An Analysis and a Solution of False Conflicts for Hardware Transactional Memory

Yuki FUTAMASE*, Masaki HAYASHI*, Tomoki TAJIMI*,
Ryota SHIOYA†, Masahiro GOSHIMA‡ and Tomoaki TSUMURA*

*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

†The University of Tokyo
Hongo 7-3-1, Bunkyo, Tokyo, Japan
Email: shioya@ci.i.u-tokyo.ac.jp

‡National Institute of Informatics
Hitotsubashi 2-1-2, Chiyoda, Tokyo, Japan
Email: goshima@nii.ac.jp

***Abstract*—Transactional memory is a promising paradigm for shared-memory parallel programming model. On TMs, transactions are executed speculatively in parallel as long as any access conflict is not detected. On general hardware transactional memories (HTMs), conflicts degrade the performance because of the overhead for retrying transactions, and it is important to avoid conflicts. HTM generally detects access conflicts on cache-line granularity, and this causes accesses on different variables that are on a cache line to be falsely detected as conflicting accesses. In this paper, we analyze how frequently such false conflicts occur and what type of coding can cause them. As a result of the analysis, we confirmed that the false conflicts account for 27.4% on average and even 99.9% at a maximum of all detected conflicts. We also propose a light-weight fine-grained conflict detection mechanism and show that it can reduce the execution cycles by 17.7% on average and 36.5% at a maximum.**

I. Introduction

Along with the spread of multi-core processors, parallel programming utilizing thread level parallelism has become increasingly important. In shared memory parallel programming model, *lock*-based mechanisms have often been used for arbitrating access to shared resources. However, lock-based synchronization may cause deadlocks, and they lead to poor scalability.

Transactional Memory (TM) [1] has been proposed as an attractive alternative to lock-based synchronization. On TM-managed systems, multiple transactions running on separate threads can speculatively access the shared memory in parallel, as long as any access conflict is not detected. TM saves both new and old values when a transaction modifies a value in the shared memory (*version management*). TM also keeps tracks of each memory access, checking whether each requested data has been accessed by another transaction or not (*conflict detection*). Hardware Transactional Memories (HTMs) [2], [3], hardware implementations of TM, offer high performance since they implement the mechanisms of version management and conflict detection in hardware. HTMs are now widely implemented in the several latest processors, such as IBM zEC12, Power8, and Intel Haswell.

Since HTM generally detects conflicts on cache-line granularity, accesses to different variables on a cache-line are falsely detected as conflicting accesses (*false conflict*). In this paper, we analyze the frequency of such false conflicts on HTM and clarify what type of coding can cause false conflicts. We also propose a fine-grained conflict detection with smaller hardware implementation cost than our previous work[4].

In summary, we will make the following contributions:

1) We analyze the frequency of false conflicts and disclose that they account for 27.4% on average and 99.9% at maximum of all detected conflicts (Sec. III).
2) We also disclose that 3.0% of the false conflicts are on non-transactional accesses to non-shared variables that essentially can never cause access conflicts (Sec. III).
3) We propose a light-weight fine-grained conflict detection mechanism (Sec. IV) and show that the execution cycles can be reduced by 17.7% on average and 36.5% at a maximum with the mechanism (Sec. V).

II. Conflict Detection on General HTM

A. Hardware Transactional Memory

With transactional memory (TM), transactions encapsulating critical sections can run in parallel as long as any memory access conflict is not detected. Developers can roughly define coarse-grained transactions unlike with traditional lock-based mechanisms and can achieve good performance owing to the transaction parallelism. TM guarantees the following properties for transactions by monitoring memory accesses:

Atomicity: Each transaction must be executed completely and must not be executed partially.

Isolation: The result of the concurrent execution of transactions must be the same as that of sequential execution.

