Transactional Value Prediction

Fuad Tabba   Andrew W. Hay   James R. Goodman

The University of Auckland
{fuad,andrewh,goodman}@cs.auckland.ac.nz

Abstract

This workshop paper explores some ideas for value prediction and data speculation in hardware transactional memory. We present these ideas in the context of false sharing, at the cache line level, within hardware transactions.

We distinguish between coherence conflicts, which may result from false sharing, from true data conflicts, which we call transactional conflicts. We build on some of the ideas of Huh et al. [1] to speculate in the presence of coherence conflicts, assuming no true data conflicts. We then validate data before committing. This dual speculation avoids aborting and restarting many transactions that conflict through false sharing.

We show how these ideas, which we call Transactional Value Prediction, can be applied to a conventional best-effort hardware transactional memory. Our preliminary model, β-TVP, does not alter the underlying cache coherence protocol beyond what is already present in hardware transactional memory. β-TVP requires only minor, processor-local modifications to a conventional best-effort hardware transactional memory.

Simple benchmarks show that β-TVP can dramatically increase throughput in the presence of false sharing, while incurring little overhead in its absence.

1. Introduction

Parallel programming is fast becoming a reality. Most processor manufactures today are producing chips with multiple cores [2–6]. However, software engineering tools have not kept up in making it easier for programmers to take full advantage of these chips. It is difficult to write correct parallel programs for reasons such as deadlock, livelock, starvation, and data races [7, 8]. It is also difficult to write efficient parallel programs for reasons such as the restrictions imposed by Amdahl's law [9, 10], convoying [8], and false sharing [6, 11–13].

Transactional Memory [14], a promising new programming model, attempts to alleviate some of these concerns. So do other mechanisms, such as Speculative Lock Elision (SLE) [15] and Transactional Lock Removal (TLR) [16].

In this paper, we investigate different ideas that could be used to improve performance within hardware transactional memory, by taking advantage of value prediction and data speculation. We explore the ideas of Transactional Value Prediction, in the context of mitigating the effects of false sharing in hardware transactions.

The inspiration for this work is that when inside a transaction, the processor is already in speculative execution mode. Therefore, it can speculate on data in ways that might be infeasible outside transactions. Such speculation would be correct if the values speculated on are not going to be produced by other transactions. If we manage to harness this observation, we could use it to improve performance by reducing memory latencies and conflicts between transactions, among other things.

Value prediction, in the context of hardware transactional memory, can be applied as long as we ensure the assumed values are correct before committing. Only then should a transaction be able to commit successfully.

One particular aspect of data speculation in general, proposed by Huh. et al. [1], is the speculation on load values, typically by conjecture from stale values in the cache, as a solution to the problem of false sharing.

The problem of false sharing, specifically at the cache line level, is not an easy problem to solve. It could degrade performance significantly [12], possibly causing transactions to completely serialize or even worse [17, 18]. False sharing has often been discovered by experts on transactional memory and parallel programming in their work [17–28]. To the best of our knowledge, no existing hardware transactional memory handles the issue of false sharing at the cache line level.

False sharing can be mitigated by careful data layout, for example, by aligning the data to cache line boundaries and padding it to fill the whole cache line. This approach increases internal fragmentation and decreases the effective cache size, partially canceling the performance gains achieved. Moreover, transactions might include code from external libraries not optimized to handle false sharing, which programmers cannot easily modify.

In our opinion, if transactional memory is to truly make it easier to write parallel programs, it must avoid the worst effects of false sharing.

We believe the techniques of Transactional Value Prediction, which we introduce by presenting an initial model,

β-TVP, could help improve performance and mitigate the effects of false sharing within hardware transactions. β-TVP is a work in progress that addresses the problem of false sharing inside transactions by applying methods built on some of the ideas in the work of Huh et al. [1].

By conjecturing about stale cache line values, β-TVP could mitigate the effects of false sharing and improve performance on several fronts. First, β-TVP reduces serialization of transactions falsely sharing the same cache lines. β-TVP allows transactions to run using stale values without stalling or aborting, while simultaneously issuing a request for the appropriate cache line permissions and data.

Second, β-TVP detects transactional conflicts based on changes in the values read from the cache, rather than relying solely on coherence conflicts. This enables β-TVP to detect transactional conflicts at any desired granularity level rather than the level of a whole cache line. Moreover, by detecting conflicts this way, β-TVP improves performance in the presence of silent stores [29] and temporally silent stores [30].

A silent store is when the same value is written to the cache line, resulting in the line being acquired exclusively without actually changing its value. Such an occurrence would stall or abort hardware transactions in different implementations, such as LogTM-SE [22] and ATMTP [31], but would not cause a β-TVP transaction to abort. Note that accommodating silent and temporally silent stores makes it possible for some *truly conflicting transactions* to execute concurrently.

