HARP: Adaptive Abort Recurrence Prediction for Hardware Transactional Memory

Adrià Armejach*† Anurag Negi‡ Adrián Cristal*§ Osman Unsal* Per Stenstrom‡ Tim Harris◇[1]
*Barcelona Supercomputing Center ‡Chalmers University of Technology ◇Oracle Labs, Cambridge
†Universitat Politècnica de Catalunya §IIIA - CSIC - Spanish National Research Council

***Abstract*—Hardware Transactional Memory (HTM) exposes parallelism by allowing possibly conflicting sections of code, called transactions, to execute concurrently in multithreaded applications. However, conflicts among concurrent transactions result in wasted computation and expensive rollbacks. Under high contention HTM protocol overheads can, in many cases, amount to several times the useful work done. Blindly scheduling transactions in the presence of contention is therefore clearly suboptimal from a resource utilization standpoint, especially in situations where several scheduling options exist.**

This paper presents HARP (Hardware Abort Recurrence Predictor), a hardware-only mechanism to avoid speculation when it is likely to fail. Inspired by branch prediction strategies and prior work on contention management and scheduling in HTM, HARP uses past behavior of transactions and locality in conflicting memory references to accurately predict conflicts. The prediction mechanism adapts to variations in workload characteristics and enables better utilization of computational resources. We show that an HTM protocol that integrates HARP exhibits reductions in both wasted execution time and serialization overheads when compared to prior work, leading to a significant increase in throughput (~30%) in both single-application and multi-application scenarios.

I. Introduction

The problem of extracting thread level parallelism through speculative execution has received a lot of attention from both industry and academia [13, 18]. In particular, Hardware Transactional Memory (HTM) [14] offers performance comparable to fine-grained locks while, simultaneously, enhancing programmer productivity by largely eliminating the burden of managing access to shared data. Recent usability studies support this thesis [8, 19], suggesting that Transactional Memory (TM) can be an important tool for building parallel applications. For these reasons, HTM is getting increasing attention from the industry [9, 10, 11], and IBM has released their first chip with built-in HTM support, the BlueGene/Q [23]. More recently, Intel has published ISA extensions (TSX) that provide support for basic HTM and lock elision, with the intention of supporting these in upcoming products [16].

An HTM system allows concurrent speculative execution of blocks of code, called transactions, that may access and update shared data. However, in the presence of data conflicts transactions may abort, i.e., the results of speculative execution are discarded. This results in wasted work, expensive rollbacks of application state, and inefficient utilization of computational resources. While conflicts due to concurrent accesses to shared data cannot be completely eliminated, mechanisms to avoid starting a transaction when it is likely to fail are necessary for maximizing computational throughput. Moreover, in scenarios where multiple scheduling options are available, having such mechanisms can expose additional parallelism and improve resource utilization.

While single application performance is still important, systems where multiple parallel applications coexist are expected to become increasingly common in the near future. The performance of HTM in scenarios with abundant transactional threads is still an open question, and solutions that provide efficient utilization of computational resources and good performance are required for TM to gain wide acceptance. In the past, considerable work has been done on contention management, but mostly in the field of Software TM (STM) [1, 12, 20]. These proposals typically react *after* aborts happen, without trying to avoid future conflicts. Conversely, a few HTM proposals exist that try to avoid execution of possibly conflicting transactions [3, 5, 24]. However, these solutions do not provide full hardware support and rely on expensive and specialized software runtime routines and data structures. Moreover, the efficacy of these proposals in scenarios with multiple concurrently executing applications is unclear.

In this paper, we introduce Hardware Abort Recurrence Predictor (HARP), a comprehensive hardware proposal that identifies groups of transactions that are likely to be executed concurrently without conflicts. Our proposal allows other threads or applications to utilize computational resources when the expected duration of contention is long, providing better throughput when running several applications, and potentially higher parallelism when several threads of the same application are available for scheduling. Moreover, HARP dynamically chooses a contention avoidance mechanism based on expected duration of contention, in order to maximize resource utilization, while minimizing the amount of wasted work due to transaction aborts. HARP avoids software overheads by using simple hardware structures to record transactional characteristics. More specifically, we notice strong temporal locality in contended addresses in transactional applications. By detecting when conflicting locations change, we can identify *when* contention is likely to dissipate.

To evaluate HARP, we compare it against "Bloom Filter Guided Transaction Scheduling" (BFGTS) [3], a state-of-the-art transaction scheduling technique, and LogTM [17], a well established HTM design. Our evaluation includes single-application setups, comprising a scenario with the same number of threads as cores, and a scenario with more threads than cores. We provide insights on when using more threads can extract additional parallelism, and show that HARP outperforms LogTM and BFGTS on average by 109.7% and 30.5% respectively. Moreover, we are the first to study the performance implications of a transactional multi-application setup where, again, our technique outperforms the other evaluated

[1]Work done while at Microsoft Research, Cambridge

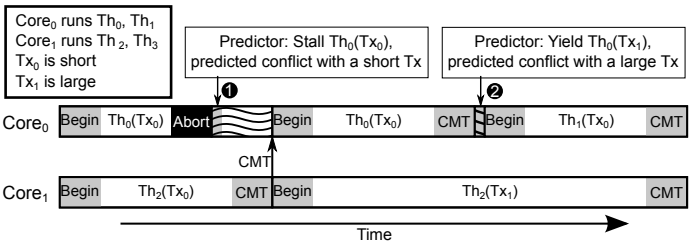

Fig. 1: Example of efficient use of computational resources.

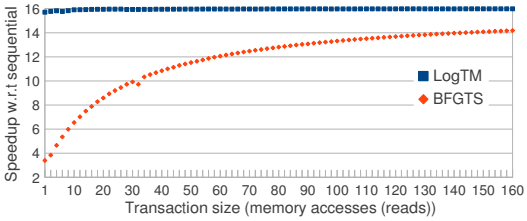

Fig. 2: Overheads of evaluated systems at different commit throughputs. Eigenbench with varying transaction sizes, 128K iterations and 16 cores.

proposals. In addition, we show that HARP is significantly more accurate in terms of predictions and resource utilization for all the evaluated setups. Compared to BFGTS, HARP has on average 1.7× and 2.2× better abort rates for single-application and multi-application workloads respectively.

II. Background and Related Work

Initial efforts on Software TM (STM) contention managers by Scherer and Scott use a set of heuristics to abort transactions and choose backoff duration when facing a conflict [20]. Further developments focused on user-level support to reduce contention, by either using runtime metrics like commit rate or dynamically discovering pairs of transactions that should not be executed in parallel [1, 12, 22]. All proposals mentioned above are reactive – imposing measures *after* conflicts happen without trying to avoid future conflicts.

In the field of HTM there has been less research on this area. Exponential backoff, as introduced in LogTM [17], is the most common contention management mechanism adopted in HTM designs. This was later used by Bobba *et al.* [7] for a thorough analysis identifying several performance pathologies present in HTM systems, including some that are closely related to contention management issues. The solutions proposed were not investigated in depth as it was not the focus of the paper.

Adaptive Transaction Scheduling (ATS) by Yoo and Lee [24] proposes queuing transactions in a centralized hardware queue if the amount of contention seen surpasses a preset threshold. ATS has little impact on performance when contention is low, and ensures single global lock performance for contended scenarios with small hardware and software requirements. However, serializing all transactions when contention intensity increases can be overly pessimistic, as not all transactions have to be highly contended. Moreover, like backoff-based policies, this mechanism is reactive and takes action *after* contention is already present in the system.