A situation where memory accesses that can violate one of the properties are detected is called *access conflict*. To be free from a conflict, one of the transactions that are concerned with the conflict should discard its execution, or *abort*. When a transaction is aborted, the old values that are updated by the transaction must be restored, and the state of the memory and the register file should be rolled back to their state of before the transaction started. Hardware transactional memory (HTM) has hardware support for the conflict detection and version management mechanisms, and costs low latency for transaction control.

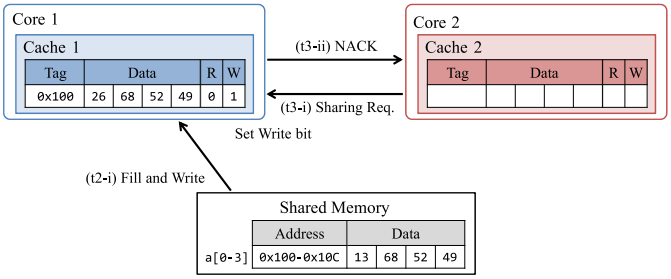

Fig. 1. How accesses to different variables cause conflict

B. Cache-Line Grained Conflict Detection on HTM

Many HTM implementations, such as IBM zEC12 [5], adopt additional bit fields on each cache line for managing transactional reads/writes and detecting conflict; they called *R/W-bit*. Other implementations, such as Bulk [6] and LogTM-SE [7], adopt filters called *signatures* to unitarily manage transactional reads/writes on cache lines. In both cases, conflicts are detected on cache-line granularity, and some memory accesses can be falsely detected as conflicts.

Figure 1 shows a brief example of how a *false conflict* is detected. Now assume that the members of array `a[]` on the shared memory are concurrently accessed by two transactions running on *Core*1 and *Core*2. Firstly, when *Core*1's transaction updates the value of `a[0]`, *Core*1 caches `a[0-3]` on a cache-line (t2-i) and sets the *W-bit* of the line (t2-ii). After then, if *Core*2's transaction tries to update the value of `a[2]`, *Core*2 sends a coherence request for sharing the cache line that contains `a[2]` (t3-i). Because the *W-bit* of the line has been set, *Core*2 receives *NACK* as the response to its request, or *Core*2's access on `a[2]` is falsely detected as a conflict (t3-ii). As shown in this example, false conflicts are due to cache-line grained conflict detection.

III. Analysis of False Conflicts

False conflicts lead to futile abort and retry of transactions and degrade the performance of HTM. To solve this, we firstly analyze the frequency of false conflicts, their impact on performance, and what type of codes will cause false conflicts.

A. Experimental Setup

We used LogTM-SE implemented on a full-system execution-driven functional simulator Simics [8] in conjunction with customized memory simulators built on GEMS [9] for analysis. The detailed configuration is summarized in TABLE I. We used a total of eight programs to be evaluated from GEMS microbench, SPLASH-2 [10], and STAMP [11]. Each benchmark program was executed with 16 threads.

B. Survey

The results with the following two HTM configurations are shown in Fig. 2 and TABLE II, and the results of (I) are normalized to the corresponding results of (B) in Fig. 2. We implemented an ideal configuration (I), which has as many Read/Write bits per cache line as the number of words in a cache line and never falsely detects conflicts.

(B) *Baseline*: LogTM-SE [7]

(I) *Ideal*: can manage transactional reads/writes on variable (or word) granularity and completely suppress false conflicts.

TABLE I
Simulator Configuration

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width/order	single/in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycles
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

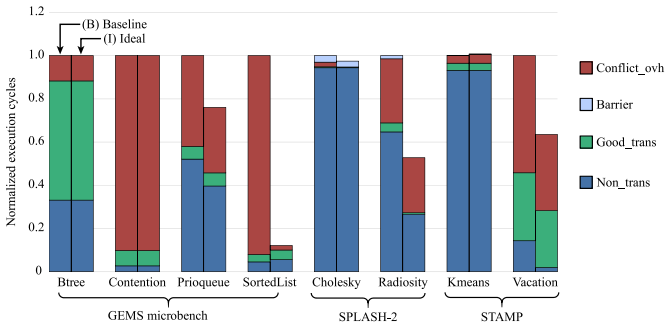

Fig. 2. Normalized execution cycles

The legend in Fig. 2 shows the breakdown items of the total execution cycles. They indicate the cycles out of transactions (*Non_trans*), the cycles in the committed transactions (*Good_trans*), the penalty overhead caused by conflicts (*Conflict_ovh*), and the barrier synchronization cycles (*Barrier*).

