Hardware Transactional Memory with Operating System Support, HTMOS

Sasa Tomic, Adrian Cristal, Osman Unsal, and Mateo Valero

Barcelona Supercomputing Center
Universitat Politècnica de Catalunya

Abstract. Hardware Transactional Memory (HTM) gives software developers the opportunity to write parallel programs more easily compared to any previous programming method, and yields better performance than most previous lock-based synchronizations.
Current implementations of HTM perform very well with small transactions. But when a transaction overflows the cache, these implementations either abort the transaction as unsuitable for HTM, and let software takeover, or revert to some much more inefficient hash-like in-memory structure, usually located in the userspace.
We present a fast, scalable solution that has virtually no limit on transaction size, has low transactional read and write overhead, works with physical addresses, and doesn't require any changes inside the cache subsystem.
This paper presents an HTMOS - Operating System (OS) and Architecture modifications that leverage the existing OS Virtual Memory mechanisms, to support unbounded transaction sizes, and provide transaction execution speed that does not decrease when transaction grows.

1 Introduction

1.1 Motivation

Transactional Memory (TM) systems can be subdivided into two flavors: Hardware TM (HTM) and Software TM (STM). HTM systems bound TM implementations to hardware to keep the speculative updated state and as such are fast but suffer from resource limitations. In this work, we propose Hardware Transactional Memory with Operating System support (HTMOS) which is complexity-effective, potentially performs on the same order of magnitude with HTM, and is flexible like STM systems. We present a fast, scalable solution that has virtually no limit on transaction size, does not prefer either directory based coherence or snooping, that has low transactional read and write overhead, that works with physical addresses, and does not require any changes inside cache subsystem. Instead, changes are done on the Operating System (OS) level - in the Virtual Memory system, and inside the processor - the TLB and in the form of additional instructions/functionality.

HTMOS involves modest architectural and OS changes to keep the additional copies of a page in memory for transactional memory. Compared to the

previous HTM proposals, HTMOS has three important advantages: (1) implicitly accommodates large transactions without getting bogged down with complex implementations which decreases the performance (as is the case for most HTMs). In fact for a 4GB address space, a transaction could be as large as 2GB. (2) is much more flexible than other HTMs. For example, different versioning schemes such as in-place update versus lazy update are trivial to implement. Most other HTM proposals embed a certain versioning scheme in silicon which makes it very difficult to implement other alternative schemes. (3) ensures strong atomicity. Non-transactional loads and stores do not conflict with transactional ones. More specifically, each non-transactional load or store can be seen like a single-instruction transaction to the transactional ones.

1.2 Previous work

Almost all current HTM implementations assume that transactions are going to be very small in size. Our assumption is that ordinary programmers will try to use transactions whenever they are not sure if they should use them, and for as big segments of the program as they can.

In the current HTM implementations, there are generally two approaches to the version management: lazy and eager, and two for conflict detection: lazy and eager. Two representatives for these are LogTM[5], from University of Wisconsin, with eager-eager, and Transactional Memory Coherence and Consistency[3], from Stanford University, with lazy-lazy conflict detection and version management. According to some researchers, the overall performance of LogTM is a little bit worse than that of TCC[1].

In a recent proposal, Unbounded Page based Hardware Transactional Memory[2] (PTM), the conflict detection is done on the level of cache-line sized blocks, and it supports unbounded transaction size. All of the transactional state is indexed by physical pages, and is maintained at the memory controller. PTM's Transaction Access Vector (TAV) double linked list tracks the accesses to a page. One shadow page for every physical page is created, which requires the eager conflict detection. The memory controller is responsible for all conflict detection, updating transactional state, and aborting/committing transactions. On abort or commit, the memory controller updates the TAV and the special summary cache for this transaction. Transactions are nested by flattening. The cost of every miss to the TAV cache increases linearly with the length of the TAV list. The length of TAV lists increases with the number of processors (transactions) accessing different blocks of the same physical page. So, if there are many (N) processors, each non-cached memory access would require N memory reads. To avoid the high cost of non-cached access, the TAV cache that was used in evaluation is fully associative 2048 entries, which is very difficult to be done using current technology.