Finally, β-TVP only needs to acquire cache lines in their correct state, whether it is a shared or an exclusive one, just prior to commit time. This reduces the window where conflicts between transactions might occur, potentially increasing concurrency.

The modifications required to implement β-TVP are limited to the local processor; no changes to the underlying cache coherence protocol are needed beyond what is already present in hardware transactional memory. The concepts presented here could equally be applied to different hardware transactional memory implementations and to lock-based mechanisms such as SLE and TLR.

Transactional Value Prediction and β-TVP are still a work in progress. We do not believe that β-TVP is the only way of taking advantage of these ideas. This is, however, the first step in our investigation.

This paper is organized as follows: section 2 describes the false sharing problem, explaining why solving this problem could improve performance and also make it easier to program. In section 3, we propose our preliminary implementation of β-TVP, with details of how it could fit in with existing hardware transactional memory. Section 4 presents our preliminary evaluation of β-TVP. Section 5 briefly describes some of the related work. Finally, we discuss some ideas for future work and end with concluding remarks.

2. The False Sharing Problem

The problem of false sharing, and its impact on performance is a well known problem [6, 11–13]. False sharing occurs when a cache line contains unique data objects being referenced by different processors. Since the cache line is the unit of granularity for coherence, these nonconflicting accesses nevertheless force serialization of access.

False sharing is not an easy problem to solve. Most solutions we have encountered in existing literature and from our own experiences are oriented towards the restructuring and padding of data, so that nonconflicting accesses to separate data objects are also nonconflicting as far as the coherence protocol is concerned.

False sharing is a tricky problem because programmers often include external library functions in their code. Even if the programmers' own code does not suffer from false sharing, by including code that does, the whole program could suffer. Often, accessing and modifying such external code is difficult or infeasible.

Huh et al. [1] observed that on a read cache miss, a processor requests, stalls, and eventually obtains both the needed permissions and data in one go. However, the processor may already have the correct data in one of its caches but without the required permissions, i.e., a stale cache line. By separating the request for the needed permissions from the use of the data, the processor does not need to stall, but can speculate using the stale data until the permissions arrive.

Speculating on stale data might, of course, be counterproductive at times. Whether such speculation improves performance or not depends on the benefit of correct speculation, the cost of recovery, and the ratio of correct to incorrect speculation [1]. Huh et al. demonstrate that their method greatly reduces performance losses due to false sharing.

Huh et al. also recognized that writing to shared data can also be broken into steps, in which the write can be performed first but not committed until permissions are acquired. This could reduce the need to stall on writes.

Huh et al.'s proposal requires additional support beyond typical microarchitecture speculation hardware [1]. However, this support exists, or would exist, in a processor that implements hardware transactional memory, such as Sun's upcoming Rock processor [31], and Azul's optimistic concurrency processors [32].

False Sharing in Transactions

False sharing is a bigger problem when it occurs in conjunction with hardware transactional memory [18]. Many hardware transactional memory implementations detect transactional conflicts based on cache coherence conflicts. Cache line permissions are usually needed for the duration of a transaction. Therefore, false sharing causes a transaction to stall while it waits for the cache line to come in. Even worse, since hardware does not distinguish between true and false
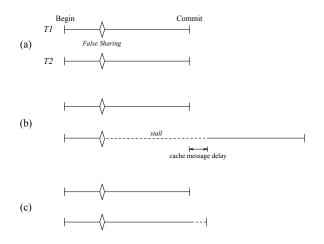
Figure 1. A demonstration of the false sharing problem with two concurrent transactions. (a) ideal case (b) hardware transactional memory (c) a solution that mitigates the problem of false sharing.

sharing, false conflicts may cause a transaction to abort or serialize as it waits for other transactions to complete [17, 18].

The example in Figure 1 shows the timeline of two concurrent transactions. These transactions access different locations within the same cache line at one point during their execution, i.e., false sharing. The transactions in this example at no point have any true conflicts.

Ideally, these transactions should be able to run completely in parallel as shown in (a). However, hardware transactional memory implementations, such as LogTM-SE, infer a transactional conflict whenever a coherence conflict is detected. Thus, such implementations do not distinguish between true and false sharing, thereby stalling transactions as shown in (b), or even aborting them.

We expect a solution to the problem of false sharing to result in an execution timeline similar to the one shown in (c). Such a solution would likely not eliminate all the delays caused by false sharing, since 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.

Since hardware transactional memory is particularly susceptible to false sharing at the cache line level, its cascading effects potentially have a much greater impact on throughput than in non-transactional applications [17, 18], including software transactional memory.

