

Transactional Conflict Decoupling and Value Prediction

Fuad Tabba, Andrew W. Hay, James R. Goodman

Under Submission

Abstract

This paper explores data speculation for improving the performance of Hardware Transactional Memory (HTM). We attempt to reduce transactional conflicts by decoupling them from cache coherence conflicts; many HTMs do not distinguish between transactional conflicts and coherence conflicts, leading to false transactional conflicts. We also attempt to mitigate the effects of coherence conflicts by using value prediction in transactions. We show that coherence decoupling and value prediction in transactions complement each other, because they both speculate on data in ways that are infeasible in the absence of HTM support.

As a demonstration of how data speculation can improve performance, we introduce DPTM, a best-effort HTM that mitigates the effects of false sharing at the cache line level. DPTM does not alter the underlying cache coherence protocol, and requires only minor, processor-local, modifications.

We evaluate DPTM against a baseline best-effort HTM, and compare it with data restructuring by padding, the most commonly used method to avoid false sharing. Our experiments show that DPTM can dramatically improve performance in the presence of false sharing without degrading performance in its absence, and consistently performs better than restructuring by padding.

1 Introduction

Parallel programming is fast becoming widespread. Most processor manufacturers today are producing chips with an increasing number of cores [16], while software engineering tools have yet to simplify the programming for these cores. Writing correct parallel programs is complicated by deadlock, livelock, starvation, and data races [31]. Making parallel programs efficient is further complicated by convoying [31] and false sharing [10, 17, 21].

Methods of aggressive speculation in hardware, such as Thread-Level Speculation (TLS) [2, 30, 51], Hardware Transactional Memory (HTM) [23], and Speculative Lock Elision [38, 45], attempts to reduce the challenges of parallel programming. However, processor manufacturers have been slow to adopt such mechanisms, because they may not perceive the cost-benefit tradeoff to be worthwhile.

In this paper, we show that many HTM proposals, and similar methods of aggressive speculation, conservatively infer transactional conflicts from cache coherence conflicts [4, 12, 15, 23, 42, 45–47], which could lead to false transactional conflicts that adversely affect performance. Drawing inspiration from Huh et al. [27], we demonstrate that by decoupling transactional conflicts from coherence conflicts, we can reduce false conflicts between transactions; and that by speculating on data values, we can reduce the delays coherence conflicts incur.

Specifically, we explain how decoupling and data speculation can improve performance in the presence of false sharing and silent stores (including temporally silent stores) [32, 33]. We justify the cost of these added mechanisms by showing that, because transactions are already in speculative mode, HTMs can speculate on data in ways that are infeasible in the absence of HTM support.

False sharing can be difficult to mitigate, and its effects are especially pronounced in HTM [24, 42]. Typical methods for mitigating false sharing require data restructuring, which goes against two promises of Transactional Memory (TM): abstraction and composition [31]. Restructuring requires low level knowledge of the system, breaking abstraction; it is also difficult to restructure external code that suffers from false sharing, which hinders composition. In our opinion, for TM to become viable for parallel programming, it should avoid the worst effects of false sharing.

This paper makes the following contributions: (i) we describe how decoupling transactional conflicts from coherence conflicts in HTM reduces apparent transactional conflicts, and how value prediction in HTM complements the decoupling by reducing the delays of coherence conflicts in transactions; (ii) to mitigate the effects of false sharing in transactions, we present DPTM, an enhanced best-effort HTM that relies on a conventional coherence protocol, and adds little processor-local modifications; (iii) we evaluate various design points for DPTM and compare them with data restructuring by padding, the most commonly used method to avoid false sharing.

Our evaluation, using STAMP [40], SPLASH-2 [56], and other benchmarks, shows that DPTM significantly improves performance in benchmarks that exhibit false sharing — more than restructuring by padding — without having adverse effects on benchmarks that do not exhibit false sharing.

2 The False Sharing Problem

False sharing at the cache line level can have a big impact on the performance of parallel programs [10, 17, 21, 25, 29]. It occurs when different processors access distinct data objects that share the same cache line, and at least one processor is modifying one of the objects. Because the cache line is the unit of granularity for coherence, such *logically* nonconflicting accesses are serialized.

The effects of false sharing are not easy to mitigate. Most methods we have encountered are oriented towards the restructuring and padding of data objects [20, 27, 28, 42, 54], so that logically nonconflicting accesses to different objects are also nonconflicting as far as the coherence protocol is concerned.

Such restructuring of data can be difficult to apply in practice. First, the programmer needs to identify the data objects that cause false sharing. Once the objects are identified, they are aligned to cache line boundaries, which requires knowledge of machine-specific details such as the cache line size.

Then typically, each object is padded so it occupies a whole cache line by itself, which increases memory use

and fragmentation, and adversely impacts locality. These changes are often machine-specific, so the benefits of code modified in such ways might not be portable. Moreover, using high-level languages, such as Java, where the underlying virtual machine or interpreter does not lay out the data structure the way it is specified in the program, can make data restructuring impractical.

False sharing may also be introduced to a program by the use of external libraries that suffer from it. Modifying external code is often difficult or infeasible.

False Sharing in Transactions

False sharing is a bigger problem when it occurs in conjunction with HTM, because it leads to the aborting or serialization of whole transactions that could have otherwise executed concurrently [24, 42]. It is a problem that has often been observed by experts on transactional memory and parallel programming [1, 5, 8, 13, 18, 20, 24, 42, 48, 49, 55, 57, 58].

The example in Figure 1 shows the timeline of two concurrent transactions. These transactions access different data on the same cache line at one point during their execution, i.e., they exhibit false sharing. The transactions in this example at no point have any true conflicts, and so any order of their component operations is allowable.

Ideally, these transactions should be able to run completely in parallel, as shown in (a). Because many HTM proposals infer transactional conflicts from coherence conflicts, such implementations do not distinguish between true and false sharing, stalling transactions (b), or aborting them altogether (c).

We expect a solution to the problem of false sharing to result in an execution similar to the one shown in (d). Such a solution would likely not eliminate all the delays caused by false sharing, because some cache data still needs to be communicated. However, it should be able to mitigate these effects by overlapping the delay with other speculative operations.

The typical solution by restructuring and padding, in addition to the difficulties it poses in non-transactional systems, also goes against some of the promises of TM: that of abstraction and composition [31].

Composition in TM is the ability to combine and use different code in transactions. By using external code that is not optimized for false sharing, or that is optimized for a different machine, the usefulness of composition in TM is undermined.

Abstraction aims to hide irrelevant complexity and detail, but restructuring requires exposure to architectural and other low level details, knowledge a typical programmer may not have, and should not need. Therefore, an HTM should also abstract away the adverse effects of false sharing.

