (e.g., TxB) of the misprediction via the NACK message. Then, the requester notifies the directory of the misprediction through the UNBLOCK message, so that the directory can invalidate the stale priority in its P-Buffer that caused the misprediction. On the other hand, if a requestor is predicted incorrectly to have a higher priority than the sharers, the request is multicasted to all the sharers as normal, i.e., the PUNO unicast mechanism is not triggered, so no abnormal correctness issues arise for this case.

The misprediction handling approach guarantees correctness with marginal performance impact due to three reasons. First, the prediction accuracy is high (90%+ hit rate in simulation). Second, some NACKs due to misprediction can be true positives anyway as the request could be nacked by other sharers if not being unicasted. Third, the invalidation and upgrading performed by the directory upon receiving the misprediction feedback do not incur a performance penalty as they are off the critical path of the coherence messages.

D. Notification

PUNO further suppresses false aborting with a notification mechanism. The sharer transaction receiving the unicasted request notifies the requester with its expected running time (T$_{est}$ in terms of cycles) through the NACK response. As the requester cannot proceed until the nacker finishes, it can leverage the notification to decide whether

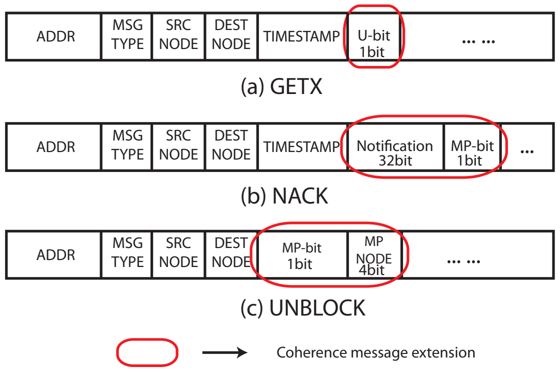

Figure 7. Protocol message extensions to support PUNO.

to backoff. If T$_{est}$ minus twice the average cache-to-cache latency (determined by network topology) is positive, it is used as the backoff period to throttle the requester polling for the cacheline. The effectiveness of notification depends on an accurate tracking of transactions' running length. Bad backoff causes the requester to either wait too long or retry too soon. Given the large variance of the transaction length within applications, averaging the lengths of all the past transactions is not sufficient.

The proposed design tracks the length of individual static transactions separately using a per-node hardware structure named the Transaction Length Buffer (TxLB), as depicted in Figure 6. A static transaction is defined in the code with `TX_BEGIN` and `TX_END` pairs. Such a static transaction is usually executed multiple times. Each execution is a *dynamic instance*. A static transaction has a TxLB entry to track the average length of its past dynamic instances. When a dynamic instance commits, its length (*DynTxLen*) is known by subtracting its beginning cycle time from the current cycle time. Then, the static transaction length (*StaticTxLen*) in the TxLB is updated using formula (1). This formula places more weight on recent dynamic instances to closely track recent execution.

$$StaticTxLen_{new} = \frac{StaticTxLen_{prev} + DynTxLen}{2} \tag{1}$$

The TxLB size can be small for less hardware overhead as workloads usually have a limited number of static transactions. For instance, Bayes, the workload with the largest number of static transactions in the STAMP benchmark, has 15 static transactions in total. In the rare case of overflow, the system can resort to a software managed structure to track transaction length.

E. Protocol Support for PUNO

PUNO requires minimal modification to the coherence protocol. The protocol state transition remains unchanged, and no extra coherence states (stable or transient) are needed. So, PUNO can work with the coherence protocols in many existing HTM designs.

Three coherence messages are extended (see Figure 7). First, the GETX message is extended with 1 extra bit (U-bit) to indicate whether it is a unicast request. The U-bit is set by the directory when the request is unicasted to a sharer. In some protocol variations, the directory sends invalidations instead of GETX to the sharers, in which the U-bit can simply be added to the invalidation messages. Second, the NACK message is extended with the notification field. This field includes the number of cycles that indicates the nacker transaction's running time. Also, a misprediction bit (MP-bit) is added to support misprediction feedback as discussed in Section 3.3. Third, the UNBLOCK message is extended with a misprediction bit (MP-bit) and a MP-node field that specifies the mispredicted unicast destination. Due to the wide on-chip channels, the extended messages can fit into the existing flits, requiring no extra flits on the network.

F. Operation Example

This subsection provides several walk-through examples to illustrate how the predictive unicast and the notification work collaboratively to mitigate false aborting.