2 HTMOS Architecture

HTMOS leverages the strong coupling between the Architecture and OS to enable the HTM design that is both fast and flexible. The core idea is to create an additional, *secondary* copy of the page, for every transaction, in case when transaction is trying to write to the page. Each transaction has its own transactional *Virtual Page Table* (VPT), which can be utilized to switch between the alternate versions of the block inside the page. Each page is subdivided into blocks of cache-line size, to reduce the granularity for conflict detection. Assuming eager conflict detection implementation, the current value of the data is in the *primary* copy of the page, so only the record-keeping information needs to be cleaned. On abort, the original values are copied from the secondary copy of the page to the primary, and record-keeping information is cleaned. The detailed explanation follows, divided into Software and Hardware sections.

2.1 Software

Operating System manages the memory required for the transactional bookkeeping. It also allocates the secondary pages used for storing the backup copies of the cachelines.

Global Transaction State Table (TST) OS has a special area of memory allocated for the TST. There is a fixed number of transactions running at the same time. This number is fixed by both hardware and software. In our implementation, we use so called transaction flattening: nesting counter is incremented on begin of a nested transaction and decremented on commit of a nested transaction. The real commit is done only when nesting counter reaches zero. Therefore, it is sufficient to have the maximum number of concurrently running transactions equal to number of processors in the system.

The global TST (Tab. 1) holds the basic information about all transactions. Every processor has access to each element of this table and, therefore, can read or set status of any transaction in the system. The table has as many entries as there are processors/transactions in the system. We will assume a 32 processor system.

Each entry of the table is extended with (currently) unused bits up to cacheline size, to minimize the false-sharing between processors.

On system startup, the OS also creates a 32 transactional VPTs. Transactional VPT holds the physical addresses for the secondary pages of this transaction.

Transactional Bitmap (TB) Transactional Bitmap is the key structure of our HTM implementation. This sparse bit-array is permanently stored in the physical memory, and holds the information about the transactional reads and writes from and to the page that it is associated with. It exists *only* for the pages that have the transactional reads and/or writes.

size	field name	possible states
52b	tx_vpt	pointer to transactional vpt
1b	active	INACTIVE / INSIDE_PROC
2b	status	RUNNING, COMMITTED, ABORTED, FREE
9b	nesting depth	0..511
64b	ws_size	$0..(2^{64}-1)$
32b	tx_blocked	bitmap of TXs blocked on this one
32b	thread_id	thread_id that is running transaction
320b	unused	-

Table 1. An entry of Transaction State Table, assuming cache-line size of 512b (64B)

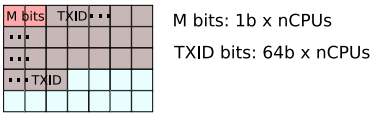


Fig. 1. Transactional Bitmap organization for *nCPUs* processors

TB is organized in the following way:

For every block in the page (standard page of 4KB is split into 64B standard cacheline-sized blocks) we have a bitmap in the following format: **1 M bit**: marks if this block has been **M**odified by some processor, **32 TXID bits** (1 bit per processor): marks which processors have read the value, or in case when M=1, then which processor (only one) is holding the modified value. This makes a total of 64 x (1+32) bits = 2112bits = 264B associated with each transactional PTE, and this is enough to cover up to 32 processors, with concurrent access to each of the 64B blocks in the page.

As can be seen on Fig. 1, when stored in the memory, M bits are grouped into one 64-bit M field, and all TXID bits are grouped in a field after that.

The motivation for the addition of the TB is the *reduction of conflict detection granularity*. The best granularity for the conflict detection is word-size, but false-conflicts are mostly tolerable[2,4] if the granularity is 64B (standard cache-line size). Therefore, we split the page of 4KB to 64B blocks.

This bitmap can be located in the memory separately from the page table. The TLB inside the processor is extended to also hold the address of this bitmap, associated with every page. On the first transactional read or write to the page, if the address of the TB for the page is uninitialized (equal to zero), the processor interrupts the OS and signals that it needs the TB for the page. The OS allocates the space and loads the new TLB entry into the processor, that is now also holding a pointer to the TB. From now on, the processor reads and updates this TB on transactional access to the page blocks.

The total occupied space by both TST and TB grows linearly with the number of processors in the system, and this dependency can be seen graphically in the Fig.2.