Blake *et al.* were the first to introduce proactive mechanisms to manage contention. Proactive Transaction Scheduling (PTS) is one such technique [5]. PTS employs a global software graph structure that maintains the confidences of conflict, with nodes representing transactions and edges representing the confidence level of a conflict reoccurring in the future. PTS can schedule more optimistically than ATS, thus attaining better performance. However, PTS needs to query a global data structure at the beginning of each transaction and update it when committing or aborting.

Bloom Filter Guided Transaction Scheduling (BFGTS) [3] outperforms PTS by employing a hardware accelerator and better Bloom filter manipulations using a metric termed *similarity* – a measure of memory locality present throughout different executions of a transaction. If two transactions with high similarity conflict, the conflict is likely to be persistent. However, this approach may not be accurate because two transactions could conflict very infrequently while still having high similarity, especially if they perform a large number of reads over the same locations. BFGTS is largely implemented using (1) software data structures that store confidences of conflict, per-transaction Bloom filters, and similarity values; and (2) runtime routines that execute when the system serializes, commits, or aborts a transaction. These routines can be larger than the transaction itself, and may not be compatible with arbitrary transactional codes (e.g., different languages). Per-core hardware support includes a list of transactions running in remote cores, an additional 2KB cache, and a Bloom filter to infer memory locality. This hardware performs a prediction in a few cycles at the beginning of a transaction, but cache misses can increase prediction latency.

III. Overview and Motivation

Overview example: Figure 1 illustrates how abort prediction enables efficient utilization of parallel resources with a simple example. It shows two cores, each executing two threads from the same application. Each thread has two transactions, where the first is short (Tx_0) and the second is long (Tx_1).

The example assumes an initial state where software threads Th_0 and Th_2 are both allowed to execute Tx_0 concurrently and eventually transaction Tx_0 in Th_0 aborts, meaning that $Core_0$ mispredicted the conflict. An HTM system without abort prediction support would now blindly try to re-execute the transaction, possibly leading to more conflicts and inefficient resource utilization. However, if the system is aware of contention it can proactively take steps to avoid it. At time ❶, $Core_0$'s predictor decides to stall the transaction because it predicts a conflict is likely to happen with a short transaction. Thus, in this case, waiting until the short transaction finishes makes sense. When $Core_1$ commits its transaction (Tx_0), its predictor allows the execution of the next transaction (Tx_1) of the same thread Th_2, and the stalled execution in $Core_0$ can be resumed with the approval of its predictor. $Core_0$ can now successfully commit its transaction, but when trying to move on to the next transaction (Tx_1), the predictor preempts the thread because a conflict is predicted using past history (explained in depth later). Now, at time ❷, the conflict is against a transaction known to be long, so the system decides to yield the thread Th_0, and Th_1 is granted permission to start execution. The example ends with both running transactions committing in parallel. Note that if Th_0 had not yielded and Tx_1 is contended, $Core_0$ would have probably wasted time or even experienced a series of aborts until $Core_1$'s transaction

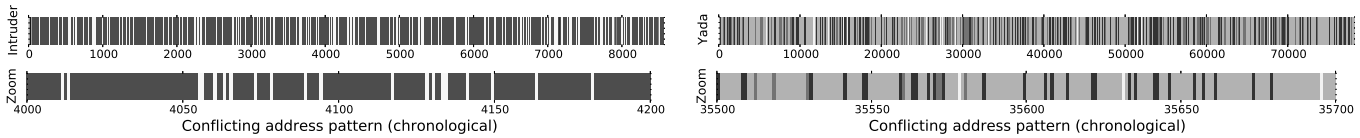

Fig. 3: Chronological distribution of conflicting addresses for a transaction of interest in Intruder (left) and Yada (right). The x axis represents cumulative abort count. Each different grey scale level represents a different conflicting address.

commits, whereas with abort prediction support a different transaction has executed and committed meanwhile.

Why do we need a hardware solution? Previous techniques rely on software components in their designs. To understand the overheads imposed by such components and the prediction mechanism in general, we perform an experiment using Eigenbench [15], a flexible exploration tool for TM systems. We configure Eigenbench to have no contention and to maximize total transactional execution time.

We evaluate LogTM and BFGTS using its best performing configuration. Figure 2 shows our experiments on a range of transaction sizes (smaller transactions demand higher commit throughput). The smallest transaction size evaluated performs one read operation and a small amount of work with the read data. Since there is no contention, LogTM scales almost linearly with any transaction size. BFGTS experiences a notable performance degradation with small and medium size transactions. Even with relatively large transactions (more than 100 reads) the performance gap under no contention is significant. The hardware accelerator of BFGTS performs a quick decision at the beginning of each transaction, however, having to interrupt the normal flow of execution on every commit (and abort) to execute additional code is the main cause of the slowdown seen in the chart. With a hardware solution we aim to minimize these overheads and deliver performance close to LogTM in uncontended scenarios.

Detecting conflict recurrence: An efficient abort prediction mechanism needs to track transaction characteristics in order to anticipate when conflicts are going to happen. It must also possess the capability to detect when conflicts dissipate. To this end, we introduce the use of *conflict lists*. A transaction's conflict list contains the last few conflicting addresses that triggered an abort; locality in such addresses is an indication that contention between two transactions is recurring in nature. These lists can be of small size, thus suitable for a hardware approach such as ours where the amount of information that can be kept is limited. To motivate this design choice, we show a study done using two of the most contended applications of the STAMP benchmark suite [8]: Intruder, a network packet intrusion detection program, and Yada, a Delaunay mesh refinement algorithm. For both applications we have looked at the history of conflicting cacheline addresses that cause an abort. More specifically, we monitored one transaction of interest (long and contended) for one of the executed threads.

Figure 3 shows two bars for each application with the chronological distribution of conflicting addresses that triggered an abort for the studied transaction. Each upper bar shows the entire sampling, while the lower bars show a magnified view of a representative region. Each address has a different grey scale level associated. The `x` axis quantifies the total number of aborts seen so far, each being triggered by a conflicting address. For better visualization, ten addresses are considered for Intruder and five for Yada, enough to cover more than 98% of the total number of aborts. As can be seen, conflicting addresses present high temporal locality, with a dominant address in both cases. These addresses with high locality are easy to capture with the proposed conflict lists.

A conflict between two transactions is likely to be persistent if one of the transactions accesses an address present in the conflict list of the other transaction, and it has likely dissipated otherwise. For example, in applications where contention is data dependent, like Yada, two concurrent transactions may conflict when operating over the same subset of data (addresses), and the conflict will likely dissipate when one of the transactions starts operating over different data (i.e, the transaction does not access addresses present in the other transaction's conflict list). Similarly, if contention is due to accessing a data structure, like in Intruder, conflicts might be present depending on which sections or nodes (addresses) of the data structure are accessed by concurrent transactions. We expect this observation to hold true for most TM use cases, as such conflicts are often unavoidable in parallel programs.