Directory updates the P-Buffer (Figure 8(a)): when the directory receives transactional GETS requests (TxGETS) from the three nodes, it updates its P-Buffer and increments the validity counters from 1 (invalid) to 2(valid). The UD pointer is pointing to the priority of Node1 because it has the highest priority.

Directory predicts the unicast destination (Figure 8(b)): when the directory receives the transactional GETX (TxGETX) from Node2, it follows the UD pointer to get the highest priority of the sharers. As the requester's priority is lower than Node1's priority as recorded in the Prio-Buffer, the directory only sends the TxGETX to Node1.

Unicast destination sends notification to the requester (Figure 8(c1)): upon receiving the Fwd_TxGETX request, Node1 resolves the conflict by nacking the request. Node1's average length is retrieved from the TxLB. The remaining running length is computed by subtracting the cycles it has already run from its average length. The information is attached to the NACK message to Node2. The transaction at Node2 enters backoff upon receiving the notification.

Unicast destination provides misprediction feedback to the directory (Figure 8(c2)): now suppose that the previous transaction (timestamp=100) on Node1 has finished executing and a new transaction (timestamp=180) starts. But the directory is not aware of the new transaction just yet and, hence, still forwards the TxGETX to Node1. Upon receiving the request, Node1 detect a misprediction of the unicast destination as the local transaction has a lower priority than the requester. Node1 nacks the request to guarantee correctness. Due to misprediction, no notification to Node2 is provided. The MP-bit of the NACK is set for misprediction feedback. After receiving the NACK, Node2 sets the MP-bit and MP-node in the UNBLOCK message. When the directory receives the misprediction feedback, it invalidates Node1's priority in the Prio-Buffer entry. The UD pointer is updated to point to Node3 because it has the highest priority now.

IV. Evaluation

A. Methodology

We conducted cycle-accurate full system simulation using SIMICS [12] and GEMS [13] to evaluate PUNO. Garnet [14] was used as the on-chip network timing model. We present results for all eight workloads from the STAMP benchmark

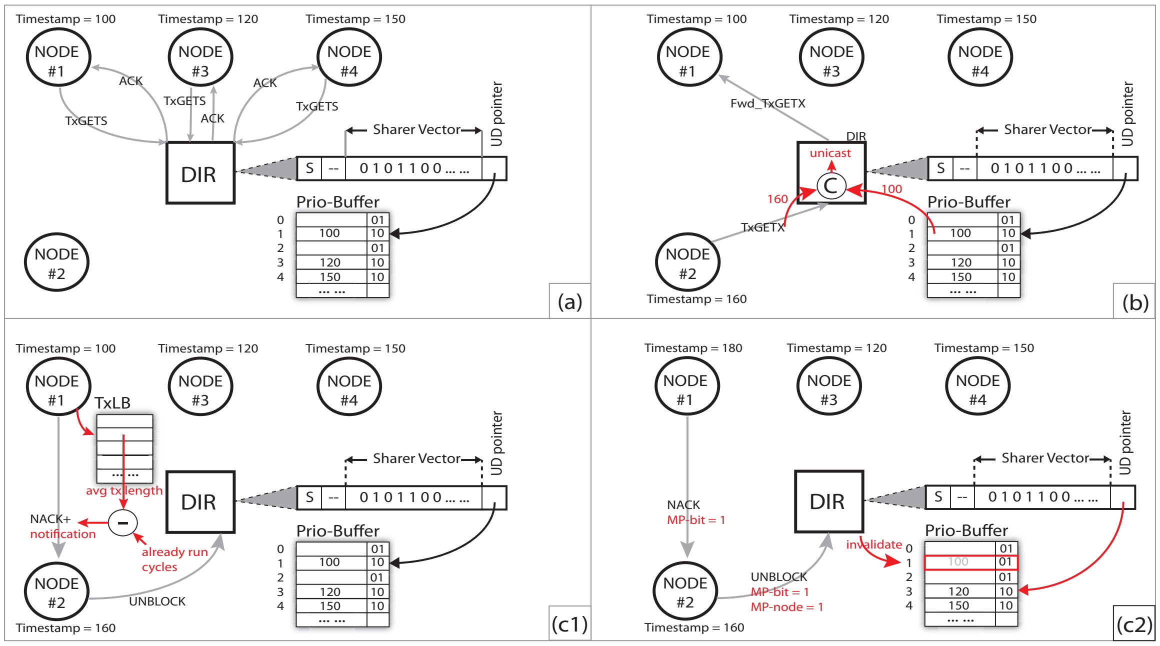

Figure 8. PUNO operation examples. All the coherence messages and states are with regard to the same cacheline. DIR: directory. C: comparator. Key operations are highlighted. Smaller timestamp indicates higher priority.