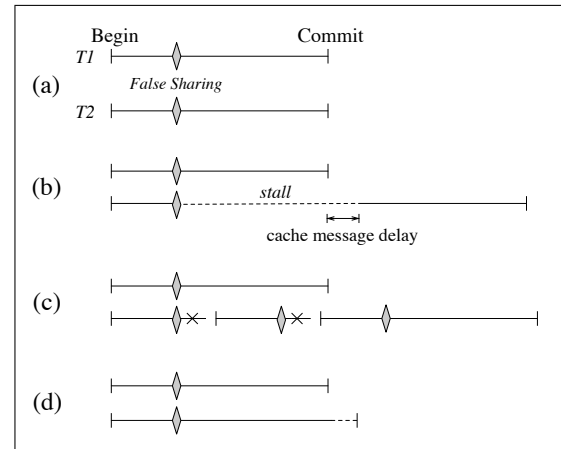


Figure 1: False sharing in HTM

3 Coherence Decoupling and Value Prediction in Transactions

From the first HTM proposal [23], the means used by many HTMs to detect cache coherence conflicts has also been employed and extended to assure that transactional conflicts are detected. Coherence conflict detection, however, is implementation dependent, usually classifying certain patterns as a coherence conflict even in some cases where no conflict exists.

One pattern is false sharing. While logically no sharing — and no conflict — occurs, this widely recognized problem is nevertheless treated as a coherence conflict. Likewise with silent stores and temporally silent stores [32, 33], where two or more threads truly share data and at least one of them writes to the data ultimately leaving its value unchanged: no conflict occurs, yet virtually all coherence implementations treat them as a coherence conflict.

None of these constitute a transactional conflict, but are characteristically treated as such because the coherence detection mechanism does not distinguish between them. Yet these patterns may result in significant performance degradation, as their occurrence during transactional speculation can cause a transaction to stall, or even to abort. Aborting may result in much more serious degradation and could, for example, be the cause of livelock.

To mitigate the effects of false conflicts in transactions, we argue that HTMs should decouple transactional conflicts from coherence conflicts. Moreover, because transactions are *already* in speculative execution mode, this makes it easier to use value prediction in transactions.

HTMs that associate transactional conflicts with coherence conflicts also associate cache line states with transactional states, i.e., modified data resides on exclusive cache lines, while read data resides on valid cache lines. These associations enable HTMs to detect potential transactional conflicts using the coherence protocol, but could result in false transactional conflicts.

One alternative is to speculate inside a transaction without this association, then restore it before the transaction tries to commit. This could be done by ensuring that cache lines representing the read and write sets are in their expected state at commit time, and that the read set data matches the current data values.

Therefore, when a cache line containing read data is invalidated, instead of aborting, a transaction continues without triggering a transactional conflict, speculating that the data it has read will still be valid. The transaction tracks which parts of the cache line contains read data; and before it can commit, the transaction must *validate* by re-acquiring the cache line to ensure that the read data is unchanged. If it has changed, only then is a transactional conflict triggered, thereby associating transactional conflicts with *modifications* to the shared data.

This same mechanism can also be used for value prediction in transactions: a transaction could speculate on any value as long as it ensures, that the read cache line is valid and holds the predicted value at the time it commits.

For example, in many HTMs, when a thread attempts to read data that is not in a valid state in its cache, it requests the cache line with the data and stalls waiting for the request. Rather than stall, a transaction could predict the data value because it *already* is in speculative execution and does not have to take an additional checkpoint. When it predicts accurately, the transaction mitigates the effects of the stall.

One method of prediction would use the data value in a *stale* cache line [27], i.e., an invalid cache line still containing the data it held when it was last invalidated. If the cache line is stale, a transaction could predict that the particular part it wants to read has not changed, and speculate using that value. Such predictions would be accurate in the cases of false sharing and silent stores [27].

When a transaction speculates on the value of a cache line, it must verify that the speculation was correct before it is able to commit. A transaction must track the parts of the cache lines on which it is speculating, and validate those cache lines by commit time. It validates the cache line by acquiring the line, and ensuring that the value it predicted matches the current data value in the cache line.

4 DPTM Description

4.1 Overview

This section demonstrates how data speculation could mitigate the effects of false conflicts by example of *Decoupling and Prediction TM* (DPTM), an enhanced *best-effort* HTM [14].

As a baseline, we assume a conventional best-effort HTM that uses eager conflict detection and lazy version management [11, 42], tracks its read and write sets in a transactional cache (such as the L1 cache) [15], and keeps transactional writes in a write buffer until the transaction commits. We also assume a cache coherence protocol that distinguishes stale cache lines from other invalid states. This baseline HTM associates transactional conflicts with coherence conflicts: when a cache line containing transactional data receives an invalidation request (due to a coherence conflict), this triggers a transactional conflict. This baseline is similar to other best-effort HTM proposals [6, 23, 26], and to the one used in Sun’s Rock processor [15].

DPTM modifies this baseline HTM as follows.

DPTM decouples transactional conflicts from coherence conflicts using value-based conflict detection [43]. The values in a transaction’s read set are monitored for change, and a transactional conflict is triggered only if a value changes. This reduces the effects of false sharing, because transactional conflicts are now restricted to changes *only* to the data used inside a transaction, rather than coherence conflicts over whole cache lines.

Moreover, DPTM can speculate when attempting a load from a stale cache line using the value of the stale data and validating it later. When speculating, it assumes that the cache line became stale because of a coherence conflict caused by false sharing. When this prediction is accurate, it could eliminate the stalling due to false

sharing.

In addition to mitigating the effects of false sharing, DPTM also mitigates the effects of silent stores. In silent stores, data values do not change; therefore, DPTM's value-based conflict detection does not consider them to be transactional conflicts. Moreover, DPTM's speculation on a stale cache line that became stale because of a silent store would likely be accurate.

4.2 Detailed Description

DPTM does not alter how the baseline HTM begins its transaction. It alters the loading and storing of values inside a transaction, the handling of coherence conflicts and how a transactional conflict is interpreted, and the process of committing a transaction.

4.2.1 Loading a Value

When a transaction in DPTM attempts to load data not present in its transactional cache, it proceeds, as in the baseline, by issuing a request for the cache line and stalling for the request. Otherwise, if the cache line is present and in a valid state, the load is a cache hit.

If the cache line is stale, and DPTM *predicts* that the value has not changed, then it may serve the load using the stale data while simultaneously issuing a cache request for the data. The load proceeds as if it were a cache hit. On the other hand, if it *predicts* that the value has changed, it behaves conventionally, as if the line were not present.

By default, DPTM speculates only on stale data that is already part of a transaction's read set. This is conservative because the transaction has already read that data, and if it has changed, the transactions must abort anyway. Therefore, it has little to lose in the case of a mis-prediction.

