Unbounded Page-Based Transactional Memory

Weihaw Chuang[†], Satish Narayanasamy[†], Ganesh Venkatesh[†], Jack Sampson[†], Michael Van Biesbrouck[†], Gilles Pokam[†], Osvaldo Colavin[‡], and Brad Calder[†][*]

[†]University of California - San Diego, [‡]ST Microelectronics, [*]Microsoft

Abstract

Exploiting thread level parallelism is paramount in the multi-core era. Transactions enable programmers to expose such parallelism by greatly simplifying the multi-threaded programming model. Virtualized transactions (unbounded in space and time) are desirable, as they can increase the scope of transactions' use, and thereby further simplify a programmer's job. However, hardware support is essential to support efficient execution of unbounded transactions. In this paper, we introduce *Page-based Transactional Memory* to support unbounded transactions. We combine transaction bookkeeping with the virtual memory system to support fast transaction conflict detection, commit, abort, and to maintain transactions' speculative data.

Categories and Subject Descriptors C. Computer Systems Organization [*C.1 Processor Architectures*]: C.1.4 Parallel Architectures

General Terms Design, Languages, Performance

Keywords Transactions, Transactional Memory, Parallel Programming, Concurrency, Virtual Memory

1. Introduction

Effective utilization of multi-core processors is stymied by the difficulty of programming multithreaded programs. Failure to obey data dependencies between threads results in race conditions, where variables are modified in an order not possible in a single thread of execution. This results in bugs that are extremely hard to detect, understand and replicate. Many serious bugs are attributed to race conditions.

Most parallel programs use locks and other synchronization primitives to impose an order between threads. Poor lock usage results in incorrect code and performance penalties [15, 11]. Coarse-grained locks result in inefficient execution causing threads to wait when they could otherwise be executed in parallel. Too fine a granularity adds programming complexity and increases the likelihood of deadlock or other incorrect behavior.

Transactions provide the appearance to any external viewer of atomic execution of code regions, simplifying the creation of parallel code. Any memory access in a transactional region that aliases with a concurrent external memory access is detected as a conflict and either the transaction aborts [11] or stalls [14], or the external memory operation stalls. Stalling or aborting on conflict serializes the data accesses, thus preventing the data races and providing atomicity for the transaction's execution.

To maintain the appearance of atomicity, transactional memories must perform two tasks: detect conflicts and recover from them. Doing this efficiently under all conceivable operating conditions is a significant challenge. We focus on hardware transactional memories, which provide these features while maintaining good performance. To facilitate conflict detection and recovery, transactional memory maintains a speculative version of memory. If a conflict is detected, the hardware can quickly abort, discarding the speculative memory modifications to recover the original memory state before the transaction started. Otherwise, it can commit quickly to promote the speculative memory to non-transactional memory. Both operations are done atomically so that all processors see consistent state before and after the operation. Virtually all recent proposals for hardware transactional memory build versioning hardware into the cache and use cache coherence for conflict detection [11, 9, 1, 15]. One critical issue is what to do when a program's transactional working set exceeds the cache capacity, or must be evicted due to a context switch. Once a transaction's data is placed in some backing store such as main memory, it is important that memory accesses that conflict with that transaction be detected.

We propose a *Page-based Transactional Memory* (PTM) technique to handle the transactional state overflow condition to minimize overhead, and handle all possible cases including *context switches, exceptions, and inter-process shared memory*. For PTM, the cache overflow bookkeeping is done at the page level, but performing conflict detection for overflowed blocks is at the cache block granularity. This distinction is one of the differences we have over prior techniques [15, 1, 14] that do both the bookkeeping and detection at the cache line level. In order for PTM to efficiently organize transactional state occurring at the cache block level the state is reduced to a boolean bit value and packed into a bit-vector for each transactional page being used.

All transactional-memory systems incur an overhead cost for committing or aborting transactions whose memory footprint includes blocks that have overflowed from the cache. Several prior techniques use some temporary backing-store in main memory such as a log [15, 1, 14], or hash table [1], and they either copy-update on commit, or copy-restore on abort. In PTM, an additional physical page is allocated for an overflowed physical page called the *shadow page*. We refer to the original physical page as the *home page*. One of the two physical pages holds the non-speculative committed data while the other holds the speculative version. Both the non-speculative and the overflowed speculative version of a cache block are kept in the same page offsets on these two pages. We investigate a PTM design, called Copy-PTM, where we use a copy-restore on abort policy. That is, when a transaction block overflows, it is always written to the home page and the old committed data is copied to the shadow page. This design enables fast commit. However, on abort the data from the shadow page needs to be restored to the home page.

Copying from main memory is expensive, and is compounded by the number of overflowed lines that must be copied. In the Copy-PTM design, the original memory block, which is being overwritten by the transaction, has to be copied once if the transaction commits (eviction) or twice if the transaction aborts (eviction and abort restoration). To address this, we investigate another design for PTM, Select-PTM, where we use a set of selection bits (one for each block in a page) to avoid copying the memory blocks. The selection bit for a memory block determines which of the two pages contain the non-speculative data. With this approach the original memory block never has to be moved, even if the transaction commits or aborts, which provides both fast commit and abort.

2. Transaction Model

Transactional memory has its roots in database transactions [5]. Transactions are sequences of memory operations that are *atomic*, *isolated* and *serializable*. Atomicity guarantees that either all or none of the operations within a given transaction take effect. Isolation ensures that no other transaction can observe the sequence of operations in a partially completed state. Having serializability means that there exists a serial execution of the transactions that has the same effect as the actual execution that occurs. Transactions simplify the task of writing concurrent programs because they ensure that each thread observes data in a consistent state and can be programmed as if other threads do not exist.

2.1 Size of Transactions

With the advent of multi-cores, exploiting thread level parallelism has become important to an increasingly broad set of programmers, including many not well-versed in parallel programming. Transactions can enable programmers to write multi-threaded code and expose such parallelism. It is not reasonable to expect them to use fine grained critical regions *safely*. Hence, we firmly believe that programmers will create large transactions (either by accident if not on purpose). For example, a straight-forward method to use transactions is to put the transaction around the start and end of a loop, and the loop may contain procedures calls that go off and touch more data than the programmer is aware of. In addition, transactions can be applied to loops to exploit thread level speculation (TLS) in order to derive parallelism in the presence of loop carried dependencies.

Harris *et al.* [10] examined optimizations for software transactional memory, and provided support for long running transactions containing millions of shared memory accesses. The approach relied solely on software support, and such long transactions had significantly greater slowdown than shorter transactions. Recently, Chung *et al.* [3] benchmarked several different styles of parallel programs such as SpecOMP programs, which have significantly larger transactional regions.

2.2 Ordered vs. Unordered Transactions

In creating transactions for our study, we supported two types of transactions: ordered and unordered. Ordered transactions constrain the sequence of commits amongst the transaction threads to enforce some programmer defined order. Unordered transactions allow the transactions to commit in any order. Ordered transactions are used by programmers when they do not know if there is a potential loop-carried dependency in a loop that they want to parallelize. In this case, they will invoke a call to create a new ordered transaction, and then each transaction created will execute in parallel, but commit in the order in which it was created. If programmers know that there are no loop-carried dependencies, then they can create an unordered parallel transaction loop, where the transactions can commit in any order.

2.3 Transactional Execution Model

We now summarize our transaction execution model and how it deals with nested transactions, non-transactional code, and how it interacts with the operating system.

2.3.1 Nested Transactions

A nested transaction is a transaction that starts within a transaction. Like many other transactional memory proposals, PTM flattens nested transactions into the outermost already existing transaction[15, 1]. This means that for nested transactions, transaction begin and end result in just incrementing and decrementing a transaction nest count maintained in the architecture. It also means that if an inner transaction aborts, all outer transactions also have to abort.

2.3.2 System Calls, Exceptions, Context Switch and Interrupts

For the transactions in this paper, we did not use system calls inside the transactions, although nothing precludes transactions with system calls from being supported by PTM. It is expected that transactional support be provided for a restricted set of system calls by operating systems. For example, Microsoft Windows Vista has software transaction support for the file system and registry system calls. For the restricted set of system calls with transaction support, the operating system must buffer the speculative state, and be able to commit and abort that state. If the system calls were to be used inside of hardware transactions, the act of committing and aborting would have to invoke the appropriate software handler to also commit or abort the system state. Investigating the use of a restricted set of system calls with hardware transactions is an interesting area of future research.