3. Transactional Value Prediction

3.1 Overview

This section introduces some of the ideas of Transactional Value Prediction by the example of β-TVP. β-TVP attempts to alleviate, and in some cases eliminate, the effects of false sharing as follows.

The first aspect is speculating on a load using stale cache line data. If a cache line is present but is stale, β-TVP allows transactions to speculate based on the stale data, validating the read data later. β-TVP assumes that the cache line was invalidated due to false sharing rather than a true conflict. If this assumption is correct, it is indeed false sharing and β-TVP could eliminate most of the effects of false sharing. If it is true sharing, then the transaction aborts. However, if β-TVP had not speculated the transaction might have stalled or aborted anyway.

For the second aspect, instead of detecting transactional conflicts using the cache coherence protocol, β-TVP detects such conflicts based on value changes. A cache coherence conflict triggers validation which will eventually compare the data read inside the transaction with the new data. This reduces the effects of false sharing since in β-TVP, transactional conflicts are restricted to changes in the data used inside a transaction rather than coherence conflicts over whole cache lines.

Another aspect is that β-TVP, conservatively perhaps, does not exclusively request cache lines that are part of a transaction's write set until commit time. This reduces the window in which conflicts might occur. This is in the spirit of Huh el al.'s suggestion that writing to shared data can be broken into steps in which a write can be performed, with the write's effects being invisible to other processors until the end of the transaction [1].

It might seem that β-TVP uses lazy conflict detection. However, lazy conflict detection is defined by Bobba et al. [33] to be that conflicts are detected when the first of two or more conflicting transactions commit. A more accurate way of describing β-TVP would be to use the taxonomy of Larus and Rajwar [8]: β-TVP detects conflicts on *validation* rather than on *open* (eager in [33]) or *commit* (lazy in [33]).

In β-TVP, a coherence conflict is *not* interpreted as a transactional conflict. Instead, a coherence conflict triggers a validation request. This validation request is served later and may or may not trigger a transactional conflict, depending on which parts of the cache line have changed. β-TVP does *not* wait until the end of a transaction to resolve such conflicts, and all conflicts must be resolved before a transaction can commit.

3.2 Detailed Description

We now describe β-TVP, an illustration that uses some of the ideas of Transactional Value Prediction, using Sun's ATMTP [31] as an example hardware transactional memory framework. We emphasize that the ideas presented in this work could apply to many speculative execution schemes. ATMTP is used mainly due to its relationship to the upcoming Rock processor.

ATMTP is a best-effort hardware transactional memory that uses eager conflict detection and lazy version management, as writes are stored in a write buffer until the transaction commits.

We note that, for the time being at least, we have refrained from adding features to β-TVP that are not relevant to Transactional Value Prediction. Such changes could disproportionately benefit both ATMTP and β-TVP.

Loading a Value

A transaction in β-TVP begins as a normal ATMTP one. If a transaction attempts to load a memory location and that location is not present in the L1 cache (rather than stale), then the protocol proceeds as normal. Moreover, if the cache line is present and is in a valid state[1], then the load also proceeds normally.

If the cache line is stale[2], β-TVP serves the load by using the stale data while simultaneously issuing a cache request for the data. The load proceeds as if it were a cache hit, and does not wait for the response of the cache request before continuing execution.

β-TVP adds additional read mark bits to all cache lines to indicate which parts of the line have been read. Each mark bit monitors reads from a subset of the cache line, i.e., the mark bit is set when its associated subset is read inside a transaction. These bits are used during validation where only the parts of the stale cache line with their mark bits set are validated.

The number of mark bits added per cache line determines the granularity level of β-TVP's conflict detection. In other words, the greater the number of mark bits the finer the granularity, and the more cases of false sharing that can be detected. This could conceivably go down to the individual bit level.

For example, assuming a 64 byte cache line and a conflict detection granularity level of 4 bytes, β-TVP requires an additional 16 mark bits in each cache line.

When a processor receives a response to a cache request, the data in the cache whose mark bits are set is validated against the new data in the response. If the validation succeeds then the transaction proceeds as normal. However, if the validation fails, the transaction aborts and the mark bits are cleared. In all cases, the old cache line data is replaced with the new data in the response.

Storing a Value

When a transaction performs a store, ATMTP would normally request exclusive access to the cache line and stall while it obtains the correct permissions, after which it would write the data to the write buffer. β-TVP stores, regardless of the current state of the line, do not generate any cache requests, and are redirected immediately to the write buffer. Thus the stall time taken is equivalent to a cache hit.