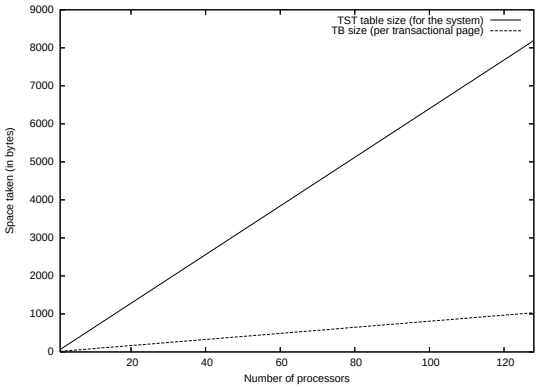


Fig. 2. Space taken for the transactional bookkeeping as a function of the number of processors in the system

As an example, let us assume non-conflicting transactional read and write to a block by e.g. processor 3. After reading from this block, the 3rd bit of TXID for this block will be 1. After writing to the block, the M bit will be 1 and 3rd bit of TXID for this block will be 1. With this simple approach it is easy to quickly detect the conflicts later.

The M and TXID bits are consulted on every (transactional or not) read or write to the block.

TB is actually associated with the physical page in the system, but is accessed only from the virtual addresses, during the translation from the virtual to the physical address. Therefore, we don't need the same number of the TBs as the number of transactionally accessed physical pages in the system. Additionally, many virtual pages can point to the same Transactional Bitmap. This allows inter-process shared memory communication, where many processes share the same TB for different virtual address spaces.

2.2 Hardware

Translation Lookaside Buffer Each entry in the TLB inside the processor is extended with one **additional bit, T**, that is actually appended to the virtual address on every TLB lookup (Fig. 3). This bit signifies if the TLB entry holds

Fig. 3. New TLB entry

the primary (T=0) or the secondary (T=1) copy of the page. The processor, itself, knows whether the lookup it wants to make is the transactional one or

not. One more addition to each entry are the **pointer to the Transactional Bitmap, TBp**, and the value of **TB**, whose functionalities are explained in 2.1.

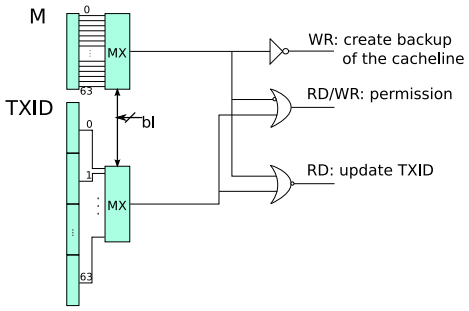


Fig. 4. Additional hardware per TLB entry, for reading and processing of M/TXID bits

A specialized hardware that can be used to process the M and TXID bits, and for conflict detection, can be seen in the Fig.4. The multiplexer for the M bits selects the proper bit with the value of **bl**, the cacheline block offset inside the page. The TXID is the array of bits, with 64 chunks of 'number of processors' bits. From each of these chunks, for conflict detection each processor is using only one bit, which is hardcoded to the multiplexer input. The multiplexers with 64 input bits and one output bit, can be multilevel to reduce the fan-in.

Effect on non-transactional reads and writes The OS flushes the TLB entry from all processors that might hold it, in the case when a pointer to TB (TBp) is changed. This ensures strong atomicity. On a conflict between non-transactional and transactional code, processor running non-transactional code determines that the transaction created a conflict, and either sends a kill_tx (abort your transaction) message to a remote processor if conflicting transaction is INSIDE_PROC or raises an ABORT interrupt if the transaction is INACTIVE.

3 Transactional Access

3.1 Begin Transaction

The transaction begins with the call to the ISA instruction (see Table2) **btx TSTp**, where TSTp is the address of the first entry in the Transaction State Table. Each processor has a unique number, CPUID or TXID, that in our implementation goes from 0 to 31. The processor locates its entry in TST, and sets the value of the *active* field for this processor as INSIDE_PROC, and the *status*

field of the transaction as RUNNING. Then it increments the *nesting depth*. After this the processor effectively enters the transactional mode and all memory reads and writes are implicitly transactional.

When the processor changes priority level (e.g. from user mode changes to kernel mode or vice-versa), or when it enters a fault: interrupt, page fault, etc. the *active* field is automatically set to INACTIVE. This is done by writing to the memory bit. Most of the time this will be just a private cache write, unless this TST entry had to be evicted.

