Enhanced Concurrency Control with Transactional NACKs

Woongki Baek Richard M. Yoo Christos Kozyrakis

Pervasive Parallelism Laboratory
Stanford University
{wkbaek, rmyoo, kozyraki}@stanford.edu

Abstract

Transactional memory (TM) is a promising technique that simplifies parallel programming by supporting atomic and isolated execution of code sections. To provide robust performance and fairness guarantees, however, TM must dynamically adjust the degree of concurrency among transactions. To design an efficient concurrency controller for TM, critical system information, such as dependencies among transactions and current utilization level of the system, should be provided in an accurate and inexpensive manner. However, efficiently obtaining such information is often the most challenging design problem.

To address the issue, we have looked at hardware features that can be used to characterize transactional behavior. In particular, we identify NACKs, which are readily available in several TM designs to handle conflicts between transactions, as the mechanism to efficiently collect the transactional information of the system.

In this paper, we propose three novel, efficient use cases of NACKs for TM systems to demonstrate the potentials of utilizing NACKs intelligently: 1) accurate deadlock detection, 2) dependency tree construction, and 3) carrier sensing. We also describe a prototype design that extends the baseline TM systems to support the proposed techniques. To evaluate the effectiveness of such approach, we use the proposed techniques to implement an enhanced concurrency controller that performs aggressive stalling, dependency chain cutting, and exponential backoff with overshoot avoidance. Our preliminary results demonstrate that the proposed techniques can significantly improve the performance of hardware and hybrid TM systems (up to 21.5%).

1. Introduction

Transactional memory (TM) [15] surfaced as a promising technology to address the difficulties of parallel programming. TM allows programmers to declare certain code sections as *transactions* that execute in an *atomic* and *isolated* way with respect to the other code sections. Controlling the concurrency among those transactions then becomes the responsibility of the system. Previous researches proposed TM implementations using hardware [1, 4, 13, 20], software [11, 14, 22], and hybrid [7, 9, 19, 23] techniques, and major vendors plan to implement TM in mass-market products [10, 16].

For a TM system to provide robust performance and fairness guarantees, the system must implement an efficient *concurrency control*. At a high level, concurrency controllers dynamically adjust the degree of concurrency within the system to increase the number of successful transaction commits. Specifically, recent work has studied on contention management techniques [12, 24] and adaptive scheduling [2, 3, 29]; they may be implemented in hardware [3, 29] or software [2, 12, 24].

To design an efficient concurrency controller, however, the controller should be provided with low-level run-time information, such as dependencies among transactions and utilization level of the system. However, obtaining such information in an accurate and inexpensive manner is often the most challenging factor in designing a TM system [5]. In particular, to facilitate low-cost TM implementations [10, 16], techniques that leverage *existing hardware features* to characterize transactional behavior would be necessary.

We notice that several TM designs utilize the underlying NACK coherence mechanism to handle conflicts between transactions [7, 20]; when a transaction (i.e., defender) receives a conflicting coherence message from another transaction (i.e., attacker), it sends a NACK to ensure serializability. The attacker then would stall until the conflict is resolved. Therefore, in essence, NACKs can be viewed as an accurate indicator to denote the dependencies between transactions and the utilization level of the system.

Although there have been researches on utilizing NACKs to support nonbusy waiting [30] and to implement conservative deadlock avoidance [20], little work has been done on the efficient use of NACKs to obtain transactional dependency information. In this paper, we suggest that NACKs can be readily used to construct such dependency information, and that the information can then be used to implement an efficient TM concurrency control.

To demonstrate the effectiveness of the intelligent uses of NACKs, we propose three novel, efficient use cases of NACKs for TM systems: 1) accurate deadlock detection, 2) dependency tree construction, and 3) carrier sensing. Due to the imprecise nature, conservative deadlock avoidance schemes on TM [1, 20] have to admit false positives. Accurate deadlock detection, on the contrary, precisely detects deadlock only when it actually occurs. This allows to implement a more aggressive stalling scheme. On the other hand, constructing a dependency tree reveals the degree of dependency for each transaction (i.e., how many transactions are stalling due to the transaction, either directly or indirectly). If the dependency on a transaction is higher than a threshold, the transaction might as well be aborted to let other transactions make progress, which would result in increased throughput. Lastly, carrier sensing enables the TM concurrency controller to estimate the current utilization level of the system and to dynamically adapt the execution.

The specific contributions of this work are:

- We propose three efficient use cases of NACKs for TM systems that include accurate deadlock detection, dependency tree construction, and carrier sensing. We also describe the design of

[Copyright notice will appear here once 'preprint' option is removed.]

