Power Efficient Hardware Transactional Memory: Dynamic Issue of Transactions

SANG WOOK STEPHEN DO and MICHEL DUBOIS, University of Southern California

Transactional Memory (TM) is no longer just an academic interest as industry has started to adopt the idea in its commercial products. In this paper, we propose Dynamic Transaction Issue (DTI), a new scheme that can be easily implemented on top of existing Hardware TM (HTM) systems, provided additional messages. Instead of wasting power and energy in transaction aborts, Dynamic Transaction Issue puts a processor core into a low-power state when there is a reasonable suspicion that the current transaction running on it will be aborted soon in the future.

We have implemented Dynamic Transaction Issue on a cycle-accurate simulator of a multicore processor system with out-of-order superscalar cores, augmented with a power package and a TM package which add accurate dynamic power estimates and a TM framework to the simulator. Our simulation results show that Dynamic Transaction Issue can achieve energy savings up to 37% from the energy consumption of a base machine with no mechanism to suppress useless aborts. We also compare Dynamic Transaction Issue with various alternative hardware TM mechanisms.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Hardware** → **Power and energy**

Additional Key Words and Phrases: Transactional memory, hardware transactional memory, dynamic issue, power consumption, energy consumption

ACM Reference Format:

Sang Wook Stephen Do and Michel Dubois. 2016. Power efficient hardware transactional memory: Dynamic issue of transactions. *ACM Trans. Archit. Code Optim.* 13, 1, Article 9 (March 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/2875425>

1. INTRODUCTION

Transactional Memory (TM) [Herlihy et al. 1993] was proposed to enhance the programmability as well as the performance of parallel programs running on a multicore or a multiprocessor system. To achieve this goal, TM removes the need for explicit locking: Mutually exclusive events are executed optimistically and corrected later if a violation of mutual exclusion is detected postmortem. As a result, TM avoids the complexity of conventional locking mechanisms aggravated when there is a need to hold multiple locks simultaneously. Some TM proposals such as Transactional Memory Coherence and Consistency (TCC) [Hammond et al. 2004] and Bulk [Ceze et al.

This material is based on work supported by the National Science Foundation under Grant No. CCF-1219186. Authors' addresses: S. W. S. Do, Ming Hsieh Department of Electrical Engineering, University of Southern California, 3740 McClintock Avenue, EEB230, Los Angeles, CA 90089-2560; email: sdo@usc.edu; M. Dubois, Ming Hsieh Department of Electrical Engineering, University of Southern California, 3740 McClintock Avenue, EEB228, Los Angeles, CA 90089-2560; email: dubois@usc.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/03-ART9 \$15.00

DOI: <http://dx.doi.org/10.1145/2875425>

2006] extend the idea to cache coherence and consistency and propose cache coherence protocols relying on the TM execution model.

In TM architectures, a finite sequence of machine instructions is enclosed in a “transaction” (or “chunk” in Ceze et al. [2006]) using special instructions such as “transaction_begin” and “transaction_end.” To guarantee the correct execution of a program, a transaction goes through three procedural phases similar to the phases in optimistic concurrency control for database systems [Kung et al. 1981]. In the “execution” phase, instructions are executed speculatively, including memory accesses. During the execution phase memory stores to shared memory remain invisible to other concurrent transactions running on other processor cores. In the “validation” phase, the transaction checks a condition that ensures its successful completion. To satisfy the condition, the transaction must not have read or written any memory location written by other concurrent transactions unless it is proclaimed the winner in such memory conflicts. The execution of a loser transaction is aborted, rolled back, and restarted from its beginning. In the final stage, the “commit” phase, the execution of the instructions in the winning transaction becomes nonspeculative and its memory updates are exposed and visible to the entire system.

To support these operations, a TM system provides at least three mechanisms. First, a conflict detection mechanism discovers any memory reference conflicts between concurrent transactions. Second, a conflict resolution mechanism selects one transaction as a winner and other participants in the conflict become losers. Third, version management keeps track of two different versions of data: new speculative data and old nonspeculative data during the lifetime of a transaction. Old data are discarded if the transaction commits and new data are discarded if the transaction aborts. Hardware Transactional Memory (HTM) systems [Ananian et al. 2005; Chafi et al. 2007; Ceze et al. 2006; Hammond et al. 2004; Moore et al. 2006; Qian et al. 2010; Rajwar et al. 2005; Yen et al. 2007] implement the three mechanisms in hardware.

HTMs are classified based on the implementations of conflict detection and version management. Using the terminology in Moore et al. [2006], “Eager” conflict detection [Ananian et al. 2005; Moore et al. 2006; Rajwar et al. 2005; Yen et al. 2007] and “Lazy” conflict detection [Chafi et al. 2007; Ceze et al. 2006; Hammond et al. 2004; Qian et al. 2010] refer to the time at which conflicts are detected between the actual time of the conflict and the time to commit. “Eager” version management [Ananian et al. 2005 (UTM); Moore et al. 2006; Yen et al. 2007] and “Lazy” version management [Ananian et al. 2005 (LTM); Chafi et al. 2007; Hammond et al. 2004; Rajwar et al. 2005] refer to the way old and new versions of the data are kept while the transaction is in progress and is not committed or aborted. Some HTM proposals [Lupon et al. 2010; Shriraman et al. 2008; Titos-Gil et al. 2011; Tomic et al. 2009] take a flexible approach between the Eager and the Lazy policies to achieve better performance.

While TM greatly improves the programmability of shared-memory multiprocessors and CMPs (Chip MultiProcessors), an obvious disadvantage of any TM system is the machine cycles wasted on transaction aborts, resulting in power and performance losses. In some worst-case scenarios, the losses can be large, as identified in Bobba et al. [2007].

One of the worst scenarios, which might occur in a TM system adopting the Lazy conflict detection scheme, in which memory stores are sent out all together after the validation phase for conflict detection, is illustrated in Figure 1. In this scenario, N transactions ($Tx1 \sim TxN$) run on N processor cores ($Pr1 \sim PrN$) and compete and conflict with each other for the same memory location(s). In the first stage, $Tx1$ is chosen as a winner by the conflict resolution mechanism and is committed successfully while $Tx2 \sim TxN$ are aborted and restarted. The aborts are denoted by “x” in the figure. This situation repeats itself until TxN is committed successfully, resulting in a total

Fig. 1. Energy loss due to repeated aborts (Lazy conflict detection).

Fig. 2. Energy loss due to repeated aborts (Eager conflict detection).

wasted energy (gray areas in the figure) of

$$\frac{(\text{energy per abort})N(N-1)}{2}, \quad (1)$$

which is $\sim O(N^2)$ if we assume that the transactions are fairly well synchronized and the energy consumed per abort is roughly the same among the aborted transactions. In the example the repeated aborts after the first one in each transaction are predictable and transactions should not restart after their first abort until all conflicts with other transactions are cleared. This is what Dynamic Transaction Issue (DTI) aims to achieve.

Figure 2 illustrates another example, in which energy waste is due to repeated aborts in a TM system adopting the Eager conflict detection scheme. In Eager conflict detection, a memory store is propagated at the execution of a store instruction during the execution phase using an underlying cache coherence protocol for conflict detection.

Table I. Repeated Aborts

Program*	B	G	I	K	L	V	Y	F	Avg
Avg. Repeated Aborts	2.9	2.0	6.6	1.2	14.5	1	2.3	3.4	4.25
Repeated Abort Percentage (%)	65	36	50	15	86	0	44	40	42

*Full program names and descriptions can be found in Table V in Section 6.1.