Conflict Management

As for handling conflicts in the cache coherence protocol: when an ATMTP transaction ($T1$) requests a cache line which is part of another transaction's ($T2$) read or write sets, whether the request is for shared or exclusive access, then the requester, $T1$, always wins, aborting the transaction $T2$.[3]

Using β-TVP, whenever a cache line in a transaction is invalidated, the contention management policy does *not* abort the transaction as it would in ATMTP, nor does it deny the invalidation request. Instead, β-TVP would invalidate the cache line, re-request it, and continue execution without stalling for the request. When the request for the invalidated cache line completes, the validation procedure mentioned earlier is triggered.

Cache Line Evictions

In ATMTP and β-TVP, if a cache line that is part of a transaction's read set is evicted from the L1 cache, the transaction has to abort. β-TVP must abort because it cannot keep track of the original value that it has read; therefore, it cannot validate it later. That said, L2 evictions, unlike in ATMTP, do *not* cause a β-TVP transaction to abort. Instead, an invalidation of the cache line is triggered, the line is re-requested and then validated.

Committing a Transaction

At commit time, ATMTP flushes its write buffer by issuing store requests of all the values in its write buffer. Because ATMTP already has all its lines in their correct commit states[4], this is sufficient to complete the transaction.

In β-TVP, 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, β-TVP employs a two stage commit for its transactions.

The first stage, β-TVP 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, if those cache lines have not already come in from the validation procedure mechanism earlier. All the while, that same validation mechanism would apply to each incoming cache line, thereby aborting the transaction if data that is part of the read set has changed.

β-TVP then issues exclusive cache requests to all lines in the cache that are part of the transaction's write set but are not in an exclusive state. Once all the cache lines are in their correct commit state, β-TVP moves to the second commit phase, which is a normal ATMTP commit.

We note that β-TVP issues only one cache request at a time and, conservatively, waits for a response before issuing another request.

---

[3] ATMTP also provides the ability to use a timestamp-based conflict management policy [33], whereby the requester only wins if it is an older transaction.

[4] Lines that are written to are in an exclusive state, i.e., E or M in a MESI protocol. Lines that have only been read are in a valid state, i.e., S, E or M in a MESI protocol.

---

[1] Shared, Exclusive or Modified in a MESI protocol

[2] present but marked Invalid in a MESI protocol

The contention management policy during β-TVP's first commit phase is different from that during a β-TVP transaction. Invalidation requests for cache lines that are either part of the β-TVP transaction's read or write sets are denied (nacked). To prevent deadlocks, the simplest thing to do would be for the committing transaction to abort if a cache request it has sent was denied during this first commit phase.

A more sophisticated contention management policy, which we have adopted, is as follows: if a cache line invalidation request comes in during the first commit phase, the transaction would acknowledge it if and only if the requester is also committing and has higher priority (e.g., as determined using timestamps [18,33]), otherwise the request would be denied.

Using this scheme, there is no need to abort in the commit phase if a request issued by the committing transaction was denied; a transaction would just reissue the request. Deadlocks cannot occur if the priority is unique (i.e. no two transactions can have the same priority level).

Miscellany

It is worth noting that since a β-TVP transaction speculates on stale data, it could cause inconsistent execution [8], which might trigger certain traps (e.g., divide by zero). This is not a problem as ATMTP, by default, aborts a transaction if it encounters such a scenario. Inconsistent execution, if left unhandled, could also cause infinite loops [8]. However, infinite loops are not a problem in β-TVP as stale data is validated within a bounded period of time. Therefore, data cannot be inconsistent indefinitely.

It is also worth noting that β-TVP assumes all conflicts are due to false sharing, probably a good assumption when false sharing is common. We believe that while the cost of speculation, even in the presence of true sharing, is low, in some cases it might be helpful if β-TVP were to recognize true sharing and handle such cases accordingly. This is one of the areas we are currently investigating.

3.3 β-TVP Architecture

β-TVP should be compatible with any cache coherence protocol with states denoting shared, exclusive, and stale cache lines. It should not matter whether it is a snooping or a directory based protocol [34–37]. Moreover, β-TVP should also be compatible with existing hardware transactional memory proposals that detect conflicts using the cache coherence protocol.

β-TVP does not modify the existing coherence protocol used in ATMTP, and adds little processor-local hardware. The additional hardware requirements are as follows.

First, β-TVP requires additional bits per cache line for the read mark bits. These bits are only necessary for the transactional cache. For instance, ATMTP's transactional cache is the L1 cache; therefore, these bits are only added to the L1 cache and not the L2 cache.

Second, we are assuming the capability of flash clearing the read mark bits. This is only desirable for performance, it is not required for correctness. If these bits cannot be flash-cleared, they could be cleared sequentially, at the cost of potential phantom conflicts in future transactions.

Third, β-TVP 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 an MSHR [38] while the validation takes place, and extra logic needs to be added to compare the values being validated.