On subsequent calls to the **btx**, the processor simply increments the *nesting depth* of the transaction.

begin transaction	btx TSTp
commit transaction	ctx TSTp
clean TB for the TXID	clean_tb vaddr, txid
unblock TXIDs blocked on the given TXID and set the TST free for the given TXID	finish_tx txid
undo the writes and clean the TB for the given TXID	undo_tx vaddr, txid

Table 2. ISA extensions

3.2 Transactional Read

On every transactional read, the processor consults the TLB entry for the page for the permission to read. If the TBp is zero (i.e. there is no TB associated with this virtual address), the processor raises an interrupt to the OS to create it. OS allocates the space for the TB, then loads the TBp and the TB into the TLB entry and returns to the same instruction that raised the interrupt. A non-transactional read would not raise an interrupt when the TBp is zero in the TLB entry. This is the only difference between the two of them. If there is non-zero TBp, it is obeyed in both cases.

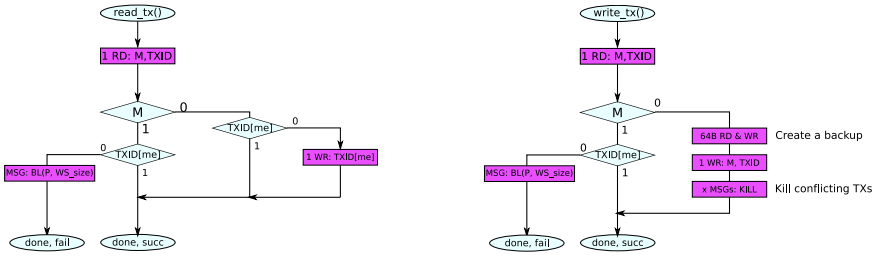
For avoiding the potential race condition when multiple processors wants to read/write to the same block, testing and setting of M and TXID bits should be done atomically.

If **M[bl]=1** for the cacheline, some processor transactionally wrote to it. The read is either permitted or denied, depending on the TXID bits for the cacheline. If some other processors transactionally wrote to this address (bit for this processor is not set), the conflict resolution protocol takes place and read is denied.

If **M[bl]=0**, read from the cacheline is allowed. If needed (determined by the TLB hardware), the TB entry for the cacheline is updated.

When the processor (e.g. P0) detects a conflict with other processor (e.g. P1), it first locates the P1's entry in TST, then reads the *active* bit of TX1. If TX1 is INACTIVE (e.g. P1 is inside the trap), then P0 sets the *status* of P1 to ABORTED and calls the OS function to initiate the abort of P1's writes. If TX1

has *active* flag set to INSIDE_PROC, then P0 sends an inter-processor interrupt (IPI) to P1 **block_tx(my_writeset_size)**, and waits. Upon receiving the interrupt, the processor P1 compares the size of the write-set of the other transaction with the size of the write-set of himself, and based on that decides which one is going to proceed. For more details about the protocol see 3.6.
Overhead of each transactional read:



(a) The overhead of the transactional read

(b) The overhead of the transactional write

Fig. 5. The overhead of the transactional access

The cost of the transactional read is not uniform as can be seen in Fig. 5a. The first read from the transaction is slowed down by one additional write to TB, and every transactional read after that is slowed down by only one additional read to the Transactional Bitmap, which should be located in cache of the processor.

3.3 Transactional Write

On every transactional write, the processor consults the TLB entry for the permission to write. For avoiding the potential race condition when multiple processors wants to read/write to the same block, testing and setting of M and TXID bits needs to be done atomically.

If the block was *not* modified (**M bit is zero**), then the processor has to copy the current value of the cache line to the secondary page and set the M and its TXID bits for this block. It also needs to inform other processors, that have read from this block, that they need to restart their running transactions, by sending the **kill_tx** (abort your transaction) to each of them or raising the interrupt ABORT TXn (see 3.5) in case they were *INACTIVE*. Then it writes to the destination cache line and increments the private *writeset_size* variable, used for the conflict resolution.

If the **M bit is one** and the *TXID is not equal to the ID of this processor*, a **conflict resolution** protocol needs to applied. The processor sends the other processor an interrupt **block_tx(my_writeset_size)**, and waits. The other processor, on this interrupt, enters the *block_tx* procedure. For further details about the conflict resolution mechanism see 3.6. On re-writing by the same processor, there is no extra overhead.

