Evaluating the Addition of Non-Transactional Loads to HTM

Yehuda Afek[1] and Hillel Avni[2]

[1] Tel Aviv University
[2] Ben-Gurion University

Abstract. A non transactional load (NTL), is a load instruction, which is invisible to the transactional system, even if done within a transaction. It ignores and suppose to be ignored by the other concurrent transactions, thus NTL does not introduce any conflicts. An analysis of the potential benefits of NTL and the issues in introducing it are discussed. The two unique characteristics of NTL are: (a) if called inside an HTM transaction, an NTL will not be added to the transaction read-set, and (b) if called inside or outside of a transaction, will not cause the abort of another concurrent transactions. Implementing (a) seems to be easy, but (b) is more difficult. If a load accesses a cache line that has been modified in a concurrent transaction T, than according to the requester wins policy [1] of the HTM, T is aborted. We suggest a potential solution to (b) and analyze it, and discuss the added value the NTL may have, e.g., in COP operations [2].

Keywords: Transactional-Memory, Consistency Oblivious Programming, Data-Structures

1 Introduction

In HTM, all addresses accessed by a transaction are marked in the cache for monitoring. To reduce the chance of contention, it is desirable to avoid marking some of the accessed addresses, as long as it does not hamper correctness. Two possible ways to exclude cache lines from the read and write sets are, (1) to unmark addresses that already joined the transaction, i.e., early release (ER), and (2) is to keep addresses from being added to the transaction data set, i.e., access the addresses with NTL that will not mark them.

NTL existed in the design of Rock and ASF HTM blocks which were canceled [3], and ASF also contained an instruction for ER, but both are absent in Intel TSX. This paper is exploring the potential benefits of adding NTL and ER, and than, after concluding that NTL is more useful than ER, considers the obstacles which make the addition of NTL complicated.

1.1 Potential Benefits

Following is a list of potential usecases for NTL and ER, and why they are justifiable or not.

Local Variables The GCC compiler STM support [4] automatically makes use of selective annotation to avoid protecting the local threads stack whenever possible. This feature can be adapted to HTM by using NTL or ER to exclude the local thread addresses from the transaction. However, the lines stay in the cache, so hardware resources are not saved, and contention is not reduced, as local variables are not contended.

HyTM Optimizations Hybrid-NOrec [5] describes the implementation of a hybrid TM system on best effort HTM. The design allows software and hardware transactions to coexist, although concurrency among such transactions is restricted rather severely. High-performance variants of this approach require the ability to issue NTL from within a transactional context. In these variants, Hybrid-NOrec polls a seqlock out of an HTM transaction in the verification of software transactions, and writes the seqlock transactionally in the commit of a hardware transaction, so with the NTL issue, the HTM committer is at risk of aborting because of a software transaction verification. Now, in the window between writing the seqlock and committing, the HTM transactions are vulnerable to abort by an STM transaction that reads the seqlock. The window is short, but it is still significant, as the STM polls the seqlock after each transactional access.

ER Data Structures ER requires a `RELEASE` instruction as was planed in ASF [3], which explicitly unmark an address. The `RELEASE` can be used to unmark addresses that are not required for the correctness of the ongoing transaction, as done in [6, 7]. The problem with HTM `RELEASE` instruction, is the composition of operations. If a transaction T executes an operation op_1 and then an operation op_2, and op_1 marks address A which is required for its safety, then if op_2 reads A and later decides to release it, op_1 will no longer be safe. With STM, there can be two instances of an address in the read-set, so ER would release only one of them and be composable, however, with HTM, there can be only one instance of an address in the cache.

COP Composition The COP method [2, 8] allows, under certain restrictions, the creation of data structures with smaller transactional footprint. With HTM and NTL, the COP operations will be composable, and as a result, more applications will be able to use them. Other techniques, which have similar ideas, will also benefit from NTL in the same way [9, 10].

We summarize that COP composition and hybrid NoRec optimization justify the addition of NTL, but not ER, to HTM. We elaborate on COP composition with NTL in Section 2.

1.2 Issues with NTL Implementation

If an NTL generates a coherency request to a cache line that has been modified inside transaction T, according to the requester wins [1] eager conflict resolution, T aborts. This violates our requirement that the NTL will be invisible to the transactional system. However, to design a load that does not generate a coherency request is not feasible, as it is like a read without a read. In Section 3 we sketch a solution that involves a change to the coherence protocol and the HTM commit microcode.

To illustrate the potential impact of this issue on an application that uses COP operations, consider a COP version of the lazy linked list [11] with three nodes, N_1, N_2 and N_3. A transaction T_1 starts by deleting N_2. For this purpose, it writes transactionally the next pointer of N_1. Then it continues to do a lot of other work. At this point, transaction T_2 performs a search for N_3 using NTL to read N_1 and N_2 next pointers. T_2 generates coherency requests to N_1 next pointer and forces T_1 to abort, although according to the COP semantics, there is no conflict.