We believe the above requirements would be the greater part of the additional logic needed for β-TVP. Some of this additional hardware could also be used for an implementation similar to the one described by Huh et al. [1], which would also be beneficial *outside* transactions.

3.4 β-TVP Design Alternatives

β-TVP is the first prototype that uses the ideas of Transactional Value Prediction. We have encountered many points where alternative design decisions could have been made.

Our initial goal was to prepare a simple prototype that requires as few changes as possible, trying not to give β-TVP an advantage over ATMTP in the absence of false sharing and contention. We do not claim any decisions we have made are optimal, and there is definitely room for improvement. Some of the design decisions we contemplated follow.

- β-TVP uses ATMTP as its baseline implementation, mainly due to its relationship with the upcoming Rock processor. ATMTP is but one of many implementations that might benefit from Transactional Value Prediction.

- We have not modified the cache coherence protocol for simplicity. Some modifications to the coherence protocol might improve performance, as demonstrated by Huh et al. [1].

- β-TVP speculates on stale cache line data. Other forms of value prediction, especially in the absence of stale data, might be better suited.

- β-TVP tracks read data using read mark bits. Other methods, such as using tables or signatures, are also possible.

- β-TVP assumes the existence of a write buffer, as it is already present in ATMTP. A write buffer is not required, and can be replaced by other means.

- β-TVP does not request cache lines that are part of the write set exclusively until commit time. Requesting cache lines exclusively before reaching the commit phase could reduce the time a β-TVP transaction would stall before committing.

- Contention management is a complex topic with different tradeoffs [33]. It is not clear whether eager vs. lazy conflict detection or version management is better. Transactional Value Prediction might affect this choice.

- β-TVP always speculates on stale cache line values. Such speculation, if wrong, could abort a transaction. Therefore, at times, it might be better to stall, or to take another checkpoint then speculate.

- When β-TVP speculates on a stale cache line, or if a cache line it has read gets invalidated, it issues a validation request immediately. In certain contexts, it might be better to defer issuing a validation request to a later time.

- β-TVP always acknowledges invalidation requests, unless a transaction is in its commit phase. It might be better to deny invalidation requests, or abort altogether, if the line being invalidated might be truly shared.

- The current cache line replacement policy treats all invalid and invalidated cache lines the same. β-TVP might benefit if the replacement policy prefers stale cache lines that are part of a transaction's read set to those that are not.

- A β-TVP transaction aborts if a cache line in its read set gets evicted, even if that line is stale. By having a solution analogous to a victim cache [39], some aborts could be prevented.

- β-TVP issues only one cache request at a time and can only have one pending request at a time. Increasing the number of cache requests it can issue and the number of requests that can be pending could improve performance.

4. Preliminary Evaluation

4.1 Simulation Environment

Our simulation framework is based on Virtutech Simics [40], in conjunction with customized memory models built on the University of Wisconsin GEMS 2.1 [41]. The simulator models processors that have best-effort hardware transactional memory support, using Sun's ATMTP simulation framework [31], itself a component of GEMS 2.1.

The simulated environment models a SPARC-V9 multi-core processor [42], with a shared L2 cache and a private transactional L1 cache. It uses a MESI directory-based cache coherence protocol.

When simulating ATMTP, we use its default parameters [31]; however, we have increased the size of the write buffer to 64 entries (from the default 32) to ensure that all transactions succeed in hardware. We have also changed the conflict management protocol to timestamp [33].

We note that while the ATMTP simulator is a Rock-like simulator, we are not trying to simulate Rock. What we are aiming for is a best-effort hardware implementation that has some restrictions that might be expected in Rock.

To simulate β-TVP, we have extended ATMTP without modifying the cache coherence protocol. We have recently finished writing the simulator additions for this proposal; at

the time of the writing of this paper, we have had limited chance to test it on a full range of benchmarks.

The next section describes the experiments we ran to obtain a preliminary estimate of the benefits of β-TVP.

4.2 Experiment Description

We have created a group of synthetic benchmarks in an attempt to cover a range of real world sharing scenarios. These benchmarks are by no means comprehensive or conclusive, but merely evidence collected to date to support our intuition.

The following experiments were run on a simulated 8-processor SPARC-V9 machine using the ATMTP environment described earlier. Each experiment involves running 1, 2, 4, and 7 threads each on a separate processor[5], with each thread performing 200 transactions. The transactions have been chosen so that the only reason they would abort is due to conflicts with other transactions, i.e., they will eventually succeed from retrying. As such, there is no need for a software fallback mechanism.