For exceptions, we execute an exception as part of the transaction that caused the exception. Whenever a processor is running a transaction thread and a context switch or interrupt occurs, recording of the current transaction state is stopped until the running of the transaction thread resumes. We correctly maintain the state of the cache blocks and the unbounded transaction state across context switches and interrupts using the techniques described in Section 4.7.

2.3.3 Interaction with Non-Transactional Code

Non-transactional memory writes to locations accessed by uncommitted transactions affect the correctness of program execution. A complete transactional system must address this issue. For our approach, the solution is simply to abort the transactions affected by a non-transactional write. Detecting non-transactional conflicts with transactions is straightforward with our approach because our transaction conflict detection mechanism monitors all non-transactional writes. If a conflict is found, it is treated like a conflict between transactions except that in this particular case the conflicting transaction has to always abort.

3. Page-Based Transactional Memory (PTM)

The purpose of Page-based Transactional Memory (PTM) is to support efficient virtualization of a transaction's execution in the presence of cache overflows, context switches, thread migration, paging and inter-process shared memory communication. PTM achieves its goal by extending the virtual memory support in the operating system to virtualize a transaction's execution.

3.1 Transaction Cache State

For handling bounded transactions, PTM assumes hardware support similar to the architectures proposed in prior work [15, 1]. To support bounded transactions, we need to keep track of the read and the write transactional states for each cache block, and use the coherence mechanism to do an eager conflict detection [15, 1]. The eager conflict detection mechanism checks for a violation on every cache coherence miss. If there is a violation, the oldest transaction always wins the conflict.

In addition to augmenting the cache blocks with the transactional states and supporting eager conflict detection, we also need a checkpoint mechanism to abort and re-execute a transaction. Our approach assumes support for checkpointing the register state when starting the execution of a transaction, similar to the earlier studies [1, 15]. Apart from such basic transactional-memory support in the processor core, PTM does not require any other significant change in the processor core, as most of its functionalities are placed in the memory controller.

In our PTM design, we take care not to adversely impact the performance of transactions whose working sets fit within the transactional cache. As long as the cache blocks accessed by a transaction do not get evicted from the transactional caches, the basic on-chip transactional memory system handles the execution of the transaction, detecting violations, and providing support for committing and aborting of cache blocks. This is similar to how the bounded transactions are handled in prior work [1, 15]. To provide this functionality, we keep a global flag indicating if any blocks have been overflowed or not, for a set of transactions in the same scope. If none of the transaction blocks overflow the cache, then when a thread misses in the cache, a conflict check does not need to be performed by PTM for the miss. The conflict check is instead handled completely by the on-chip transactional memory system. Only when a transactional block (read or written by a running transaction) has been evicted does our PTM mechanism come into play.

3.2 Home and Shadow Pages

A key difference between our approach and the prior techniques is how we maintain the transactional information for the transaction blocks that have been evicted from the cache. A transaction block is a block of memory accessed by a transaction that is still executing.

For an evicted transaction block we need to maintain the following information in a data structure: (1) the speculative data for the block, if it has been written by a transaction, and (2) a list of all the transactions that either read from or wrote to the evicted transaction block, as required for conflict resolution.

We will now describe how we store the speculative data for a transactional block when it gets evicted from the transactional cache. We observe that, for a set of transactions that are currently executing, there can be only one transactional writer to an address at any instant of time (otherwise, a conflict would be detected and one of the two conflicting transactions would have been aborted). Therefore, all that we need for any physical page accessed by a transaction is an additional page that can hold a transactional version of data for the memory blocks in the page. We call the original physical page the *home page*, and the additional physical page allocated as the *shadow page*.

Figure 1 shows an example of the PTM data structures used to maintain the unbounded transactional memory. On the left side of the figure, we show the page tables used to perform the traditional virtual to physical page translation. We also have another structure called the Shadow Page Table (SPT), which contains one entry for every physical page of memory, and is indexed with the physical page number. In the SPT entry, we store the address of the allocated shadow page, the home page's address, and some additional information required to maintain the unbounded transactional states. There is a valid bit associated with the shadow page pointer, since not every SPT entry will have a shadow page allocated for it. Since the SPT is indexed by the physical page number, we can access information about the transactional memory block given its physical or virtual address. Given a physical address, we can directly index into the SPT. Given a virtual address, we can use the page table to get the physical address and then access the corresponding SPT entry.

When a page is allocated, its corresponding allocated physical page entry in the SPT is initialized and marked as valid. When a dirty transactional block is first evicted for a page used within a transaction, PTM allocates a shadow physical page, a pointer to it is stored in the SPT, and the shadow pointer is marked as valid for that SPT entry. For example purposes, we show in Figure 1 an SPT entry for a physical page address "0x0000000" containing the shadow physical page address "0xFE03000". The corresponding speculative transactional block and the non-speculative block can then be kept track of in the two pages (home and shadow). Note, the physical shadow page that was allocated does not have a valid SPT entry. Only the home physical pages have valid SPT entries, which are marked as valid when the home physical pages are allocated. In addition, not all SPT entries have a valid (allocated) shadow page. If there are transaction blocks evicted from the cache that were only read (not written), then they may have an SPT entry without a shadow page allocated for it. In this case, the SPT entry serves the purpose of finding the transaction access information for the home page (what blocks were read, and by which transaction), which we describe later in Section 3.3.

3.2.1 Copy-PTM

Now that we have the shadow page, the question is where to store the speculative transaction blocks that have been evicted from the cache? One could have the policy where the speculative blocks are stored in the shadow page, and on commit they are copied back to the home page. We originally examined this design, but found the cost of commit to be higher than we desired. We hope to see many more commits than aborts when using transactions and we do not want to slow down the execution of transactions that are doing useful work. We therefore want to optimize the performance of committing, and start running the transactions in order if aborts are too frequent.

The first PTM approach we examine is called *Copy-PTM*. In this approach we copy the home block to the shadow page when a dirty transaction block overflows, and then store the speculative block in the home page. This policy enables fast commits, since the blocks that we want to commit are already in the home page. It requires that we make a copy of the non-speculative block from the home page to the shadow page when a dirty transaction block is evicted for the first time in a transaction. Then on abort we have to pay a penalty because we have to copy the non-speculative blocks, which were overwritten in the home page, back from the shadow page to the home page.

3.2.2 Select-PTM

The more aggressive solution we examine is to allow both the home and the shadow page to contain speculative and non-speculative blocks and use *Selection Vectors* to maintain them. We call this *Select-PTM*.

In Select-PTM, both speculative and non-speculative blocks are allowed to exist in either the home or the shadow page. We use a *Selection Vector* to indicate which of the two pages contain the non-speculative block and the speculative block. The selection bit vector is stored along with the shadow page pointer in the SPT structure as shown in Figure 1. Each bit in the selection vector represents a memory block in the page. We chose the size of the memory block to be the same as the cache block size of the outermost transactional cache in the processor, but our design does allow for larger or smaller memory blocks to be used.

A bit in the selection vector tells us which of the two pages, the home or the shadow page, contains the current committed data for the memory block for which the bit corresponds to. If a bit in the selection vector is set, then this means that the non-speculative data for that memory block resides in the shadow page and that the home page should be used for holding the speculative version and *vice versa*.