Table I
BENCHMARK INPUT PARAMETERS

Benchmark	Input Parameters	Abort %
Bayes	32 var, 1024 records, 2 edge/var	97.1%
Intruder	2k flow, 10 attack, 4 pkt/flow	77.6%
Labyrinth	32*32*3 maze, 96 paths	98.6%
Yada	1264 elements, min-angle 20	47.9%
Genome	32 var, 1024 records	1.3%
Kmeans	16K seg. 256 gene. 16 sample	7.4%
SSCA2	8k nodes, 3 len, 3 para edge	0.3%
Vacation	16K record. 4K req. 60% coverage	38%

Table II
SYSTEM CONFIGURATION

Unit	Value
Core	16 Sun UltraSPARC III+ cores, 1GHz
L1 Cache	32 KB, 4-way associative, write-back, 1-cycle
L2 Cache	8 MB, 8-way associative, 20-cycle latency
Coherence	MESI protocol, static cache bank directory
Memory	4 GB, 4 memory controller, 200-cycle latency
Network	2D mesh, DOR, VC flow control, 4-stage router
PUNO	16-entry P-Buffer; 32-entry TxLB

suite [15] widely used to evaluate HTM designs. Table I lists the benchmark details. The baseline CMP architecture is depicted in Figure 9. Each of the 16 nodes comprises a SPARC core with private L1 and shared L2. The shared L2 follows the static non-uniform cache architecture [10] and maintains coherence using the MESI directory protocol similar to the SGI Origin protocol [16]. Every memory block is statically assigned to a home node based on the memory address. The processor implements hardware support for a log-based HTM in which pre-transaction states are written to a software log while speculative states are propagated to the memory eagerly. The baseline also uses a hardware buffer to store the pre-transaction states to support fast abort recovery. Conflicts are detected eagerly using the coherence protocol. Upon a conflict, the receiver transaction resolves the conflict using the time-based policy [11]. To mitigate conflicts, a nacked requester node backoffs for a fixed 20 cycles before retrying the request. The performance of the

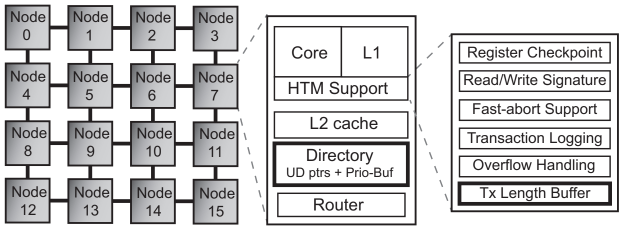

Figure 9. The baseline chip multiprocessor architecture. PUNO augmentation in bold rectangles.

baseline HTM is comparable to that of contemporary eager HTM designs (e.g., FASTM [7]). The underlying 2D mesh on-chip network uses dimension-order routing and virtual channel flow control. The pertinent characteristics of the system configuration are in Table II. We implemented PUNO on top of the baseline system in the simulator. It takes one cycle for the directory to access the P-Buffer and one cycle to determine whether to unicast the request. The remaining PUNO operations (i.e., notification, accessing UD-pointer) do not add latency as they can either run in parallel with the rest of the system or operate off the critical path. PUNO is compared against two other existing mechanisms that can reduce transaction aborts: 1) *Random backoff* [17]: aborted transactions enter randomized linear backoff before restarting. Transactions that abort frequently will have longer backoff. 2) *Read-Modify-Write predictor*(RMW-Pred) [5]: transactions exhibiting the read-modify-write memory access pattern can request exclusive permission upon the read, thereby avoiding abort due to the later dueling write. Each node has a RMW predictor to track up to 256 load instructions.

B. Reduction in Transaction Abort

One of the main design objectives for PUNO is to mitigate unnecessary transaction aborts. Figure 10 shows the impact of PUNO on transaction aborts. It is observed that,