Associated with all cache lines in the transactional cache are read mark bits that indicate which parts of each line have been read [39]. Each bit monitors reads from a subset of its cache line, i.e., the bit is set when its associated subset is read inside a transaction. These bits are used for validation, as only the parts of stale cache lines with their associated bits set need to be validated.

The number of read mark bits added per cache line determines the granularity level of DPTM's transactional conflict detection; the greater the number of bits the finer the granularity, and the more cases of false sharing that can be detected.¹ This could conceivably go down to the individual bit level.

When a processor receives a response to a cache request, the data in the cache whose read mark bits are set is validated against the returned data. If the data has not changed, validation succeeds and the transaction proceeds as normal. If the data has changed, validation fails, the transaction aborts, and all read mark bits are cleared. In all

¹For example, for a 64 byte cache line and a conflict detection granularity level of 4 bytes, DPTM requires an additional 16 bits for each cache line.

cases, the old cache line data is replaced with the returned data.

4.2.2 Storing a Value

When a transaction performs a store, the baseline HTM requests exclusive permissions for the cache line and stalls while it obtains these permissions, after which it stores the data in the transactional write buffer. Stores in DPTM do not request exclusive permissions at the time of the store; instead, a transaction stores the data in the write buffer immediately and does not stall. However, the transaction must obtain exclusive permissions for all cache lines in the write buffer before it can commit. DPTM, as a starting point in its design, requests exclusive permissions at commit time.

As an extension, DPTM can choose to request exclusive permissions immediately, and does not need to stall for the request. If it postpones the request until commit time, it cannot overlap the stalling for the request with other operations.

Either way, the time taken for the store instruction is equivalent to the time taken for a cache hit.

4.2.3 Conflict Management

When a transaction in the baseline HTM receives an invalidation request for a cache line that is a part of its read or write sets, it invokes a conflict resolution mechanism. If the mechanism decides to acknowledge the request, the receiver's cache line is invalidated, aborting its transaction.

Using DPTM, a transaction that receives an invalidation request acknowledges the request and does *not* abort, anticipating that the invalidation might be due to a coherence conflict caused by false sharing, a silent store, or that the sender of the invalidation is a *doomed transaction* [31], i.e., a transaction that will eventually abort. Therefore, the transaction schedules a request for the now invalidated cache line and continues execution.

When the transaction's request is finally served and it receives the current cache line data, the transaction validates that the read data on that cache line, as specified by the read mark bits, is unchanged. If it has changed, the transaction aborts, and the read mark bits are cleared.

4.2.4 Committing a Transaction

At commit time, the baseline HTM flushes its write buffer by writing all the values in its write buffer to memory. Because this HTM already has all its cache lines in their correct commit states, whereby the coherence states are associated with their transactional states, this is sufficient to complete the transaction.

In DPTM, when a transaction is ready to commit, parts of its read set might not be in a valid state, and parts of its write set might not be in an exclusive state. Therefore, it employs a two stage commit.

In the first stage, DPTM attempts to restore the baseline HTM association of ensuring that all its cache lines are in their correct commit states. It first issues shared cache requests to all stale lines in the cache that are part of the transaction's read set but not its write set. All the while, each incoming cache line is validated, aborting the

transaction if data that is part of its read set has changed. DPTM then issues exclusive cache requests to all cache lines that are part of the transaction’s write set but are not already in an exclusive state.

Once all the cache lines have been validated and are in their correct commit states, DPTM moves to the second commit phase, which is the same as the baseline HTM’s commit. When it finally commits, it clears all the read mark bits.

During the second commit phase, as in the baseline HTM, DPTM must keep all read lines in a valid state, and all written lines in an exclusive state. This results in the serialization of the commit phases in the presence of false conflicts — compared with the serialization of *whole* transactions in the baseline HTM.

The contention management policy during DPTM’s commit phases is different from the one during a transaction. Invalidation requests for cache lines that are part of a transaction’s read or write sets are denied. To prevent deadlock, the simplest policy is for the committing transaction to abort if a cache request it has sent was denied.

A more elaborate policy, which DPTM uses during its commit phase, is as follows: if a cache line invalidation request comes in during the first commit phase, the transaction acknowledges it only if the requester is also committing and has higher priority, otherwise it denies the request. Therefore, deadlock cannot occur if priority is unique. Invalidation requests during the second commit phase, as in the baseline HTM, are always denied; at that point the transaction is guaranteed to commit and there is no fear of deadlock.

4.3 Additions and Design Alternatives

This section discusses some of the DPTM design alternatives we have investigated.

4.3.1 Eager or Lazy Conflict Detection

DPTM detects transactional conflicts on changed values rather than on coherence conflicts. Because value changes are observable only once a transaction commits, DPTM’s conflict detection is *lazy* [31], unlike the baseline HTM which uses eager conflict detection. Because both eager and lazy conflict detection have been shown to be superior under different conditions [9, 11, 40, 50, 53], we introduce *SendSets*, an addition to DPTM which allows eager conflict detection while retaining the benefits of lazy conflict detection.

SendSets leverages the information in a transaction’s read and write sets, as represented by the read mark bits and the write buffer. When a transaction in DPTM using *SendSets* issues a cache request, it includes with the request this information as a bit map.

The transaction receiving the request determines whether there could be a transactional conflict by using the read and write set information in the bit map. If it determines that there may be a conflict, it decides, based on priority, whether to deny the request or acknowledge it.

Even if the transaction acknowledges the request it does not abort, anticipating that the request is due to a silent store, or that the sender of the invalidation request might be a doomed transaction. This allows the transaction to

eagerly detect conflicts and prioritize requests accordingly, and benefit from the laziness of value-based conflict detection.

4.3.2 Improving Prediction Accuracy by Sending Updates

Because DPTM speculates using stale cache line data, the accuracy of such predictions can increase if the values in the stale cache lines are kept up to date. When a processor requests exclusive permissions to write to a cache line, it could send the value it intends to write along with its request [27]. The receiver, upon receiving this request, would update the value of its now stale cache line.

Other alternatives might go even further; for example, processors could keep track of other processors they have invalidated, and broadcast any subsequent updates of the cache line to those processors [27].

4.4 DPTM Architecture

DPTM is compatible with existing best-effort HTM proposals that associate transactional conflicts with coherence conflicts. DPTM is also compatible with any cache coherence protocol with states denoting cache lines that are not present, present but invalid, valid, and exclusive². It does not matter whether it is a snooping or a directory based protocol.

DPTM does not require modifications of the coherence protocol typically used in HTMs, and adds little, processor-local, hardware. The additional hardware requirements are as follows.

DPTM requires additional bits per transactional cache line for the read mark bits, and assumes the ability of flash-clearing the read mark bits. Flash-clearing is desirable for performance, not required for correctness. If these bits cannot be flash-cleared, they could be cleared sequentially, at the cost of either stalling while the bits are cleared, or potential false conflicts in future transactions for the locations whose bits are still set.