In Figure 2, a store in Tx1 conflicts at first with Tx2 and Tx1 is aborted. Then, after Tx1 is restarted, the same store in Tx1 again conflicts with Tx2 and Tx1 is again aborted. In this situation, Tx1 could be aborted multiple times by Tx2, wasting energy because of wasted cycles in the execution of aborted transactions, as shown in the gray area. For as long as Tx2 is running these conflicts resulting in the abort of Tx1 are predictable. Tx1 should not restart until Tx2 is finished. DTI deals with this case, too.

To observe how many consecutive aborts of the same transaction actually occur in benchmark applications on our base machine, we measured the number of such repeated aborts. Our base HTM machine (described in more detail in Section 6) has no support for suppressing restarts of aborted transactions. The results are presented in Table I. The second row “average repeated aborts” gives the average number of aborts repeated consecutively before a transaction commits given that the transaction aborts at least once. Thus these numbers do not include transactions that commit on their first execution. For example, in Bayes, an abort is repeated about 3 times in a row on average. In Vacation, every abort is committed successfully in the next try without repetition; thus only one abort is counted each time. Overall, the benchmark programs experience 4.25 consecutive aborts before committing among all transactions that abort at least once. The third row shows the fraction of aborted transactions that experience more than one consecutive abort. For example, 65% of the aborted transactions in Bayes experience at least two aborts in a row. Overall 42% of the aborted transactions suffer from multiple consecutive aborts for all the programs.

To remedy the problem of multiple consecutive aborts, we propose a power-efficient Hardware Transactional Memory which reduces the energy wasted on transaction aborts. More specifically, we introduce a hardware mechanism called DTI that dynamically suppresses the reissuing of an aborted transaction if there is a strong possibility that the transaction will be aborted again after the initial abort.

The rest of the paper is organized as follows. In Section 2, we explain the dynamic issue mechanism in detail. Section 3 describes microarchitectural modifications to an existing TM system. In Section 4, we discuss overheads incurred by DTI and mechanisms are proposed to reduce their impact. Section 5 is on related work. Section 6 provides experimental results comparing DTI and some other alternatives with a base machine that has no mechanism to suppress the reissue of aborted transactions. Finally, Section 7 concludes the paper with the summary of our contributions and proposes future work.

2. DYNAMIC TRANSACTION ISSUE (DTI)

In DTI, a transaction is not restarted immediately once it is aborted if there is a reasonable suspicion that the transaction will conflict with another transaction in the future. Instead, the transaction is suppressed from restarting until the suspicion is gone. During the time of suppression, the processor core waits for a wakeup signal in a power-saving mode, thus saving power/energy. To predict a future conflict, two sets of information, conflict history and currently running transaction IDs (TxIDs), are maintained in each core.

2.1. Conflict History

We maintain a conflict history record in each processor core using a unique transaction TxID assigned to every new transaction. A “new” transaction means a transaction

not restarted after an abort. Aborted transactions are not assigned a new TxID but keep their old TxID until committing successfully. The conflict history is recorded in each core by pairing a TxID with every processor core ID (PID). For example, when an aborted transaction has a conflict with a transaction with TxID 123 from processor 3 ($PID = 3$), TxID 123 is recorded in a record entry indexed by PID 3. We only keep track of the most recent conflict from a processor core and so the total number of entries in the local record equals the number of processor cores in the system minus one (The local core does not have an entry for itself.) In the previous example, if a new conflict is detected with a transaction TxID = 456 from the same processor ($PID = 3$), “456” overwrites the old TxID (123) in the local conflict history. This is based on the assumption that a conflict with a transaction having a new TxID implies that the old transaction on the same remote core was successfully committed or descheduled from the core and is no longer active. When the current transaction on a processor core is committed successfully, the current conflict history record in the core is cleared as a new transaction is assumed to have no relation to past history.

To detect and record a conflict, a TxID is sent along with a memory store address or a signature as was done in Yen et al. [2007]. Store addresses can be sent over a conventional invalidation-based cache coherence protocol in Eager conflict detection HTM systems in which such addresses are sent out to other transactions during the execution phase; data are not sent at that time but will be exposed later, after the commit phase. In the case of Lazy conflict detection, we need to send out store addresses during the execution phase because in current protocols they are sent out only after the validation phase by successfully committing transactions. Otherwise, no conflict information can be collected from aborted transactions since aborted transactions would not propagate their stores. Moreover, new conflicts caused by an already committed transaction are not useful to other transactions. For example, in Figure 1, conflict information from Tx1 received by Tx2, ..., TxN during Tx1's commit phase is stale and possibly detrimental since Tx1 is already committed. Furthermore, transactions (Tx2, ..., TxN) need to collect conflict information from each other, otherwise the aborted transactions (Tx2, ..., TxN) will be restarted blindly and aborted repeatedly, which does not resolve the issue. If conflict information is communicated at the time of commit only, then the conflict information from aborted transactions would never be gathered because only committed, winning transactions would have a chance to send store addresses for conflict histories in the Lazy detection scheme. Losing a competition does not have to be useless. We want to use the conflict information from aborted transactions as well. We will evaluate the overheads caused by these extra messages in Section 4.2.

2.2. Conflict Prediction

In addition to a conflict history record in each core, a record of TxIDs currently running on all cores is maintained locally within each processor core for conflict prediction. When a new transaction starts its execution, its TxID, generated dynamically, is broadcast to all other processor cores. The current TxIDs of all other cores are stored in a local record called the “currently running transactions” record; like the history conflict record, record entries are indexed by a PID. When a new transaction is started in a core its TxID is broadcast to all other cores to notify them. In response, remote cores overwrite the TxID indexed by the PID in their currently running transactions record. In Section 4.2.2, we will discuss the overheads of TxID broadcasts and introduce a technique that can avoid the broadcasts.

There are no ordering requirements between a TxID broadcast and the execution of the current transaction on a core because TxIDs are only used to predict whether an aborted transaction will have a conflict again if it were restarted immediately. For example, a transaction, which has broadcast its TxID earlier, can still issue loads and stores regardless of the pending TxID broadcast. Also, there is no ordering required

between TxID broadcasts, allowing a transaction to commit even if multiple TxID broadcasts are outstanding.

When a transaction is about to restart after its latest abort, TxIDs from the conflict history record and the currently running transaction record are indexed by the same PID and are compared. If the comparison results in a match, then a future conflict is predicted and the processor core enters an idle state or any energy saving mode. $(N - 1)$ such comparisons are done if there are N cores in the system.

Deadlocks may occur. For example, two cores could enter the idle state and their records show conflict with each other. Thus they will stay in the idle state forever because neither will commit and send a new TxID to the other. To avoid deadlocks and guarantee forward progress, a higher priority is given to older TxID. In other words, to prevent transactions from waiting for each other in idle state, if the TxID associated with a core is older than the TxIDs causing future conflicts, then the core does not enter an idle state but instead restarts its transaction.

2.3. Wakeup from Idle State

Instead of probing, which requires continual inquiries, we advocate a signaling scheme triggered by an event to wake up a processor core from the idle state. More specifically, a change in the currently running TxID record triggers a new prediction of future conflicts. When a new TxID arrives at a core in idle state, it updates the local currently running transactions record and triggers a local comparison between TxIDs in the conflict history record and the record of currently running TxIDs. If no future conflict is predicted, then the core wakes up and resumes execution by restarting the pending aborted transaction. Otherwise, the core stays in the idle state.