HARP versatility: HARP is largely decoupled from specific HTM conflict detection and management protocols, requiring just the knowledge of conflicting addresses that trigger an abort. This information is, typically, easy to gather in most designs. Lazy conflict detection has been found to make a system more robust under high contention [8, 21]. This is because one transaction aborts only because another transaction has successfully committed. Though a lazy system as a whole makes progress, individual threads waste substantial computational resources due to aggressive speculation. Simpler HTM implementations tend to use eager conflict detection – e.g., implementations based on extensions to traditional cache coherence protocols. A mechanism like HARP that aims to (a) prevent concurrent execution of conflicting transactions, (b) provide low abort rates, and (c) swap potentially conflicting transactions for useful work; makes an eager system become robust under high contention. In addition, eager systems present the following advantages: (a) can benefit from fast local commits, and (b) eager conflict detection lets HARP take informed decisions earlier regarding the course of execution. For these reasons we frame our study in eager systems.

A hardware approach like HARP transparently provides support for arbitrary transactional codes (i.e., different languages or compilers), which may not be compatible in a software-based approach with specialized routines. In addition, HARP does not need to interrupt the normal flow of execution on the core on every commit and abort as previous techniques require [3, 5]. Finally, HARP's prediction latency and bookkeeping operations are not affected by inherent overheads present in software routines, e.g., cache misses.

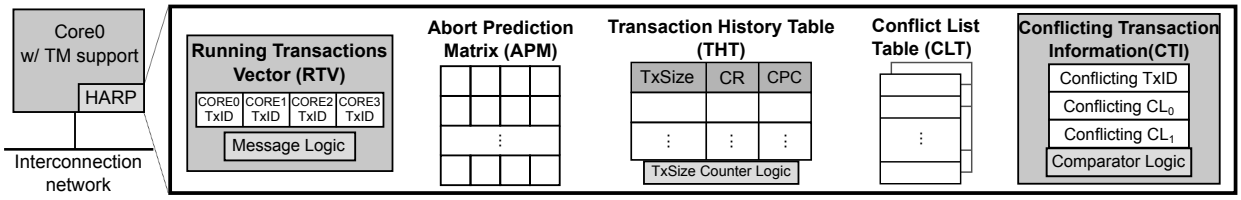

Fig. 4: Extensions to a TM-aware core. Assuming a 4-core system for the RTV and a 2-way CLT. The APM, THT, and CLT have the same number of entries.

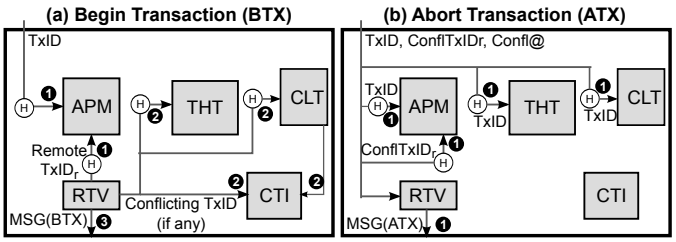

Fig. 5: Schematic overview of communication between HARP hardware structures. A subset of the bits from the TxID (PC) are used to index the APM, THT, and CLT – denoted as *H* (hash function) in the figure.

IV. HARP Design and Operation

A. HARP Hardware Structures

Figure 4 illustrates the necessary per-core hardware structures to implement HARP. These structures track important information about current and past transactional executions. The Running Transactions Vector (RTV) has as many entries as cores and tracks a list of transactions currently running on remote cores. Each entry stores a static identifier (i.e., the program counter) of a remote transaction (if any) termed *TxID*'s. The Abort Prediction Matrix (APM), Transaction History Table (THT), and Conflict List Table (CLT) are tagless structures with the same number of entries, which are indexed by TxID. The APM contains a 2-bit saturating counter in each cell. Each counter indicates the confidence of conflict between two transactions. The THT and the CLT store past information from previously executed instances of the transactions. Each entry of the THT contains the following per-transaction information: (a) the *average size* (TxSize) of committed instances, (b) a 4-bit saturating counter that indicates the *contention ratio* (CR), and (c) a 4-bit saturating counter indicating the number of *consecutively predicted conflicts* (CPC) by HARP. The CLT contains conflict lists stored in a set associative manner. Each entry of a set stores an address of the transaction's conflict list (last few addresses that caused an abort). Finally, a few additional registers and some glue logic is necessary. These registers, collectively called Conflicting Transaction Information (CTI), are used to store the TxID and conflict list of a possibly conflicting transaction upon a predicted conflict.

Figure 5 shows a communication overview between HARP structures during transactional operations. At the beginning of a transaction (Figure 5a) a prediction is performed. ❶ The RTV and APM are used to determine if a remote transaction has a high confidence of conflict with the transaction starting locally. If a conflict is found to be likely, ❷ information about the conflicting transaction is gathered from the THT to decide whether to stall or yield the thread. Additionally, the conflict list is read from the CLT and stored in the CTI. Otherwise, if no conflict is predicted, ❸ a non-blocking message is sent through the coherent interconnect to inform remote cores to update their RTVs, and the transaction starts its execution.

On transaction abort (Figure 5b), after the speculative state is rolled back, ❶ the confidence of future conflict between the two transactions is incremented in the APM, statistics in the THT and the conflict list in the CLT are updated, and a message is sent to inform remote cores to update their RTVs. On transaction commit, the previously conflicting TxID (if any) stored in the CTI is used to update the confidence of future conflict, the average transaction size is updated in the THT, and a message is sent to inform remote cores.

B. HARP Operational Details

Performing a prediction: Figure 6 details with a flowchart the process of predicting whether a transaction *TxID* will conflict or not. HARP iterates over the RTV until a conflict is found or the end of the RTV is reached (conflict not predicted). The APM is indexed by $TxID$, the corresponding row of the matrix can be seen as the set of confidences that $TxID$ might conflict with remote transactions. To know if a conflict with a remote transaction $TxID_r$ is likely to happen, $TxID_r$ is used to index by column, obtaining the cell with the confidence of conflict. The confidences are represented using 2-bit saturating counters, where the two upper states predict conflict and the two lower states predict no conflict. If a conflict is not predicted, the transaction can start its execution. Otherwise, if a conflict is predicted, HARP uses the local knowledge stored in the THT and CLT to infer the transactional characteristics of the remote conflicting transaction. The conflicting transaction identifier and its conflict list are stored in the CTI to later adjust confidences of conflict at commit time. If the size of the conflicting transaction exceeds a threshold, an exception is thrown and its handler will yield the thread in a similar way `pthread_yield()` does. Otherwise, HARP will stall the execution until the conflicting transaction is no longer running. Note that the CTI registers are part of the thread context, i.e., they are saved and restored on a context switch.

Identifying persistent conflicts and committing: We can distinguish between two kinds of running transactions: (a) the ones that start without predicting any conflict, and (b) those that execute after stalling or yielding due to a prediction (serialized). If the transaction was serialized, it has valid CTI data in the registers. Throughout the execution of a serialized transaction, the memory requests are compared against the addresses in the conflict list (CTI registers) of the previously predicted conflicting transaction. This is a crucial point to learn if a conflict has dissipated or is still present. If the transaction accesses an address present in the CTI conflict list, it means that the conflict is potentially persistent, and the transaction had a chance to execute simply because a potentially conflicting transaction instance was not concurrently running; in