Whenever a transaction modifies a cache block and evicts it, the block is copied to the speculative location in memory, which is either the home or the shadow page depending upon the state of the bit in

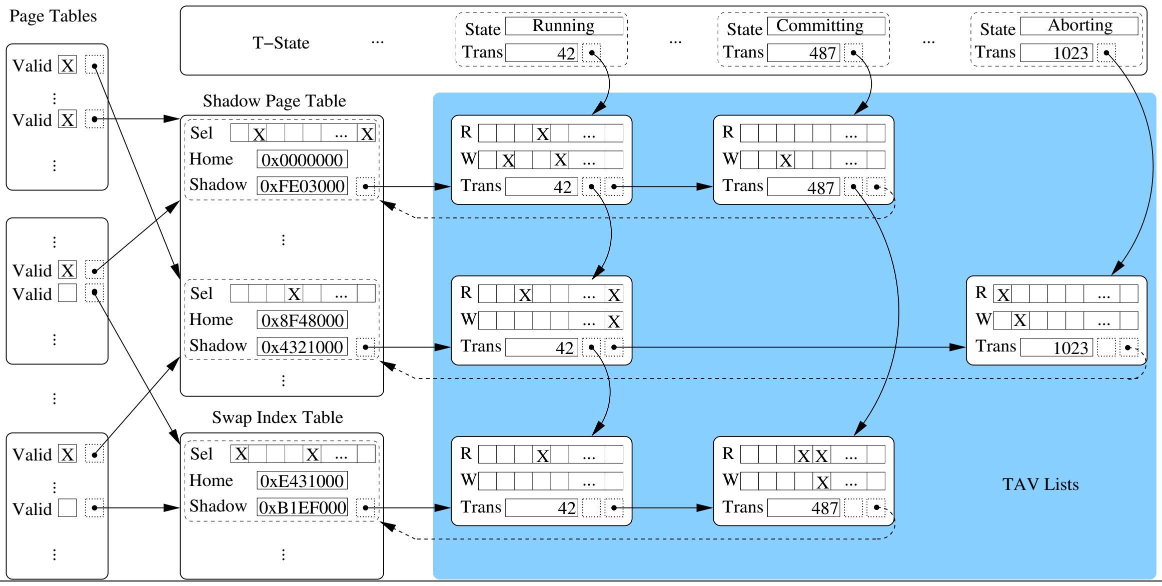

Figure 1. PTM structures. Physical page numbers and swap file offsets are obtained from the page tables and used to index into the Shadow Page Table (SPT) and the Swap Index Table (SIT) respectively. An entry in the SPT and SIT tables for a page indicate the locations of the shadow page and contain a Selection Vector in which each bit indicates which of the two pages contain the committed version of a block in the page. An entry also points to a Transaction Access Vector (TAV) List, which contain one node per transaction that has accessed the list's page, but was not able to keep the accessed blocks in the cache. The nodes in a TAV list indicate the transactions in question and contain the Read and Write Vectors to mark the accessed blocks that do not stay in the transactions' cache. The T-State table is indexed by a transaction number and contains the state of each transaction. An entry in the T-State table links to a list of TAV nodes that were overflowed by the transaction.

the selection vector. Similarly, while fetching data from memory we can determine where to find the committed and speculative copies based on the state of the bit in the selection vector.

When a dirty transaction block is evicted, we write the block to the speculative location. We write the speculative block to the home page if the bit is set, or to the shadow page if the bit is clear. When a transaction aborts, nothing needs to be done, since the bits in the selection vectors for the pages touched by the transaction are already pointing to the non-speculative blocks. On commit, however, we must go through the selection vectors and toggle the bit corresponding to the overflowed memory blocks that were written by that transaction, though hardware can accelerate this, which we describe later.

3.2.3 Trade-offs Between Copy-PTM and Select-PTM

The PTM structures required for both Copy-PTM and Select-PTM are the same structures shown in Figure 1, except that Copy-PTM does not need the selection vector in each SPT entry.

In Select-PTM, the benefit of using a selection vector and allowing committed blocks to reside on either of the two pages (the home or the shadow page) is that it does not have to copy non-speculative blocks during eviction and abort, as described earlier for Copy-PTM. The downside to using the selection vector is that a non-speculative block can now reside in either the home or the shadow page, and we need to have an efficient way of finding the correct physical address to fetch the block, given the virtual address. The policy PTM enforces is that even when a block is fetched from a shadow page, the physical address seen by the cache hierarchy and the TLB structures is the home page physical address corresponding to that block. This allows Select-PTM to only have to perform TLB translation to the home page as in a conventional design. Then Select-PTM will monitor the block addresses at the memory controller to decide where to fetch the correct blocks from (the home or the shadow page). How this is done is described in Section 4.2.2. Therefore, the advantage of the Copy-PTM approach is that, since the committed blocks are always on the home page, it does not have to deal with this address translation issue, and it does not have to maintain the selection vectors.

3.3 Conflict Detection using Transaction Access Vectors

In addition to keeping track of the speculative data for the overflowed cache blocks, we must also keep track of the information about the list of the transactions that read or write to an overflowed cache block. To accomplish this task, we maintain a Transaction Access Vector (TAV) data structure as shown in Figure 1. Each TAV node in the data structure is for a transaction and for a page that a transaction has overflowed. The TAV contains a read vector and a write vector for the page. Each bit in the read/write vectors corresponds to a cache block in the page and it tells us if the cache block was read or written by a transaction.

The read and the write bits are set when the blocks accessed within a transaction are evicted from the cache. For example, Figure 1 shows that the transaction 42 read the 4th block and wrote the 2nd and 5th block in the virtual page with the physical home address "0x0000000" and the shadow page address "0xFE03000". When a read or write (executed in a transaction's code or even in the non-transactional code) misses the cache, PTM is consulted with the home page's physical address checking these read and write vectors to determine if there is a conflict. Note, we only need to check for conflicts in PTM if there is a live transaction and if a transaction has overflowed the cache.

All the TAV nodes corresponding to a transaction are linked together (vertical links in the Figure 1). Given the transaction number, we can find all the TAV nodes for that transaction. The TAV nodes corresponding to a page are linked together (horizontal links in the Figure 1). Thus, for a given TAV we can find its corresponding SPT entry. The same horizontal link is also used to find the TAV nodes of other transactions that have also accessed the same physical page, which enables us to determine the conflicting transactions.

TAV organization: Let us summarize the TAV data structure organization. An entry in the SPT structure contains a pointer to a linked list of these access vectors (transaction access vector (TAV) list). These are the horizontal linked lists in the Figure 1 and the last node in the list points back to the SPT entry. Each node in the TAV list is for a transaction that had at least one overflowed block for that page in the past. A node in a TAV list contains a transaction's read and write access bit vector, where each read/write bit corresponds to an overflowed cache block in the page and tells us if the overflowed cache block was read or written by the transaction. In addition, each entry also contains a transaction identifier, constructed by the TM hardware, which enables us to determine the transaction to which the TAV read and write access vectors belong. A node in a TAV list is updated when a transactional cache block is evicted, and freed when the corresponding transaction either commits or aborts.

Conflict detection using TAV: If a read or write (executed in a transaction or in the non-transaction code) misses the cache, and there exists an overflowed block, PTM is consulted with the physical address to resolve any potential conflict. PTM uses the physical address to index into the SPT structure to get the pointer to the TAV list. Each node in the TAV list corresponds to a transaction that has overflowed a read or write to the page, and has to be examined to determine if there is a conflict. If the current memory operation that triggered a miss is a read, then there is a conflict if there exists a node in the TAV list with the write bit set for the accessed memory block and the transaction identifier is different from the current read's transaction identifier. Conflict detection for a transaction write is similar, and we detect a conflict if there exists a node in the TAV list with either the read or the write bit set for the accessed memory block, and the transaction identifier differs.

In Section 4, we describe how information in the TAV list can be summarized into one vector and cached in a hardware structure to perform efficient conflict detection. These summary vectors are also used with the selection vector to determine which of the two physical pages to fetch from on a cache miss for Select-PTM.

3.4 Commit and Abort

To commit or abort a transaction, we use the vertical links shown in the Figure 1. The head of the vertical list is maintained in the *T-State* structure and it also contains the transaction identifier along with its current status, which is atomically set to either *committing* or *aborting* before processing the TAV list.

Select-PTM: On commit, we traverse the vertical list for the committing transaction and free the nodes in the list. In addition, we update the selection vectors as needed. This is achieved while traversing each node in the vertical list, where we access the TAV node's corresponding SPT entry by following the horizontal list. We update the selection vector for that SPT entry if the committing transaction has overflowed any dirty block for the page corresponding to the SPT entry. On abort, we also have to traverse the vertical TAV list and free the TAV nodes. But, unlike what we did for commit, we do not have to update the selection vectors.

Copy-PTM: On commit and abort we traverse the vertical list and free the TAV nodes. For commit, we do not need to do any additional work, since there are no selection vectors. On abort however, we need to restore the original non-speculative blocks to the home page, for those overflow blocks that were written by the transaction.