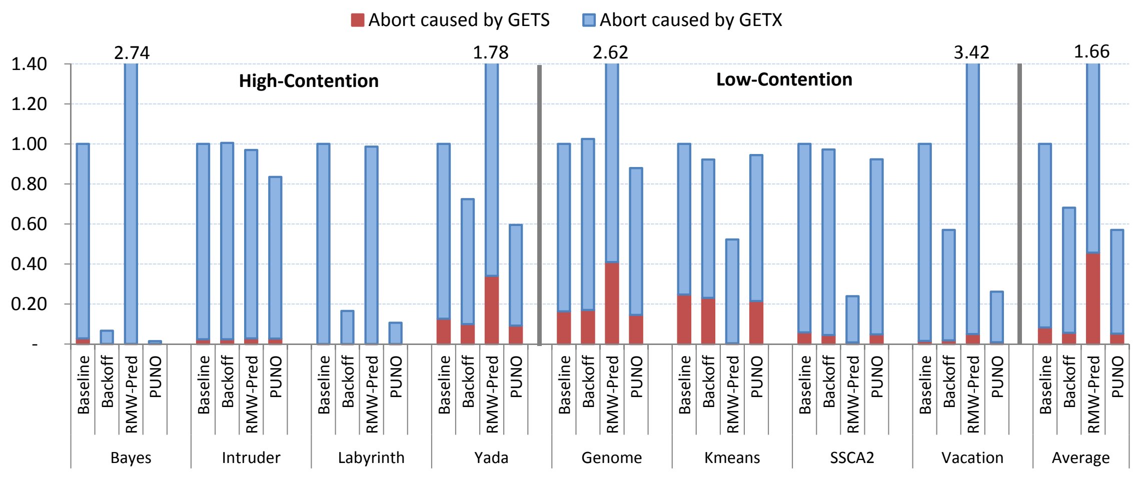

Figure 10. Normalized transaction abort count.

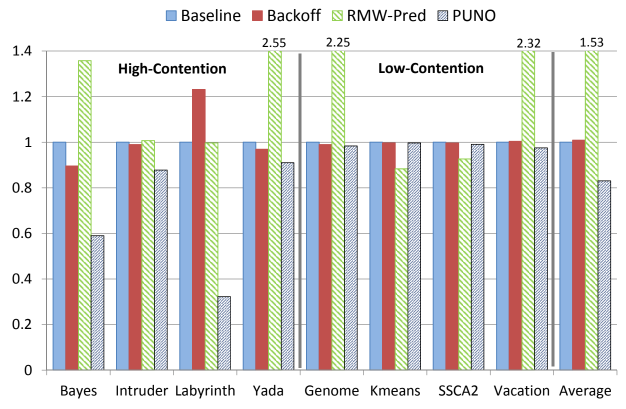

Figure 11. Normalized on-chip network traffic.

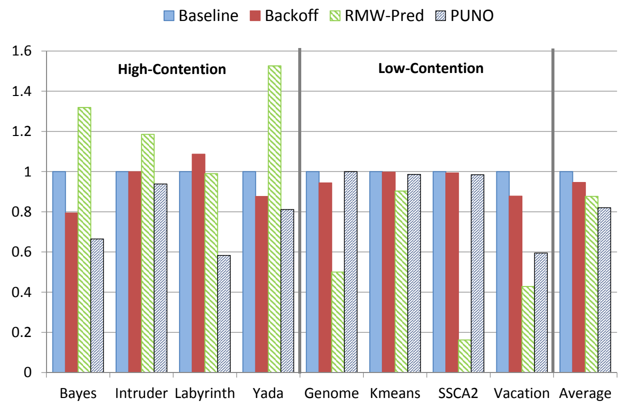

Figure 12. Normalized cycle count when the directory is blocked while servicing transactional GETX.

on average, PUNO reduces transaction aborts by 43% (up to 98%) compared with the baseline. In particular, PUNO is effective in reducing aborts caused by the transactional GETX requests which are the main causes of most transaction aborts in the workloads. PUNO achieves significant abort reduction in the high contention benchmarks (61% less aborts). This result is expected as workloads with high contention usually incur more false aborting due to frequent transaction writes and extensive read-read sharing.

PUNO incurs an average of 17% fewer aborts compared with random backoff, indicating that the notification-guided backoff scheme of PUNO is more effective in avoiding conflicts. In the random backoff scheme, the backoff period is determined by local transaction statistics such as number of retries. Nonetheless, the backoff period should essentially be dependent on the remote nacker transaction with which the local transaction has data conflicts. In PUNO, a local transaction receives reliable information from the notification from the remote transactions so that it can better optimize the backoff period.