When a transaction is followed by nontransactional code, it is necessary to invalidate the TxID associated with the transaction to avoid indefinite waiting. Let us assume that a processor core is in idle state due to a future conflict with TxID “x” running on another core with PID “y.” The waiting core will keep waiting until “y” commits or deschedules “x”, issues a new transaction “z”, and sends this new TxID “z” to the waiting core to invoke the wakeup mechanism. If there is no such “z”, hence no delivery of the new TxID (note that “x” might be the last transaction scheduled on “y”), then it looks to the waiting core as if “x” is running forever and the waiting core will never wake up. So, to prevent this undesirable situation, processor core “y” needs to send a message invalidating “x”, which updates other processor cores’ records whenever a core exits a transaction and starts the execution of nontransactional code.

2.4. Transaction Flow

Figure 3 shows the flow of transaction executions with DTI. When a transaction is new, that is, not restarted after an abort, it follows the normal flow of execution, which results in a Tx abort or commit. If a transaction was aborted during its previous execution, then it next enters the execution phase or idle state based on DTI conflict prediction. When a TxID update is received and no future conflicts are predicted, the core is awakened from the idle state and restarts the execution of the current aborted transaction. TxIDs are broadcast whenever a new transaction starts execution or nontransactional code follows a transaction.

2.5. Application Example

Figure 4 illustrates how DTI saves the energy wasted in the transaction aborts of Figure 1. When Tx1 is committed, the other transactions, Tx2 ~ TxN, are aborted as in Figure 1. However, at this time, the aborted transactions have the information about past conflicts as well as the currently running Tx’s in other cores, which are used to predict future conflicts. Consequently, only Tx2, which has the highest priority

Fig. 3. Transaction flow.

Fig. 4. Energy savings with DTI in the example of Figure 1.

among all remaining transactions, is allowed to restart execution. Meanwhile, the cores running $Tx_3, \dots, Tx_N$, enter an idle state. The times spent in idle states are shown in the darker areas in Figure 4. When Tx_2 is committed a new TxID or an invalidation from Tx_2 is broadcast from the core that committed Tx_2 and this event triggers the waiting cores ($Pr_3 \sim Pr_N$) currently in idle state to reevaluate and repredict the possibility of future conflicts. Tx_3 has the highest priority at this moment, and so it resumes execution. The same procedure is repeated until Tx_N , the transaction with the lowest priority, is committed successfully. The total energy wasted on aborts is now reduced to $(Energy\ per\ abort)(N - 1)$, which is $\sim O(N)$ and $\frac{2}{N}$ of the energy consumption in Eq. (1).

Fig. 5. Logical modules in a processor node with DTI.

Regarding the scenario of Figure 2, Tx1 knows the fact that Tx2 has aborted Tx1 and Tx2 is still running at the moment right before its first restart, and Tx1 enters an idle state instead of restarting right away, thus saving energy.

2.6. Prediction Accuracy

As in most hardware prediction schemes based on dynamic information, understanding the impact of the accuracy of conflict predictions in DTI is important to identify possible drawbacks as well as future improvements. The two major DTI mispredictions are false-negative and false-positive alarms.

2.6.1. False-Negative Alarms. False-negative alarms occur when DTI falsely predicts that there is no conflict with a concurrent transaction. For example, in Figure 4, TxN needs to know that it had a conflict with Tx2 in order to avoid restarting. However, somehow, if the store address causing the conflict was delayed or missing in the transfer from the Tx2 core to the TxN core, or if Tx2 never sent the conflicting address because Tx2 was aborted by Tx1 before reaching the instruction generating the conflicting address, then there will be a false-negative alarm. In the latter case, we might let Tx2 run until the end of the transaction and send store addresses even after its abort to collect the addresses of upcoming stores. However, the execution time overheads offset the gains of higher prediction accuracy. False negatives do not affect correctness but they affect the performance of the system by enabling some transaction aborts that could have been avoided.

2.6.2. False-Positive Alarms. A false-positive alarm happens when a transaction is predicted to have a conflict with other transactions but the conflict does not happen. The main result is a delayed update of the currently running TxID records, which in turn delays the restart of aborted transactions and perhaps increases the overall execution time. Different from the false-negative case, a false positive could affect the correctness of a program if the update of the currently running TxIDs never occurs such as in the case of a transaction followed by nontransactional code. The solution to this problem was explained in Section 2.3.

3. MICROARCHITECTURE

3.1. Overall Architecture

Figure 5 shows the microarchitecture of a processor node equipped with DTI partitioned into logical modules. The cores execute machine instructions from their private caches.

The TM module is part of existing HTMs and provides TM services for the processor core such as conflict detection and conflict resolution. The Conflict Prediction Module (CPM), which can be easily integrated into an existing HTM system, is responsible for maintaining data structures for conflict history, for currently running TxIDs and logic for conflict prediction, as well as other necessary functions such as TxID generation. CPM is the only addition to a traditional HTM to support DTI. It works closely with the processor core and the TM module by exchanging control signals such as the wakeup signal from CPM to the core when a conflict is removed. The processor node is connected to other nodes via an interconnection network.

3.2. Conflict Prediction Module

As shown in Figure 5 the Conflict Prediction Module has five basic functions: maintaining a Conflict History vector, maintaining a Running History vector, predicting conflicts, generating TxIDs, and receiving TxIDs from other cores.

3.2.1. Conflict History Vector. Conflict history is stored in a vector with entries indexed by a processor core ID (PID) uniquely associated with a processor core. The number of entries is the same as the number of cores (minus 1) in the multicore system. Each entry keeps a TxID with which the current transaction has most recently conflicted with until the TxID is replaced by a new conflicting TxID. Flushing logic is added to clear all the entries at a transaction commit.

3.2.2. Running TxID Vector. The hardware structure of this vector is the same as for the conflict history vector. However, the content differs. In this case, a TxID stored in a vector entry is the TxID of the currently running transaction on the processor core with the corresponding PID.

3.2.3. Prediction Logic. Two comparisons are needed to predict future conflicts. First, the entries with the same index (the same PID) from the conflict history vector and the running TxID vector are compared in parallel. If one TxID stored in an entry in the conflict history vector indexed by PID i matches the TxID stored in the running TxID vector indexed by the same PID i , then a future conflict is predicted since the same transaction with a past conflict is still running. $(N - 1)$ such comparisons are conducted, where N is the number of cores. Second, the core's TxID is compared with the TxIDs causing future conflicts. The processor core enters an idle state unless its TxID has the highest priority. The comparison logic is invoked whenever an aborted transaction is restarted or an update to the running TxID vector is received.

3.2.4. TxID Generation. A TxID is needed for two purposes: to identify conflicting transactions and to make priority decisions. Ideally, a global timestamp can serve directly as a TxID generator. Alternatively, a core can obtain a global TxID value by updating a global variable accessible by all cores with a special instruction such as a "read and increment" or "fetch and add" instruction which reads the shared variable and increments it by one atomically. Also, a processor core may generate a TxID locally by combining its core ID with a local timer value to assign a unique identification; in this case priority is determined by the core ID.

3.2.5 Receiving TxIDs. As TxIDs are generated in ascending order (at least those generated in the same processor core) it is possible to commit the current transaction while multiple TxIDs are still outstanding in the same core. When TxIDs for a specific core with the same PID are received out of order, a smaller TxID is overwritten by a larger TxID and a smaller TxID is simply ignored. Extra care is needed for transactional code followed by nontransactional code as discussed in Section 2.3. In this case, we need

Fig. 6. Store-Load skew.

to buffer a TxID invalidation message if the message is received before the TxID is invalidated.

