TagTM - Accelerating STMs with hardware tags for fast meta-data access

Srđan Stipić*, Saša Tomić*, Ferad Zyulkyarov*, Adrián Cristal*†, Osman Unsal*, Mateo Valero*
{srdjan.stipic, sasa.tomic, adrian.cristal, osman.unsal, mateo.valero}@bsc.es
feradx.zyulkyarov@intel.com⋆
*Barcelona Supercomputing Center †IIIA - Artificial Intelligence Research Institute - CSIC - Spanish National Research Council

***Abstract*—In this paper we introduce TagTM, a Software Transactional Memory (STM) system augmented with a new hardware mechanism that we call GTags. GTags are new hardware cache coherent tags that are used for fast meta-data access. TagTM uses GTags to reduce the cost associated with accesses to the transactional data and corresponding metadata. For the evaluation of TagTM, we use the STAMP TM benchmark suite. In the average case TagTM provides a speedup of 7-15% (across all STAMP applications), and in the best case shows up to 52% speedup of committed transaction execution time (for SSCA2 application).**

I. Introduction

The industry shift towards chip multiprocessors (CMPs) has put many researchers to study new techniques which would make parallel programming easier. One such technique is Transactional Memory (TM) [8], [10]. TM is a concurrency control mechanism which abstracts the complexity of programming with locks by using a much simpler programming interface based on transactions. Implementations of TM exist in both hardware and software. Hardware TM (HTM) systems are realized in form of extensions at the micro-architectural level and software TM (STM) are realized entirely in software. HTMs have good performance but are complex to implement and are not flexible (may not respond well to future software requirements). On the other side, while STMs are flexible and can be easily tuned to a specific application without requiring any architectural changes, they are orders of magnitude slower than HTM [5]. It is notoriously known that the poor performance of STMs is due to the overheads which are incurred during the maintenance of transactional metadata. After a quick profiling we found that transactional overheads can constitute up to 5 times of the total program execution confirming conclusions which already exist [5], [16] (see Figure 1).

In a nutshell, STM systems need to maintain transactional metadata to detect conflicts for every memory word. This metadata is accessed on every transactional operation (i.e. `tx_read` and `tx_write`) thus doubling the cost of the memory operations inside transactions. In addition, typical STM implementations store the transactional metadata separately from its data. Because of this poor locality the accesses to metadata can be further penalized with a cache miss while loading the actual data.

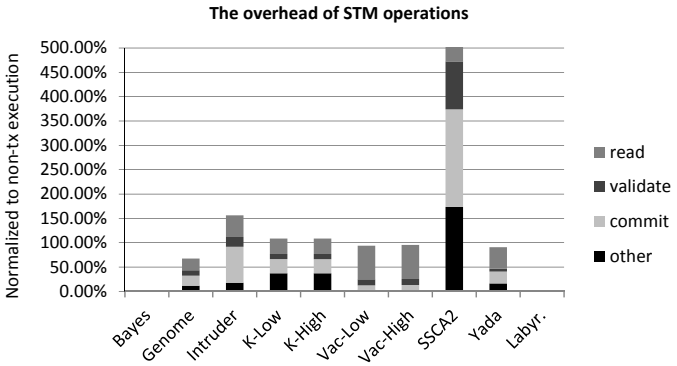


Fig. 1. STM overheads running one CPU.

In this paper, we propose architectural extension to reduce the STM overheads of accessing the metadata. In particular, the extensions allow (*i*) accessing both the data and its metadata with a single memory operation instead of two and (*ii*) fetching both the data and the metadata together thus eliminating the cache misses due to poor spatial locality. We describe our extensions which we call *Global Tags (GTags)* in Section II. GTags extend the computers' memory hierarchy with coherent metadata tags and a set of instructions to operate them. In Section III we introduce TagTM, an extension of STM library with GTags. We use GTags for accelerating the maintenance of the transactional metadata. In Section IV we present an evaluation of TagTM, in Section V we discuss the related work, and in Section VI we conclude.