Previous work [5] demonstrates the effectiveness of RMW-Pred to reduce conflicts in expertly optimized workloads that have very low contention and fine-grain transactions. Our evaluation results echo this finding as RMW-Pred reduces transaction abort significantly in Kmeans and SSCA2, both consisting of short transactions with few conflicts (for instance, the transaction abort rate is 0.3% in SSCA2). However, the results also suggest that RMW-Pred is inefficient in workloads with frequent conflicts among coarse-grain transactions. RMW-Pred tends to convert read-read sharing to write-read conflicts by obtaining exclusive permission upon loads. As the abort rate is already very sensitive to the number of conflicts in most contemporary and expected future TM applications, RMW-Pred exhibits many more transaction aborts (e.g., 2X more in vacation) than the other mechanisms.

C. Reduction in Network Traffic

Figure 11 shows the normalized on-chip network traffic measured in router traversals by all the network flits. As can be observed, PUNO eliminates 33% (up to 68%) of the traffic in high-contention benchmarks compared with the baseline scheme. Across all the workloads, the network traffic is reduced by an average of 17%. The traffic reduction is due to three facts. First, PUNO replaces the wasteful multicast of GETX requests with unicast when possible. Second, the notification mechanism suppresses unnecessary transaction polling. Third, the reduction in transaction aborts translates to less futile traffic from aborted transactions.

In comparison with random backoff, PUNO reduces the network traffic by 34% in the high-contention workloads. As both random backoff and PUNO significantly reduce the transaction aborts (see Figure 10), the difference in network

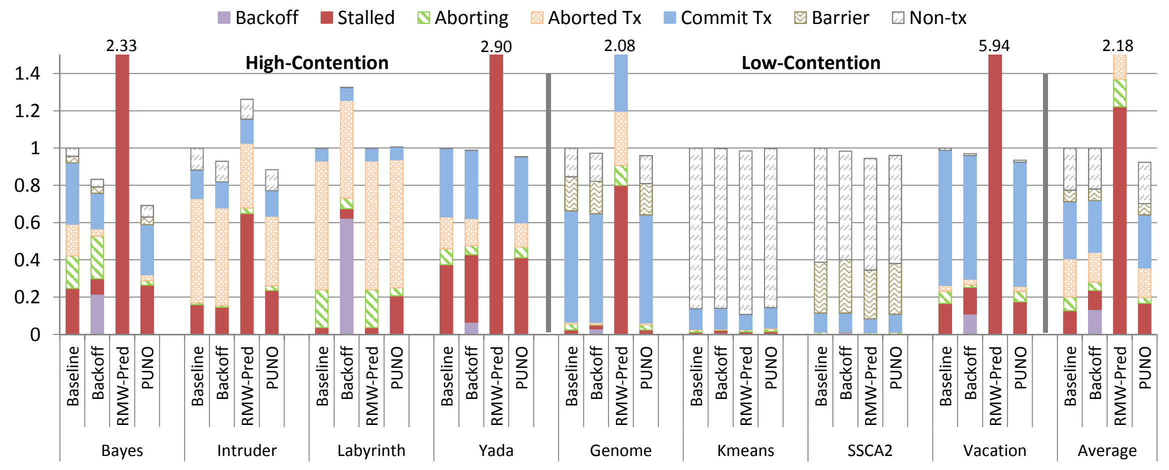

Figure 13. Normalized execution time.

traffic reduction largely comes from the difference in the traffic from committed transactions. The backoff period is an important factor in determining the traffic generated by the committed transactions. The results in Figure 11 show that the notification-guided backoff in PUNO is more effective than the random backoff in throttling transaction traffic.

D. *Reduction in Directory Blocking*

When the directory forwards a GETX request to the sharer nodes, it cannot service subsequent requests to the same cacheline until the requester sends an UNBLOCK message to the directory after receiving responses from all the sharers. Reducing directory blocking leads to potential performance gains in workloads bounded by memory bandwidth. Figure 12 shows the impact of PUNO on directory blocking. The values are obtained by averaging the number of cycles during which directory entries stay in a blocking transient state when servicing transactional GETX requests. As can be seen, PUNO eliminates 18% (up to 42%) of such directory blocking compared with the baseline. This improvement is mainly because the unicast of GETXs minimizes the number of sharer nodes that need to respond to the requester. Statistically, the expected waiting time for response from a single sharer is shorter than the waiting time for responses from multiple ones. Thus, the directory blocking time is reduced with a minimized set of sharer nodes. In particular, transactions in Labyrinth read in the entire global maze grid and write to a small portion of the grid. So, the writer transactions in the baseline need to wait for responses from a large number of sharers before unblocking the directory. In contrast, the predictive unicast significantly minimizes the sharer transactions to respond to the request, thereby reducing the waiting time. Thus, PUNO incurs 42% less directory blocking in Labyrinth. The reduction in directory blocking allows more requests to be serviced instead of waiting, increasing the concurrency in the cache system.