For sufficiently long running programs, it is imaginable to have a TxID roll over its maximum value so TxID counters or timestamps overflow. When this happens two transactions could have the same ID at the same time, thus confusing the conflict resolution logic as well as the priority logic. A counter or timestamp with 32 bits takes 2^{32} values. Thus in 16 cores each running one transaction, this should not be a problem. There are simple workarounds (as for all techniques relying on hardware counters) if this is a real problem.

4. OVERHEADS

DTI has some overheads: performance, intercore message traffic, area, and power/energy.

4.1. Performance Overheads

Performance overheads of DTI are increased execution times mainly due to the actions of the Conflict Prediction Module in Figure 5. The false-negative and false-positive predictions are a source of performance overheads as described in Section 2.6. However, these factors are related to the timely delivery and handling of TxID information and depend on implementation details such as the underlying interconnection network but are not inherent to DTI.

Figure 6 illustrates a scenario causing performance overheads due to the inherent shortcomings of DTI. In Figure 6, two transactions, Tx1 and Tx2, just had a conflict for memory address X, and Tx2, which was aborted by Tx1, is now about to make a decision whether to restart or idle at time T1. Tx1 has higher priority than Tx2. With DTI, Tx2, which keeps Tx1's ID in its conflict history vector as well as in the currently running transactions vector, enters an idle state at time T1 and later wakes up and restarts execution at time T2 after it is notified of Tx1's commit. However, it was actually safe to restart Tx2 at T1 because the load to X occurs after Tx1's commit, and hence there was no need for Tx2 to idle between T1 and T2. So time $(T2 - T1)$ is a performance overhead, an unnecessary delay.

Several observations are warranted from this example. The overhead is not caused by a false-positive alarm since there is an actual conflict between the two transactions. Instead it is caused by the timing of the conflict and the lack of information about the actual timing. This timing information is hard to collect. From the fact that Tx2 keeps the conflict information of "X" with Tx1, the Load of X in Tx2 must have occurred before the end of Tx1 in the previous run otherwise the conflict on X could not have been detected and recorded. In the repeat run, the Load of X was somehow delayed enough in Tx2 to safely load X without aborting Tx2. However, it is hard to predict the exact timing of accesses to X as any event might happen between the Tx2 begin and the Load of X in consecutive runs of the same transaction. To avoid this overhead, we would have to track conflicts on each memory access. For example, we could allow Tx2 to begin but make a decision of whether to continue at every memory load or store. We

leave this avenue of research to future work. It will require further modifications and simulations and may be impractical.

Another comment regarding this performance overhead is that, at the moment when the above event occurs, it is not clear how the delay will affect the overall performance in terms of execution time because it may be possible that the delay improves overall performance by avoiding other Tx aborts even though, in general, serializing transactions does not beat parallelizing transactions first and then serializing them only when a conflict is detected in terms of execution time. The occurrences of such delays as well as their effects are more obvious in hindsight than in foresight.

4.2. Message Overheads

To maintain the information needed for conflict prediction, the following message overheads are incurred, depending on the conflict detection policy of the underlying TM system.

4.2.1. Propagating Memory Addresses for Stores. If we implement DTI on a TM system relying on eager conflict detection, no additional messages are necessary as the conflict detection mechanism requires to send store addresses using a conventional cache coherence protocol during the execution phase regardless of DTI; we just need to piggyback TxIDs to the coherence messages already needed, just increasing a packet size by a few bytes.

In a TM system with lazy conflict detection, additional messages not in the base TM system are needed. These messages are due to write hits in the local cache during the execution phase since these write hits, which are purely local and circumscribed within the local node boundary in the base TM, now have to propagate outside local boundaries. To reduce the number of additional messages, we introduce the following technique with modest hardware modification. We propose that a processor node sends store addresses attached with a TxID over the network in bulk at the end of a transaction execution, regardless of whether the transaction commits or aborts, instead of sending out a store address for each individual write hit during execution. Cores send out store addresses in a message at transaction abort in addition to at transaction commit, with an additional bit to differentiate between these two types of messages: A message originating from an aborted transaction does not trigger a transaction abort but only updates the conflict history vectors in remote cores. This technique reduces the store traffic in DTI drastically, as simulation results will show in the evaluation section.

4.2.2. Propagating TxID at Tx Begin and Tx End. To keep the information of transactions running locally in every processor node current, each node broadcasts a TxID at the beginning of a new transaction and also at the commit of a transaction if the transaction is followed by nontransactional code. We denote these events as TxID_B and TxID_C. Our goal is not to keep such information in all cases but to predict future aborts using that information. A TxID_B is not sent to a processor node running an independent transaction, a transaction without conflicts with others. Also, a TxID_B is not sent out until the first store address is sent out for conflict detection. Thus a separate message for TxID_B other than a message to propagate a store address to update the conflict history vectors is not needed. No additional message or broadcast under both Lazy and Eager conflict detection is needed: TxIDs are sent only to the nodes currently sharing memory blocks. By the same token, no additional message for TxID_C is needed with Lazy conflict detection as a transaction has to send out store addresses for conflict detection at the commit time anyway. However, additional messages are needed for TxID_C in Eager conflict detection in which a transaction commits silently.

Fig. 7. Conflict prediction module.

Table II. Hardware Components

Component	Circuitry	Activity Frequency ²
TxID	4 byte register	Updated on every new transaction
Conflict History Vector	$N \times 4$ byte registers	Updated on a new detected conflict
Running TxID Vector	$N \times 4$ Byte registers	Updated on every TxID_begin and TxID_commit
4 Byte Comparator	32-bit XOR	When an aborted transaction restarts
Counter ¹	$\log_2 N$ bit counter register	For each comparison of vector elements

¹If one 32-bit comparator is used to repeat N comparisons to compare the two vectors.²How often an activity is incurred, later represented as activity factor “a.”

TxIDs must be sent to processor nodes which have conflicted with the committing transaction.

4.3. Hardware Overheads

Hardware overheads comprise additional design costs and the area for the additional components of the conflict prediction module shown in Figure 7. The design cost is marginal because the operation of the prediction module is very simple: It compares two vector elements by bitwise exclusive OR. As for the area overhead, we show the necessary circuitry in Table II, assuming 32-bit TxIDs and N processor cores. For example, if there are 16 cores in the system, we need two vector storage components, each of which has the size of 60 bytes ($15 \text{ cores} \times 4 \text{ bytes}$) and 15 comparators or 1 comparator with a counter register.

4.4. Power Overheads

Power overhead derives from sending and receiving the additional messages as described in the message overheads section and the activity of the circuitry in Table II. We will estimate these overheads by the number of additional messages and the activity frequency factors in the evaluation section.

5. RELATED WORK

Several TM proposals aim to avoid redundant transaction aborts by throttling the issue of transactions, from a simple back-off mechanism, such as the random back-off of aborted transactions in Bobba et al. [2007] to an adaptive scheduling technique based on transaction abort and commit rates [Yoo et al. 2008] to more complex proactive transaction scheduling by profiling the pattern of conflicts between transactions [Blake et al. 2009] to a reordering scheme based on work stealing at the thread level [Ansari et al. 2009]. These proposals mainly focus on performance improvement by scheduling transactions to avoid transaction aborts. By contrast, our paper proposes a simple hardware scheme to save energy in HTM systems, targeting consecutive aborts of the

same transaction. We believe that DTI is the first effort addressing energy savings in such systems using a hardware dynamic transaction issuing algorithm.