We are comparing the throughput of ATMTP, β-TVP with false sharing conflict detection granularity of 4 bytes (one word[6] — TVP-4), and also 64 bytes (a whole cache line — TVP-64). TVP-64 is not an attempt to mitigate the effects of the false sharing; however, it is used as a control experiment to account for the different contention mechanisms used in ATMTP and β-TVP.

Below is a description of the experiments we ran.

False sharing followed by no sharing: All threads start by incrementing different parts of the same one cache line, followed by incrementing 39 different cache lines.

No sharing followed by false sharing: All threads start by incrementing 39 different cache lines, followed by incrementing different parts of the same one cache line.

False write sharing: All threads increment *different* parts of the same 40 cache lines.

True write sharing: All threads increment the *same* part of the same 40 cache lines.

Read-write false sharing: The first thread increments 40 different cache lines, while all other threads read the same 40 cache lines. However, the reads and writes (increments) are to *different* parts of the same cache lines.

Read-write true sharing: The first thread increments 40 different cache lines, while all other threads read the same 40 cache lines. However, the reads and writes (increments) are to the *same* parts of the same cache lines.

---

[5] The simulated environment runs more smoothly with a dedicated processor for kernel-related events, as recommended by the Wisconsin GEMS group.

[6] The definition of a *word* is architecture dependent. The SPARC Architecture Manual, version 9 [42], defines a *word* as a quadlet (4 bytes). This is the default size of an integer (`int`) on a SPARC-V9 platform.
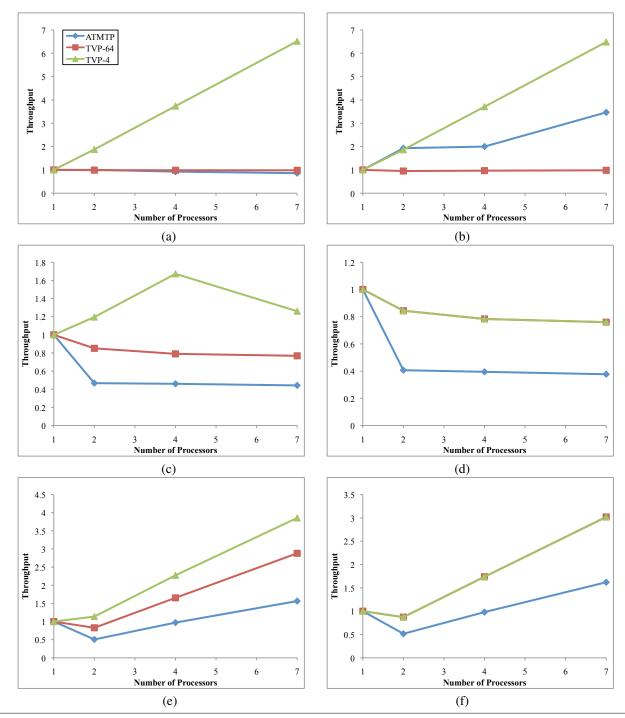
Figure 2. Comparative throughput of different schemes, normalized to the throughput of a single processor (higher is better). (a) false sharing followed by no sharing (b) no sharing followed by false sharing (c) false write sharing (d) true write sharing (e) read-write false sharing (f) read-write true sharing

4.3 Preliminary Results and Analysis

Figure 2 presents the results from the experiments mentioned earlier. The results show the throughput of the different tests normalized to the throughput of a single processor, which is the same in all cases of a single benchmark.

A small amount of false sharing can have a big impact on throughput as seen in (a) and (b), which outline the results for having only one falsely shared cache line among no sharing at all. If false sharing occurs at the beginning of a transaction as in (a), then ATMTP fully serializes the

transactions. On the other hand, if false sharing occurs at the end of a transaction as in (b), then there is more parallelism in ATMTP but throughput still does degrade.

We note that in (b), ATMTP throughput does not improve going from 2 processors to 4. Investigating this showed that in this particular example, transactions are synchronized in a pattern at 4 processors that is causing a higher number of them to abort, due to true data conflicts.

In this case of having little false sharing (a,b), using TVP-4 seems to have completely mitigated the effects of false sharing, almost achieving perfect parallelism. TVP-64 underperforms ATMTP in (b), probably due to differences in conflict management.

The chart in (c) shows throughput with false sharing over a big write set. ATMTP throughput degrades, compared to the single-threaded case, since the cache lines thrash between threads. TVP-4 significantly mitigates this effect. However, TVP-4 does not provide perfect parallelism since the cache lines still need to be exclusive at commit time. TVP-64 outperforms ATMTP in this scenario, probably because of the different contention management. We note that TVP-4's throughput drops going from 4 to 7 processors, possibly due to the higher level of contention over the cache lines during the commit phases of the different processors.