E. *Impact on Performance*

Figure 13 presents the normalized execution time. As it is observed, PUNO achieves an average of 12% (up to 31%) performance improvement over the baseline scheme in high-contention workloads. Across all the workloads, PUNO improves the performance by an average of 8%. The performance advantage of PUNO stems from the fact that it succeeds in suppressing false aborting, thereby improving transaction throughput.

Compared with the random backoff scheme, PUNO performs consistently better in all the workloads. It is worth noting that, although random backoff mitigates aborts in Labyrinth (see Figure 10), it hurts performance by limiting the concurrency among transactions. Execution statistics of random backoff show that transactions in Labyrinth spend more time in backoff than in execution. Hence, compared with PUNO, the random backoff is too conservative and less effective in Labyrinth, which represents workloads with extremely high contention.

Compared with the RMW-Pred scheme, PUNO performs better in six out of eight workloads. In the remaining two workloads (Kmeans and SSCA2), the performance advantage of RMW-Pred is very marginal (less than 1.6%). As discussed in Section 4.2, RMW-Pred performs well in Kmeans and SSCA2 as it can mitigate conflicts in workloads with very low contention. However, as observed in Figure 13, RMW-Pred incurs a performance penalty (1.83X slow down) in high-contention workloads due to the extra conflicts caused by upgrading GETS requests early on.

Note that the performance improvement is not necessarily proportional to the reduction in transaction aborts. For instance, PUNO eliminates more than 90% of the transaction aborts and reduces the execution time by 31% in Bayes while it eliminates 40% of the aborts and reduces the execution time by only 5% in Yada. The reduction of transaction aborts may not be translated directly to a performance advantage as transactions surviving the abort can be stalled due to conflicts with other transactions. Nonetheless, the amount of wasted transaction work is reduced.

F. *Transaction Execution Efficiency*

Transactions can be stalled instead of aborted to avoid discarding valid transaction computation. The interesting tradeoff between abort and stalling reveals an opportunity to improve the efficiency of transaction execution. To evaluate the efficiency of transaction execution, we measure the number of cycles in transactions that commit and the number

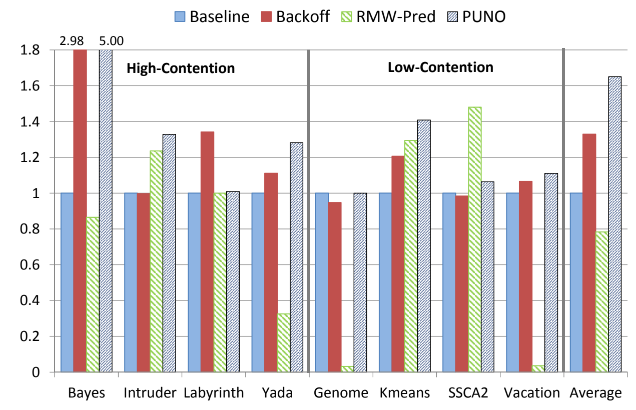

Figure 14. Normalized transaction G/D ratio indicating the efficiency of transaction execution (the larger the better).

of cycles in transactions that are aborted due to conflicts. The former metric is named *good transaction effort*, while the latter is named *discarded transaction effort*. The ratio of the two efforts, namely the *G/D ratio*, signifies whether the system can execute transactions with minimal waste. A large G/D ratio indicates that a significant amount of transactional computation is valid and committed to the memory eventually. In contrast, a small G/D ratio suggests that a sizable amount of transactional computation is wasted. Figure 14 shows the G/D ratio of the four designs. As can be observed, on average, the G/D ratio of PUNO is higher than the baseline, random backoff and RMW-Pred schemes by 1.65X, 1.24X and 2.11X respectively. This result highlights that PUNO improves execution efficiency due to its capability to mitigate false aborting.

G. Hardware Overhead

The implementation of PUNO introduces little area and power overhead. The Prio-Buffer, TxLB and UD pointers are the main contributors of the extra area and power dissipation. We estimate the area and power of the structures using a commercial memory compiler with a clock frequency of 2.3GHz and Vdd value of 0.9V. Table III reports the area and power estimation of PUNO targeting 65nm technology. The configuration of the Prio-Buffer and TxLB is identical to that used in the full system simulation, as shown in Table II. The area and power of UD pointers are overestimated as each pointer is set to 8 bits instead of 4 bits due to constraints of the memory compiler. The overhead estimation is derived by comparing with the Sun Rock processor [18] which is a 16-core chip multiprocessor with HTM support. The Rock processor is clocked at 2.3GHz and fabricated using 65nm technology. Each of the 16 cores has an area of 14,000,000um^2 and a power dissipation of 10W. The overhead estimation in Table III shows that PUNO incurs less than 0.41% more area and 0.31% more power, which further justify its deployment into future HTM designs.