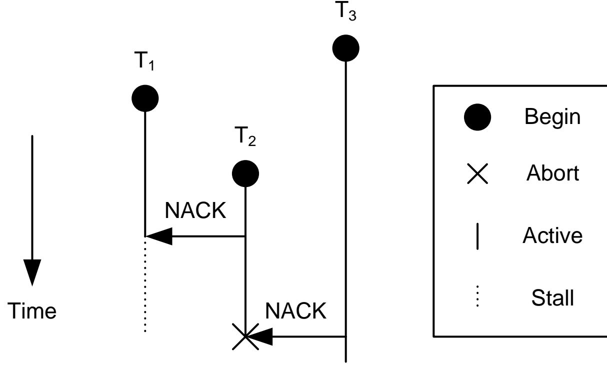

Figure 1. An Unnecessary Abort of a Transaction due to Conservative Deadlock Avoidance.

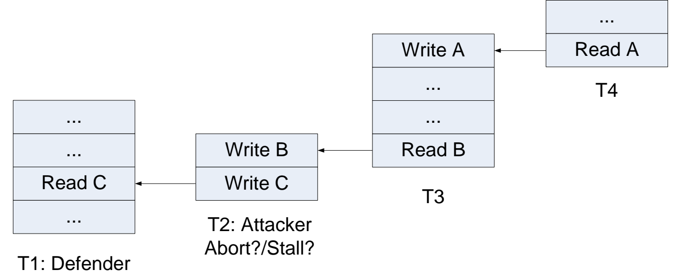

Figure 2. Cascaded Stalls of Transactions.

hardware and software components to enhance TM systems to enable these use cases.

- Using the proposed techniques, we implement an enhanced concurrency controller that performs aggressive stalling, dependency chain cutting, and exponential backoff with overshoot avoidance.
- We provide our preliminary results to demonstrate the performance potentials of the NACK use cases. We show that the proposed techniques can improve the performance of both hardware and hybrid TM systems.

The rest of the paper is organized as follows. Section 2 discusses the three efficient use cases of NACKs. Section 3 describes the design of hardware and software components, and Section 4 provides motivating results. Section 5 reviews related work. Finally, Section 6 concludes and discusses future research direction.

2. Efficient Use Cases of NACKs for TM

In this section we propose the three use cases of NACKs: accurate deadlock detection (Section 2.1), dependency tree construction (Section 2.2), and carrier sensing (Section 2.3).

2.1 Accurate Deadlock Detection

For hardware TM systems with eager version management and pessimistic conflict detection (eager HTM systems) [20, 28], performance is more susceptible to conflict resolution policy [5]. A commonly used policy for this type of implementations is to stall the attacker while performing conservative deadlock avoidance [4, 20]. In this policy, if a transaction sends a NACK to a logically older transaction, it sets its `possible_cycle` flag. If the transaction receives a NACK from another logically older transaction, it aborts because it could induce deadlock.

Due to the conservative nature of this policy, in some cases, transactions that do not actually cause deadlock should still be aborted. When frequent, these false positives may eventually degrade performance.

Figure 1 illustrates such an example. After T2 sends a NACK to T1, which is logically older, it sets its `possible_cycle` flag. When T2 receives a NACK from another logically older transaction, T3, T2 must conservatively abort to avoid a possible deadlock scenario, even though there is no actual deadlock in the system. If T2 were able to commit successfully, this unnecessary abort would clearly degrade the performance.

To eliminate the performance degradation due to conservative deadlock avoidance, we need an accurate deadlock detection (ADD) mechanism. A common method to implement an ADD mechanism is to construct a wait-for graph (WFG) that encodes the dependency information in the system; if a cycle is detected in WFG, there is a deadlock. And to construct a WFG, an accurate dependency indicator is required to track the system-wide, up-to-date dependency information.

NACK messages can be used as such an indicator, since they essentially represent dependencies among transactions. Simply, if T1 receives a NACK from T2, it indicates that T1 would attempt to wait for T2. Hence, an ADD mechanism can be built on top of NACK messages.

If there exists an ADD unit in the system that can exactly tell whether introducing a new dependency (i.e., T1 $\rightarrow$ T2 in this example) can induce deadlock or not, the system would abort a transaction only when definitely necessary, and allow transactions to stall more aggressively. This reduction in unnecessary aborts will result in improved performance.

2.2 Dependency Tree Construction

For eager HTM systems [20, 28], acquiring an exclusive access to a memory (cache) block amounts to acquiring a lock. Under such systems, attacker stalling policy [20] can form a potentially large dependency tree where one transaction may make many other transactions stall, either directly or indirectly.

Figure 2 depicts such a scenario. In this case, Transaction 4 (T4), which does not have a direct dependency on T2, still has to indirectly stall for T2 since T3, which T4 has a direct dependency on, is stalling for T2. This dependency *transitively* propagates through the dependency tree, and the transaction positioning at the root of the (sub)tree in effect appears to hold all the memory blocks that its dependents have acquired.