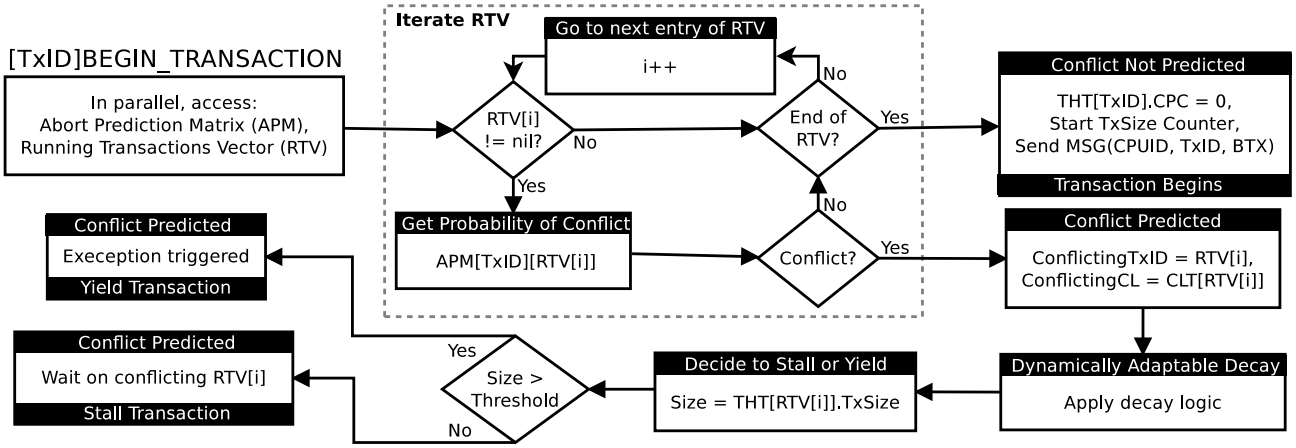

Fig. 6: Flowchart depicting the process of performing a prediction in HARP for a certain transaction *TxID*.

this case, the confidence of conflict is increased at commit time. If the transaction does not access an address in the CTI conflict list, it means that the conflict between the two transactions is perhaps no longer present, and the confidence of conflict is decreased. Additionally, at commit time the average transaction size and the contention ratio (CR) are updated, the CTI registers are also cleared.

Aborting a transaction: When a transaction aborts due to a conflict, the aborting core increases the confidence of conflict between the two transactions in the APM. The contention ratio (CR) in the THT is incremented, and the transaction's conflict list is updated in the CLT with the conflicting address. Since conflict lists can have repeated elements, the replacement policy is simple. There is no need to do a look up before replacing; instead, an LRU bit decides which entry is replaced. The broadcast message sent when a transaction aborts is slightly larger, it also contains the core identifier and TxID of the remotely conflicting transaction, and the conflicting address. In this manner, besides remote cores updating their RTVs, the remotely conflicting core can also update the confidence of conflict and the conflict list of the remotely conflicting transaction in its local structures. These remote updates on abort are important because they make a transaction aware of a potential conflict and a conflicting address.

Non-blocking communication: When a core starts or exits (commits or aborts) a transaction, communication with remote cores is necessary to keep the RTVs updated. This communication is done via small broadcast messages that include the core identifier, the TxID, and the action being performed (e.g., committing). These messages are non-blocking, which can lead to outdated information in remote cores for a small window of time, but this is not a correctness issue and far less critical to performance than adding synchronization. The number of such messages is small when compared to coherence messages (~1% on average in our simulations). Moreover, a large number of simultaneous messages implies a high commit rate, where HARP would not need to interfere. In high contention scenarios, HARP serializes conflicting transactions, which reduces the number of messages. These facts suggest that communication is not a limiting factor for the design to scale (see Section V-F for related evaluation).

During the process of predicting a conflict, committing, or aborting, all information is available locally. Such a distributed approach eliminates synchronization overheads between cores and contention when accessing the hardware structures.

Dynamically adaptable decay: The decay targets transactions where contention varies with time, allowing them to execute optimistically faster when contention dissipates. As shown in Figure 6, the decay is applied after a conflict is predicted and implements a simple algorithm as follows: if the number of consecutively predicted conflicts by HARP is at least equal to the transaction's contention ratio, the confidence for the recently predicted conflict is decremented and the CPC counter is reset. Otherwise, the CPC counter is increased. This enables transactions that commit often to decrement their confidences of conflict faster, while contended transactions will need to predict a larger number of consecutive conflicts in order to see their confidences of conflict decremented by the decay. As contention increases, the chances to apply the decay decrease at a faster rate, since having a large number of consecutive predicted conflicts is increasingly unlikely.

Execution example: Figure 7 presents a self-contained step-by-step example of HARP's operation.

V. Evaluation

A. Simulation Environment

To evaluate HARP we compare it to two HTM baselines, LogTM [17], a well established system; and a state-of-the-art transaction scheduling technique: Bloom Filter Guided Transaction Scheduling (BFGTS) [3]. In our experiments, both HARP and BFGTS use the LogTM architectural framework for basic TM support. We use the M5 full-system simulator [2]. This simulator was made publicly available by the BFGTS authors [4], thus assuring the BFGTS baseline is faithfully modeled. Queuing delay and resource contention in the memory subsystem and in added structures has been accounted for. The simulation parameters are detailed in Figure 8.

We use the best performing BFGTS configuration, which skips most calculations in software routines when there is low contention. HARP's prediction cost is modeled as one cycle per lookup in the APM, i.e., 15 cycles in the worse case. Lower prediction cost can be achieved by fetching the entire row of the APM, filtering the columns of interest, and using a set of comparators in parallel – trading hardware footprint for