3.5 Paging and the Freeing the Shadow Pages

Since the blocks representing a page are split across the home and shadow pages, we need to correctly deal with paging those pages in and out, as well as how to free the shadow pages.

3.5.1 Paging

To deal with the paging out of transaction pages, we actually have two tables, the *Shadow Page Table* (SPT) and the *Swap Index Table* (SIT). The first is indexed using the physical address (when the page is in main memory) and the second is indexed using the swap index number (when the page is swapped out to disk).

The swap index number is the number used by the operating system to keep track of the pages that are swapped out. It is equivalent to the physical page number. The difference between the two is that the swap index number refers to a location on the disk, but the physical page number refers to a location in the main memory. Thus, when a page is swapped out of the main memory to a location in the disk, the swap index number corresponding to that location is stored in place of the physical page number in the page table entry. When an application refers to a swapped-out page, the swap index number is used to locate the paged out data and swap the page back in to the main memory. The new location in main memory referred by a physical page number is stored in the page table entry.

In PTM, the shadow and the home page cannot be swapped out independent of each other. If one of the pages is swapped out, both pages have to be swapped out. The operating system does not consider the shadow pages to be candidates for swap out. The operating system only makes decisions about swapping out home pages. When a page is swapped out, if the page has a valid SPT entry, then it has to be copied to a SIT entry. The index for the SIT entry is the swap index number corresponding to the location in the disk that is allocated to hold the swapped-out home page. If there is a valid shadow page for the home page, then it is also swapped or garbage collected (see below). If swapped out, then its SIT shadow pointer is used to point to where the shadow page is stored on disk. When a transaction page is swapped back in, the SIT entry is copied to the SPT entry corresponding to the newly allocated physical home page. If the SPT entry has a shadow page, it is also allocated a physical page, and its shadow pointer is updated in the home page's SPT entry.

3.5.2 Freeing Shadow Pages

Copy-PTM frees a shadow page when there are no more transactions using it, which is determined by the NULL TAV Link.

Similarly, for Select-PTM, a shadow page can be freed when there are no more transactions using it. That is, the page has only one version (committed version) for each memory block in the page. However, since the committed blocks can reside in both the home and the shadow page, we need to copy the contents from the shadow page back to the home page before we can free the shadow page.

We examined two different policies for freeing shadow pages for Select-PTM. One approach is to merge the home and the shadow pages together when the home page is swapped out by the operating system. To accomplish this, when a home page is swapped out, if it has a corresponding shadow page and there are currently no transactions using that page (determined by the NULL TAV link), then the operating system stores the valid blocks in the shadow page to the backing store location that is allocated for the home page. The SIT entry is updated to indicate that the page does not have a shadow page anymore and the selection vector is also cleared. This completes the process of freeing a shadow page.

Another approach to free a shadow page for Select-PTM is to lazily migrate the committed blocks to the home page. Whenever a non-speculative dirty block is written back to main memory, we can force it to be written back to the home page, even if the bit in the selection vector points to the shadow page. After writing back the cache block, the bit in the selection vector is toggled to indicate that the committed copy is in the home page. This allows the memory blocks to be gradually merged back to the home page when they are read and written. Eventually, when the selection vector is completely clear (all the blocks are now in the home page), the shadow page can be freed.

3.5.3 Shared Memory Inter-Process Communication

Since the SPT entry (or SIT entry) and the TAV list are maintained for a physical page (or a swapped out page) rather than a virtual page, conflicts between transactions executing in two different processes accessing the same physical page can be detected. Thus, PTM supports shared memory inter-process communication.

4. PTM Hardware Implementation

This section examines the hardware changes necessary to support PTM. We modify caches and the cache coherence protocol in a way similar to other hardware transactional memory models. In addition, paging and swapping are changed in ways that the operating system needs to be aware of. We also add hardware to the memory controller to cache the transactional state.

4.1 When Everything Fits in the Cache

Each processor core is largely unaware of the memory controller's PTM hardware. All requests for cache blocks use the home page address. Each core can detect transaction conflicts within its cache through the existing cache coherence mechanism. Each cache line contains a valid bit, coherence state bits to support MOESI, a Transaction ID, and bits indicating if the transaction read or wrote the block.

A transaction may complete without overflowing its cache. When a dirty block commits and it has never overflowed the cache, no work needs to be done by PTM. The block is just marked as non-speculative, and at that point it is treated as a normal cache block. It will continue to reside in the processor core's cache until the cache sets overflow or another core requests the block. When a cache miss results in a conflict with another block in a cache, we use a Virtual Transaction Supervisor (VTS) to arbitrate which transaction to abort. The aborted transaction's cached data is invalidated in the cache.

4.2 VTS Caches

In order for PTM to provide efficient unbounded transactional memory, we provide hardware support to make the following tasks efficient:

- **Fast Conflict Detection** - When a transaction scope has overflowed the cache we need a way to quickly determine a violation when processing a cache miss. We therefore cache in the memory controller, the summary information for the transaction blocks that have been read and written for recently accessed pages.
- **Fast Commit and Abort** - We need to have the ability to quickly commit or abort a transaction, and to let future execution continue, while the overflow data structures used by the transaction are cleaned up.
- **Fast Selection Between Home and Shadow Page for Select-PTM** - We need to be able to quickly choose between the home and the shadow page when fetching a block from memory. To achieve this, the memory controller caches the information needed to correctly choose between the home and shadow page for recently accessed pages.

To provide the above functionality, PTM uses a Virtual Transaction Supervisor, which is shown in Figure 2. The VTS is part of the memory controller for a snoopy architecture, and part of the directory controller for a directory based system. VTS has two main caches. A cache of the shadow page table entries and a cache of the current transaction access vectors. We describe them as if they are updated on-demand, but performance improvements can be had by prefetching data into the caches.

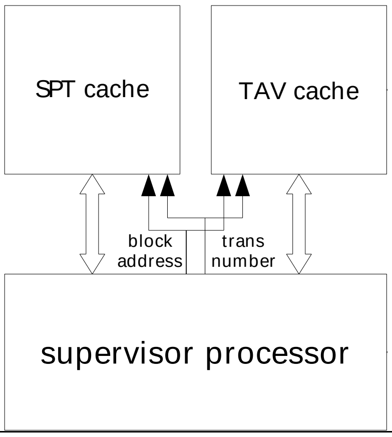

Figure 2. The Virtual Transaction Supervisor (VTS) has a memory backed cache holding the SPT entries and the TAV nodes.

4.2.1 Transaction Access Vector (TAV) Cache

The first cache is called the *Transaction Access Vector* (TAV) cache and is used to hold the nodes in the TAV lists in memory. An entry in the TAV cache corresponds to a TAV node shown in Figure 1. The TAV cache entry contains the read and write transaction vectors for a page accessed by a transaction. The TAV cache is indexed by the physical page number, and is tagged by the physical page number and the transaction ID. This allows multiple TAV nodes for the same physical page, corresponding to different transactions, to be stored in the cache at the same time. Indexing by the physical page allows PTM to quickly find all of the cached TAV nodes for that page.

The TAV cache is an important component in the PTM architecture for providing fast conflict detection. When there is a cache miss, the resulting memory request may need to determine exactly which transactions were prior readers or which transaction was a prior writer to the block. In this case, if the TAV nodes for the page of the block are found in the TAV cache, the read and write vectors for the page can be quickly examined to determine the conflicting transactions (if there are any).

When a TAV cache entry is evicted and the access vectors have been updated (the entry is dirty), the access vectors need to be written back to their corresponding TAV entries in memory.

4.2.2 Shadow Page Table (SPT) Cache

The second cache structure, called the *Shadow Page Table* cache, is used to cache the entries in the SPT structure, which was described in Section 3.2. The SPT cache is indexed by the physical page number, and is used to quickly determine conflicts.

When there are overflowed transactions being executed, we allocate an SPT cache entry for every non-transactional page and home page accessed. This is needed because non-transactional cache misses, which are executing while there are evicted transactional blocks, still need to be checked for conflicts. For non-transactional pages, the SPT cache entry allocated for it is used to quickly identify this.