TABLE II shows the false conflict rate with (B). We define the false conflict rate as the ratio of the number of *NACK*s caused by false conflicts to the total number of *NACK*s.

The result shows that false conflicts account for 99.9% at a maximum and 27.4% on average of the total conflicts, and fine-grained conflict detection can reduce the execution cycles by 87.9% at a maximum and 24.7% on average.

C. Detailed Examination

Prioqueue and **SortedList** particularly show high false conflict rates, and the number of execution cycles of (S) is significantly lower than (B). These workloads have transactions that frequently access multiple member variables in a structure defined as a global variable. Accessing these variables placed in the adjacent memory space causes false

TABLE II
FALSE CONFLICT RATE

workload	Btree	Contention	Prioqueue	SortedList
rate	0.6%	0.0%	45.8%	99.9%
workload	Cholesky	Radiosity	Kmeans	Vacation
rate	3.7%	66.5%	0.3%	29.8%

```
for (i = 0; i < numOperation; i++) {
  action = generates_random_value();
  switch (action) {
    case MAKE_RESERVATION: {
      BEGIN_TRANSACTION(0);
      for(n = 0; n < numQuery; n++){
        types[n] = generates_random_value();
      }
      ... /* make reservation of car/room/flight */
      COMMIT_TRANSACTION(0);
      break;
    }
    ... /* other cases */
  }
}
```

Fig. 3. A transaction in Vacation

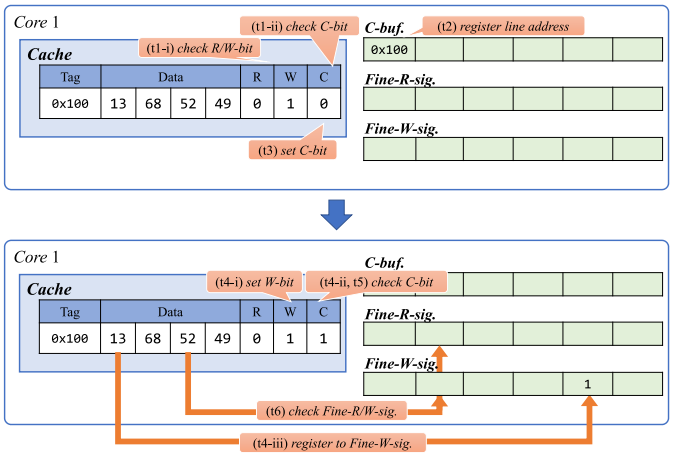

Fig. 4. Fine-grained conflict detection

conflicts. In such a case, false conflicts can be avoided by appropriately padding between variables in the structure, but performance may decline due to reduced cache efficiency.

Since **Vacation** has a distinguishing feature that *Non_trans* cycles are largely different between (B) and (I), we examine it in detail. It is a workload that emulates an online booking system. It has three transactions, and the transaction shown in Fig. 3 is most frequently executed; accounts for 98% of the total count of transaction execution. The transaction manages reservation entries, and the entry size is 16 bytes. Hence, four entries at most can be placed in a cache line and accesses to them in the transaction lead to false conflicts.

In addition, the difference of *Non_trans* indicates that execution outside transactions can also suffer conflicts even though it is not transactional. We investigated the details and discovered that it is due to an array of a structure; each thread uses only one element of the array indexed by its thread ID. Accesses to the elements never bring conflicts, but a coarsely defined transaction includes an access on an element and this can stall a non-transactional access on another element that is located in the same cache line.