V. Related Work

Herlihy and Moss introduced Transactional Memory as a new concurrency control mechanism to provide lock-free synchronization [19]. In the past decades, extensive research was conducted on implementing high-performance HTM systems [6], [20], [21], [22], [23], [7], [24]. In particular,

Table III
Area and power overhead estimation

Components	Area (um^2)	Power (mW)
Prio-Buffer	4700	7.28
TxLB	5380	7.52
UD pointers	47400	16.43
Overall	57480	31.23
Overhead	**0.41%**	**0.31%**

LogTM is a representative design that piggybacks onto the conventional directory-based cache coherence protocol for conflict detection. The transactional write requests are always forwarded by the directory to all the sharer transactions. The baseline scheme in our experiment follows the same conflict detection mechanism. As HTM designs usually piggyback onto the directory protocol to detect conflicts, their performance is susceptible to false aborting.

Bobba et al. [5] studied the pathological behaviors due to transaction conflicts. Then, the authors propose a set of techniques to mitigate conflicts. We have compared two of their techniques in our evaluation to show that our mechanism could reduce more transaction aborts in high contention wordloads. Titos et al. [25] noticed that the high abort rate not only hurts performance but also incurs excessive on-chip network traffic, which has a significant energy implication. More recently, as HTM becomes available on commercial implementations [2], performance studies on those systems [26] indicate that HTM performance is sensitive to conflict and abort rates. Beyond previous studies on transaction conflicts, our analysis is this work shows the inherent difference between the coherence scheme and conflict detection in eager HTM creates a significant source of unnecessary conflicts and aborts.

Reducing transaction aborts is one of the main design objectives in many high-performance HTM designs. [8], [27], [28] provides hardware support for a hybrid-mode execution of transactions. While the hybrid approach could boost performance, the complexity of implementing an unbounded lazy mode can be considerably high. PUNO can be implemented with the hybrid HTM designs to mitigate false aborting in the eager execution mode. Also, conflicts can be reduced through contention management. Scherer and Scott [17] introduced and analyzed a variety of reactive conflict resolution policies. Contention management can be proactive. The ATS [29], BFGTS [30] and PTS [30] schemes implement either low overhead counters or sophisticated filters to track and predict conflicts between transactions, thereby mitigating aborts by serializing transactions with a high probability to conflict. The basic idea of PUNO is orthogonal and complementary to these proactive contention management mechanisms. To the best of our knowledge, PUNO is the first to identify and combat false aborting.

As PUNO uses a prediction scheme to decide the unicast destination, we also look at related works on destination-set prediction. Acacio et al. [31] proposed a two-level predictor to predict the possible node that can service a request. Therefore, the request can be sent directly to the predicted node. Besides, Martin et al. [32] leveraged workload characteristics to predict the destination of coherence requests. The authors proposed a cluster of predictor designs to predict the destination nodes of coherence requests in order to remove the directory indirection. Each of the predictor designs targets a specific sharing characteristic in the workloads. However, both approaches, if implemented in HTM-enabled microprocessors, are incapable of avoiding false aborting as coherence protocols always attempt to notify all the sharers with the write requests to maintain coherence. In contrast, PUNO takes advantage of additional information from transaction execution to reduce false aborting.

VI. Conclusion and Future Work

HTM designs typically piggyback onto the cache coherence protocol for conflict detection. In this paper, we identify an intrinsic mismatch between the coherence protocol and eager conflict detection of HTM, which leads to a performance and energy pitfall called false aborting. We propose the Predictive Unicast and Notification (PUNO) scheme to combat false aborting. First, PUNO replaces the wasteful multicast of transactional write requests with unicast, thereby preventing the requests from disrupting the execution of concurrent transactions unnecessarily. Second, a proactive notification scheme restrains transaction polling, thereby further suppressing false aborting. PUNO does not require modification to the existing coherence protocol states or transitions. Full-system simulation demonstrates that, compared with a typical high performance HTM design, our approach reduces transaction aborts by 61% in benchmarks representative of future TM applications. Meanwhile, the network traffic is reduced by 32%. These improvements are achieved with a mere 0.41% area overhead.