II. Global Tags (GTags)

In this this section we introduce global tags (GTags), a mechanism for annotating transactional data at cache line granularity across the different levels of the memory hierarchy. In our design GTags extend every cache line with 32-bit tags.[1] These tags are associated with the data in the cache lines and

⋆This work was conducted while Ferad Zyulkyarov was in Barcelona Supercomputing Center.

978-3-9810801-8-6/DATE12/

[1]The tags can be 32 or 64 bits. In the case of TagTM 32-bits were sufficient.

New ISA instructions	Description
ldt r1 ← T[r2]	Load tag
stt T[r1] ← r2	Store tag
cast T[r1] ← r2 if T[r1] == r3	Compare tag and swap tag
ldtv r2 ← T[r1], r3 ← M[r1]	Load tag and value
sttv T[r1] ← r2, M[r1] ← r3	Store tag and value
castv T[r1] ← r2, M[r1] ← r3 if T[r1] == r4	Compare tag and swap tag and value

TABLE I
GTAGS MANIPULATION INSTRUCTIONS

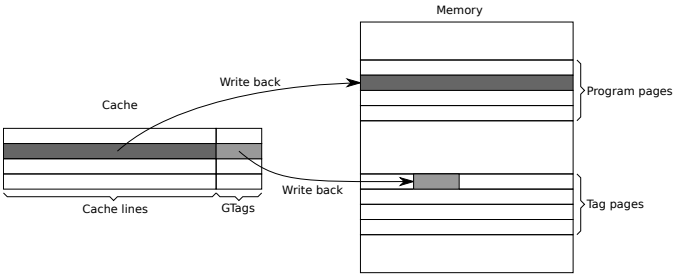


Fig. 2. Cache line eviction

kept always coherent. When a cache line is invalidated, its tag is also invalidated and vice versa.

GTags are accessed and modified with the special tag manipulation instructions which are shown in Table I. Furthermore, all instructions in the table are executed atomically. All the instructions that operate on the addresses that fall in the same cache line share the same tag.

Extending memory hierarchy with GTags requires small architectural changes which affect the CPU caches and the memory controller. In our particular case, we store GTags at the data part of the cache lines. Such a design decision allows reading and writing the GTags with simple load/store like instructions and does not require any changes to the otherwise complicated cache lookup logic.

In the main memory, we store the tags separately from the data in special GTag-pages. The GTag-pages are allocated statically and occupy the end of the physical address range. This allocation enables easy mapping of the physical address to the corresponding tag. The memory controller does a simple shift operation and adds an offset to calculate the physical address of the corresponding tag. In our simulations, we model a memory controller which can automatically fetch and store both the data and the tag from their corresponding pages (Figure 2). With such functionality, the tags are preserved when the cache lines and the tags are evicted from the cache.

III. TAGTM

In this section we introduce TagTM, the modified version of TinySTM. The goal of this section is to demonstrate how to use GTags to improve the performance of an STM library. In Section III-A we explain how TinySTM works and in Section III-B we describe the bottlenecks in the implementation of TinySTM and later we show how to extend transactional operations with GTags.

A. TinySTM

Felber et al. proposed TinySTM[12], a lightweight and efficient word-based STM system. TinySTM implements timestamp-based versioning algorithm. It utilizes a shared array of locks (SAL) to manage concurrent accesses to memory. Each lock covers a portion of the address space. The least significant bit of the lock indicates if the lock is owned, and the remaining bits of the lock store the version number that corresponds to the commit timestamp of the transaction that last updated the memory location covered by the lock.[2]

[2]For the full implementation, please refer to the original paper.

B. Bottlenecks in TinySTM

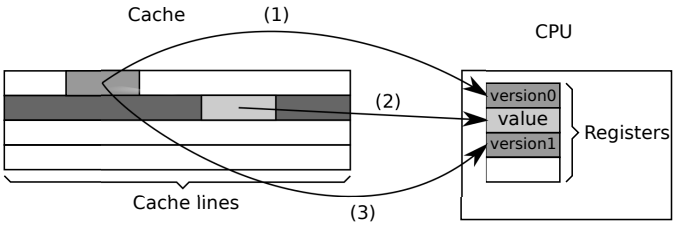
Figure 1 shows the breakdown of runtime for STAMP applications using TinySTM for a single processor run. The overheads are divided in 4 important parts: `tx_read`, `tx_validate`, `tx_commit`, and other. It is important to note that all these overheads do not exist in non-transactional execution. `tx_read` is a dominant overhead for Genome, K-means, Vacation, and Yada. The total overhead can be up to 69.82% (in the case of Vacation-high) compared to the non-transactional execution time. `tx_commit` is a dominant overhead for Intruder, and SSCA2, and adds up to 200.4% (in the case of SSCA2). `tx_validate` is a significant overhead for SSCA2 (98.59%).