Since such a coding style is not strange and array elements are generally located sequentially in the shared memory, many programs can suffer this type of false conflict. Although developers can avoid it by utilizing padding with minute attention, the productivity that HTM offers will be severely detracted. Hence, we will propose a solution for false conflicts in the following.

IV. FINE-GRAINED CONFLICT DETECTION

A. Implementation

We evaluated an ideal performance described in Sec. III by using a pair of *R-* and *W-bit* per word. It is a naive implementation, but costs large area overhead. Cache lines where false conflicts occur should be limited, and scaling *R/W-bits* for all lines is futile.

Inspired by LogTM-SE [7] and BFGTS [12], we propose an implementation that adopts Bloom filter [13] for fine-grained conflict detection and can manage only cache lines where false conflicts occur with small hardware extension. We install the following additional hardware units to each core for the implementation:

- *C-buf (conflict address buffer)*: temporarily manages cache line addresses where conflicts are observed in the current running transaction.
- *C-bit (conflict bit)*: a flag installed to each cache line; it indicates whether a conflict has occurred on the line.
- *Fine-R/W-sig. (Fine-grained Read/Write signature)*: Bloom filters that manage accessed addresses for fine-grained conflict detection.

B. Execution Model

The proposed mechanism firstly manages the cache lines where conflicts have occurred in past and adopts fine-grained conflict detection only on such cache lines. We explain the execution model by using Fig. 4 where how the same example shown in Fig. 1 is managed by the proposed mechanism.

1) Managing Conflicted Cache Lines: Now assume that *Core*1 receives an access request for `a[2]` after updating `a[0]` as same as the example described in Sec. II-B. In that case, *Core*1 checks not only *R/W-bit* but also *C-bit* (t1-i/ii). Since the value '0' of *C-bit* means no conflict has occurred on the line, *Core*1 normally detects conflict on cache line granularity and returns *NACK* as usual, and the line address is registered on *C-buf* (t2).

After then, when committing the transaction, *Core*1 sets *C-bits* of the cache lines registered on *C-buf* (t3) for adopting fine-grained conflict detection to the lines.

2) Conflict Detection using Bloom Filter: Assume a situation where *Core*1 executes the transaction and receives an access request for `a[2]` after updating `a[0]` again. When updating `a[0]`, *Core*1 knows the cache line has a conflict

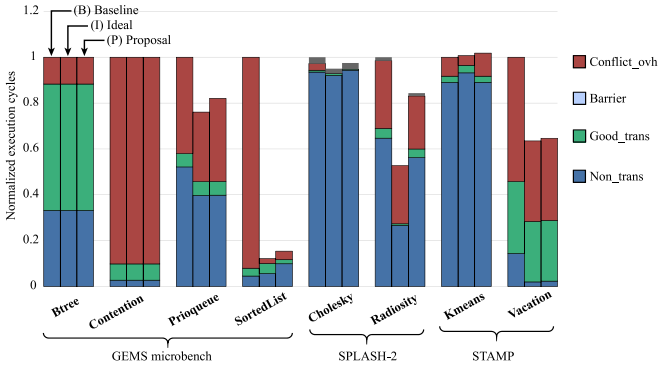

Fig. 5. Normalized execution cycles with the proposed mechanism.

in past by checking *C-bit* of the line. Hence, *Core*1 registers the address of `a[0]` to *Fine-W-sig.* for fine-grained conflict detection on this line (t4-i/ii/iii). Receiving a request for `a[2]`, *Core*1 checks the *C-bit* (t5). Because it is set, *Core*1 tries fine-grained conflict detection by searching the requested address of `a[2]` from *Fine-R/W-sig.* (t6). It is not registered on them, no conflict is detected and the request can be granted. In this way, accesses to different variables in the same cache line can avoid false conflict.