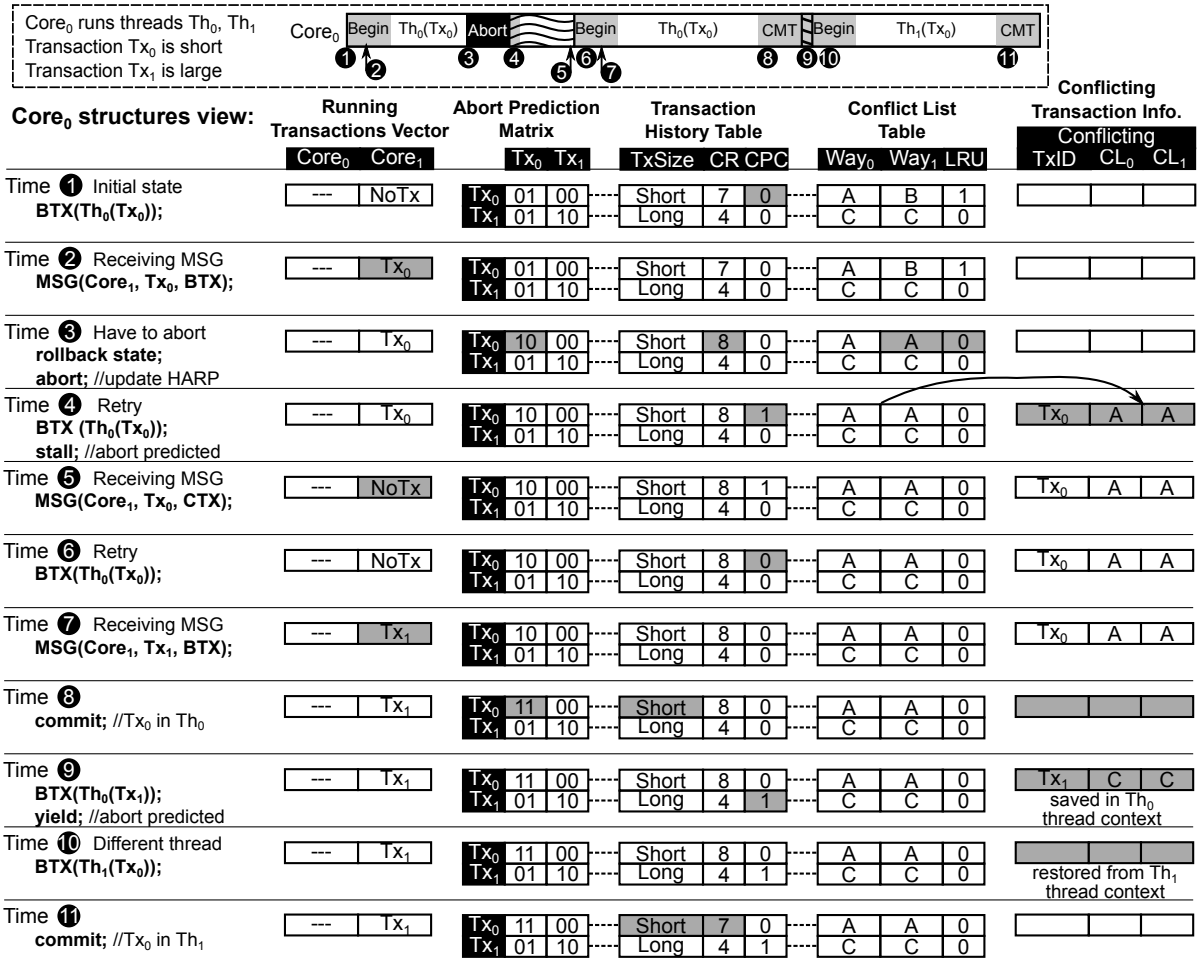

Fig. 7: HARP execution diagram for a two core system. The box at the top depicts a sequence of events for $Core_0$, matching those presented in Figure 1. The rest of the figure shows changes in $Core_0$'s HARP hardware structures at each step (shaded areas), outgoing messages are not shown. The transaction begin at time ❶ triggers the predictor, since no other transactions are running on the system, it can start normally. At time ❷ a remote message from $Core_1$ is received and the RTV is updated accordingly. At time ❸ the transaction aborts due to a conflict with Tx_0 running on $Core_1$. At time ❹ the transaction tries to restart, but this time the RTV is not empty, a conflict is predicted and the CTI registers populated. Since the conflict is predicted against a transaction marked as "short" in the THT, the execution is stalled. Later, at time ❺, a message is received indicating that the conflicting transaction has finished, allowing $Core_0$ to retry again and start ❻. At time ❼, a message is received indicating $Core_1$ started to execute Tx_1, updating the RTV. At time ❽, the running transaction in $Core_0$ commits with valid CTI information because it was serialized. In this example, we consider that during the execution address A was touched, making the previously predicted conflict potentially persistent, so the confidence of conflicting again in the future is increased. At time ❾, $Core_0$ tries to start Tx_1, but a conflict is predicted with a large remotely running transaction, yielding the current thread. Note that before yielding, the CTI info is populated and will be saved as part of the thread context when yielding. At time ❿, a new thread Th_1 is granted execution, restores CTI information (null in this example), and starts executing Tx_0. The transaction commits at time ⓫, updating local information.

prediction latency. The transaction size threshold that decides when to stall or yield is set to half the average time it takes the kernel to perform a context switch in our system. Note that after stalling, the transaction is not guaranteed to execute as a new abort could be predicted. This transaction size threshold allows for at least two consecutive stalls before having a penalty larger than yielding.

We use the STAMP [8] benchmark suite with nine different benchmark configurations. Figure 9 describes the input parameters used and the number of transactions defined in each benchmark. We exclude *Bayes* because of its non-deterministic exiting conditions, leading to inconclusive results due to high runtime variability, as noted by many researchers [3, 6, 8].

B. Comparison of Hardware Costs

Figure 10 shows the storage requirements for HARP and BFGTS. Implementing HARP requires an additional storage of 2.06KB on each core, roughly 3% of a 64KB L1 cache. HARP requires less storage than BFGTS. This is because BFGTS uses an additional 2KB cache to speedup accesses to its software data structures. Moreover, a cache needs additional logic (e.g, tags), not considered in this comparison.

C. Evaluation Methodology

Our evaluation includes three different system setups: (a) a setup with a single-application using the same number of threads as cores, (b) a setup with a single-application where four threads are assigned to each core, and (c) a setup with two different applications where one thread of each application is assigned to each core, i.e., two threads per core each from a different application (multi-application workloads). While single-application performance is still critically important, we believe that for TM to be widely accepted, it also needs to deliver good performance in such multi-application scenarios.

Cores	16 in-order 2GHz Alpha cores, 1 IPC
L1 Caches	64KB 2-way, private, 64B lines, 1-cycle hit
L2 Cache	16MB 16-way, shared, 64B lines, 32-cycle hit
Memory	4GB, 100-cycle latency
Interconnect	Shared bus at 2GHz
Linux Kernel	Modified v2.6.18
HARP Structures	64 entries for APM, THT, and CLT 2 addresses per conflict list
BFGTS Structures	2048bit signatures for BFGTS commit routines 2KB 16-way confidence cache, 64B lines, 1-cycle hit

Fig. 8: Simulation parameters.

Benchmark		**Input parameters**	**Num Tx**
Genome	(G)	-g4096 -s32 -n524288	5
Intruder	(I)	-a10 -l32 -n8192 -s1	3
KMeans-High	(K)	-m15 -n15 -t0.05 -i random50000_12	3
KMeans-Low		-m40 -n40 -t0.05 -i random50000_12	3
Labyrinth	(L)	-i random-x96-y96-z3-n128.txt	3
SSCA2	(S)	-s15 -i1.0 -u1.0 -l3 -p3	3
Vacation-High	(V)	-n8 -q10 -u80 -r65536 -t131072	1
Vacation-Low		-n2 -q90 -u98 -r65536 -t131072	1
Yada	(Y)	-i ttimeu10000.2	6

Fig. 9: STAMP input parameters and number of transactions.

Hardware structure	**Equation of cost**	**Cost (bytes)**
Running Transactions Vector	16 entries × (1 TxID/entry × 48 bits/TxID)	96
Abort Prediction Matrix	64 entries × (64 counters/entry × 2 bits/counter)	1024
Transaction History Table	64 entries × ((1 counter/entry × 16 bits/counter) + (2 counters/entry × 4 bits/counter))	192
Conflict List Table	64 entries × ((2 addresses/entry × 48 bits/address) + 1 LRU bit/ entry)	776
Conflicting Transaction Information	(1 register × 48 bits/register) + (2 registers × 64 bits/register)	18
HARP Total Storage	Sum of the above	**2.06 KB**
BFGTS Total Storage	RTV-like structure (96 bytes) + Additional confidence cache (2 KB) + Bloom filter (2048 bits)	**2.34 KB**

Fig. 10: HARP and BFGTS hardware costs for one core.

$$\text{Efficiency ratio} = \frac{\text{useful\_tx (cycles)}}{\text{useful\_tx} + \text{wasted\_tx} + \text{abort recovery} + \text{stall/yield/backoff} + \text{BFGTS commit routine (cycles)}} \quad (1)$$

In fact, as parallel programming becomes ubiquitous, future systems would have several multithreaded applications running concurrently in the common case. To the best of our knowledge, we are the first to study multi-application transactional scheduling in an HTM environment.