The chart in (d) shows throughput with true sharing over a big write set. Unsurprisingly, performance degrades regardless of the mechanism used. That said, TVP-4 and TVP-64 outperform ATMTP, probably for the same contention management reasons mentioned earlier. Observe that TVP-4 performs no worse than ATMTP despite speculating incorrectly about the presence of false sharing.

When there are many readers and a single writer as in (e) and (f), ATMTP does not scale well since that one writer cannot run in parallel with any readers. TVP-4 allows that writer to run in parallel with other readers if it is false sharing. If it is true sharing, then TVP-4 and TVP-64 only serialize the writer and readers during the writer's commit phase, rather than the whole duration of the writer's transaction.

It is worth noting that the thread running on the first processor in (e) and (f) is a writer thread. From that point on, all new threads are read-only threads. This explains why performance degrades in some cases going from 1 to 2 processors, but improves after that point.

We have covered a variety of different sharing scenarios, but this is by no means conclusive. There probably are scenarios where ATMTP would outperform β-TVP. We think that such scenarios, however, are rare and that even in their presence β-TVP would not suffer much. This is a work in progress, and we are still in the course of investigating different possibilities using more benchmarks.

5. Prior Work

Huh, Chang, Burger, and Sohi's work on Coherence Decoupling [1] proposes a solution to the problem of false shar-

ing. One of the methods they suggest is speculating based on the values of stale cache lines. We have applied some of the concepts they propose to transactional memory, taking advantage of the speculative execution inherent in it. Unlike their work, we do not alter the underlying coherence protocol beyond what is already present in hardware transactional memory.

Torrellas, Lam, and Hennessy [43] propose some solutions to the false sharing problem. Their work investigates the relationship between false sharing and spatial locality, and proposes compiler modifications that optimize the layout of shared data in cache lines to mitigate its effects.

Kadiyala and Bhuyan [44] propose a hardware solution to the problem. Their work suggests maintaining coherence on small cache lines, while using larger lines containing several of these small lines as the unit of transfer. They argue that this would reduce false sharing while retaining the benefits of having larger cache lines.

Lepak, Bell, and Lipasti [29] explore the recurrence of previously seen values in a program. They also explore new definitions of false sharing based on changes, or lack thereof, in the values being stored. Lepak and Lipasti [30] exploit this phenomenon in their work on the MESTI protocol to reduce memory traffic and improve performance. Their work, however, is not based on speculative execution.

Olszewski, Cutler, and Steffan propose JudoSTM [45], a dynamic binary-rewriting software transactional memory that detects conflicts based on value changes. By using value-based conflict detection, JudoSTM also improves performance in the presence of silent stores. However, JudoSTM does not address the problem of false sharing.

6. Future Work

Transactional Value Prediction is still a work in progress. There are many issues we intend to address in the near future and alternative design options we intend to explore, such as the ones in section 3.4.

We have presented the ideas of value prediction and data speculation in hardware transactions, mainly in the context of mitigating the effects of false sharing. We believe there may be other forms of taking advantage of these ideas, and we intend to explore these venues.

Speculating in the case of true sharing, rather than false sharing, might not always be a good idea. We would like to investigate in more depth the impact of such misspeculation, and look into methods that differentiate between cases of true and false sharing, and deal with them appropriately if needed. One such method would be to extend the contention management policy to disallow sharing of cache lines that might be involved in true sharing.

This could be achieved in several ways, for example, by counting the number of set read mark bits in the cache line. If the number of set bits exceeds a certain threshold, all requests to that line would be denied or redirected to the

contention management mechanism. Another way would be to maintain a history of true conflicts in cache lines, and use that history as a predictor of false sharing.

The preliminary β-TVP results presented in this paper are based on a set of synthetic benchmarks we have created. We are in the process of testing β-TVP using a subset of the STAMP benchmarks [46], as well as some of the microbenchmarks used in many transactional memory evaluations, such as red-black trees, linked-lists and chained-hash tables [47]. We note some of these benchmarks may already be tuned to work around false sharing; therefore, we need to investigate the best way of using these benchmarks to evaluate our ideas.

We have chosen ATMTP as the framework to use with β-TVP, mainly because it simulates an environment similar to the upcoming Rock processor. However, these ideas could apply to many different hardware transactional memory implementations, and other similar lock-based mechanisms such as SLE. We are also considering applying these ideas to LogTM-SE [22].

LogTM-SE allows the eviction of cache lines that are part of the read set; such an eviction would abort a β-TVP or ATMTP transaction. An example of applying β-TVP to LogTM-SE could handle evictions by invoking nested transactions whenever the value prediction mechanism is used. Thus, only values that are part of the nested transaction's read set cannot be evicted without aborting the nested transaction.