C. Using GTags in TinySTM

In Section III-B we show that the performance-critical operations in TinySTM are `tx_read`, `tx_commit`, and `tx_validate`. All these operations can benefit from the use of GTags. GTags and timestamp-based versioning STMs are a natural fit because the SAL can be stored in the tags, next to the actual data. Because of the tight coupling of the tags and the data in the cache lines, the lock access will force the inclusion of the memory location to the cache. This acts as "free" prefetching of the corresponding memory location that will be used in the transaction. This improves the performance of `tx_read` operation that is on the critical path of the transactional execution (`tx_read` always reads the lock from the SAL to get the version number). With GTags, the combined write-back of the tag and the new memory value saves one write to the memory per write-set entry. As we demonstrate later, this can provide big improvements of the `tx_commit` operation in large transactions. `tx_validate` also benefits from GTags by having better cache locality. In the following sections we explain in detail how to extend TinySTM with GTags.

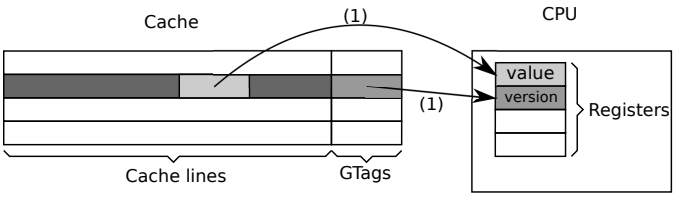
D. Improving the tx_read operation

In typical lazy versioning TM systems, an unmodified `tx_read` operation consists of three phases: (*i*) the query of the write-set phase, (*ii*) the address and version read phase, and (*iii*) the read-set update phase. On Figure 3, we can see the pseudo code for the `tx_read` operation while the address and version read phase of `tx_read` is represented explicitly in the code, and graphically. This part of the code can not



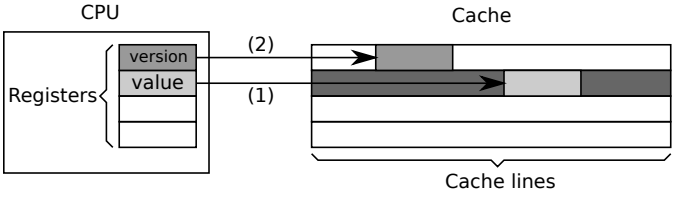

```
    tx_read(addr) {
      <write_set_query>
    read_value_start:
(1)   version0 = load_lock(&addr);
(2)   value = *addr;
(3)   version1 = load_lock(&addr);
      if (version0 != version1) { goto read_value_start;}
      <read_set_update>
      return value; }
```

Fig. 3. Tx_read in lazy versioning STM.




```
    tx_read(addr) {
      <write_set_query>
(1)   load_tag_and_value(addr, &version, &value);
      <read_set_update>
      return value; }
```

Fig. 4. Tx_read in TagTM.




```
    tx_commit() {
      try_acquire_locks();
      tx_validate();
      for (ws_entry in write_set) {
(1)     *ws_entry.addr = ws_entry.value;  // update memory
(2)     *ws_entry.lock_addr = new_version;  // lock release }}
```

Fig. 5. Tx_commit in lazy versioning STM.

be optimized by the compiler, because the exact ordering of the read instructions is necessary for the correct execution of `tx_read`.