DPTM also requires the ability to validate cache lines that are part of a transaction's read set against incoming data. The incoming data could be buffered in a miss status handling register (MSHR) while the validation takes place. DPTM also requires logic to compare the values being validated, but could be designed to leverage existing logic in a processor.

As for some of the design alternatives, *SendSets* requires the ability to include the read and write sets as a payload with cache requests. Sending updates also requires the ability to add the value being stored as an extra payload to exclusive cache requests and invalidations.

5 Evaluation

This section reports on our analysis of DPTM. We compare different design alternatives for DPTM and compare them with a best-effort HTM, focusing on false sharing in particular.

²We note that the extra state introduced for stale data is readily distinguishable in most cache coherence protocols, which conveniently group them together, but in fact can easily distinguish them.

5.1 Experiment Environments

The simulation framework we use is based on Virtutech Simics [34], and the University of Wisconsin GEMS 2.1 [36]. The simulator models processors that have best-effort HTM support using Sun’s ATMTP [41], itself a component of GEMS. The simulated system is a SPARC/Solaris Sun Fire server; the simulated environment parameters are given in Table 1.

We use ATMTP to simulate the baseline HTM described. To simulate DPTM, we have extended ATMTP without modifying its cache coherence protocol. ATMTP models an eager conflict detection and lazy version management Rock-like best-effort HTM. ATMTP uses the L1 cache as its transactional cache, which limits transactional reads and writes by its size and associativity. ATMTP keeps transactional writes in a write buffer until the transaction commits, which also limits transactional writes. Because ATMTP models a best-effort HTM, we use a single global lock as a fallback mechanism when unable to complete a transaction in hardware.

5.2 Benchmarks

For our evaluation, we use STAMP [40], SPLASH-2 [56], micro [22, 35], and our own *SharingPatterns* benchmarks.

We test the STAMP benchmarks (*Table 2–top*), using the parameters suggested by their creators for both *low* and *high* contention tests [40]. We have excluded `bayes` and `yada` from our tests, because their transactions are too large to complete successfully in hardware using ATMTP.

STAMP was written by transactional memory experts; its benchmarks exhibit no false sharing inside transactions. However, code not written by experts is unlikely to be optimized for false sharing. We use these benchmarks, anyway, to investigate how DPTM behaves in the absence of false sharing.

The SPLASH-2 suite was not originally meant for evaluating TM. Others have reported that `raytrace` is particularly susceptible to false sharing in HTM [42]; therefore, we thought it would be interesting to include it in

Item	Model
Processor	in-order, single-issue, single-threaded, CMP
L1 cache	private, transactional, 128 kB, 4-way split, 1 cycle latency
L2 cache	shared, 2 MB, 8-way split, 20 cycle latency
Cache line size	64 bytes
Memory	8 GB, 450 cycle latency
Transactional write buffer	32 entries, 256 for <code>labyrinth</code>
Conflict resolution priority	timestamp [9]
Function calls in transactions	allowed [41]

Table 1: Simulated machine configuration

Benchmark	Tx Length	R/W Set	Contention
<code>genome</code>	medium	medium	low
<code>intruder</code>	short	medium	high
<code>kmeans</code>	short	small	low
<code>labyrinth</code>	long	large	high
<code>ssca2</code>	short	small	low
<code>vacation</code>	medium	medium	low/medium
<code>raytrace</code>	short	small	medium
<code>hash</code>	very short	small	low
<code>list</code>	medium	medium	high
<code>redblack</code>	short	medium	low

Table 2: Qualitative summary, relative to STAMP, of the benchmarks’ runtime transactional characteristics: length of transactions (number of instructions), size of the read and write sets, and amount of contention. cf. [40]

our evaluation (*Table 2–middle*). We use a small image, *teapot*, as an input.

The micro-benchmarks are ones we ported from the Java-based DSTM [22] to C (*Table 2–bottom*). They are concurrent sets implemented with: a chained hash table, a single sorted linked list, and a red-black tree. In these benchmarks, each thread randomly inserts, deletes, or looks up a value in the range of 0–255. The *low* contention distribution of operations is 1:1:1 (insert:delete:lookup), and the *high* one is 1:1:0.

We have also created a group of benchmarks, *SharingPatterns*, to cover a range of sharing patterns. These benchmarks are not meant to represent realistic workloads, but to exaggerate these patterns to better observe the behavior of DPTM in the presence of false and true sharing. The patterns *SharingPatterns* exhibits are the following.

Sharing followed by no sharing: all transactions start by incrementing a value residing on the same cache line, followed by incrementing 19 different lines.

No sharing followed by sharing: all transactions start by incrementing values residing on 19 different cache lines, followed by incrementing the same line.

Write sharing: all transactions increment values residing on the same 20 cache lines.

The sharing part of each of the three patterns above is tested with false sharing, where processors increment different values on the same cache line; and with true sharing, where processors increment the same value on the same cache line.

5.3 Experiments and Results

The outcome of the experiments, running 16 user threads on 16 processors, is shown in Figures 2, 3, and 4. We measure the time spent in transactional workloads, and present the results of 10 runs with pseudo-random perturbations [3], plotting the standard deviation as error bars.

5.3.1 The effect of different sharing patterns

Because we designed DPTM with the goal of mitigating the effects of false conflicts, we start by looking at different patterns of false and true sharing. We expect DPTM to improve performance over the baseline HTM in the presence of false sharing, and that it would be comparable in the case of true sharing. We also expect that as the amount of false sharing increases, the gap between DPTM’s performance and the baseline HTM would also increase.

For these experiments (*Figure 2*), we use the *SharingPatterns* benchmarks, which present an exaggerated form of different sharing patterns. We run the benchmarks using the baseline HTM, and DPTM with conflict detection granularity levels, in bytes, of 4 (one word), 8, 16, 32, and 64 (one cache line); in the presence of false sharing, we expect performance to improve the finer the granularity.