The most obvious negative effect of stalling a transaction with high degree of dependency is that it also stalls all the dependents, regardless of actual dependency. However, the more subtle effect is that the other transactions not yet joined the dependency tree are soon likely to join the tree as well, which would force the entire system into high congestion.

In this sense, to improve the overall performance, it would be better to abort the highly depended transaction early in time. To accomplish this, the underlying contention management scheme should be provided with the dependency tree to make an informed decision whether to stall or abort a transaction.

For systems that already implement NACKs, the necessary dependency information can be easily constructed without the significant modification of the existing coherence protocol; each NACK contains the peer-to-peer information as to which transaction is stalling for who, and by collecting the information we can construct a coherent view of the dependency. Thus formed dependency information can be used to implement a flexible contention resolution scheme that would determine whether to stall or abort an attacking transaction based on the dependency degree.

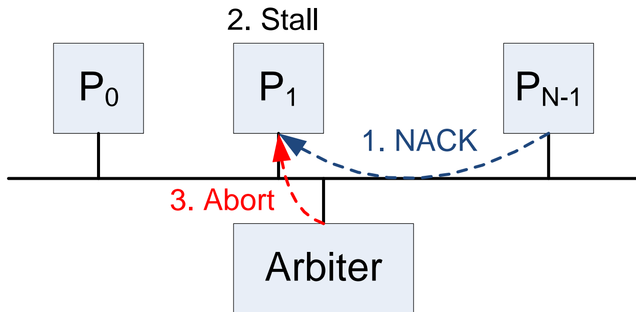

Figure 3. High-Level Overview of Accurate Deadlock Detection.

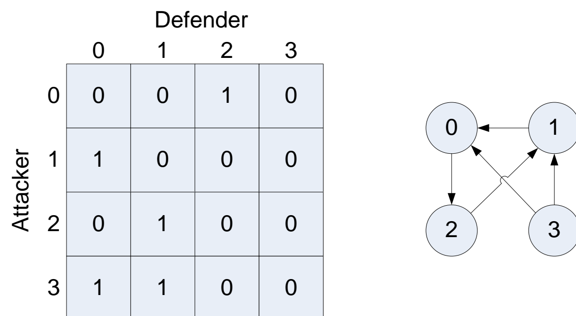

Figure 4. The Wait-For Table and the Resulting Graph.

2.3 Carrier Sensing

To provide robust performance across a wide range of workloads, TM systems must accurately detect under/overutilization of the system, and dynamically adjust the concurrency level according to the observed *degree of contention* [29].

More specifically, recent work utilized abort [29] or commit [2] rates to measure the degree of contention in the system. While these metrics can be effective, they are fundamentally reactive in the sense that they provide a decision point only after a certain number of repeated transactional aborts or commits have occurred. In short, they do not allow the concurrency control to take effect at the moment of conflict detection. Moreover, they cannot deal with fine-grain problems such as underutilization due to a long backoff.

Researchers in communication and networking areas have dealt with similar problems for years, and adopted the *carrier-sensing* technique as one of the key techniques for dynamic concurrency control in communication systems [17]. In this approach, each transmitter senses a shared medium, and probabilistically sends a packet, only if the medium is idle.

Similarly, we can view concurrent transactions as competing over a shared medium/resource (e.g., ownership over shared memory objects). In this view, NACKs can be utilized as an indicator to estimate the current utilization level of a TM system: high NACK count indicate high contention, thus utilization, and low NACK count low utilization. With the information, the controller could efficiently adjust the concurrency level. As an example, shown below, a TM concurrency control routine could use the number of snooped NACK messages per period to dynamically control the number of active threads in the system.

```
if(num_nacks_per_period>high_threshold){
  decrease_number_of_active_threads();
}
else if(num_nacks_per_period<low_threshold){
  increase_number_of_active_threads();
}
```

In Section 4.4, we discuss a technique that incorporates carrier sensing to eliminate the underutilization problem of exponential backoff.

3. System Design

In this section we describe how the proposed use cases (Section 2) can be practically implemented on a TM system. Hardware and hybrid TM designs that utilize hardware NACKs as their conflict detection mechanism could readily implement our design.

3.1 Accurate Deadlock Detection

Figure 3 shows a high-level overview of accurate deadlock detection (ADD). For simplicity, we assume a broadcast-based interconnection network. However, distributed implementations for

```
bool reduceGraph(G) {
 // flag is set when G is updated
 bool flag=remove_all_sources_and_sinks(G);
 return flag;
}

bool detectDeadlock(G) {
 reducible=false;
 do {
  reducible=reduceGraph(G);
 } while(reducible)
 if (G.size()==0) {
  // empty after the reduction process
  // so, there is no deadlock
  return false;
 } else {
  // non-empty after the reduction process
  // so, there is deadlock
  return true;
 }
}
```

Figure 5. Deadlock Detection Algorithm.

directory based systems are also possible. We also assume one active transaction per processor, and the information on transactional events such as commit and abort are broadcasted.