Among all the TM scheduling proposals, the proactive scheduling paper [Blake et al. 2009] is more closely related to this paper in that proactive scheduling collects and uses the information of past transaction conflicts and currently running transactions. Proactive scheduling differs from DTI in the following aspects. Proactive scheduling is heavily reliant on software mechanisms incurring software overheads such as scanning a software graph structure at every transaction begin, while DTI relies singly on simple hardware mechanisms without software overheads. Moreover, proactive scheduling targets HTMs with Eager conflict detection, while DTI is applicable to systems with either Eager or Lazy conflict detection. Finally, proactive scheduling improves execution time by switching threads, which causes thread switching overheads and relies on abundant threads while DTI targets the reduction of energy/power consumption by putting processor cores into an energy saving mode or idle state without any other major activities. Consequently, our mechanism is easy to port to existing HTM systems, regardless of the characteristics of the underlying system, with minimal modifications of hardware and no modification of the software stack and also without incurring much execution overheads.

HARP [Armejach et al. 2013] is also directly related to our work as it relies on hardware-only mechanisms to predict future conflicts using past conflict information. DTI specifically focuses on issues related to consecutive aborts of the same transaction, which can be dealt with minimal system information and modest hardware cost, targeting only previously aborted transactions and tracking only the most recent conflicts. By contrast, HARP keeps track of large amounts of stored information in each core regarding transaction executions in the whole system in order to schedule transactions. DTI can manage the necessary information in a small, simple, distributed hardware structure composed of two vectors, each of them requiring fewer than 100 bytes (based on the assumption that TxID size is 32 bits (4 bytes) and there are 16 processor cores) in each core. In the case of HARP, a bigger hardware structure, estimated to be 2.06KB per core according to the proposal, is needed to maintain more detailed information such as transaction size, contention ratio, number of consecutively predicted conflicts, and more for multiple transactions as opposed to the last conflicting transaction per core in DTI. By targeting aborted transactions only, DTI does not interfere with the execution of committing transactions, which incur no overheads in DTI but does in HARP. The main goal of DTI is to save power and energy consumption in a multicore system that runs one thread per core, though it might run on a system with multiple threads per core with the thread-switching decision delegated to software. Consequently, our scheme fits well in the context of relatively small systems that cannot afford a big power/energy budget such as portable machines. Different from DTI, HARP focuses mainly on enhancing system performance, especially for systems running multiple applications, each of which is multithreaded.

Several papers address the topic of power and energy savings in TM.

Sanyal et al. propose “Clock Gate on Abort (CGA)” [Sanyal et al. 2009] to improve the energy efficiency of HTM by clock gating or by turning off processor cores for aborted transactions. The mechanisms in CGA are developed for a specific HTM system, Scalable TCC [Chafi et al. 2007], which is known to be a suboptimal design due to its limited parallelism and scalability. By contrast, DTI is readily applicable to any HTM system and is not limited by the underlying TM system. Moreover, DTI relies on up-to-date dynamic information to decide whether to put a processor core in an idle state instead of simply putting the cores into an idle state immediately after a transaction is aborted like is done in CGA. DTI is more responsive to changes in current conditions for cores in an idle state by snooping any changes that might wake up the core. In Sanyal

et al.'s approach a core in idle state waits for a timer preset at a fixed amount of time¹ to expire. This approach is very similar to the blind simple back-off schemes mentioned earlier. Finally, DTI is based on a distributed control mechanism while Sanyal et al.'s approach relies on a centralized control mechanism delegated to directory modules, which avoids broadcast messages² but incurs delays due to congestion.

Ferri et al. [2010] proposes TM architectures well suited to embedded multicore systems, with an emphasis on energy, performance, and complexity. The authors discuss various techniques for energy efficiency such as shutting down the Transaction Cache when not in use. The restart of an aborted transaction is triggered after a simple random exponential back-off period during which the CPU stays in a low-power mode. Their proposed embedded architectures emphasizing energy efficiency are good targets for DTI.

Gaona et al. [2014] propose Selective Dynamic Serialization (SDS), an energy reduction scheme targeting an Eager-Eager³ HTM system, specifically LogTM-SE [Yen et al. 2007]. The scheme serializes transactions (instead of retrying immediately) when a counter incremented at the detection of a conflict (NACK_SDS) or of an abort (ABORT_SDS) saturates to a preset value. Energy is saved by putting cores in a low-power mode during the time it waits for its turn, using a hardware record based on transaction priority for each conflicting address. The proposal adopts a token-based scheme, in which the highest priority transaction wakes up the next highest priority transaction waiting for permission to resume in line, as compared to DTI where a processor core decides locally whether to restart a transaction based on the information of past conflicts and current transactions. SDS stalls transactions during execution when a conflict is detected (address-base) while DTI restricts the restart of aborted transactions at their beginning (transaction-base) and, consequently, is applicable to both Lazy and Eager conflict detection TMs.

Hourglass [Liu et al. 2011], like DTI, focusses on repeated aborts with a simple contention management policy evaluated in a Software Transactional Memory (STM) system. When transactions are aborted consecutively over a certain threshold number, they are marked as "toxic transactions." A toxic transaction acquires a token that prevents any other transaction from starting execution except those that have already started execution. Once the toxic transaction is committed successfully, the token is released to let other transactions proceed. There are several differences between Hourglass and DTI. First, Hourglass does not use conflict history for conflict prediction. Second, all concurrent transactions are serialized as compared to only transactions with conflict history in DTI. Third, a central arbiter is necessary in the case of multiple pending toxic transactions.

Table III compares the choices made in DTI and in several other proposals.

6. EVALUATION

6.1. Experimental Setup

6.1.1. Simulation Setup. We implemented and simulated DTI on top of the SESC simulator [Renau et al. 2005], a cycle-accurate simulator for out-of-order cores in multicore configurations, augmented with the following packages.

First, we enable the TM package [Poe et al. 2008] on top of the base simulator using "–enable-transactional." The package includes a TM framework that incorporates the

¹Though the amount of time itself is determined dynamically based on the current contention level, once a core enters idle state, it waits for the amount of time determined at the time of entrance.

²Broadcast messages are also avoidable in our approach using the techniques in Section 4.2.2.

³Eager conflict detection and Eager version management.

Table III. Design Space in Hardware Transactional Memory Systems

Design Space	DTI	HARP	CGA	Ferri's	SDS
Target TM System	Any	Any	Lazy conflict detection (Scalable-TCC)	Eager conflict detection	Eager conflict detection (LogTM-SE)
Target Transaction	Aborted transactions	All transactions	Aborted transactions	Aborted transactions	Transactions with conflicts
Conflict Information Used/Kept	The most recent	Previously executed instances of transactions	The most recent	N/A	All currently NACKed conflicts for each block address
Scheduling Methodology	Local nodes decide dynamically whether to restart or not.	Local nodes decide dynamically whether to stall or yield thread.	Directory modules decide whether to restart with back-off timer.	Random back-off with exponential distribution; serialization; Lazy conflict resolution	Stalls conflicting transaction except a winner, which wakes up the highest priority one among the stalled.

common operations of TM into the core of the SESC simulator such as taking a checkpoint at a transaction begin and restoring the system context back to the checkpoint at a transaction abort. Within this framework, functions related to the execution of individual transactions such as `beginTransaction()`, `abortTransaction()`, and `commitTransaction()` are already implemented. We added DTI on top of this platform.