To improve the performance of the `tx_read` operation, we use GTags' *load tag and value* instruction (Table I) to combine the read of the memory address and the read of the corresponding lock. This improves the performance of `tx_read` by reducing the number of cache accesses in the address and version read phase, which reduces the number of cycles spent waiting for the memory subsytem. The use of GTags in `tx_read` is presented on Figure 4 where the single *load tag and value* instruction is executed to load the address value and version, instead of 3 separate instructions. This simplifies the `tx_read` operation and improves its performance. Because `tx_read` operation is on the critical path in STMs [18], this has positive impact to the application's performance. In Section IV, we show the reduction in the number of executed cycles for the second phase of `tx_read`.

E. Improving the tx_commit operation

In typical lazy versioning TM systems, unmodified `tx_commit` consists of three phases: (*i*) the lock acquisition phase, (*ii*) the validation phase, and (*iii*) the write-back[3] phase (Figure 5). The write-back phase of `tx_commit` is represented explicitly in the code, and graphically.

To improve the performance of `tx_commit` operation, we use GTags' *store tag and value* instruction to combine the write to the memory address and the write to the corresponding lock. This improves the performance of the write-back phase by reducing the number of executed instructions, and the number of updated cache lines, and by releasing the lock earlier and thus publishing the results sooner.

The use of GTags in `tx_commit` is presented in Figure 6. The code for the lock acquisition phase and for the validation phase is the same in both versions of the code. The difference exists in the write-back phase. In this phase, the original `tx_commit` operation without GTags has to execute two memory writes per write-set entry, one for the memory update and other for the lock release operation[4]. `tx_commit` with GTags can execute just one *store tag and value* instruction that will update the memory reference and will release the lock. Because the `tx_commit` operation is on the critical path in STMs, this has positive impact to the application's performance. In Section IV, we show the reduction in the number of cycles for the write-back phase.

F. Modifying remaining transactional operations

The implementation of `tx_validate`, `tx_write`, and `tx_abort` operations is the same in TinySTM and in TagTM, with one small difference. In original TinySTM, these operations access the locks from the SAL, and in TagTM, these operations access the locks stored in the cache line tags. GTags improve the performance of `tx_validate` operation indirectly, by having better cache locality than original TinySTM. The improvement is attributed to better use of the cache line associativity because the locks stored in tags do not compete with data for the associativity (Figure 9). In Section IV, we show the reduction in a number of executed cycles the for the validation phase.

[3]The write-back phase includes the release of the acquired locks from the lock acquisition phase.

[4]The release also updates the version number.

Feature	Description
Processors	1 to 32 DECAlpha cores, in-order, single-issue
L1 Cache	64KB, private, 4-way assoc., 64B line, 2-cycle access
Coherence protocol	MESI protocol
L2 cache	8MB, shared, 32-way assoc., 64B line, 16-cycle access
Memory	300-cycle off chip access
GTags	Every cache line is extended with 32-bit tag.

TABLE II
THE SIMULATION PARAMETERS.

IV. EVALUATION

For the evaluation we use the M5 full system simulator [3]. The bus-based coherency protocol is replaced with directory-based MESI cache coherence protocol. We use in order DECAlpha CPU cores extended with the new instructions for tag manipulation. We extend the cache lines to store tags and extend the cache coherence protocol to make data and tags cache coherent. The configuration parameters used for the simulation are shown on Table II.

Minh et al. created STAMP [4], a state-of-the-art benchmark suite for evaluating TM systems. We use STAMP to compare the unmodified TinySTM against TagTM. We use the recommended input parameters for the application in STAMP for simulation run [4].

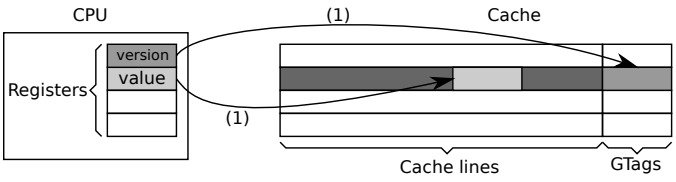
A. Transactional operations performance improvements

Figures 7, 8, and 9 show the time spent in memory hierarchy for the address and version read, the write-back, and the validation phases in STAMP benchmarks respectively. The X axis of the graph shows the benchmarks with TinySTM and TagTM, interleaved. The Y axis depicts the time that is normalized to TinySTM. The cycles spent for memory accesses are broken down in three parts for L1, L2, and main memory accesses. The results are shown for the benchmarks running with one thread on one CPU, in order to see better the pure effect of GTags on transactional operations.