2 Composable COP Requires NTL

A COP operation, is based on a data-structure operation op. We split op to a read only prefix op^{ROP} and to the writing suffix op^{C}. To run op inside a transaction just execute op^{C} after op^{ROP}. However, the COP version of op, which is embedded in a transaction T, T_{op}, performs the following steps:

- **In non-transactional mode:** Execute $T_{op^{ROP}}$ and record its output. This part is done without any synchronization, and may pass through inconsistent states and return inconsistent output.
- **In transactional mode:** Verify that ROP output is consistent, and if it is not, abort, otherwise execute $T_{op^{C}}$.

In [2] the correctness and progress of COP are already demonstrated. We remind the reader that the verification not only ensures T that the op^{ROP} output was consistent, but also adds the addresses that prove it to the read set for monitoring of this consistency. Here we assume the correctness and focus on the performance and composability of COP with HTM, which will require NTL.

The only way to compose COP operations in current TSX, is the one proposed by [9], i.e., execute all ROP parts of the composed operations before starting the transaction, then, inside the transaction, verify their output and complete updates. This method allows composition only if an operation is not writing data that may later be accessed by another operation in the same transaction.

To demonstrate this restriction, we split each COP operation op_k, which executes in transaction T, T_{op_k}, to $T_{op_k^{ROP}}$ and $T_{op_k^{C}}$ (verify and complete). Now, assume op_1 precedes op_2, and op_1 is writing data that op_2 is reading. According

to [9], the transaction T, which executes op_1 and then op_2, will execute the following sequence. **xstart** means HTM goes into transactional mode, and **xend** is HTM commit:

$$T_{op_1^{ROP}} \rightarrow T_{op_2^{ROP}} \rightarrow \textbf{xstart} \rightarrow T_{op_1^{C}} \rightarrow T_{op_2^{C}} \rightarrow \textbf{xend}$$

As $T_{op_2^{ROP}}$ must execute before $T_{op_1^{C}}$, op_2 will not see op_1 updates, and T will not be correct.

If instead of op_1, T will execute any other transactional code, we will have to call **xstart** before op_2, so op_2^{ROP} will be in transactional mode. For example, if T dequeues V and then inserts V to a RB-Tree with a COP operation, then this operation will not benefit from the usage of COP.

Using NTL allows the composition of any COP operation, with any other operations, by using NTL in the ROP. Now T executes the ROP with NTL, so we call the ROP of op op^{ROP-N}. If T tries to execute the COP operation op_2 after the COP operation op_1, it goes through the following sequence:

$$\textbf{xstart} \rightarrow T_{op_1^{ROP-N}} \rightarrow T_{op_1^{C}} \rightarrow T_{op_2^{ROP-N}} \rightarrow T_{op_2^{C}} \rightarrow \textbf{xend}$$

As $T_{op_1^{C}}$ executes before $T_{op_2^{ROP-N}}$, and as both $T_{op_1^{C}}$ and $T_{op_2^{ROP-N}}$ execute in the context of T, $T_{op_2^{ROP-N}}$, correctly sees the updates of both $T_{op_1^{C}}$ in the local cache.

3 A Possible Solution to the NTL Issues

Theoretically, it would be possible to solve the issue raised in Section 1.2 if one could modify the MESI cache coherency protocol and the HTM commit microcode, as follows; We replace the Exclusive (E) state from MESI with a *locally-transactionally-modified* (L) state. The L state resembles the Shared (S) state in the sense that it never writes back, but also has a similarity to the Modified (M) state, as the value in the cache is different from the value in main memory. For any given pair of caches, the permitted states of a given cache line are as follows:

	M	L	S	I
M	X	X	X	V
L	X	V	V	V
S	X	V	V	V
I	V	V	V	V

The M, S and I states stay as in MESI, and L is the state of a *modified marked* cache line. In HTM, the commit microcode needs first to change the state of the lines in state L to state M and then send a coherency request for each of them. Then it proceeds to the standard commit sequence. The HTM stays requester

wins, and not committer wins, and once the line moved to the M state, it can be aborted by a concurrent load.

We note that the practicality of the suggestion is not clear to us. MLSI introduces engineering challenges, such as changing the states and sending the requests efficiently, and probably other, maybe unsolvable, difficulties.

In addition, The MLSI tolerates the existence of doomed transactions, i.e., transactions that are in a deadlock and eventually have to abort. For example, two transactions that have the same cache line in L state. However, unlike their STM counter parts, these transactions see consistent states. They do not cause memory corruption, as they eventually abort. A similar issue is discussed in [12] where they propose Speculative Modified (SM) to allow more readers to commit successfully, by buffering the writes.