The contents of an SPT cache entry is shown in Figure 3. An SPT cache entry contains the shadow page number (if there is a valid one). In addition, it contains a write summary vector and a read summary vector. The write summary bit vector for a page is an OR of all the transaction write access vectors that exist in the TAV list for the page. This provides immediate identification for a cache block that a transaction has speculatively overflowed that block. The read summary vector is a single bit vector where each bit indicates if there has been at least one overflow transaction read for that block.

For Select-PTM, the SPT entry also has the selection vector as shown in Figure 1. When an SPT cache entry is evicted, the corresponding selection vector in memory is updated if the SPT cache's selection vector is dirty. The SPT cache entry for Copy-PTM is the same as Select-PTM, but without the selection vector.

The SPT cache stores the information for the most recently accessed pages. A miss in the SPT cache requires the VTS to lookup the shadow page table to find the SPT entry, calculate the write and read summary vector from the TAV list, and then update the SPT cache. While the read and write vectors are calculated from the transaction's read and write access vectors for that page, TAV cache entries are created (if they do not exist) for each TAV node corresponding to that page.

4.3 Cache Eviction

When a cache block read/written by a transaction is evicted, the VTS takes action in response to the coherence message triggered as a result of the eviction. The coherence message will contain the physical address of the home page and is also piggy-backed with the transaction identifier. When a block is evicted, we do not need to check for a violation. We only need to check for a violation for the read or write cache miss. The following actions need to be taken on eviction.

When an *unmodified block* is evicted in the normal MOESI protocol, there is no need to generate a coherence message, but in our case, when a cache block read by a transaction is evicted it has to generate a coherence message to inform VTS to keep track of the overflow information. However, the data block is not written back because the cache block was not modified. When the VTS receives the coherence message for the unmodified transactional block, it will update the read transaction access vector in the TAV cache corresponding to the transaction that accessed the evicted cache block. Also, the read summary vector in the SPT cache for the physical page of the cache block is also updated. Note, when an unmodified non-transactional block is evicted, no coherence message is sent.

When a *modified transaction block* is evicted, we write the transaction access vector in the TAV cache and update the write summary vector in the SPT cache. If a shadow page has not been allocated for the home page, then one is allocated at this time. The modified cache block then needs to be written to the page that is supposed to hold the speculative version. For Select-PTM, the selection vector indicates which page (home or shadow) to write the speculative block to, and the write is done to the speculative location. For Copy-PTM, the block is always written to the home page. For Copy-PTM, we need to determine when we need to copy the non-speculative block to the shadow page on eviction. This is done by checking the write summary vector for the modified block being evicted. If the bit is not set, then this is the first modified overflow of that block, so we first copy the non-speculative block to the shadow page. We can then write the evicted block to the home page and set the write summary vector bit. If the bit is set, and there is no conflict, then we do not have to perform any copy, and the evicted block is written to the home page.

When a *modified non-transaction block* is evicted, we always write the block to the home page for Copy-PTM, and we do not need to do any SPT cache lookup. For Select-PTM, we first need to perform a SPT cache lookup, and use the selection vector to determine which page to write the block to, which is the non-speculative location.

4.4 Cache Miss

There are two operations that need to be performed on a cache miss. The first operation identifies from which of the two pages we need to fetch the data to serve the cache miss. The second operation detects any potential conflict. We initiate the fetch for the data block from memory in parallel with the conflict resolution and hold back the coherence reply with the data until the conflict is resolved.

4.4.1 Finding the Block to Fetch on a Miss

To fetch a block in Copy-PTM, we always fetch the block from the home page.

For Select-PTM, on a miss we need to look up the selection vector and write summary vector in the SPT cache. We XOR the bit in the write summary vector and the bit in the selection vector for the current cache block request, and the resulting bit value determines the page (home or shadow) we want to read the block from. This logic is shown in Figure 3.

4.4.2 Conflict detection

Read Miss: If the memory access is a read to a memory block, then there is a conflict only if there exists an uncommitted transaction that has modified the memory block (RAW conflict). To determine this, first we examine the bit in the write summary vector that corresponds to the memory block being accessed. If the bit is not set, then there is no conflict. If the bit is set, then there are two possible cases. Either the transaction that is currently accessing the block has itself modified the memory block in the past, or the block has been modified by another transaction. There exists a conflict only in the latter case. To determine which case it is, we look up the block's physical address with the current transaction ID in the TAV cache. If there is a match, then we check to see if the current transaction is the owner of the write. If so, then there is no conflict. If not, there is a conflict and we find the conflicting transaction. If we get a miss in the TAV cache, the VTS has to perform a hardware walk on the TAV list, starting from the shadow page table entry, to find out the conflicting transaction, and the TAV structures found are put into the TAV cache.

We assume MOESI protocol. In PTM, a transactional read miss request to a block that has already been overflown by a different transaction is not granted exclusive permission even if there are no sharers in the system (that is, no processor in the system has read permission). This is required because the transaction that gets the block might later write to it and at that time we have to make sure to resolve any potential conflict that may exist with that write. However, if there are no transactional read overflows to the block and if there are no other sharers in the system, then the read miss request can be granted exclusive permission.

Write Miss: If the memory access is a write to a memory block, then there is a conflict if there exists an uncommitted transaction that had read (WAR conflict) or written the memory block (WAW conflict). An SPT cache lookup is performed to examine the write and read summary bit vectors. If the write summary bit is not set, and if the read summary bit is not set, then we know there is no conflict.

If the write summary bit is set, we need to lookup the write access vector in the TAV cache to see who the writer was. If the TAV write vector shows that the same transaction was the prior overflowed writer, then there is no conflict. If not, then we know there is a WAW conflict, and one of the transactions must be aborted.

If the write summary bit is not set, but the read summary bit is set, then we look through the TAV list to see who the readers are. If the current transaction is the only reader, then there is no conflict, otherwise there is a WAR conflict, and one of the transactions has to be aborted.

4.4.3 Arbitration

When conflicts are detected, the oldest transaction wins the arbitration causing the younger conflicting transactions to abort, thereby guaranteeing forward progress, as any long waiting thread eventually becomes the oldest. Unique transaction identifiers generated se-

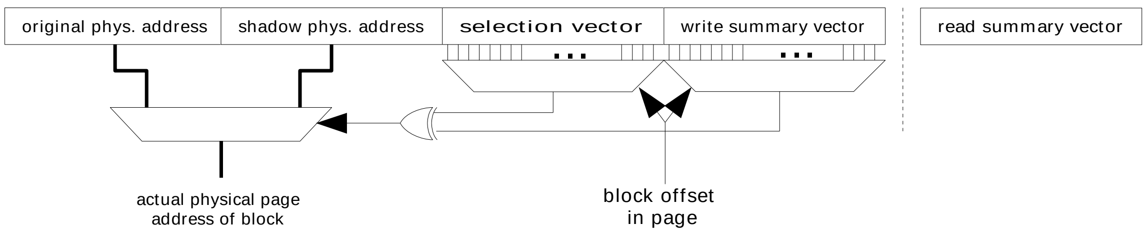

Figure 3. SPT cache entry. The SPT cache entry stores the selection vector, the write summary vector, and the read summary vector for a page.

quentially at the transaction start allows us to determine the age of the transaction. This also supports ordered transactions described in Section 2, by assigning the identifiers to match the program defined ordering. When a transaction is aborted and restarted, it maintains the transaction identifier that was originally assigned to it.

4.5 Commit and Abort

On commit, all of the cache blocks with the transaction ID are specified as no longer being speculative, and the transaction ID is cleared. On abort, all of the cache blocks with the transaction ID that are dirty are invalidated. Those that are not dirty just have their transaction ID cleared.

To process the PTM state on commit or abort, the VTS will first atomically change the status of the transaction in the T-State structure shown in the Figure 1. This is referred to as the logical commit/abort by VTM [15]. Once the transaction has been logically committed or aborted, the thread can continue its execution. The TAVs of the transaction are lazily freed on commit and abort. Before freeing a TAV node, we update the read and write summary vectors in the SPT cache as necessary. During this lazy commit, if another transaction accesses a "not-yet-committed" memory block (in cache or in main memory) it sees that there might be a conflict. However, while resolving the conflict, PTM knows that the conflicting transaction ID has already committed, when it looks up cached T-State structure in VTS. The transaction that has the outstanding miss is made to wait until the commit for that page finishes. After the commit for the conflicting transaction is over, the stalled transaction can continue its execution with the committed data block. After abort or commit, if the shadow page does not contain any committed blocks, then the shadow page is put on the free list and the SPT entry is updated.