Second, we use the Power package [Brooks et al. 2000], which is enabled by “enable-power,” to estimate dynamic power and energy consumption during the execution of a program by keeping track of units accessed during each clock cycle. The dynamic power consumption is estimated by

$$P_d = CV_{dd}^2 \alpha f, \quad (2)$$

where P_d is the dynamic power consumption; C is the load capacitance calculated by the capacitance models as described in Brooks et al. [2000] for the various hardware structures defined by the hardware parameters set in the SESC configuration file, `sesc.conf`; V_{dd} is the supply voltage; α is an activity factor indicating the average switching activity in every clock tick, ranging from 0 to 1 and dependent on the execution behavior of each benchmark; and f is the clock frequency. For circuits that precharge and discharge on every cycle, “ α ” is set to 1.

Tables IV and V summarize the configurations for the hardware platform and the benchmark applications. To help understand the scalability of the benchmarks, Figure 8 shows their execution times with varying number of threads, from 1 to 16, normalized to those with one thread on the base system, a TM system with no back-off or suppression of transaction restarts. For Bayes (“B”), the execution times are normalized to the execution time with two threads due to a simulation error with one thread.

6.1.2. Hardware Setup. To evaluate our approach and compare it to other proposals, we implemented and compared the following hardware-only schemes.

Base Machine with No Back-off. Each core of the base machine has a TM module implementing Lazy conflict detection and Lazy version management using bus contention for transaction conflict detection as in TCC [Hammond et al. 2004]. A transaction is

Table IV. Simulation Configuration

Feature	Description
Processor	16 cores ¹ , out of order execution with issue width 4
L1 cache	Private, 32 KB size, 2-way associative,
Instruction	32 B block, LRU replacement, 1 cycle hit delay
L1 cache	Private, 32 KB size, 4-way associative,
Data	32 B block, LRU replacement, 2 cycle hit delay
L2 cache	Shared, 1 MB size, 8-way associative,
Combined	32 B block, LRU replacement, 3 cycle hit delay, 11 cycle miss delay, 64 MSRH entries
Memory	Perfect memory, 64 B block, 500 cycle round trip time
TM module	Lazy conflict detection
	Lazy version management
Interconnection	Bus

¹Eight cores were used for Bayes and Yada due to scalability and simulator errors with 16 cores.

Table V. Benchmark Applications

Program	Abbreviation	Contention	Input argument
Bayes	B	High	-v32 -r1024 -n2 -p20 -s0 -i2 -e2
Genome	G	Low	-g256 -s16 -n16384 -t16
Intruder	I	High	-a10 -l4 -n2048 -s1
Kmeans	K	Low	-m40 -n40 -t0.05 -p16 -i random-n2048-d16-c16.txt
Labyrinth	L	High	-i./random-x32-y32-z3-n96.txt
Vacation	V	Low/Medium	-n2 -q90 -u98 -r16384 -t409
Yada	Y	Medium	-a20 -i./633.2
Fmm*	F	28088 locks	two cluster, plumber, 16384, 1e-6, 16, 5, .025, 0.0, cost zones

*Only application from SPLASH-2 [Woo et al. 1995], the others are from the STAMP [Minh et al. 2008] benchmark suite.

Fig. 8. Scalability with the number of threads on the base machine.

restarted immediately after it is aborted. There is no scheduling algorithm or back-off scheme.

Linear Back-off. A transaction is backed-off by a number of cycles linearly dependent on the number of aborts experienced consecutively. For example, if the linear constant is L and the number of consecutive aborts is N , then the processor core stays in the

Fig. 9. Dynamic power consumption.

idle state for ($L \times N$) cycles before it restarts the aborted transaction. L is set to eight cycles in our simulations.

Exponential Back-off. This is the same as the linear back-off scheme except that the number of cycles in idle state increases exponentially as L^N , where L is the radix (which is set to two in our simulations) and N is the number of consecutive aborts. In this scheme a large number of consecutive aborts are dealt with more aggressively but a small number of consecutive aborts are dealt with less aggressively than in the linear back-off scheme.

Random Back-off. Originally proposed in Bobba et al. [2007] to remedy one of the TM pathologies identified in the paper, this scheme handles the situation in Figure 1 by idling processor cores for a random number of cycles. In our simulations a random number is generated between 1 and 10 using the C++ `rand()` function and then is multiplied by the number of consecutive aborts to calculate the number of idle cycles.

Dynamic Transaction Issue (DTI). In this scheme, a processor core enters and wakes up from idle state when certain conditions are met using the Conflict History vector and the Running TxID vector as described earlier in this paper. We implement DTI on top of the base machine.

6.2. Results and Analysis

6.2.1. Power and Energy. In this section, we show and analyze simulations results for dynamic power and energy consumption of the benchmarks on different machines.

Figure 9 compares the dynamic power consumptions of the schemes described in Section 6.1.2, abbreviated as “No” for the No back-off scheme (Base), “Lin” for the Linear back-off scheme, “Exp” for the Exponential back-off scheme, “Ran” for the Random back-off scheme, and “DTI” for our DTI scheme. These abbreviations are also used later in the paper. Power consumption is normalized to the power consumption of the No back-off scheme, the base machine. As the figure shows, DTI outperforms all the back-off schemes for all benchmark programs in terms of dynamic power consumption (by up to 38.2% in Intruder) except for Fmm where the Linear back-off scheme consumes about 0.05% less power than DTI.

In the back-off schemes including DTI, the idle state is implemented by stopping instructions in the fetch stage. A processor core enters the idle state by stopping fetching instructions. At that time, all prior in-flight instructions proceed to the retirement stage. Once the core exits the idle stage, the execution resumes by restarting the fetching of instructions. With this method, we can accurately model an idle state which does not incur dynamic power overhead to resume execution. An alternative to this

Fig. 10. Overall dynamic energy consumption (transactional and non-transactional codes).

policy is to enter the core into a sleep mode, which might save more power but would affect performance whenever the core is wakened up.

We now evaluate DTI in terms of dynamic energy consumption, which is calculated by multiplying the dynamic power consumption by the execution time in cycles (power $\times$ delay model). Figure 10 shows the total dynamic energy consumptions of the various schemes during the entire execution of each benchmark, normalized to the energy consumptions of the No back-off policy (Base). DTI achieves energy savings for some programs, most notably for Intruder by about 37%, and also for Yada and Labyrinth by 12% and 8%, respectively. The Linear back-off, the Exponential back-off, and the Random back-off schemes show similar energy consumption except for Intruder and Yada. In Intruder, the Linear scheme, the Exponential scheme, and the Random scheme improve energy consumption by 4%, 20%, and 5%, respectively, as compared to the No back-off scheme. In Yada, energy consumption is slightly higher with the Exponential and Random Back-off schemes. This is possible if a back-off scheme increases execution time, for example, by increasing the number of aborts.

Energy savings are very similar to power savings in most cases, as demonstrated by comparing Figures 9 and 10. Variations are less than 1%. One of the reasons for these small differences is that DTI (and other schemes) does not predict future conflict with perfect accuracy, and some mispredictions may slightly increase execution times and thus increase energy consumption as compared to power consumption. For example, in Intruder, while the power consumption of DTI decreases by 38% relative to the No back-off scheme (Base), energy consumption decreases by 37%. This slight difference can be attributed to the increased execution time of DTI relative to the execution time of the No back-off scheme. The increase in execution time is about 1%. In summary, the relative energy consumptions are similar to the relative power consumptions. This implies that DTI does not sacrifice performance in order to save power.