In the presence of false sharing, DPTM with granularity of 4 to 32 bytes always performs significantly better

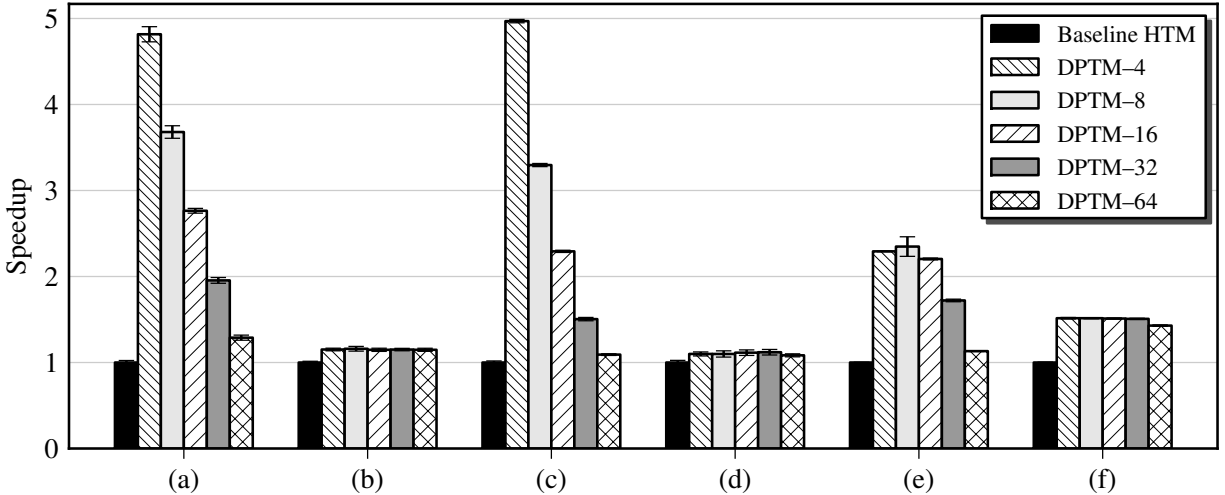


Figure 2: The speedup of the SharingPatterns benchmarks running at granularities of 4 to 64 bytes, relative to the baseline HTM. (a/b) false/true sharing followed by no sharing (c/d) no sharing followed by false/true sharing (e/f) false/true write sharing

than the baseline HTM (Figure 2: a, c, e). DPTM also performs at least as good, if not better, for true sharing (b, d, f). The improvement in the case of true sharing is an anomaly due to DPTM’s lazy conflict detection, where its transactions are not aborted, as in the baseline HTM, when a cache line is invalidated by a doomed transaction.

We note that DPTM’s gains are comparable whether false sharing occurs at the beginning of a transaction (a), or at the end (c). Because contention in these benchmarks is high, false sharing adversely affects performance in the baseline HTM regardless of where it occurs inside a transaction. This also shows that even one instance of false sharing can be detrimental to performance.

As for the effect of different granularity levels, we find that for the false sharing patterns that do not perform many shared stores (a, c), the improvement gained by finer granularity is more pronounced than where shared stores dominate (e). Moreover, where shared stores — and false conflicts — dominate, DPTM is less effective in mitigating their effects. This is due to DPTM’s serialization of commit phases in the presence of false conflicts, which reduces the gains in such workloads.

5.3.2 The effects of restructuring

We have seen that DPTM has potential for improving performance in the presence of false sharing. In the next set of experiments (Figure 3), we investigate how DPTM performs using a more diverse set of benchmarks.

We investigate the amount of false sharing present in these benchmarks. Where it is present, we try to mitigate it by padding, and where it is not, we try to instigate it by removing existing padding. This allows us to study the efficacy of using padding, particularly when compared with using DPTM, to mitigate false sharing. We expect DPTM to be at least comparable to, if not better than, padding in mitigating false sharing, because padding could have adverse effects on locality.

We note that the STAMP suite was developed by TM experts for evaluating TM proposals, so we expect it

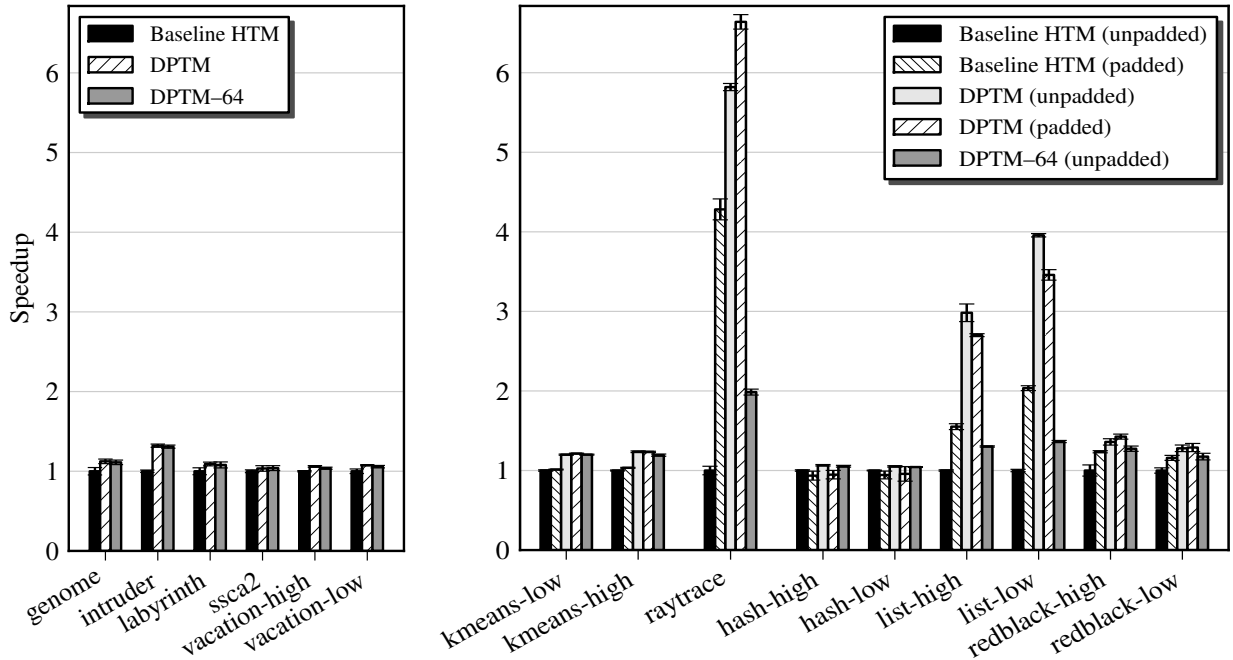


Figure 3: The speedup of running the benchmarks, padded and unpadded where applicable, relative to the baseline HTM. The benchmarks on the left do not exhibit false sharing inside transactions.

would be optimized to mitigate false sharing. By analyzing its code, we observed that some of its structures are padded and aligned to cache line boundaries. The only padded data structures used inside transactions, as far as we could tell, are in `kmeans`. We created a modified version of `kmeans` with this padding removed. We still use the rest of the STAMP benchmarks in our evaluation, even though we do not expect any performance gains, to study the effect our changes might have in the absence of false conflicts.

Even though the SPLASH-2 suite was written for evaluating multiprocessors, `raytrace` is known to suffer from false sharing in the context of HTM [42]. To compare against mitigating the effects of false sharing by restructuring, we created a modified version of `raytrace` in the manner Moore et al. [42] describe.