In our design, there exists a centralized *arbiter* that maintains the dependency information among processors, and arbitrates when there is a conflict. The arbiter snoops all coherence messages and updates its *wait-for table* (W) when it snoops a NACK message, to build the up-to-date dependency information of the system.

Figure 4 shows an example of W and its resulting wait-for graph (WFG). Row i of W encodes processors on which P_i is waiting for. Column j represents the processors that P_j has caused to stall. For example, $W(i, j)$ is set when attacking processor i is stalling for defender P_j. When P_i commits or aborts, column i and row i are cleared. Each node in WFG represents a processor and edges between the nodes represent the wait-for relationship — head denotes the defender, and tail denotes the attacker. The problem of detecting any deadlock in the system is equivalent to the problem of finding any cycle in the corresponding WFG [26]. For example, the WFG in Figure 4 has a cycle ($P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_0$) and equivalently, there is a deadlock in the system.

When the arbiter snoops a new NACK message, it detects a deadlock in the system by running the deadlock detection algorithm shown in Figure 5 over W. At each iteration of the main loop, the

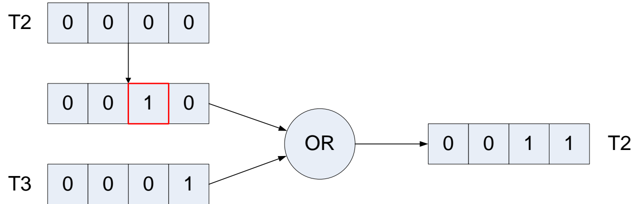

Figure 6. Propagation of Dependency Information.

algorithm attempts to reduce the WFG by removing all source (i.e., no incoming edges) and sink (i.e., no outgoing edges) nodes and their associated edges from the graph. If there is any update at an iteration, it proceeds to the next iteration. If not, it terminates the main loop. Note that the runtime complexity of the algorithm is $O(P)$, where P is the number of processors, since at least one node is removed from WFG at each reduction step, if there is any update. After the entire reduction process, if the resulting WFG is non-empty, it indicates that there is at least one cycle in WFG and the algorithm reports that there is deadlock in the system. Then, the arbiter sends a NACK message to an attacking processor to make it abort. If the resulting WFG is empty, there is no deadlock in the system and the attacking processor can stall.

There are several hardware implementations that detect deadlock in the system in a fast and efficient way [25, 27]. In particular, the data structure and logic of our arbiter are similar to the deadlock detection unit [25] that was implemented in hardware in an area- and delay-efficient way. In addition, since the deadlock detection process is not on the performance critical path (i.e., any delay in deadlock detection can be considered as additional stalling cycles), we believe the arbiter can be implemented with reasonable hardware complexity. However, we do plan to explore the hardware complexity and overhead in a later version of this work.

3.2 Dependency Tree Construction

For a system that utilizes NACKs to implement transactional conflict detection, the dependency tree information can be constructed in a distributed fashion. Similar to Section 3.1, we assume that the begin, abort, and commit of a transaction is broadcasted through a shared interconnect. We also assume that only one active transaction is supported per processor.

Each processor maintains a bit vector where each bit denotes a processor in the system. This bit vector is flash-cleared when a transaction begins, aborts, or commits.

To maintain the direct dependency information, a processor sets the corresponding bit to 1 whenever it sends a transactional NACK to another processor. And when a transaction abort is broadcasted over the interconnect, each processor clears the corresponding bit if it is set to 1.

To propagate indirect dependency information, each processor then includes this bit vector in its coherence request messages. For example, if an attacking transaction tries to acquire a memory block that is held by a defender, the attacking processor would send its bit vector along with the exclusive ownership request.

Figure 6 depicts the dependency information propagation in the context of Figure 2. In particular, the figure captures the moment when T4 is stalling for T3, and T3 is requesting an ownership over a block held by T2. Note that the least significant bit of T3's bit vector is set to 1 to denote that T4 is stalling for T3.

When the defender processor decides to NACK the attacker, it would first set the corresponding bit to 1 in its bit vector. The defender would then perform a bitwise OR between its bit vector and the attacker's bit vector, to store the new value in its own bit vector. The defender will then send a NACK to the attacker.