We have presented our ideas mainly in the context of false sharing, and to a lesser extent in the context of silent stores. We are also investigating how Transactional Value Prediction could improve performance and make it easier to program in other contexts.

7. Concluding Remarks

In this workshop paper, we have introduced Transactional Value Prediction, ideas for data speculation and value prediction in hardware transactional memory. We presented these ideas mainly in the context of mitigating the effects of false sharing. However, we believe that there may be other ways of using Transactional Value Prediction to improve performance of hardware transactions.

We have also explained how false sharing, at the cache line level, could negatively affect both performance and ease of programming. Therefore, we believe it is an important issue to address in hardware transactional memory.

We developed a preliminary proposal, β-TVP, as one example that uses some of these ideas in an attempt to address the problem of false sharing. We have demonstrated that, at least in some cases, β-TVP can alleviate, or even eliminate, the effects of false sharing.

References

[1] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar Sohi. Coherence decoupling: making use of incoherence. *ACM SIGARCH Computer Architecture News*, 32(5), 2004.

[2] David Geer. Chip makers turn to multicore processors. *IEEE Computer*, 2005.

[3] Anwar Ghuloum. Unwelcome advice, 2008.

[4] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[5] Geoff Koch. Discovering multi-core: Extending the benefits of Moore's law. *Technology*, 2005.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

[7] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 1993.

[8] James R. Larus and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 1967.

[10] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *IEEE Computer*, 2008.

[11] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 422 – 431, 1988.

[12] William J Bolosky and Michael L Scott. False sharing and its effect on shared memory. 1993.

[13] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[14] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. *Proceedings of the 20th Annual International Sympsium on Computer Architecture*, pages 289–300, 1993.

[15] Ravi Rajwar and James Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, 2001.

[16] Ravi Rajwar and James Goodman. Transactional lock-free execution of lock-based programs. *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002.

[17] Maurice Herlihy and J Eliot B Moss. System for achieving atomic non-sequential multi-word operations in shared memory. *US Patent 5,428,761*, 1995.

[18] Kevin Moore, Jayaram Bobba, Michelle Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[19] Tim Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[20] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. *Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[21] William N. Scherer III, Doug Lea, and Michael L. Scott. A scalable elimination-based exchange channel. *Proceedings of the Workshop on Synchronization and Concurrency*, 2005.

[22] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272, 2007.

[23] Hany Ramadan, Christopher Rossbach, Donald Porter, Owen Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: transactional memory for an operating system. *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[24] Dan Grossman. The transactional memory / garbage collection analogy. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, 2007.

[25] Jayaram Bobba, Neelam Goyal, Mark Hill, Michael Swift, and David Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[26] Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. Kicking the tires of software transactional memory: why the going gets tough. *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008.

[27] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, 41(6):26–37, 2006.

[28] Enrique Vallejo, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero. Hybrid transactional memory to accelerate safe lock-based transactions. *Workshop on Transactional Computing (TRANSACT)*, 2008.

[29] Kevin Lepak, Gordon Bell, and Mikko Lipasti. Silent stores and store value locality. *IEEE Transactions on Computers*, 50(11):1174 – 1190, 2001.

[30] Kevin Lepak and Mikko Lipasti. Temporally silent stores. *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002.

[31] Mark Moir, Kevin E. Moore, and Daniel Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. *The third annual SIGPLAN Workshop on Transactional Memory*, 2008.

[32] Brian Goetz. Optimistic thread concurrency. Technical report, Azul Systems, 2006.

[33] Jayaram Bobba, Kevin Moore, Haris Volos, Luke Yen, Mark Hill, Michael Swift, and David Wood. Performance pathologies in hardware transactional memory. *Proceedings of the 34th international symposium on Computer architecture*, 2007.

[34] James Goodman. Using cache memory to reduce processor-memory traffic. *Proceedings of the 10th annual international symposium on Computer architecture*, 1983.

[35] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *In Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[36] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, C-27(12):1112 – 1118, 1978.

[37] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Proc. of the National Computer Conference*, volume 45, pages 749–753, 1976.

[38] David Kroft. Cache memory organization utilizing miss information holding registers to prevent lockup from cache misses. *US Patent 4,370,710*, 1983.

[39] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *17th Annual International Symposium on Computer Architecture*, 1990.

[40] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustab Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 – 58, 2002.

[41] Milo Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa Alameldeen, Kevin Moore, Mark Hill, and David Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4), 2005.

[42] The SPARC architecture manual, version 9. 2000.

[43] Joseph Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651 – 663, 1994.

[44] Murali Kadiyala and Laxmi N. Bhuyan. A dynamic cache sub-block design to reduce false sharing. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 313 – 318, 1995.

[45] Marek Olszewski, Jeremy Cutler, and JG Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[46] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, 2008.

[47] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003.