The micro benchmarks were originally written in Java, and therefore are not padded. In porting them, we created a modified version where each object is padded and aligned to the cache line boundary.

We compare the padded with the unpadded versions (where applicable), running on the baseline HTM, DPTM with conflict detection granularity of 4 bytes (one word — *DPTM*), and 64 bytes (one cache line — *DPTM-64*). We use *DPTM-64* to isolate the effects *DPTM* has on false sharing from other effects, such as *DPTM*'s lazy conflict detection, because *DPTM-64* detects conflicts at the granularity level of a whole cache line.

To better understand the impact of false sharing on the benchmarks' transactions, Table 3 presents an analysis of the impact false sharing has on the abort rate of transactions in *DPTM-64*. We present this analysis for *DPTM-64* because it uses the same mechanisms as *DPTM-4*, yet detects conflict at the granularity level of a whole cache line. Therefore, *DPTM-64* isolates the effects of false sharing in transactions from the effects of all the other

changes DPTM makes to the baseline.

The results of evaluating DPTM using STAMP show that DPTM is consistently faster than the baseline HTM, with the exception of `ssca2`, where they are comparable. This improvement is not due to the mitigation of false sharing or silent stores, but to DPTM’s lazy conflict detection.

For `kmeans`, padding improves performance only slightly. False sharing in `kmeans` does not occur often, because its transactions are small with low contention, and its transactional objects that suffer from false sharing are only slightly bigger than a cache line.

As for the SPLASH-2 benchmark `raytrace`, false sharing has a significant impact. Padding mitigates much of its effects. Furthermore, the cache miss statistics show that padding does not adversely affect locality in this case.

DPTM significantly speeds up `raytrace` compared with the baseline HTM when running either the padded or the unpadded version. Most of this improvement is due to mitigating the effects of false sharing.

DPTM is faster when running the padded version compared with the unpadded version of `raytrace`. This shows that DPTM was able to mitigate most of the effects of false sharing, but not eliminate them. DPTM serializes the commit phases of transactions with false conflicts, whereas the padded version has no false conflicts that cause commit phases to serialize.

For the micro benchmarks running on the baseline HTM, the padded versions of `list` and `redblack` are significantly faster, because padding mitigates false sharing. As for `hash`, there is no significant difference because contention is low and its transactions are very short.

DPTM running the unpadded micro benchmarks is faster than the baseline HTM running either the unpadded or the padded versions. Interestingly in `list`, DPTM performs better using the unpadded version whereas the opposite is true for the baseline HTM; DPTM was able to mitigate the effects of false sharing as well as take advantage of the locality afforded by using the unpadded version.

An observation regarding value prediction, we have until now presented the results using a conservative DPTM, which speculates only on the values already part of a transaction’s read set. Most of the gains so far, even in the presence of false sharing, are due to the decoupling of transactional from coherence conflicts; value prediction

Benchmark	False Sharing Aborts	True Conflict Aborts
genome	16%	18%
intruder	7%	104%
labyrinth	16%	300%
ssca2	1%	3%
vacation-high	19%	415%
vacation-low	15%	415%
kmeans-high	5%	18%
kmeans-low	5%	8%
raytrace	67%	28%
hash-high	2%	0%
hash-low	1%	0%
list-high	313%	251%
list-low	269%	156%
redblack-high	17%	9%
redblack-low	12%	6%

Table 3: DPTM-64’s abort rates, relative to the number of committed transactions, broken down by cause. The first column presents the percentage of aborting transactions that abort because of false sharing. The second column presents the percentage of transaction that abort because of true conflicts.

accounts for less than 5% of the *additional* gains presented.

We also experimented with an aggressive approach, which *always* speculates on the values in a stale cache line, with the results (not shown) consistently slower. This is not surprising, because mis-prediction has the high cost of aborting the whole transaction. Others have also observed that for data speculation to be effective, it should be throttled to avoid the high cost of mis-prediction [52].

5.3.3 The effect of design alternatives

In Section 4, we have discussed some of DPTM’s design alternatives. We evaluate the alternatives (*Figure 4*) against the basic DPTM with conflict detection granularity of 4 bytes.

First, we evaluate issuing exclusive requests immediately on stores (*GetX*), rather than waiting until commit time. Issuing the requests immediately can hide the latency for the request. But if contention is high, the transaction may lose exclusive permissions and need to reissue the request later.

We find no significant differences except in `intruder`, `kmeans`, `raytrace`, and `redblack-high`, where *GetX* is noticeably faster. These benchmarks have short transactions, so the window for losing exclusive permissions is small. On the other hand, `hash` is not benefitting much from *GetX*, because its transactions are very short, so there is little time to hide any request latency.

Because *GetX* performs as well as, if not better than, the basic DPTM, the remaining experiments also issue exclusive requests immediately on stores, and are compared against *GetX*.

Next, we evaluate *SendSets*, which makes DPTM more eager by taking advantage of a cache line’s read and write set information in detecting transactional conflicts. As mentioned in Section 4.3.1, eager and lazy conflict detection are superior under different conditions; therefore, we expect *SendSets*’s performance to be in line with more eager conflict detection schemes.

For the benchmarks we use, *SendSets* is generally comparable to *GetX*, except for `intruder`, `raytrace`, and `list-high`. There is more improvement in `list-high`, a long benchmark with high contention, because eager conflict detection aids higher priority transactions, i.e., older transactions in this case. On the other hand, `intruder` and `raytrace` are slower, because requests are often denied by doomed transactions, wasting work.

Finally, we investigate ways to improve the accuracy of value prediction in DPTM. We first model the instantaneous and free broadcasting of all cache line updates to all processors that have the cache line in a stale state (*Seer*). We use this *unrealistic* approach to evaluate the effects of such a broadcasting mechanism regardless of its cost, in order to give us intuition regarding the benefits of sending updates in general. We then compare *Seer* against the more realistic approach of sending updates only with invalidation requests (*SendUpdates*).

When using either *Seer* or *SendUpdates*, DPTM *always* speculates on stale cache line data, unlike the basic DPTM. Out of all our experiments, *Seer* performs the best.