Simulated System Settings	
CPU	16 x86 cores Single-issue, in-order Non-memory IPC=1
Per Core L1 Cache	4-way unified, 64 KB 1-cycle hit latency
Shared L2 Cache	4-way unified, 8 MB 10-cycle latency
Memory	100-cycle latency
Interconnect	Shared bus, 6-cycle arbitration latency
HTM Settings	
Version Management	Eager
Conflict Detection	Pessimistic
SigTM Settings	
Signature Register	2048 bits
Hash Function	Permuted cache line address

Table 1. Simulation Settings.

Note that the bit vector now represents the transitive dependency relation. Again, in Figure 6, we can see that the resulting bit vector correctly depicts that both T3 and T4 are stalling for T2. Counting the number of bits set would give the number of direct/indirect dependents that are stalling for a particular transaction. A transaction could then utilize this information to decide whether it should stall or abort.

3.3 Carrier Sensing

To implement optimizations using carrier sensing, a hardware performance monitoring counter per processor and its associated instructions are required. The performance monitoring counter simply counts the number of snooped NACK messages. In addition, two instructions are required to read or reset the counter. Since many modern commercial processors provide a variety of performance monitoring units and software interface [8], in most cases the carrier-sensing functionality can be included just by adding an additional performance monitoring event.

4. Motivating Results

Section 4.1 describes the simulation framework we use in this work. We implemented the techniques described in Section 3 on top of this simulator. Sections 4.2–4.4 provide our preliminary performance results.

4.1 Methodology

For our experiments, we use an execution-driven simulator. Table 1 specifies the simulated machine. Our simulation framework models a multi-core system based on x86. The processor model assumes an IPC of 1 for all non-memory instructions, but we do model all the memory hierarchy timings and congestions through event queueing.

Similar to [20], our HTM system uses eager version management and pessimistic conflict detection. Private L1 caches provide the necessary TM bookkeeping functionalities to implement an HTM system. The baseline conflict resolution policy is to stall the attacker with conservative deadlock avoidance; the attacking transaction stalls for a memory block unless it risks a possible deadlock. The stalling mechanism is implemented by the defender sending a NACK message to the attacker.

To study the effectiveness of the carrier-sensing technique, we also experiment with a hybrid TM system that models eager SigTM [6]. The hybrid TM also utilizes the NACK mechanism to handle conflicts between transactions. The hardware signature register is 2048 bits, and we use permuted cache line address as the

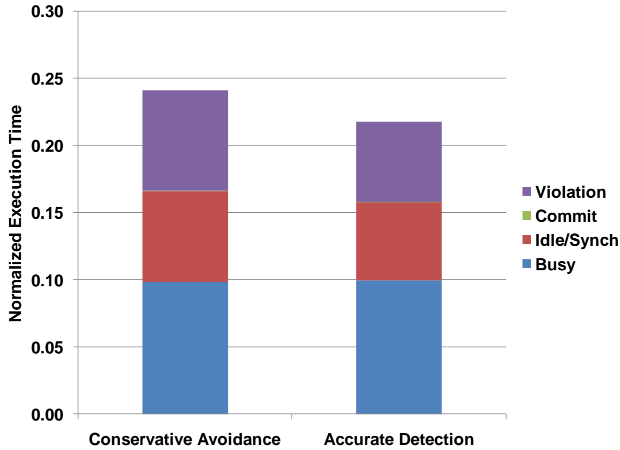

Figure 7. Execution Time Breakdown of Genome Benchmark.

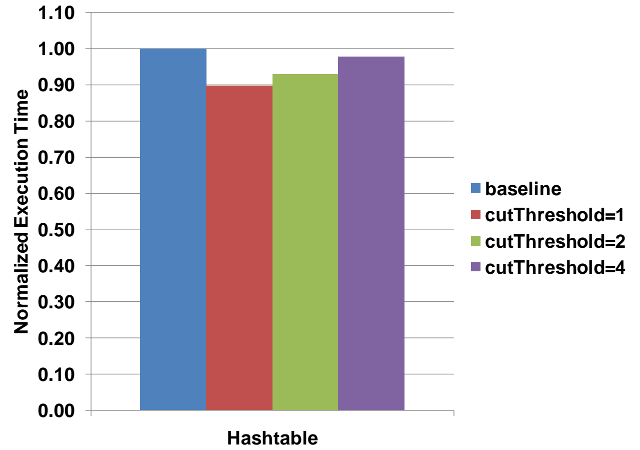

Figure 8. Dependency Chain Cutting with Varying Cut Thresholds.

hash value. For performance comparison, we implement both randomized linear and exponential backoff as contention management schemes.

For all the experiments, we use 16 processors. However, since the performance of larger scale systems would be more susceptible to the effectiveness of TM concurrency control, we expect the proposed techniques to be more beneficial on such systems. Finally, we use Genome and Kmeans benchmarks from the STAMP benchmark suite [6] and a Hashtable benchmark.