For Select-PTM, as the TAV structures are committed for a transaction, the corresponding pages in the SPT cache and TAV cache are processed to correctly update the selection vector in the cache (if there is an SPT cache hit) and in memory (if there is a SPT cache miss). On abort, the selection vectors do not need to be update.

In the case of Copy-PTM, on abort we need to restore the original cache blocks that were overwritten by the transaction in the home page from the shadow page. We walk the TAV list and use the write vector to determine which blocks to restore from the shadow page to the home page. On commit, no data needs to be copied.

4.6 VTS Implementation for Snoopy-based and Directory-based Systems

To implement VTS as part of a snoopy architecture we integrate VTS into the memory controller. This is straightforward for a centralized controller, but it is also possible if there are multiple memory controllers. For multiple memory controllers, if the memory controllers are associated with particular regions of physical memory, this means a partitioned and distributed SPT cache and TAV cache. If instead the memory controllers are associated with particular cores rather than memory regions, this means distributed SPT and TAV caches with a dedicated coherence network among them.

For a directory protocol, the VTS would be distributed among the directories and implemented in the directory controller. Essentially, the SPT cache and the TAV cache in a directory will be caching the information corresponding to the physical pages maintained by that directory. The directory based VTS implementation requires some additional hardware support to perform arbitration to resolve conflicts. The additional support is required to ensure that all commits and aborts will be serialized correctly to guarantee atomic commit and aborts. Each directory entry has an overflow bit, which is set when the corresponding memory block overflows. Cache overflow due to a cache miss triggers a coherence request. When a cache miss coherence message reaches a directory, and if the overflow bit is set, the VTS associated with the directory is consulted to resolve conflicts. Thus, selecting between the home and the shadow page and resolving the conflicts can all be done as before. In addition, the shadow page for a home page is allocated so that they reside in the same directory controller.

Processing cache overflows and non-conflicting cache misses does not involve the supervisor processor, unless there is a miss in the SPT cache or TAV cache. If that is the case, then the supervisor processor needs to fill in the entries. The only other main functionality the supervisor processor does is to perform the TAV list walks on commit or abort. For snoopy and directory, we make sure that all of the TAV entries for a transaction are to the same memory/directory controller. An issue to keep in mind here is that the supervisor processor needs to have low enough occupancy to not become a bottleneck.

4.7 Efficient Context Switching

Context switches can be handled by just forcing an overflow of all the cache blocks read/written by a transaction. We assume physically indexed caches. Prior schemes like VTM [15] require the ability to translate the physical address to the virtual address as their overflow structures are virtually indexed. In comparison, PTM can update SPT entries and TAV entries using just the physical address.

On context switches, we avoid overflowing the cache blocks by tagging the transactional cache blocks with the transaction identifiers. In this case, the normal cache coherency conflict detection mechanism will be able to identify conflicts with the cache blocks that were not overflowed when the transaction was context switched out.

When a transaction begins, PTM takes a checkpoint of the architectural register states in the processor so that on an abort they can be restored. To support context switches for a transaction, we save and restore the transaction's checkpointed register state. In PTM, the T-State, which contains an entry for each transaction is used to save the checkpointed register state of a transaction when it is context switched out.

5. Prior Hardware Unbounded Transaction Work

In this section, we describe related work, and provide a detailed description of VTM, which is the prior unbounded transactional memory approach that we compare against.

5.1 Related Work Outside of Transactional Memories

Chang, *et al.* [2] introduced support for efficient locks in hardware. A bit or a bit-vector is associated with each page by extending page tables and the TLB. A memory word is locked by setting its corresponding bit. The primary similarity between PTM and the IBM 801 processor is that PTM also associates a bit-vector with each transactionally touched page. However, the PTM extensions are used for supporting unbounded transactions, as opposed to locks.

5.2 Related Work on Unbounded Transactional Memory

Recently, many hardware transactional memory techniques have been proposed [6, 7, 9, 13, 3, 1, 15, 14]. Due to space limitations, we only provide a summary of four prior techniques that focused on large or unbounded transactions: UTM [1], VTM [15], LogTM [14], and LTM [1].

UTM was one of the earliest approaches to completely support unbounded transactions. UTM uses its *XState* data structure to log all transaction-related information. Each memory block has a *log pointer* associated to the list of transactions that accessed it. All writes done inside a transaction modify the memory in place, storing a copy of the old non-speculative value in the *XState Log*. This approach makes abort a costly operation, though commit can be done very efficiently. UTM requires multiple memory lookups to traverse the log pointer on abort, since it does not cache the log entries, although it could potentially do so. The UTM approach can support most system events, including overflows, context switches, process migration, and paging. Their approach requires significant hardware changes including globally unique virtual addressing.

LTM [1] supports reasonably large transactions with the memory footprint size comparable to that of physical memory. LTM uses the memory coherence protocol to detect conflicts. It uses an overflow bit in caches to let the coherence protocol know if there is potentially a conflicting overflowed transactional block. LTM stores all the overflowed speculative values in a memory-based hashed data structure until the transaction commits. This approach results in an efficient abort operation, but the commit operation can incur high overhead as the new values need to be copied from the backup structures to their corresponding memory locations. LTM can avoid conflict-detection overhead for non-overflowed blocks using its overflow bits, but it must do multiple memory lookups to resolve conflicts for the overflowed blocks. LTM cannot support transactions longer than a time slice or with footprints larger than physical memory.

LogTM [14], like LTM [1], supports reasonably large transactions that fit in the physical memory. LogTM uses a directory-based coherence protocol for conflict detection. It makes in-place memory updates for overflowed speculative values and hence, abort can potentially be a high overhead operation. Also, aborts are handled in software with LogTM, which makes them costly. LogTM can do efficient conflict detection using the directory state, but it requires that transactional state is never paged out. The LogTM approach does not handle thread migration, context switches and paging. To ameliorate the abort cost, LogTM stalls the transaction whenever possible instead of aborting it.

5.3 VTM

In this paper we compare our approach to VTM [15]. The VTM approach provides an efficient and nearly complete handling of unbounded transactions. The key structures needed to implement VTM are an in-cache hardware transactional memory system, and a set of hardware and software structures to handle transactional overflow and context switching. VTM is oriented towards in-cache TM with eager conflict detection, but is otherwise mostly agnostic about the particulars of the in-cache hardware transactional memory system.

The software structures for VTM consist of transactional state information (XSWs), a table tracking overflowed blocks and their original values (XADT), an overflow counter, and a counting Bloom Filter (XF). Unlike PTM, the addresses tracked by VTM for overflowed blocks are virtual. Instances of the software structures reside in the virtual address spaces of each transactional application, and are shared among the threads. The hardware structures needed for VTM are an XADT walker that performs lookups on overflowed state in the XADT and walks the XADT on commit and abort, and a cache of meta-data for overflowed blocks, called the XADC. The bloom filter XF is used to reduce the frequency of having to access the XADT when doing conflict detection. A set of counters in the XF will be incremented when a cache block is overflowed, and are decremented lazily during commit or abort. A value of zero means that there is no overflow block, and a non-zero value means that there may be an overflow block.

The XADT log table contains the virtual addresses, transaction state, and data of the overflowed cache lines, buffering all speculative state. VTM uses the old value of the transaction-modified memory, also stored in the XADT, to detect non-transactional code interaction with transactional code. Whenever a transaction encounters a read or write miss, the XF will be consulted to determine if the memory block being accessed may have been overflowed in the past. If so, the corresponding entry, if any, in the XADT will be looked up to resolve the potential conflict. VTM accesses the XADT via the XADT hardware walker.