Figure 11 shows the dynamic energy wasted in aborted transactions only, normalized to the case of No back-off. For all applications DTI reduces the wasted energy due to aborts more than the other schemes because it adapts to future transaction execution using past information rather than relying on a static back-off mechanism. Figure 11 displays the differences between the various schemes better than Figure 10 because Figure 10 shows the energy consumed during the entire execution of each benchmark program while Figure 11 concentrates on the energy spent on aborted transactions only. If transactions are a small portion of an entire execution, then we expect little improvement from any of the back-off schemes, including DTI. For example, only 0.5% and 1.3% of the entire execution times of Bayes and Vacation are transactions in the No back-off scheme, so the bars for Bayes and Vacation in Figure 10 are flat. Moreover, the fraction of aborted cycles to total cycles in transaction executions (aborted cycles

Fig. 11. Dynamic energy wasted on aborted transactions.

Fig. 12. Dynamic energy spent on all transaction executions –aborted or committed.

Fig. 13. Average consecutive aborts.

plus committed cycles) is another factor that affects overall energy consumption. For example, in Fmm, only 0.1% of total transaction execution cycles is spent in aborted transactions. In Figure 12, which is closer to Figure 10 than Figure 11, Fmm shows no improvement with any back-off scheme. This does not mean that DTI or the other back-off schemes are ineffective but that there is no need for them, since the underlying basic TM system with no back-off already runs smoothly without losing machine cycles on wasted work or aborted transactions.

To confirm that DTI actually solves the problem illustrated in Figure 1, Figure 13 shows the average number of consecutive aborts of the same transaction with each scheme, normalized to that of the No back-off scheme. The figure shows how many times

Fig. 14. Execution times.

an aborted transaction repeats aborts consecutively on average before it is committed successfully. For example, in Bayes, with the No back-off scheme, aborted transactions experience repeated aborts 2.91 times consecutively before committing. In the case of Vacation, every aborted transaction commits successfully right after the initial abort in every scheme, so the bars of Vacation are equal in the figure. DTI reduces consecutive aborts more effectively than any other Back-off scheme. This is also reflected in the energy consumptions in Figure 10. The Exponential scheme somewhat works but not as well as DTI.

6.2.2. Performance. We now evaluate the performance overheads of back-off systems by comparing the execution times of benchmark programs with different policies.

Figure 14 shows the execution times of the benchmark programs, normalized to those of the No back-off scheme. We could not find significant variations among the schemes; the variations range from -1.4% (Intruder with Exponential Back-off) to $+2.5\%$ (Yada with Random Back-off). With DTI, the range is from -0.2% to $+1.5\%$, with an average of $+0.4\%$. These results confirm once more that DTI does not sacrifice performance to save power. Note that performance can be affected by the underlying TM system, in particular by the version management policy. Because our base machine adopts lazy version management in which transaction aborts are fast and local, performance gains with DTI are not guaranteed in all cases. For example, when two transactions with conflict history are about to restart execution after an abort, DTI serializes the two transactions, while the base machine with no back-off reexecutes the two transactions in parallel. Whether there is an actual conflict or not, the resulting execution time of DTI is the sum of the execution times of the two transactions, whereas, in the base machine, the execution time is the sum of the execution times of the two transactions only when there is an actual conflict. This can occur because the prediction of DTI is not perfect.

Figure 15 compares the number of committed cycles (cycles spent in committed transactions) spent by all the transactions in the No-Backoff scheme and in DTI. The figure shows that the number of committed cycles does not noticeably differ in the base machine (No-Backoff) and in DTI. Thus DTI does not interfere with the normal flow of execution of benchmark programs when there are no conflicts among concurrent transactions. DTI only targets aborted transactions and does not affect independent transactions.

Figure 16 shows transaction commit rates calculated by dividing the number of transactions that successfully commit by the number of transactions that reach the end of their execution (either aborted or committed) in all back-off schemes, normalized to the same rates in the No back-off scheme. In Labyrinth, DTI achieves a better commit

Fig. 15. Committed cycles.

Fig. 16. Transaction commit rates.

Fig. 17. Message overhead percentage.

rate than the No back-off scheme by about 30%. In the figure, DTI is shown to improve the commit rates in all the benchmark programs because of the conflict prediction mechanism. With DTI, transactions have a better chance of committing than with the other schemes once they reach the end of their execution. The commit rates of the No back-off scheme are 0.90, 0.95, 0.73, 0.91, 0.68, 1.00, 0.98, and 1.00 for the benchmark programs in the order of the horizontal axis of the figure.

6.2.3. Message Overheads. Figure 17 shows the message overheads during the execution of transactions with DTI. The figure shows the fraction of extra messages needed to update the data structures of the conflict prediction modules, as discussed in Section 4.2, and the fraction of L1 miss request messages plus the commit request messages. L1 misses by store instructions do not contribute to extra messages since they go

Table VI. Message Overheads with the Reduction Technique

Program	B	G	I	K	L	V	Y	F
Overhead	0.001%	0.027%	0.190%	0.066%	0.000%	0.000%	0.002%	0.009%

Fig. 18. Message overhead in data transferred.

through the network anyway in the No back-off policy. In some programs, the fraction of extra messages due purely to DTI is nonnegligible. Overall, extra messages contribute more than 20% to overall traffic. In particular, in Fmm, the fraction of extra messages is 45.2%, which is a huge increase in message traffic in the network and could affect overall performance and power.

The technique proposed in Section 4.2.1, in which store address packets are sent in bulk not only at transaction commits but also at transaction aborts, drastically reduces the number of packets. Addresses originated from aborted transactions do not abort the transaction at the receiving nodes but are only used to update the conflict history vector of the receiving nodes. The fractions of extra messages with bulk transfer of stores addresses on aborts are given in Table VI. In the table, the fractions of extra messages are now negligible, close to zero. This is because only one message, which possibly contains multiple addresses, is sent out at a transaction abort, as compared with sending individual store messages separately during the execution of the transaction.

To compare the message overheads in terms of number of bytes transferred, Figure 18 compares the number of extra bytes transferred (in the payload only) in DTI with and without the message overhead reduction technique, normalized to the case without the reduction technique. We assume 32-bit addresses. With the reduction technique, the total payload is measured by counting the number of distinct block addresses in messages sent at transaction aborts and multiplying the number by 4 (32-bit address). Without the reduction technique, the total payload is calculated by multiplying the number of messages sent for every store during the execution phase by 4. In the figure, we observe that the payload size in bytes is cut dramatically with the reduction technique.

A major factor contributing to the dramatic reduction of bytes transferred in Figure 18 is the temporal locality of memory stores: The same memory address may be written repeatedly during the execution of a transaction, and the more frequent such repetitions are, the better the reduction technique works. To quantify the temporal locality of stores, we measured the ratios between the number of stores and the distinct cache block addresses in a transaction on average, reported in Table VII. For example, in Bayes, only 0.5% of the total stores in a transaction targets distinct block addresses.

Finally, we compare the power consumption, the energy consumption and the performance of DTI with and without the message overhead reduction technique. This is

Table VII. Ratios of Write-Set Size to Stores Per Transaction

Program	B	G	I	K	L	V	Y	F
Overhead	0.005	0.130	0.094	0.135	0.003	0.020	0.010	0.278

Table VIII. Power, Energy, and Performance with Message Reduction

Program	Power		Energy		Performance	
	N. R. ¹	R. ²	N. R.	R.	N. R.	R.
B	1	1.00	1	1.00	1	1.00
G	1	1.00	1	1.00	1	1.00
I	1	1.16	1	1.15	1	0.99
K	1	1.00	1	1.00	1	1.00
L	1	1.00	1	1.00	1	1.00
V	1	1.00	1	1.00	1	1.00
Y	1	1.01	1	1.02	1	1.01
F	1	1.00	1	1.00	1	1.00
Avg.	1	1.02	1	1.02	1	1.00

¹No Reduction.²Reduction.

necessary because the reduction technique may change the relative timings between the executions of transactions to a point that it may affect the results (power, energy, and performance) noticeably. Table VIII shows the measurements for DTI with the reduction technique normalized to DTI without the reduction technique. In the table, no noticeable change is observed except for Intruder, in which the power and energy consumptions are increased by 16% and 15%, respectively. However, the power and energy consumptions of Intruder with the reduction technique are still better than the No Back-off scheme by 25%, with 1% increase in performance.