4.2 Accurate Deadlock Detection: Aggressive Stalling

To showcase the usefulness of accurate deadlock detection (ADD), we implement aggressive stalling with ADD on an HTM, and compare the performance against stalling with conservative deadlock avoidance (CDA). Figure 7 shows the results with Genome benchmark, where the execution time with 16 processors is normalized to the sequential execution time. Each bar is broken down into 1) time for useful instructions and cache misses (Busy), 2) idle and synchronization time, 3) time spent for commit, and 4) time spent on aborted transactions (Violation).

As shown in the figure, the main performance benefit is from the reduction in violation cycles. Our results indicate that aggressive stalling with ADD reduces the number of aborted transactions by 20.5%, compared to stalling with CDA. This means that allowing more transactions to stall using ADD is beneficial, since many of them can eventually reach to a successful commit. Overall, when compared to stalling with CDA, aggressive stalling with ADD improves the performance of Genome application by 9.9%.

4.3 Dependency Tree Construction: Dependency Chain Cutting

We model the dependency tree construction technique described in Section 3.2 on an HTM system. To utilize this information, we also implement a dependency chain cutting mechanism in hardware. Under this scheme, an attacking transaction is aborted if it 1) receives a NACK from an older transaction, and 2) it has more than a designated number of dependents. For the rest of the cases the attacker would stall unless it risks a possible deadlock.

We use the Hashtable microbenchmark to compare performance. In the benchmark, concurrent transactions insert data into a shared hash table. The hash table is designed as buckets of linked lists, and each transaction performs a predefined number of data insertions. The benchmark input is constructed to cause high contention among transactions. We measure the execution time taken to successfully commit a thousand transactions.

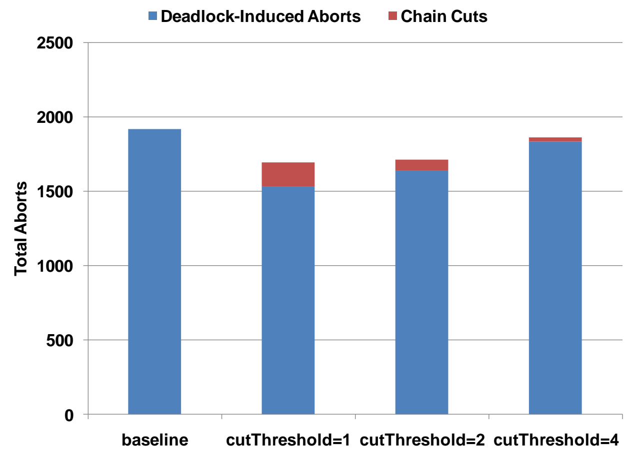

Figure 9. Breakdown of Aborts on Hashtable Benchmark.

Figure 8 shows the performance of the dependency chain cutting method over varying chain cut thresholds. The execution time is normalized to the baseline. In the figure, baseline denotes the case where we stall the attacker transaction unless it risks a potential deadlock. cutThreshold=1 represents the case where we cut the dependency chain by aborting the attacker if it has at least 1 dependent. cutThreshold=2,4 represent the case where we abort the attacking transaction with at least 2 and 4 dependents, respectively. baseline could be considered as a special case of chain cutting where the cut threshold is set to infinity.

For this particular benchmark, cutThreshold=1 gives the best performance improvement; compared to the baseline, the speedup is 10%. cutThreshold=2 and cutThreshold=4 schemes bring about 7% and 2% speedup, respectively. The diminishing performance improvement over the increasing threshold is due to the fact that the actual occurrence of dependency cutting reduces when we increase the threshold.

For the same benchmark, Figure 9 shows the breakdown of aborts into those induced by conservative deadlock avoidance, and those by proactive dependency chain cutting. When the cut threshold is 1, there are 159 chain cuts in total. The cut instances are

```
void OrigExponentialBackoff(numRetries){
 stall=pow(2,min(numRetries,M));
 for(i=0;i<stall;i++){
  // idle
 }
}

void ExponentialBackoffWithCS(numRetries){
 stall=pow(2,min(numRetries,M));
 for(i=0;i<stall;i++){
  // periodically senses the medium
  if(i%period==0){
   numNacksInPeriod=ISA_READ_NACK_CNT();
   if(numNacksInPeriod<low_threshold){
    // if the medium is underutilized.
    // terminate the backoff early
    break;
   } else {
    ISA_RESET_NACK_CNT();
   }
  }
 }
}
```

Figure 10. Baseline Exponential Backoff and Exponential Backoff with Carrier Sensing.

reduced to 78 and 28 for thresholds 2 and 4, respectively. As less dependencies are cut, performance converges back to the baseline.