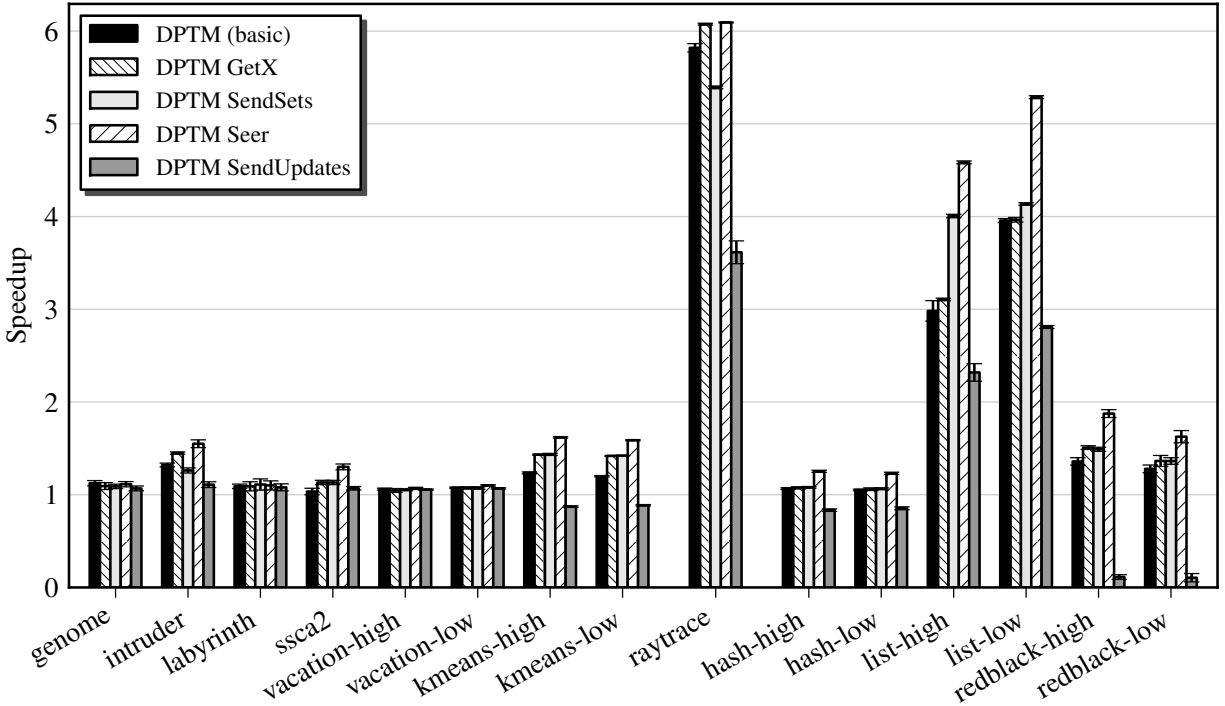


Figure 4: The speedup of design alternatives for DPTM relative to the baseline HTM (not shown)

The gains from value prediction when using *Seer* are substantially higher than in the basic DPTM. For example, in *ssca2*, value prediction now accounts for 50% of the additional gains over the baseline HTM, and 40% in *redblack*, and 30% in *kmeans* and *linklist*. This unrealistic approach demonstrates that broadcasting updates has potential for improving performance gains from value prediction.

On the other hand, *SendUpdates* overall performs poorly; transactions frequently abort because of mis-speculation, especially in *redblack*. This implies that while the sending of updates seems promising, *SendUpdates* is not sufficient to capture this potential.

Finally, to better appreciate the effect the different aspects the design of DPTM has on performance, Figure 5 presents a breakdown of the different speedup components in *Seer*. We present the breakdown for *Seer* because its component parameters perform the best across all workloads in these experiments. Therefore, a breakdown of its components is well suited to examine the relative benefits of the different design options we investigate.

6 Related Work

Other HTMs are also capable of fine-grained conflict detection, e.g., *TCC* [19, 39], *Bulk* [11], and *RETCON* [7]. *TCC* associates fine-grained read bits for conflict detection with each cache line; however, *TCC* proposes an unconventional approach for memory consistency and cache coherence, where transactions are the basic unit of parallel work. *Bulk* hashes a transaction’s access information, and uses this hash for conflict detection. *RETCON*, which is concurrent with this work, symbolically tracks modifications and constraints that a transaction applies to variables, and uses value-based conflict detection. In contrast to these proposals, we directly address the problem

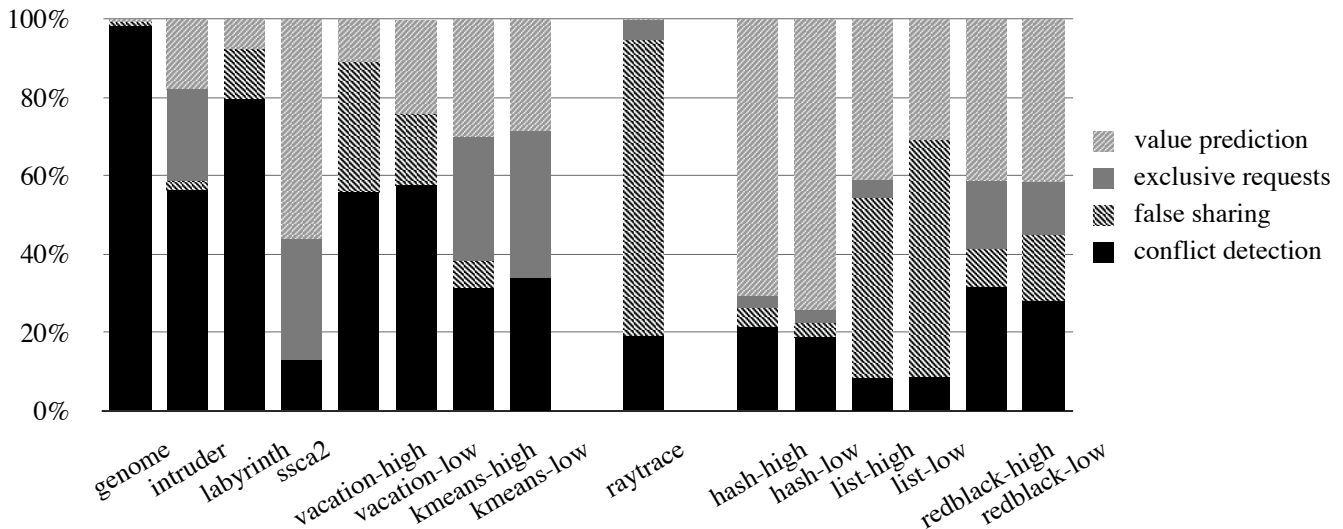


Figure 5: A breakdown of the speedup components of Seer. The conflict detection component is due to the difference in DPTM’s conflict detection from the baseline hardware. The false sharing component is due to the transactions not aborting because of false sharing. The exclusive requests component is due to issuing exclusive requests immediately on stores, rather than waiting until commit time. The value prediction component is due to the reduction in load latency when speculating on data values.

of false sharing using a best-effort HTM with a conventional coherence protocol. We also propose using value prediction to reduce latencies incurred with false sharing, by speculating on the values of stale cache line data.