4 Simulation and Evaluation

To examine the potential contribution of COP to applications, we added the COP RB-Tree from [2] to the STAMP testing suite. Inside the STAMP, we executed Yada, Intruder, and Vacation tests. These tests are the ones that use a map, which is implemented as an RB-Tree. We simulate the NTL over GCC STM, by using TM-Pure [4] attribute for the ROP function. Our goal is to demonstrate the benefit of NTL and COP in some applications.

We execute the standard configuration of Vacation (*vacation-high* from [13]). Each transaction in this application is accessing several 1M RB trees, several times each, and these transactions are a significant portion of the workloads.

	GCC-COP	GCC-STM
Transactional Loads	0.4 G	2.4 G
Aborts Rate	0.5%	3.0%

Fig. 1: STAMP Vacation Statistics (G = 10^9)

In Figure 1 we count transactional loads and aborts for the Vacation benchmark. We count the transactional loads when the whole application is executing on a single thread, to get the most accurate number. The aborts count is taken when all eight hardware threads execute. We see that plain STM is performing more than five times the transactional loads of COP with NTL, and there are six times more aborts in plain STM.

In Figure 2 we compare the conflicts rate on a COP red-black tree vs. plain STM one. The tree keys range is 1K and it is half full, thus it is small and has a lot of mutations, which cause relatively high contention. Each update, i.e., insert or delete is a transaction that consists of the update and four lookups, to demonstrate COP operations composition. As each transaction is retried up to 20 times, the number of conflict aborts can be higher that the number of successful transactions. We can see that contention on the plain STM is rising much faster and on 100% updates, i.e., 50% inserts and 50% deletes, STM has 6 times more aborts than COP.

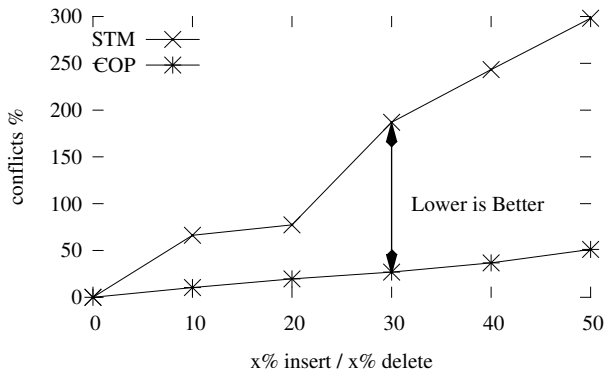

Fig. 2: Aborts vs. update rates on a small red-black that executes on eight threads.

5 Conclusion and Future Work

In this paper we showed that adding NTL to HTM will allow applications to use COP data structures inside transactions. We showed with the STAMP benchmark, that using COP, can significantly reduce the contention in a transactional application. We also observe and discuss the difficulties in implementing NTL, which may overshadow its potential benefits.

6 Acknowledgements

We are grateful to Raanan Sade from Intel for pointing out the issues with NTL Implementation.

References

1. : Intel architecture instruction set extensions programming reference. (`http://software.intel.com/file/41604`)
2. Afek, Y., Avni, H., Shavit, N.: Towards consistency oblivious programming. In: OPODIS. (2011) 65–79
3. Christie, D., Chung, J.W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Rivière, E.: Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In: Proceedings of the 5th European Conference on Computer Systems. EuroSys '10 (2010) 27–40
4. Riegel, T.: Software Transactional Memory Building Blocks. PhD thesis, Technischen Universitat Dresden, geboren am 1.3.1979 in Dresden (2013)
5. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. In: ASPLOS. (2011) 39–52

6. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: DISC. (2009) 93–107
7. Afek, Y., Morrison, A., Tzafrir, M.: Brief announcement: view transactions: transactional model with relaxed consistency checks. In: PODC. (2010) 65–66
8. Avni, H., Kuszmaul, B.C.: Improving htm scaling with consistency-oblivious programming. In: TRANSACT. (2014)
9. Xiang, L., Scott, M.L.: Composable partitioned transactions. In: WTTM. (2013)
10. Xiang, L., Scott, M.L.: Compiler aided manual speculation for high performance concurrent data structures. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '13 (2013) 47–56
11. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Proceedings of the 9th International Conference on Principles of Distributed Systems. OPODIS'05, Berlin, Heidelberg (2006) 3–16
12. Armejach, A., Titos-Gil, R., Negi, A., Unsal, O.S., Cristal, A.: Techniques to improve performance in requester-wins hardware transactional memory. ACM Trans. Archit. Code Optim. **10** (2013) 42:1–42:25
13. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: IISWC. (2008) 35–46