For the first setup where the same number of threads as cores is used, it is inefficient to yield threads when aborts are predicted. In order to compare BFGTS and HARP fairly, we disable the yield option for this particular setup. This can be accomplished by letting the kernel scheduler notify the hardware when yielding is not useful, as the scheduler would have the knowledge to make such decision. We expect such operating system support to be present in an HTM system. For the multi-application setup, we had to modify the design of BFGTS because the original proposal was not able to deal with multiple applications. In addition, we allow BFGTS to yield. Originally the library would not yield when the number of threads is not larger than the number of cores for a particular application; but we observed that yielding judiciously benefits BFGTS when threads from different applications are available.

We provide execution time breakdowns, scalability analysis, and statistics for the evaluated workloads. Execution time breakdowns are normalized to LogTM, and the following components are shown – non-transactional time (*non-tx*), barriers time (*barrier*), useful transactional time (*useful-tx*), wasted work from aborted transactions (*wasted-tx*), time spent in abort recovery (*abort recovery*), time spent due to contention management handling (*stall/yield/backoff*), and time spent by BFGTS in the software commit routine. Prediction cost was not visible in charts and it is attributed to other components based on prediction outcome, e.g., to *useful-tx* if the transaction starts and commits. The statistics that we show include a metric that captures how effective contention management is in BFGTS and HARP. This metric, shown in Equation (1), is an efficiency ratio that compares the amount of useful cycles with the inherent design overheads due to bad predictions and serialization costs that lead to inefficient resource utilization.

D. Single-Application Results

One thread per core: Figure 11 presents the execution time breakdown for the evaluated workloads. Overall, the backoff strategy employed by LogTM fails to manage contention and exhibits a large amount of wasted work and serialization overheads (backoff time) when compared to BFGTS or HARP. Dynamically avoiding the execution of transactions that are likely to fail improves performance and scalability by over 2× on average (see Figure 12), while abort rates diminish by 6×, as shown in Figure 13. These are clear indicators that proposals like BFGTS and HARP are likely to have a significant impact when applied to any HTM system.

Performance improvements of HARP when compared to BFGTS are due to (a) comprehensive hardware support, yet with a smaller hardware footprint than BFGTS (see Section V-B), thus avoiding software data structures and runtime routines; and (b) greater prediction accuracy by focussing only on addresses that actually cause contention. HARP performs better than BFGTS for all the evaluated workloads, attaining 30.5% performance improvement on average.

The BFGTS commit routine accounts for a significant amount of the execution time in workloads with small transactions like *Intruder* (27%) and *KMeans-High* (11%). This is because the time spent in the routine, which is used to adjust confidences of conflict, is constant and cannot be amortized when executing short transactions. Hence, in general, workloads with small transactions are penalized using BFGTS. However, HARP use of conflict lists results in a small, fixed maintenance cost that does not depend on workload characteristics. Having a better transactional scheduling policy and fewer aborts can also reduce non-transactional and barrier time. By executing only those transactions that are likely to commit, interactions with non-transactional code are minimized, e.g., the number of stalls when trying to access transactionally modified data is reduced. In addition, fewer aborts can reduce overall load imbalance, as it happens in *Vacation*.

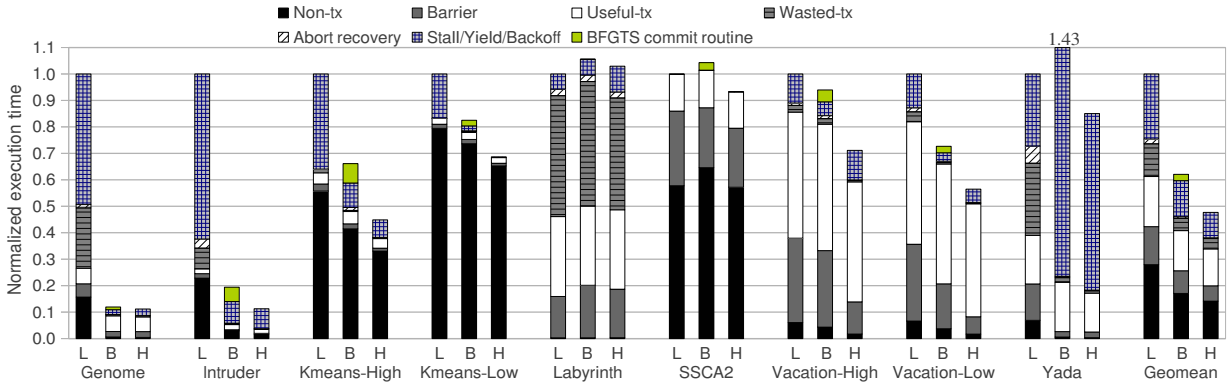

Fig. 11: Normalized execution time breakdown for 16 threads in single-application workloads. **L** – LogTM; **B** – BFGTS; **H** – HARP.

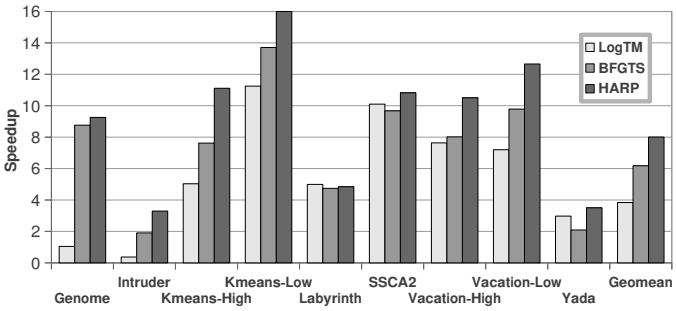

Fig. 12: Speedup of 16-threaded executions compared to sequential execution.

Benchmark	Abort Rate (%)			Efficiency Ratio	
	LogTM	BFGTS	HARP	BFGTS	HARP
Genome	65.3	3.6	3.7	0.64	0.65
Intruder	70.2	14.6	7.3	0.12	0.17
KMeans-H	23.9	9.9	5.3	0.20	0.34
KMeans-L	13.0	3.9	0.5	0.39	0.89
Labyrinth	15.5	7.8	12.7	0.35	0.36
SSCA2	0.0	0.0	0.0	0.83	1.00
Vacation-H	11.6	7.0	2.4	0.79	0.79
Vacation-L	10.0	3.2	1.2	0.87	0.89
Yada	56.8	6.6	5.0	0.13	0.18
Geomean	**11.3**	**3.3**	**1.9**	**0.38**	**0.48**

Fig. 13: Benchmark statistics for evaluated systems.

Regarding higher prediction accuracy, HARP offers promising abort rates (see Figure 13), obtaining near-linear speedup in *KMeans-Low*. Moreover, these improvements in abort rate are not due to overserializing transactions; as our efficiency ratio demonstrates, HARP is 1.27× more efficient than BFGTS in terms of useful computational cycles. This indicates that the conflict lists and the dynamically adaptable decay quickly adjust the confidences of conflict in accordance with actual contention levels that are present at any given time. In fact, in workloads like *KMeans* and *Yada* where contention varies with time, the decay allows to optimistically execute transactions faster when necessary – e.g., in *Yada* BFGTS overserializes transactions that could run in parallel (note the large stall time), but HARP decay logic detects this fact, allowing parallel execution while maintaining a lower abort rate.

Four threads per core: We execute the benchmarks with 64 threads, pinning 4 threads to each core. Both BFGTS and HARP present similar execution time breakdowns for all the benchmarks when compared to their 16-threaded executions. HARP attains an average speedup of 25.8% over BFGTS due to no software runtime overheads and less serialization (stall and yield time) as a result of better predictions, with average abort rates of 4.1% for BFGTS and 2.8% for HARP.