If no blocks are currently overflowed, then conflict detection beyond the in-cache mechanism consists only of checking the overflow counter. When there are overflows, VTM can avoid the overhead of performing conflict-detection for addresses that have never overflowed by filtering out queries to those addresses using the XF, but it requires XADT look-ups to resolve conflicts for overflowed cache blocks. We model using an XAD Cache (XADC), similar to the cache structure used in PTM. The XADC stores the meta-data for the most recently accessed evicted transaction blocks, and a pointer to the XADT structure for that block in memory. The meta-data describes what transactions have read the overflowed block, and, if the block was dirty, which transaction wrote it. When a query to the XF says that there may be an overflowed block, we look up the block being loaded in the XADC. If there is a hit, then we have all of the information to determine if there is a conflict, and a pointer to the data blocks in memory to load the speculative block if needed.

VTM stores the new speculative value in their overflow data structure and the memory is updated on transaction commit. This allows fast aborts, but results in memory-copying overhead at the time of commit. VTM can hide some of this cost by doing a lazy commit, but the memory updates still consume bandwidth, and all the transactions that need to access a memory block modified by a committed transaction, but yet to be updated in memory, have to stall. In comparison, PTM does not involve any data movement at the time of commit. One can potentially change VTM to write speculative blocks to memory when a block is evicted, and only store the non-speculative block in the XADT structure. This would make it similar to our Copy-PTM approach, but with XADT and XF structures, and an approach along these lines was proposed by Zilles and Baugh [18].

VTM virtualizes the execution of transactions across most system events, which include cache overflows, context switches, process migration and paging. However, they require that the cache blocks touched by the transaction be evicted from caches and invalidated before the transaction is context-switched out. Further, VTM needs to record virtual addresses for locally cached transactional blocks so that it can do the reverse address translation from physical address to virtual address. This enables VTM to evict all the cache blocks read or written by a transaction that is being context-switched out. In contrast, we propose the use of additional tags for cache blocks to support context switches without unnecessarily evicting transactional cache blocks as explained in Section 4.7. Even if we do not use the tags, flushing the blocks touched by a transaction entry is simple, as the PTM structures are indexed by the physical addresses. Therefore, for physically tagged caches we do not have the complexity of doing reverse address translation.

As the VTM data structures are held in the private address spaces of processes, VTM cannot offer transactional guarantees for inter-process communication through shared memory segments mapped to different virtual addresses. Our PTM technique, on the other hand, is built on the top of the existing virtual memory and hence can provide virtualization across processes.

5.3.1 Modeling VTM

In order to evaluate the performance of PTM, we constructed a VTM model based on the description in [15]. We use the same in-cache hardware transactional memory model for both PTM and VTM. This is a more optimistic model for VTM than that featured in [15]. We assume the presence of transaction IDs in the cache, which can be used to avoid having to flush all transactional data on every context switch. We also assume for the VTM model that the XF counting Bloom filter has been implemented in dedicated hardware. We model an XF with 1.6 million entries. We also assume an XADC to cache the meta-data for the overflowed blocks.

When checking for conflicts, if all of the block's XADT entries have their meta-data cached in the XADC, then the conflict resolution is done in the time it takes to do the cache lookups. If there is an XADC miss, it requires a reconstruction of meta-data via traversal of the XADT, similar to creating a SPT cache entry from our TAV structures for PTM. When walking the XADT for commit or abort, we assume that each XADT entry lookup requires a single main memory access, and that the number of memory accesses is equal to the number of XADT entries traversed.

VTM, like PTM, supports a lazy commit, changing the status of a transaction atomically via an atomic memory operation on the transaction's status word XSW and updating all other data and structures lazily. However, since it has buffered all overflowed speculative values in the XADT, VTM must actually copy the speculative data to the original memory location on commit. This occupies bus resources, even when doing the commit lazily. As bus contention in our memory model leads to performance degradation, we also consider adding data buffering to the XADC to hold the speculative and non-speculative block in addition to the meta-data. Because this secondary cache acts like a victim cache, we refer to this variant as Victim-VTM (VC-VTM) in our results, with the baseline VTM labeled simply as VTM. Blocks in the victim cache are marked as being committed instantly, and later written back to memory when evicted from the cache. Currently executing transactions can then use the blocks found in the victim cache, instead of having to wait for them to be committed. We found this to significantly reduce the commit delay penalty for VTM.

6. Results

This section evaluates the performance of PTM, demonstrating that it efficiently supports virtual transactional systems without incurring high overhead. For this evaluation, we used programs from the SPLASH-2 [17] benchmark suite to evaluate PTM.

6.1 Simulation Platform

We modeled a CMP system using Virtutech Simics [12] based on Enterprise machines running RedHat Linux 7.3, and extended the model to simulate PTM and VTM. The entire system has 4 nodes, each with two levels of private cache. The L1 cache is 16KB direct-mapped with a 1-cycle latency, while the L2 cache is 256 KB 4-way set associative with 6-cycle latency. Coherency is maintained at the L2 cache using a snoopy-based MOESI protocol. The augmented L2 cache blocks contain transactional read and write bits that are used to track transactional read and write accesses similar to prior work [8, 14]. In addition, each cache block contains a transaction ID, a valid bit and the bits to implement MOESI protocol. Each node in the system is a single-issue in-order pipeline. We simulate a 512 entry fully associative TLB where each page is of size 4 KB.

We added features to Simics to support a transactional memory system. In particular, we modified the Simics instruction decoder to recognize the instructions *Begin* and *End*, which are used in the program to specify the begin and end of the transactions respectively. Our simulation of PTM and VTM assumes that the processor has a fast register checkpointing mechanism.

The Chip-Multiprocessor (CMP) memory hierarchy is supported by a high speed on-chip bus and a low speed main memory bus. We simulate a high speed on-chip bus connecting the four CPUs and the on-chip memory controller with a minimum round-trip latency of 20 cycles. The memory controller contains the PTM caches and the ancillary hardware. In the MOESI protocol that we model, a cache miss request can be sourced from other caches containing a valid copy instead of having to access the much slower external memory. We assume access to main memory has a minimum latency of 200 cycles, but up to three requests can be pipelined simultaneously.

The PTM hardware in the memory controller handles transactional coherence requests using the SPT cache and TAV cache to speed up the process. We simulate a 512 entry SPT cache and 2048 entry TAV cache. Both are fully associative. A miss requires that we access the shadow page table in memory. From the shadow page table we can get access to all of the TAV structures for that page. To ensure fairness, in our simulation of VTM, we use an XADC [15] of capacity equal to the combined capacities of the SPT and TAV cache. The victim cache, and the hardware resources to implement it, are used only for the Victim-VTM results. Those extra hardware resources not used by PTM.

6.2 Characterizing Transactional Applications

We studied the behavior of the transactional memory regions by using Splash-2 [17] programs. We first removed all the locks from the programs. We then parallelized each program using transactions. We made use of two instructions, `Begin` and `End`, which specify the begin and end of a transaction respectively. To parallelize the code, we focused on creating critical transaction regions similar to how the average programmer might go about doing this. We wrapped each loop body with a transaction, so that each iteration of the loop can be executed in parallel. If there are loop carried dependencies, we used ordered transactions to enforce correct dependencies.

Various program characteristics relevant to PTM are presented in Table 1. The second column in the table indicates the number of committed transactions per application, and the third column presents the number of aborted transactions. Both these results demonstrate the significant amount of transactional activity in our benchmarks. We present the results for system effects in the fourth and fifth column of Table 1, listing the number of exceptions and context switches seen by the program – the fact these system effects exist is a motivation for our proposal's support for virtualizing unbounded transactions.

The sixth column, titled "pages", presents the memory footprint in terms of the number of unique pages accessed during the course of

Application	Transactions		System		Memory				
	commit	abort	exception	context-switch	pages	pg-x-wr	conservative	ideal	mop/evict
fft	34	5	595	52	1041	551	52.9%	9.5%	87.5
lu	656	0	17754	1079	2311	2130	92.2%	3.6%	95.3
radix	70	17	615	116	771	629	81.6%	2.0%	246.3
ocean	877	282	7417	1421	14966	6769	45.2%	0.2%	15.8
water	59	8	32	127	241	110	45.6%	2.6%	4926.3

Table 1. Transactional memory execution behavior for loop regions in the SPLASH-2 programs. The entries in the table are organized in three sets. The first set describes the transactional behavior of the applications, the second set describes the system behavior, and the third set provides information about the memory footprint of the transactions.

entire program execution by both transactional code as well as non-transactional code. This does not include the shadow pages used. The seventh column "pg-x-wr" shows the total number of unique pages updated by just the transactional writes.