Our future work includes optimizing and leveraging the unicast destination predictor design to predict transaction behaviors under high contention. The prediction can be used to perform coherence actions speculatively to accelerate inter-transaction communication. Furthermore, reducing the coupling between conflict detection and coherence protocols could be promising in improving the performance-per-joule of HTM-enabled multiprocessors.

References

[1] "http://www.top500.org/."

[2] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, P. Boyle, N. Chist, C. Kim, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, and G. Chiu, "The IBM Blue Gene/Q Compute Chip," *Micro, IEEE*, 2011.

[3] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance SPARC CMT Processor," *Micro, IEEE*, vol. 29, no. 2, March-April 2009.

[4] I. Corp., "Intel Architecture Instruction Set Extensions Programming Reference," February 2012.

[5] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Procs. of the Int'l. Symp. on Computer Architecture*, 2007.

[6] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Procs. of the 12th Int'l. Symp. on High Performance Computer Architecture*, 2006.

[7] M. Lupon, G. Magklis, and A. Gonzalez, "FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery," in *Procs. of the Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[8] M. Lupon, G. Magklis, and A. Gonzalez, "A Dynamically Adaptable Hardware Transactional Memory," in *Procs. of the 43rd Int'l. Symp. on Microarchitecture*, 2010.

[9] C. Jacobi, T. Slegel, and D. Greiner, "Transactional Memory Architecture and Implementation for IBM System z," in *Procs. of the 45th Int'l Symp. on Microarchitecture*, 2012.

[10] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," in *Procs. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[11] R. Rajwar and J. R. Goodman, "Transactional Lock-free Execution of Lock-based Programs," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, 2002.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "SIMICS: A Full System Simulation Platform," *Computer*, vol. 35, 2002.

[13] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Comput. Archit. News*, vol. 33, November 2005.

[14] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A Detailed On-chip Network Model inside a Full-system Simulator," in *Procs. of the Int'l. Symp. on Performance Analysis of Systems and Software*, 2009.

[15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Procs. of Int'l. Symp. on Workload Characterization*, 2008.

[16] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, May 1997.

[17] W. N. Scherer III and M. L. Scott, "Advanced Contention Management for Dynamic Software Transactional Memory," in *Procs. of the 24th Symp. on Principles of Distributed Computing*, 2005.

[18] M. Tremblay and S. Chaudhry, "A Third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC Processor," in *Solid-State Circuits Conference, 2008. Digest of Technical Papers. IEEE International*, 2008.

[19] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," in *Procs. of the 20th Int'l. Symp. on Computer Architecture*, 1993.

[20] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Procs. of the 13th Int'l Symp. on High Performance Computer Architecture*, 2007.

[21] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Procs. of the 31st Int'l Symp. on Computer architecture*, 2004.

[22] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory," in *Procs. of the 13th Int'l. Symp. on High Performance Computer Architecture*, 2007.

[23] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware Transactional Memory for Increased Concurrency," in *Procs. of the 41st Int'l. Symp. on Microarchitecture*, 2008.

[24] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory," in *Procs. of the 34th Int'l. Symp. on Computer Architecture*, 2007.

[25] J. R. T. Gil, M. E. A. Sanchez, and J. M. G. Carrasco, "Characterization of Conflicts in Log-based Transactional Memory (LogTM)," in *Procs. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008.

[26] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," in *Procs of the 21st Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[27] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. García, and P. Stenstrom, "ZEBRA: a Data-centric, Hybrid-policy Hardware Transactional Memory design," in *Procs. of the Int'l Conf. on Supercomputing*, 2011.

[28] L. Zhao, W. Choi, and J. Draper, "SELTM: Selective Eager-Lazy Management for Increased Concurrency in Transactional Memory," in *Procs. of the International Parallel and Distributed Processing Symposium*, 2012.

[29] G. Blake, R. Dreslinski, and T. Mudge, "Proactive Transaction Scheduling for Contention Management," in *Procs. of the Int'l Symp. on Microarchitecture*, 2009.

[30] G. Blake, R. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," in *Procs. of Int'l. Symp. on High Performance Computer Architecture*, 2011.

[31] M. E. Acacio, J. González, J. M. García, and J. Duato, "Owner Prediction for Accelerating Cache-to-cache Transfer Misses in a cc-NUMA Architecture," in *Procs. of the Conf. on Supercomputing*, 2002.

[32] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using Destination-set Prediction to Improve the Latency/bandwidth Tradeoff in Shared-memory Multiprocessors," in *Procs. of the 30th Int'l Symp. on Computer architecture*, 2003.