However, an interesting point is to determine if such an overcommitted system is beneficial by comparing these workloads to their 16 threaded counterparts. Workloads with few transactions are not likely to benefit from an overcommitted system. This is the case of *Vacation*, which only has one transaction defined in the code, hence less room for improvement when switching to a different thread. Also workloads like *SSCA2* and *KMeans-Low* where contention is minimal cannot scale further, and the overheads of managing additional threads can hurt scalability – e.g., in *SSCA2* there is a significant loss of scalability from 10× to 3.5× (see Figure 14).

Yada exhibits significant benefits for all the evaluated systems when using 64 threads, as Figure 14 shows, where striped bars indicate configurations with better scalability than in the 16-threaded setup. *Yada* has the largest number of transactions (six). Moreover, its transactions are large with moderate contention. With these characteristics it is easier to find additional parallelism when switching between different threads, because the chances of executing a non-conflicting transaction are higher. In addition, large transactions help amortize yield time costs. *Yada* is the only benchmark that significantly improves its efficiency ratio when using 64 threads, from 0.18 to 0.31 for HARP. Our results suggest that large transactional codes, with medium or large transactions, may be necessary to benefit from overcommitted setups. This is likely to become a common case as more transactional applications become available.

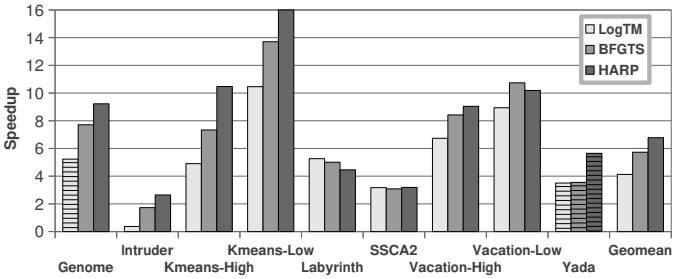

Fig. 14: Speedup of 64-threaded executions compared to sequential execution. Striped bars indicate significant performance boost compared to 16-threaded executions.

E. Multi-Application Results

In this setup, each core executes two threads from different applications. We only consider the '-High' versions of *KMeans* and *Vacation*, and evaluate all the possible combinations of 2 applications out of the 7 possible, which amounts to 21

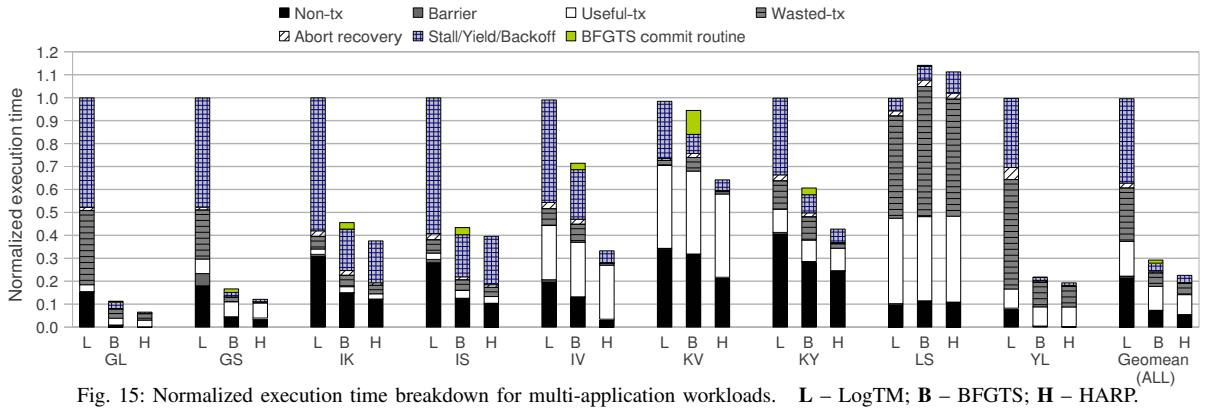

Fig. 15: Normalized execution time breakdown for multi-application workloads. **L** – LogTM; **B** – BFGTS; **H** – HARP.

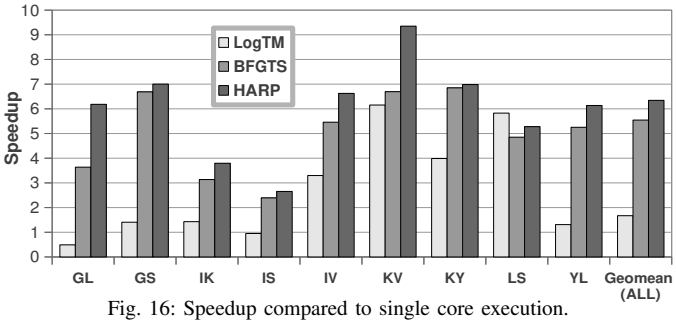

Fig. 16: Speedup compared to single core execution.

Benchmark	Abort Rate (%)			Efficiency Ratio	
	LogTM	BFGTS	HARP	BFGTS	HARP
GL	90.1	32.4	3.7	0.28	0.46
GS	34.8	2.9	1.1	0.54	0.81
IK	46.9	21.4	15.2	0.09	0.09
IS	43.2	17.9	14.7	0.11	0.10
IV	37.6	25.6	3.1	0.40	0.79
KV	17.1	11.6	2.8	0.57	0.86
KY	23.2	8.3	4.4	0.29	0.54
LS	0.0	0.0	0.0	0.36	0.37
YL	94.2	41.5	3.1	0.39	0.45
Geomean (ALL)	**24.1**	**7.3**	**3.3**	**0.38**	**0.47**

Fig. 17: Benchmark statistics for evaluated systems.

different workloads. The workloads are named with the initials of each application, the legend is in Figure 9 – e.g., 'GL' executes *Genome* and *Labyrinth*. To make accurate measurements, we synchronize the two applications at the beginning of their parallel sections. When an application reaches the end of its parallel section, that application is no longer considered for execution. Similarly, when a core finishes all of its threads (applications), that core is considered to be available for other tasks, and hence does not contribute to the execution time. To measure scalability, the slowest core is considered.

Figure 15 shows the execution time breakdown and Figure 16 the scalability results. We show a representative selection of 9 workloads, plus the geometric mean which considers the 21 evaluated workloads. LogTM fails to deliver good performance, experiencing a large number of aborts and high backoff overheads. Thus, policies that cannot dynamically decide what is the best course of action are not suitable for future systems where parallel applications might be dominant. However, BFGTS and HARP deliver higher performance because they can swap potentially wasted computation for potentially useful work.

HARP performs better than BFGTS for all the evaluated workloads, achieving a 29.5% improvement on average. This is due to four main reasons. First, BFGTS is overly pessimistic in general, leading to a larger serialization time (stall and yield). We observe a notably larger number of predicted conflicts in *GL*, *GS*, *KV*, *KY*, and *IV*; in the latter BFGTS predicts 4× more conflicts. Second, HARP makes better predictions than BFGTS; as Figure 17 indicates, even though HARP predicts a lower number of conflicts, it still attains remarkably better abort rates. Hence, HARP allows for increased parallel execution of transactions while keeping lower abort rates. Third, BFGTS decides whether to stall or yield depending on the number of cache lines touched by the transaction, which we find is less accurate than HARP's approach that uses actual execution time. Finally, as observed before, small transactions (*Intruder* and *KMeans*) penalize BFGTS performance by increasing the software commit routine time.