We estimate the worst case upper bound on additional pages allocated due to allocation of the shadow pages. The upper bound is shown in column eight with the title "conservative". The upper bound is computed as the fraction of transaction's footprint (shown in column six) and the entire program execution's footprint (shown in column seven). The column "ideal" shows the percent increase in the number of pages if all of the shadow pages created for a transaction were instantaneously committed or garbage collected when a transaction commits. To calculate this number we determine the average number of pages that are live at any instant for the transactions. We treat this number to be the additional number of shadow pages that are live at any instant and calculate the increase in page overhead accordingly.

The last column "mop/evict" in the table describes the frequency of cache block evictions. The results are shown in terms of how many memory operations occur between evictions. For example, `radix` shows that it evicts a block every 246 memory operations. This is one measure of how much work the overflow transactional memory has to perform. In the worst case (`ocean`), we see that a cache block is evicted for every 16 memory operations.

6.3 PTM Performance Comparisons

To determine the usefulness of the proposed PTM, we simulated the performance of PTM comparing it against the prior technique VTM and lock-based multi-threaded execution. Figure 4 shows the speedup over a single thread of execution for five SPLASH-2 benchmarks. In this and our other figures, we abbreviate Select-PTM as *Sel-PTM*. We first show the speedup of using the default p-thread locks. Using fine grain locks we can achieve a speedup of 134% on average. This approach does not have the overhead of the transactional execution, speculative aborts, and the overhead of buffering the overflowed blocks, although lock-based execution lacks the deadlock-free execution guarantees that transactional memories provide.

The baseline VTM shows decent speedups for three of the benchmarks, but we do not see any speedup for VTM on `fft` and `ocean`, due to the overhead of commits. In comparison, if a victim cache is used with the XADC to hold the recently evicted transaction blocks, we achieve speedup for all benchmarks over single threaded execution. This is because currently executing transactions can access the overflowed but not-yet-committed blocks from the victim cache, while those blocks are being committed. Thus, for VC-VTM we see an average speedup of 72% reflecting the benefits of overlapping execution with physical commit to reduce the commit cost.

Our results for Copy-PTM show an average speedup of 116% and for Select-PTM we observe 220% speedup. The difference between the two is directly attributable to the additional overhead Copy-PTM incurs for copying blocks to the home block on evictions and restoring them on aborts. Note that we do not use a victim cache

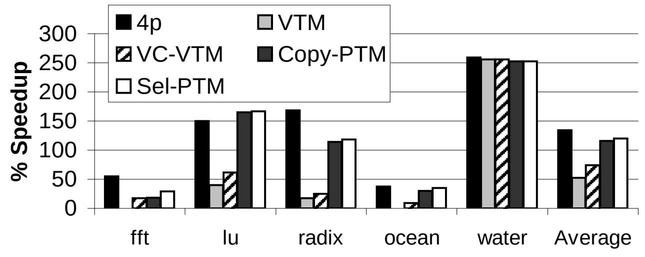

Figure 4. Comparing TM speedup for lock-based multithreading, (base) VTM, Victim-Cache VTM, Copy-PTM and Select-PTM. Speedup is over single threaded execution.

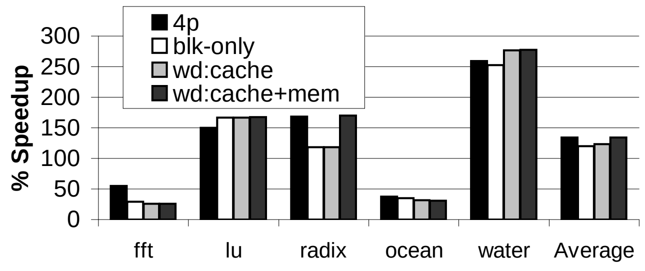

Figure 5. Advantage of conflict detection at the word granularity.

for the PTM results. One of the main differences between VTM and Copy-PTM is that Copy-PTM incurs a penalty on abort, whereas VTM incurs a penalty on commit. In the future we plan to compare against a variant of VTM that does in-place speculative updates, so that the main penalty is due to abort and not commit. We expect this approach to perform closer to Copy-PTM.

Since coherence is done at the cache block granularity, there can be false conflicts detected due to false sharing. This can lead to unnecessary aborts, which incur extra run-time overhead [13]. It has been shown that this overhead can be reduced by for tracking conflicts at the word granularity [9, 8].

For the results we discussed thus far, we used a cache block of size 64 bytes. Let us say a transaction read/wrote to one of the words in the 64 bytes, and then it was followed by another transaction that tried to write to a different word in the same 64 byte cache block. Clearly, there was no conflict. However, our conflict detection mechanism based on block sized coherence messages and PTM data structures would detect a false conflict and unnecessarily abort one of the transactions, because the conflict mechanism operates at the cache block granularity.

Figure 5 shows the performance of modeling conflicts at the word granularity compared to Select-PTM. Results are compared against only using block granularity `blk-only`, and using p-threads locks. The first approach we examine, `wd:cache`, performs cache coherence at the word granularity, but still keeps track of transactional information for overflowed blocks at the block granularity (64 bytes). As a result, this leads to more coherence traffic, which we modeled, and also adds additional complexity to a directory system. This resulted in only minor speedups, because evicting a block with multiple writers would cause an abort, since the overflowed PTM structures would only kept track of one writer per block.

We then examined keeping track of transactional information even for the overflowed blocks in PTM at the word granularity, which is `wd:cache+mem` in Figure 5. Resolving conflicts at the granularity of words is useful especially for programs such as `radix` and `water`. For `radix`, this resulted in 170% speedup over single threaded execution, which is a significant improvement over 116% speedup from tracking all conflicts at the block level.

While the problem of false conflicts due to detection granularity is highly benchmark dependent and not universal, it does affect programs like `radix` dramatically. Techniques explored in prior work should help reduce false conflicts, either by changing data structure alignments [16] via the compiler, or allowing more than one processor to own sub-partitions of the cache block [4].

7. Conclusion

With the advent of multi-cores, extracting task level parallelism is going to be crucial. To meet this goal, transactions can help common programmers to write multi-threaded programs. However, support for unbounded transactions is crucial to develop a good transactional programming model.

In this paper, we proposed a system design called PTM that extends existing virtual memory support to support unbounded transactions. In PTM, when a transactionally modified cache block is evicted, we allocate a shadow page, which can be used to hold the speculative block. In addition, we aggregate and maintain all of the transactional information on a page-level granularity. The PTM structures are integrated with the virtual memory system, allowing direct access to the transactional data for a page with both the virtual and physical address of the page.

The first approach we examined is Copy-PTM, in which on a transactional dirty block overflow, a copy of non-speculative block is first backed up in the shadow page. On commit, the backed up copy can be discarded, but on abort it has to be restored in the home page. This allows commits to be fast, but aborts can be slow. We optimized this design in Select-PTM, where the two versions of data are allowed to be spread across the home and the shadow pages. To determine which of the two pages contain the block to be fetched, we used a selection bit vector. Select-PTM is efficient for performing both commit and abort operations, as it does not have to physically copy the data between the two pages. Also, on dirty block eviction the non-speculative data need not be backed up.

Our PTM solution integrates well with the virtual memory and paging subsystems, and is a promising option for supporting unbounded transactional memory.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. This work was funded in part by grants from ST Microelectronics, Microsoft, and Intel.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, 1988.

[3] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[4] C. Dubnicki and T. J. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992.

[5] J. N. Gray. *Operating Systems: An Advanced Course*, chapter Notes on Database Operating Systems, pages 393–481. Springer-Verlag, Berlin, 1978. R. Bayer, R. M. Graham, and G. Seegmuller, editors.

[6] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *Mico's Top Picks, IEEE Micro*, 24(6), nov/dec 2004.

[7] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 32(5):58–69, 1998.

[9] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI-06 Programming Languages Design and Implementation*, pages 14–25. ACM Press, 2006.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[12] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[13] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005.

[14] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

[16] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Computers*, 43(6):651–663, 1994.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36. Association for Computing Machinery, 1995.

[18] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and nontransactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.