6.2.4. Power Overheads. As discussed in Section 4.4, there are two major sources of power overheads: the power for communicating extra messages and the power for activating the prediction logic. With the message reduction technique, the power overhead of extra messages is marginal as shown in Table VI. The power overheads of the prediction logic are also marginal due to the relatively small capacitance and the activity factor (“ α ” in Equation (2)) of the logic, which is limited by the number of transaction commits and aborts, ranging from 0.000 to 0.051 with an average of 0.015.

6.2.5. Effect of the Number of Threads. To help better understand how DTI performs for applications with varying number of threads, Table IX summarizes results from simulations with 2, 4, 8, and 16 threads and cores. Each number is normalized to the base machine with no back-off. Some entries are not available because we could not run the simulator for some benchmarks and some thread numbers. Overall, dynamic energy savings, especially on aborted cycles, increase as the number of threads increases due to the increased level of contention in the base machine. The execution times are mostly unaffected as DTI targets only aborted transactions with low performance overheads. The dynamic energy wasted on aborts in DTI increases at times for Bayes, Genome, and Fmm, presumably because the prediction accuracy of the scheme and the relatively small number of aborts enhance the effects of false negatives.

7. CONCLUSION

In this paper, we propose a DTI scheme that can be easily implemented on top of existing HTM systems. Instead of wasting dynamic power on transaction aborts, DTI puts a

Table IX. Simulation Summary with Varying Number of Threads

<i>N</i>	2				4				8				16			
<i>P</i>	ET	TE	ETx	EA	ET	TE	ETx	EA	ET	TE	ETx	EA	ET	TE	ETx	EA
<i>B</i>	1	1	1	1.00	1.00	1.00	1.12	1.61	1.00	1.00	0.94	0.85	N/A	N/A	N/A	N/A
<i>G</i>	1	1	1	1.00	1.00	1.00	1.00	1.26	1.00	0.99	0.99	0.90	1.01	1.00	0.99	0.83
<i>I</i>	1	1	1	1.00	1.00	0.99	0.97	0.91	1.01	0.84	0.68	0.51	1.02	0.63	0.42	0.29
<i>K</i>	1	1	1	0.79	1.00	1.00	1.00	0.74	1.00	1.00	0.99	0.68	1.00	1.00	1.00	0.78
<i>L</i>	1	1	1	1.00	N/A	N/A	N/A	N/A	1.00	1.00	1.00	1.00	1.01	0.92	0.91	0.91
<i>V</i>	1	1	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.92
<i>Y</i>	1	1	1	1.00	0.98	0.92	0.90	0.72	1.00	0.88	0.85	0.69	N/A	N/A	N/A	N/A
<i>F</i>	1	1	1	1.00	1.00	1.00	1.00	1.72	1.00	1.00	1.00	1.10	1.00	1.00	1.00	0.71

N: Number of threads; *P*: Programs; ET: Execution Time; TE: Total dynamic Energy; ETx: Dynamic Energy on Transactions; EA: Dynamic Energy on Aborts; N/A: Not Available.

processor core into idle state when there is a reasonable suspicion that the current transaction will be aborted soon in the future, thus saving the power consumption associated with aborts. The power/energy savings come without noticeable performance and communication overheads. This paper makes two major contributions: proposing a simple hardware scheme saving power/energy consumption on existing HTM systems and comparing the scheme with various alternative hardware mechanisms from a power/energy perspective.

Future work includes, but is not limited to, obtaining additional simulation results for various types of HTM platforms including commercial HTM implementations to evaluate and understand how DTI can save power/energy on these platforms. Comparisons with related proposals such as those described in Section 5 are needed, too. Finally, incorporating DTI into HTM systems for energy-sensitive architectures such as those in smartphones will be a direction for our future research work.

REFERENCES

- C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded transactional memory. In *HPCA 11*. 316–327.
- Mohammad Ansari, Mikel Lujan, Chritos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. 2009. Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory. In *HiPEAC*. 4–18.
- Adria Armejach, Anurag Negi, Adrian Cristal, Osman Unsal, Per Stenstrom, and Tim Harris. 2013. HARP: Adaptive abort recurrence prediction for hardware transactional memory. In *HiPC’13*. 196–205.
- Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. 2009. Proactive transaction scheduling for contention management. In *Micro 42*. 156–167.
- Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. Performance pathologies in hardware transactional memory. In *ISCA 34*. 81–91.
- David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA 27*. 83–94.
- Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Knule Olukotun. 2007. A scalable, non-blocking approach to transactional memory. In *HPCA 13*. 97–108.
- Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA 33*. 227–238.
- Cesare Ferri, Samantha Wood, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. 2010. Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. In *Journal of Parallel and Distributed Computing (JPDC)* 70, 10 (Oct. 2010), 1042–1052.
- Epifanio Gaona, J. Rubén Titos-Gil, Juan Fernández, and Manuel E. Acacio. 2014. Selective dynamic serialization for reducing energy consumption in hardware transactional memory systems. In *the Journal of Supercomputing* 68, 2 (May 2014), 914–934.

- Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Knule Olukotun. 2004. Transactional memory coherence and consistency. In *ISCA 31*. 102–113.
- Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*. 289–300.
- H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM TODS*, 6, 2 (Jun. 1981), 213–226.
- Yujie Liu and Michael Spear. 2011. Toxic transactions. In *TRANSACT'11*.
- Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. 2010. A dynamically adaptable hardware transactional memory. In *MICRO-43*. 27–38.
- Chi Cao Minh, Jae Woong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08*.
- Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based transactional memory. In *HPCA 12*. 254–265.
- James Poe, Chang-Burm Cho, and Tao Li. 2008. Using analytical model to efficiently explore hardware transactional memory and multi-core co-design. In *SBAC-PAD*.
- Xuehai Qian, Wonsun Ahn, and Josep Torrellas. 2010. ScalableBulk: Scalable cache coherence for atomic blocks in a lazy environment. In *MICRO-43*. 447–458.
- Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. Virtualizing transactional memory. In *ISCA 32*. 494–505.
- Jose Renau, Basilio Fraguera, and Wei Liu. 2005. SESC simulator. Retrieved from <http://sourceforge.net/projects/sesc/>.
- Sutirtha Sanyal, Sourav Roy, Adrian Cristal, Osman S. Unsal, and Mateo Valero. 2009. Clock gate on abort: Towards energy-efficient hardware transactional memory. In *IPDPS'09*. 1–8.
- Arrindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. 2008. Flexible decoupled transactional memory support. In *ISCA 35*. 139–150.
- Ruben Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. 2011. ZEBRA: A data-centric, hybrid-policy hardware transactional memory design. In *ICS'11*. 53–62.
- Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrian Cristal, Osman Unsal, Tim Harris, and Mateo Valero. 2009. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO-42*. 145–155.
- Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and methodological considerations. In *ISCA 22*. 24–36.
- Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA'13*. 261–272.
- Richard M. Yoo and Hsien-Hsin S. Lee. 2008. Adaptive transaction scheduling for transactional memory systems. In *SPAA'08*. 169–178.

Received May 2015; revised January 2016; accepted January 2016