It is also important to point out that the overall performance shows high correlation to the number of aborts, and that the chain cutting approach actually results in reduced number of aborts. In plain sight, we could expect the chain cut method to generate more aborts, since we are injecting aborts even when we do not risk a deadlock. However, as can be seen from Figure 9, the artificial injection of aborts actually results in reduced number of total aborts. This can be attributed to the fact that the proactive dependency cutting prevents the system from entering the highly congested mode of operation.

4.4 Carrier Sensing: Exponential Backoff with Overshoot Avoidance

To demonstrate the usefulness of the carrier-sensing technique, we apply it to eliminate the underutilization problem (i.e., *overshooting* of the backoff length) of exponential backoff (EB). EB is a widely used contention management scheme for TM systems [24]. While EB is in general more effective than linear backoff (LB) on workloads with long and highly conflicting transactions (since it quickly escapes from high contention), it can suffer from a temporal underutilization, as it rapidly increases the backoff length.

To eliminate this problem, we propose an exponential backoff scheme with carrier sensing (EB-CS). Figure 10 compares EB and EB-CS. In the main loop of backoff, EB-CS periodically senses the utilization of the system and early terminates the backoff if it detects underutilization. With this approach, EB-CS can eliminate the overshooting of the original EB while keeping the benefits of EB in quickly escaping high contention.

We implement EB-CS on eager SigTM, and measure the performance using Kmeans benchmark. For this experiment, we use 1024 for `period` and 3 for `low_threshold`. Figure 11 shows the result. In addition to the segments described in Section 4.2, each bar has additional segments such as the time executing read barriers (RB) and write barriers (WB), and the time spent in backoff. In

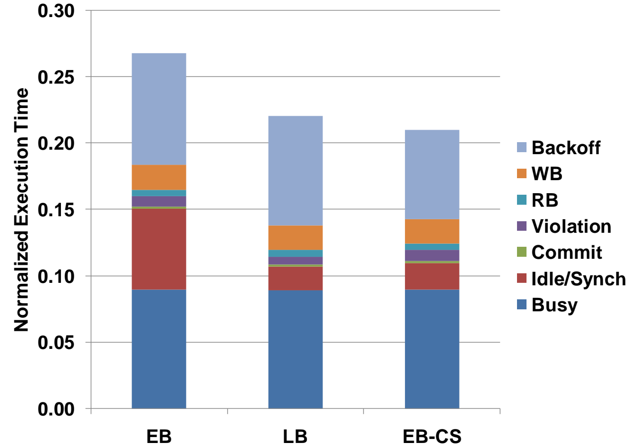

Figure 11. Execution Time Breakdown of Kmeans Benchmark.

the figure, EB and EB-CS present the execution time of the two EB implementations. For completeness, we also include the execution time with linear backoff (LB).

Compared to EB, EB-CS improves the performance by 21.5%. The performance improvement is mainly from the reduction in Idle/Synch and Backoff cycles. With EB, there are several transactions that suffer from overshooted backoff. Since it causes a temporal load imbalance in the system, EB leads to high Idle/Synch cycles. In contrast, with periodic carrier sensing, EB-CS eliminates the overshooting problem by early terminating the backoff when underutilization is detected.

When comparing LB against EB, we observe that LB generally outperforms EB on Kmeans, since the workload exhibits relatively low contention and the majority of its transactions are short [6]. EB-CS, on the contrary, manages to provide similar performance by eliminating the overshooting problem of EB.

5. Related Work

The effects of contention on the performance of TM systems have been studied extensively [2, 5, 14, 24, 29, 30]. Especially, Herlihy *et. al.* [14] propose a *contention manager* based approach on STM where a software conflict resolution routine performs localized, peer-to-peer conflict resolution. Bobba *et. al.* [5] delineate performance pathologies in HTM systems, and discuss possible solutions for reducing the contention. Regardless of the implementation, these approaches usually boil down to two major issues: 1) obtaining the dependency information, and 2) enforcing a priority mechanism. In this paper we show that NACKs can be effectively used to obtain such dependency information.

Under the TM context, the use cases of NACKs have been limited to stalling an attacking transaction [20, 21, 28]. While Moore *et. al.* utilize NACKs to implement a conservative deadlock avoidance policy [20], little work has been done on how TM concurrency controllers can effectively exploit the information available from NACKs. This paper discusses novel use cases on utilizing NACKs in the context of contention management: for accurate deadlock detection, dependency tree construction, and carrier sensing.