Labyrinth and *Intruder* have lower scalability and significantly larger execution time than *KMeans* and *SSCA2*. Hence, scalability for *IK*, *IS*, and *LS* tends to be close to that seen in *Labyrinth* and *Intruder* for single-application (Figure 12). However, for combinations where the execution time is more evenly distributed, like *IV* and *KY*, we can observe how scalability is significantly higher than the one reported for *Intruder* and *Yada* respectively. *YL* achieves 6.1× speedup, higher than both *Yada* and *Labyrinth* when executed as single applications.

F. Sensitivity analysis

System parameters: We evaluate our technique changing two major system parameters. First, we modified the size of HARP hardware structures to have no collisions (i.e., two different TxID's mapping to the same entry) for the multi-application setup, since for single-application no collisions were found. Our results with no collisions did not show any significant changes in the abort rates of the affected multi-application workloads. This is because very few collisions were present in the first place, one in *GS* and one in *GY*.

Second, we looked into conflict lists size sensitivity. Throughout our evaluation, we have used conflict lists of size 2. We evaluate single-application workloads with conflict lists of size 1 and 4. Low contention applications like *SSCA2* are not affected by the conflict lists size, due to their low conflict rates. High contention applications like *Labyrinth*, *Yada*, and *Intruder* did not experience significant variation either due to a single dominant conflicting address, as shown in

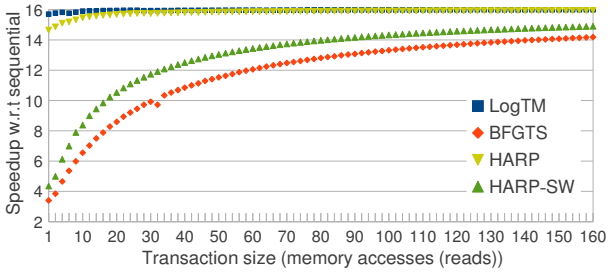

Fig. 18: Communication and prediction overheads of evaluated systems at different commit rates. Using Eigenbench with varying transaction sizes, 128K iterations and 16 cores.

Figure 3. However, '-High' versions of *KMeans* and *Vacation* present moderate contention and show a significant drop in performance when using conflict lists of size 1. This is because they have a larger set of conflicting addresses, with no dominant address, which makes HARP schedule too optimistically. Overall, we find that conflict lists of size 2 offer the best trade-off between performance and hardware cost.

Communication and prediction overheads: We expect uncontended scenarios demanding high commit throughput to expose communication and prediction overheads. We repeat the experiment from Section III, see Figure 18, adding HARP and a version of HARP that stores and maintains the THT and CLT structures in software (HARP-SW). HARP experiences a 7% slowdown for the smallest transaction size, due to communication and prediction latencies not being amortized. However, HARP rapidly closes the gap in performance with respect to LogTM, confirming that broadcast messages do not hinder scalability. In contrast, both HARP-SW and BFGTS have a severe performance drop, mainly due to additional code executed at commit time, which can make executed transaction several times larger. HARP-SW remains slightly better than BFGTS because its software operations are simpler.

Multi-application using four applications: We also evaluate a multi-application setup using four applications concurrently, which amounts to 35 different workloads. HARP again outperforms BFGTS by 20.3% on average, and attains scalability similar to that seen in the two application setup, 6.5×. In this scenario collisions did not affect performance either.

VI. Conclusions

In spite of much research, HTM performance is susceptible to degradation when contention is present. Moreover, parallel programming is becoming the norm, and systems with several parallel applications will be increasingly common. Techniques that minimize the amount of wasted work due to misspeculation and maximize computational resource utilization are necessary for TM to gain wide acceptance.

This work proposed HARP, a hardware mechanism that efficiently predicts future conflicts and avoids speculation when the probability of contention is high. The resources thus freed are, when it is deemed advantageous, utilized to schedule possibly non-conflicting codes, thereby improving concurrency and throughput. The design provides seamless support for both single-application and multi-application scenarios. Our investigation has shown that HARP outperforms, by a substantial margin, both LogTM, a popular HTM proposal, and BFGTS, the state-of-the-art proactive transaction scheduling scheme prior to this work. This is achieved with modest hardware support comprising three simple tagless structures in each core. Since HARP does not rely on software runtimes and data structures, it presents little management overhead, while simultaneously keeping the architecture relatively independent of the software that runs on it. In addition, HARP predictions can be leveraged to implement aggressive power saving schemes when no useful computation can be scheduled. We see this area as a potential direction for future work.

Acknowledgements

We would like to thank reviewers for their comments and valuable feedback. Adrià Armejach's work is supported by the Ministry of Science and Technology of Spain and the European Union under contracts TIN2007-60625 and TIN2008-02055-E. Anurag Negi's work at Chalmers is supported by grants from the Swedish Foundation for Strategic Research (SSF) under the SCHEME project.

References

[1] M. Ansari *et al.*, "Steal-on-abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering," in *HiPEAC'09*.
[2] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 simulator: Modeling networked systems," *IEEE Micro*, 2006.
[3] G. Blake, R. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," in *HPCA-17*, Feb. 2011.
[4] G. Blake, "A Hardware/Software Approach for Alleviating Scalability Bottlenecks in Transactional Applications," Ph.D. dissertation, 2011, University of Michigan.
[5] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *MICRO-42*, Dec. 2009.
[6] C. Blundell, A. Raghavan, and M. M. K. Martin, "RetCon: Transactional Repair without Replay," in *ISCA-37*, Jun. 2010.
[7] J. Bobba, K. E. Moore, H. Volos *et al.*, "Performance Pathologies in Hardware Transactional Memory," in *ISCA-34*, Jun. 2007.
[8] C. Cao Minh *et al.*, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Intl. Symp. on Workload Characterization*, 2008.
[9] J.-W. Chung *et al.*, "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory," in *MICRO-43*, Dec. 2010.
[10] C. Click, "Azuls experiences with hardware transactional memory," in *HP Labs - Bay Area Workshop on Transactional Memory*, 2009.
[11] D. Dice *et al.*, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS-14*, Mar. 2009.
[12] S. Dolev *et al.*, "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *PODC-27*, Aug. 2008.
[13] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. Morgan and Claypool Publishers, 2010.
[14] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *ISCA-20*, May 1993.
[15] S. Hong *et al.*, "Eigenbench: A simple exploration tool for orthogonal TM characteristics," in *IISWC*, Sep. 2010.
[16] Intel Corporation, "Transaction Synchronization Extensions (TSX)," in *Intel Architecture Instruction Set Extensions Programming Reference*, Feb. 2012, pp. 506–529, http://software.intel.com/file/41604.
[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *HPCA-12*, Feb. 2006.
[18] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *MICRO-34*, Dec. 2001.
[19] C. Rossbach, O. Hofmann, and E. Witchel, "Is Transactional Memory Programming Actually Easier?" in *WDDD-8*, Jun. 2009.
[20] W. N. Scherer III and M. L. Scott, "Advanced Contention Management for Dynamic Software Transactional Memory," in *PODC-24*, Jul. 2005.
[21] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *Intl. Conf. on Supercomputing*, Jun. 2009.
[22] N. Sonmez *et al.*, "Taking the heat off transactions: Dynamic selection of pessimistic concurrency control," in *IPDPS-23*, May 2009.
[23] A. Wang, M. Gaudet *et al.*, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *PACT-21*, 2012.
[24] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *SPAA-20*, Jun. 2008.