C. Area Cost Estimation

The naive implementation we adopted for evaluating an ideal performance in Sec. III-B uses a pair of *R/W-bits* per word. The simulator configuration has 32 kB D1 cache and its line size is 64 B. Hence, required additional memory cell for the naive implementation can be calculated as $(32\mathrm{kB}/64\mathrm{B}) \times 2 \times (64\mathrm{B}/32\mathrm{bit}) = 2\mathrm{kB}$.

In contrast, only two 1024 bit length Bloom filters and additional small logic are required for our proposal. This means that the proposed implementation only costs one eighth memory cells compared with the naive implementation, and it will be energy efficient.

V. Experimental Evaluation

We implemented the mechanism described in Sec. IV and evaluated it with the same setup as that presented in Sec. III-A. Figure 5 shows the results with the following three HTM configurations. It shows that the execution cycles are reduced by 36.5% at a maximum and 17.7% on average with (P).

(B) *Baseline* (reproduced from Fig. 2)
(I) *Ideal* (reproduced from Fig. 2)
(P) *Proposal*

The mechanism we proposed largely reduced the execution cycles of the programs that show high false conflict rate in TABLE II, and the performance is comparable to an ideal configuration (I) even though its area overhead is remarkably smaller than that of (I).

Only with **SortedList**, the performance is inferior to (I). It is caused by many false positives on the Bloom filter because we used a very simple hash function that consists of only bit shift and XOR to suppress the latency of hash value calculation.

It is known that false positives can be drastically reduced by increasing the number of hash functions. Hence, the performance will reach to the ideal by designing more appropriate function and/or increasing the number of functions.

VI. Conclusion

In this paper, we analyzed the frequency and performance impact of false conflicts on HTM and discussed what type of coding is apt to cause such conflicts, by examining typical benchmark programs.

We disclosed that false conflicts account for a large percentage of all detected conflicts, and some of them are detected on even non-transactional memory accesses outside transactions. We also proposed an implementation of fine-grain conflict detection and introduced that it can reduce the execution cycles by 35.5% at a maximum and 17.7% on average. Our future work includes reducing false positives of the Bloom filter by using multiple hash functions and optimizing them.

Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Numbers JP17H01711, JP17H01764, and JP17K19971.

References

[1] M. Herlihy *et al.*, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, May. 1993, pp. 289–300.
[2] T. Knight, "An architecture for mostly functional languages," in *Proc. ACM Conference on LISP and Functional Programming (LFP'86)*, 1986, pp. 105–112.
[3] L. Hammond *et al.*, "Transactional Memory Coherence and Consistency," in *Proc. 31st Annual Int'l Symp. Computer Architecture (ISCA'04)*, Jun. 2004, pp. 102–113.
[4] S. Horiba, H. Asai, M. Eto, T. Tsumura, and H. Matsuo, "Fine-Grain Conflict Management for Hardware Transactional Memory Systems Employing Eager Version Management," in *Proc. 4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA2013), held in conjunction with HiPEAC'13*, Jan. 2013.
[5] C. K. Shum, F. Busaba, and C. Jacobi, "IBM zEC12: The Third-Generation High-Frequency Mainframe Microprocessor," *IEEE Micro*, vol. 33, no. 2, pp. 38–47, Mar. 2013.
[6] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Proc. 33rd Annual Int'l Symp. on Computer Architecture (ISCA'06)*, Jun. 2006, pp. 227–238.
[7] L. Yen *et al.*, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Proc. 13th Annual Int'l Symp. on High Performance Computer Architecture (HPCA-13)*, Feb. 2007, pp. 261–272.
[8] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[9] M. M. K. Martin *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.
[10] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. 22nd Int'l. Symp. on Computer Architecture (ISCA'95)*, 1995, pp. 24–36.
[11] C. C. Minh *et al.*, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, Sep. 2008.
[12] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," in *Proc. 17th Int'l Conf. on High-Performance Computer Architecture (HPCA-17)*, 2011, pp. 75–86.
[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692