Koskinen and Herlihy [18] also discuss an efficient deadlock detection scheme and its use on the improvement of STM performance. However, the implementation purely relies on software mechanisms for dependency information propagation and deadlock detection. Our paper builds on this approach by utilizing hardware NACK messages to construct a deadlock detection mechanism. This mechanism can be implemented on a wide range of hardware

and hybrid TM systems that utilize NACKs. Moreover, we describe a hardware structure that can efficiently calculate the existence of a deadlock, given the wait-for graph.

Utilizing NACKs for contention resolution has another benefit in the sense that we can design more proactive contention managers or runtime schedulers. Compared to those approaches that utilize the indirect outcome of contention as the concurrency feedback [2, 29], NACK based approaches can be more effective, because the conflict resolution can take place at the exact moment when a conflict has been detected.

6. Conclusions and Future Work

Obtaining the transactional behavior information in an accurate and efficient fashion allows to construct an effective concurrency controller. In this paper, we identified that NACKs readily available in several TM designs can efficiently provide the critical system information, such as dependencies among transactions and the current utilization level of the system.

Specifically, we proposed the three efficient use cases of NACKs for TM systems: 1) accurate deadlock detection, 2) dependency tree construction, and 3) carrier sensing. Unlike conservative deadlock avoidance, accurate deadlock detection precisely detects the deadlock in the system, without any false positives. Dependency tree construction enables each transaction to track its degree of dependency. Carrier sensing can estimate the current utilization level of the system. We described our prototype design that extends hardware and hybrid TM systems to support the three proposed techniques. We also used the proposed techniques to implement an enhanced concurrency controller that performs aggressive stalling, dependency chain cutting, and exponential backoff with overshoot avoidance. Our preliminary results showed that the baseline TM systems readily benefit from the proposed techniques (up to 21.5% speedup).

As our future work, we plan to study the effectiveness of the proposed techniques with a variety of TM workloads on a larger scale system. We also plan to investigate the additional hardware complexity and overheads required to implement the proposed techniques in more detail.

Acknowledgments

Woongki Baek was supported by a Samsung Scholarship and an STMicroelectronics Stanford Graduate Fellowship. Richard Yoo was supported by a David and Janet Chyan Stanford Graduate Fellowship. This work was supported by the Stanford Pervasive Parallelism Laboratory.

References

[1] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th international symposium on high performance computer architecture*, pages 316–327, 2005.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th European conference on parallel processing*, pages 719–728, 2008.

[3] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd annual international symposium on microarchitecture*, pages 156–167, 2009.

[4] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th annual international symposium on computer architecture*, pages 24–34, 2007.

[5] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual international symposium on computer architecture*, pages 81–91. 2007.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE international symposium on workload characterization*, pages 35–46, 2008.

[7] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on computer architecture*, pages 69–80. 2007.

[8] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on low power electronics and design*, pages 221–226, 2005.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on architectural support for programming languages and operating systems*, pages 336–346, 2006.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on architectural support for programming languages and operating systems*, pages 157–168, 2009.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th international symposium on distributed computing*, pages 194–208, 2006.

[12] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *OOPSLA 2005 workshop on synchronization and concurrency in object-oriented languages*. 2005.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on computer architecture*, pages 102–113, 2004.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on principles of distributed computing*, pages 92–101, 2003.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

[16] Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions. 2012.

[17] L. Kleinrock and F. Tobagi. Packet switching in radio channels: Part I–carrier sense multiple-access modes and their throughput-delay characteristics. *IEEE transactions on communications*, 23(12):1400–1416, 1975.

[18] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *Proceedings of the 20th annual symposium on parallelism in algorithms and architectures*, pages 297–303, 2008.

[19] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th symposium on principles and practice of parallel programming*, pages 209–220, 2006.

[20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th international symposium on high performance computer architecture*, pages 254–265, 2006.

[21] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference*

on architectural support for programming languages and operating systems, pages 359–370, 2006.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th symposium on principles and practice of parallel programming*, pages 187–197, 2006.

[23] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th annual international symposium on microarchitecture*, pages 185–196, 2006.

[24] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual symposium on principles of distributed computing*, pages 240–248, 2005.

[25] P. H. Shiu, Y. Tan, and V. J. Mooney III. A novel parallel deadlock detection algorithm and architecture. In *Proceedings of the 9th international symposium on hardware/software codesign*, pages 73–78, 2001.

[26] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. Wiley, 7th edition, 2002.

[27] X. Xiao and J. J. Lee. A novel O(1) deadlock detection methodology for multiunit resource systems and its hardware implementation for system-on-chip. *IEEE transactions on parallel and distributed systems*, 19(12):1657–1670, 2008.

[28] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th international symposium on high performance computer architecture*, pages 261–272, 2007.

[29] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th annual symposium on parallelism in algorithms and architectures*, pages 169–178, 2008.

[30] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *the 1st workshop on languages, compilers, and hardware support for transactional computing*, 2006.