Concurrently with and independently of our work, Pant and Byrd [44] proposed using value prediction in HTM, in a manner very different from ours both in design and in purpose. Their proposal does not address false sharing in transactions, but uses value prediction to reduce load latencies by predicting future updates. It also requires extensive modifications to the underlying hardware: the value predictor is located at the memory level, near the memory or directory controller; requires the directory to be able to observe all stores, which could be a bottleneck; limits the number of transactions concurrently involved in value prediction; requires additional hardware modifications to the directory to keep prediction history and make further predictions; and requires a *nacking* coherence protocol [42], as well as further changes to that protocol.

Outside the context of HTM, Huh et al. decouple the use of a cache line’s data from obtaining permissions for that line to mitigate the effects of false sharing [27]. They propose speculating on the values of stale cache lines, as well as mechanisms of sending write updates and forwarding modified data. In their work, latency tolerance for the coherence permissions was low, effectively limited to the parallelism in the existing reorder buffer. HTM naturally provides a larger scope, and so is better suited for coherence decoupling. Moreover, the additions required for our proposal can be used to mitigate the effects of false sharing *outside* transactions, in the same manner Huh et al. propose.

Value prediction has also been explored in the context of TLS in works by Knight [30], Akkary and Driscoll [2], Martin et al. [37], Cintra and Torrellas [13], and Steffan et al. [52], among others.

EazyHTM separates conflict detection from conflict resolution in an HTM, allowing it to detect conflicts eagerly, but act on them lazily [53]. Our work on *SendSets* serves a similar purpose using a different technique.

On the software side, Torrellas et al. [54] propose some solutions to the false sharing problem using compiler modifications that optimize the layout of shared data in cache lines to mitigate its effects. Olszewski et al. [43] propose *JudoSTM*, a software TM that uses value-based conflict detection, and is capable of improving performance in the presence of silent stores.

7 Concluding Remarks

We have demonstrated how data speculation in HTM, by example of DPTM, has the potential for improving performance, particularly in the presence of false conflicts. Benchmarks that exhibit false sharing show dramatic gains, whereas ones that do not exhibit false conflicts are not harmed by the alternative designs — and some even benefit from having lazy conflict detection.

We demonstrated how DPTM could mitigate the effects of false conflicts inside transactions. The modifications DPTM needs could also be applied to mitigate the effects of false sharing *outside* transactions, in the same manner proposed by Huh et al. [27], improving performance over a wider range of workloads.

Although DPTM can significantly mitigate the effects of false sharing, this mitigation is not perfect, because it serializes the commit phases of transactions with false conflicts.

DPTM can also improve performance in the presence of silent stores. Others have noted that silent stores are common in certain workloads [13, 52]; however, they were not common in the benchmarks we used. We are interested in evaluating DPTM on such workloads in the future.

References

- [1] A. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. *PLDI*, 2006.
- [2] H. Akkary and M. Driscoll. A dynamic multithreading processor. *MICRO*, 1998.
- [3] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. *HPCA*, 2003.
- [4] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. *HPCA*, 2005.
- [5] C. Ananian and M. Rinard. Efficient object-based software transactions. *SCOOOL*, 2005.
- [6] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. 2008.
- [7] C. Blundell, A. Raghavan, and M. Martin. RETCON: Transactional repair without replay. *ISCA*, 2010.
- [8] J. Bobba, N. Goyal, M. Hill, M. Swift, and D. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. *ISCA*, 2008.
- [9] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood. Performance pathologies in hardware transactional memory. *ISCA*, 2007.

- [10] W. Bolosky and M. Scott. False sharing and its effect on shared memory. 1993.
- [11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *ISCA*, 2006.
- [12] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *ASPLOS*, 2006.
- [13] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. *HPCA*, 2002.
- [14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *ASPLOS*, 2006.
- [15] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *ASPLOS*, 2009.
- [16] D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 2005.
- [17] J. Goodman and P. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. *ISCA*, 1988.
- [18] D. Grossman. The transactional memory / garbage collection analogy. *OOPSLA*, 2007.
- [19] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *ISCA*, 2004.
- [20] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. *DISC*, 2002.
- [21] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. 2006.
- [22] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. *PODC*, 2003.
- [23] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. *ISCA*, 1993.
- [24] M. Herlihy and J. Moss. System for achieving atomic non-sequential multi-word operations in shared memory. *US Patent 5,428,761*, 1995.
- [25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [26] O. Hofmann, C. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. 2009.
- [27] J. Huh, J. Chang, D. Burger, and G. Sohi. Coherence decoupling: making use of incoherence. *ASPLOS*, 2004.
- [28] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *PPoPP*, 1995.
- [29] M. Kadiyala and L. Bhuyan. A dynamic cache sub-block design to reduce false sharing. *ICCD*, 1995.
- [30] T. Knight. An architecture for mostly functional languages. *LFP*, 1986.
- [31] J. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2007.
- [32] K. Lepak and M. Lipasti. Silent stores for free. *MIRCO*, 2000.
- [33] K. Lepak and M. Lipasti. Temporally silent stores. *ASPLOS*, 2002.
- [34] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 2002.
- [35] V. Marathe, M. Spear, C. Heriot, and A. Acharya. Lowering the overhead of nonblocking software transactional memory. *TRANSACT*, 2006.
- [36] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood.

Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. 2005.

- [37] M. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. *MICRO*, 2001.
- [38] J. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. *ASPLOS*, 2002.
- [39] A. McDonald, J. Chung, H. Chafi, C. Minh, B. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. *PACT*, 2005.
- [40] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. *IISWC*, 2008.
- [41] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. *TRANSACT*, 2008.
- [42] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. *HPCA*, 2006.
- [43] M. Olszewski, J. Cutler, and J. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. *PACT*, 2007.
- [44] S. Pant and G. Byrd. Extending concurrency of transactional memory programs by using value prediction. *CF*, 2009.
- [45] R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. *MICRO*, 2001.
- [46] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. *ASPLOS*, 2002.
- [47] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *ISCA*, 2005.
- [48] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. *ISCA*, 2007.
- [49] W. Scherer, D. Lea, and M. Scott. A scalable elimination-based exchange channel. *SCOOOL*, 2005.
- [50] A. Shriraman, S. Dwarkadas, and M. Scott. Flexible decoupled transactional memory support. *ISCA*, 2008.
- [51] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. *ISCA*, 1995.
- [52] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving value communication for thread-level speculation. *HPCA*, 2002.
- [53] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. *MICRO*, 2009.
- [54] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 1994.
- [55] E. Vallejo, T. Harris, A. Cristal, O. Unsal, and M. Valero. Hybrid transactional memory to accelerate safe lock-based transactions. *TRANSACT*, 2008.
- [56] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *ISCA*, 1995.
- [57] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. *HPCA*, 2007.
- [58] R. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H.-H. Lee. Kicking the tires of software transactional memory: why the going gets tough. *SPAA*, 2008.