Obviously, there is a need for the translation of one virtual address to two or more physical addresses. This is accomplished with the different Page Directory Base Registers (PDBR). Standard PDBR in x86 architectures is the CR3 register. Alternatively, transactional PDBR is defined uniquely for every transaction, in the TST (see 2.1).

The cost of transactional write is not uniform as can be seen in Fig. 5b. First write to the block is slowed down by a write to TB, creating a backup of the block in the secondary page and notifying all dependent transactions/processors to abort transactions. Every next write by the same transaction to the same block is slowed down by only one extra read.

3.4 Commit Transaction

A call to **commit_tx()** is translated to the processor instruction ***ctx TSTp***, which decrements a transaction nesting depth. If the nesting depth is not yet zero, there is no side-effect of instruction call. If the nesting depth becomes zero, then this is the outermost commit (when transactions are nesting). In that case, the processor executes the OS function "**commit(TXn**)". In this function, the OS iterates through the list of all secondary pages and for each of them calls the instruction **clean_tb vaddr, TXID**, in which the processor cleans all the bits for the given TXID. When the OS finishes iterating, it executes *finish_tx*, which informs every transaction blocked on this one that they can proceed, and sets the transaction state in the TST as FREE and INACTIVE. The cleaning of the transactional VPT can be done by other idle processor.

3.5 Abort Transaction

Abort transaction is implemented as an interrupt, to allow non-transactional reads and writes that have a conflict with a transactional write, to undo the transaction, restore original value of the block and then proceed with execution. In the global TST it sets the transaction status to ABORTED, loads the correct virtual page table base address into the PDBR (CR3 register on the x86 architecture) of the chosen processor and then starts iterating through the Transactional VPT, and for every page in it issues a processor the instruction ***undo_tx vaddr, cpuid***, which for each block (bl) in the page: restores a backup copy if it exists, and clears the M/TXID bits of the block (same activity as for clean_tb)

When the OS finishes iterating through the Transactional VPT, it executes *finish_tx*, which informs every transaction blocked on this one that they can proceed, and sets the transaction state in the TST as FREE and INACTIVE.

3.6 Conflict resolution: Block Transaction IPI

Conflict resolutions are usually done either by the system clock, by virtual time (timestamp), by transaction size, by transaction execution time, or other criteria. We have adopted the conflict resolution by write-set size. Whenever a transaction

modifies a block for the first time (and creates a backup value of the block), it increments the private writeset_size counter. The value of this counter is used to determine the priority of transactions. The transaction with bigger value of writeset_size has the higher priority in a conflict.

When a processor, for instance P1, receives the **block_tx (writeset_size)** IPI from the other processor, for instance P0, it compares the write-set size of P0 (P0_writeset_size) with the write-set size of P1 (P1_writeset_size), and if P0_writeset_size is greater then the P1_writeset_size, it aborts (retries) the current transaction. Otherwise, if P0_writeset_size is less or equal to the P1_writeset_size, then P1 puts the P0 into the waiting list. In that case, P1's write-set is greater in size, and the other processor should wait until the completion of transaction inside P1.

This protocol gives the priority to the transactions with the bigger write set. At the same time, it orders transactions so that dead-lock is avoided.

4 Conclusions and future work

We presented a new synergistic hardware-software solution for providing Unbounded Page based Hardware Transactional Memory. Leveraging existing Virtual Memory mechanisms, it should allow constant transaction execution speed, regardless of the transaction size, and with small overhead for transactional reads and writes. It does not require any changes of the currently highly-optimized caches. It requires some relatively small changes in current Operating System implementations, to allocate and manipulate the memory space for bookkeeping and secondary pages.

References

1. Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M., and Wood, D. A. Performance pathologies in hardware transactional memory. In *ISCA* (2007), pp. 81–91.
2. Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Biesbrouck, M. V., Pokam, G., Calder, B., and Colavin, O. Unbounded page-based transactional memory. *SIGARCH Comput. Archit. News 34*, 5 (2006), 347–358.
3. Hammond, L., Carlstrom, B. D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., and Olukotun, K. 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*. ACM Press, Oct 2004, pp. 1–13.
4. McDonald, A., Chung, J., Chafi, H., Cao Minh, C., Carlstrom, B. D., Hammond, L., Kozyrakis, C., and Olukotun, K. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Sept 2005.
5. Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. Feb 2006, pp. 254–265.