Figure 7 suggests that GTags reduce the number of memory accesses in "address and version read" phase of `tx_read` operation because of the improved spatial locality of data and tags. Vacation and Bayes benchmarks show the biggest time reduction (up to 41.93% for Vacation-low) because GTags successfully eliminate the accesses to main memory. The other applications show a modest improvement of execution time, which is attributed to the reduction in the number of accesses to the L2 cache.

Figure 8 suggest that GTags reduce the number of memory accesses in the write-back phase of `tx_commit` operation because of the improved spacial locality of the data and tags. Almost all the benchmarks show reduction in accesses to main memory and to L2 caches. This presents a large performance improvement of 58.96% (geometric mean of all the applications) for the write-back phase.

Figure 9 suggests that GTags reduce the number of memory accesses in the "validation" phase of `tx_commit` operation because the number of accesses to main memory is greatly




```
tx_commit() {
  try_acquire_tags();
  tx_validate();
  for (ws_entry in write_set) {
    // update memory and release lock
(1) store_tag_and_value(ws_entry.addr, new_version, ws_entry.value); }}
```

Fig. 6. Tx_commit in TagTM.

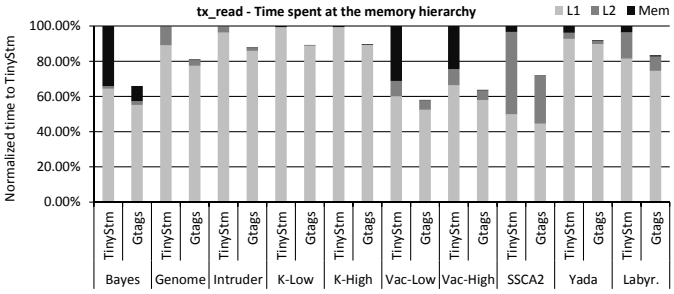


Fig. 7. `tx_read` - Time spent in memory hierarchy. Time is normalized to TinySTM.

reduced. The validation routine has to traverse all the locks that are stored in the transaction's read-set during the short time interval which will populate the caches with the locks from the SAL. This will kick out some of the transactional data from the caches to the main memory. TagTM does not exhibit this problem because the locks stored in the tags do not compete with transactional data for the caches, therefore GTags effectively increase the associativity of the caches for transactional applications. The performance benefit of GTags for the validation is 26.76% for the applications with small and medium read-sets (geometric mean for Kmeans-high, Kmeans-low, Vadation-high, and Vacation-low). The benefit is bigger for the applications with medium and large read-sets and is 85.55% (geometric mean for the rest of the STAMP applications).

B. Transaction execution performance improvements

Figure 10 shows the speedup of TagTM over TinySTM for committed transactions. The X axis shows STAMP applications running from 1 up to 32 threads. The Y axis shows the speedup gain provided by GTags. The right most columns on the graph represent geometric mean of the speedup of the applications. Three applications from STAMP (Kmenans, Vacation, and SSCA2) show similar behaviour when running on different number of CPUs. All of them have more or less constant performance improvement while the number of CPUs change (except Kmeans on 32 CPUs). SSCA2 shows the best speedup (of 52%). This improvement comes from the fact that SSCA2 has the biggest transactional overhead

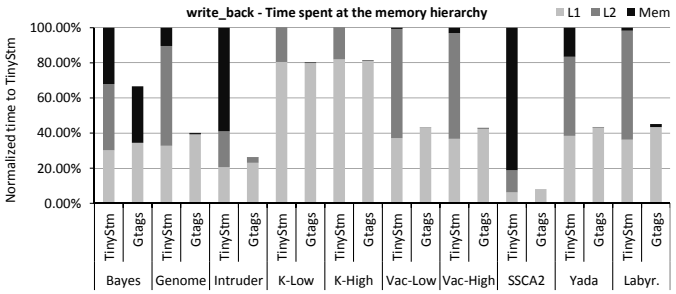


Fig. 8. `write_back` - Time spent in memory hierarchy. Time is normalized to TinySTM.

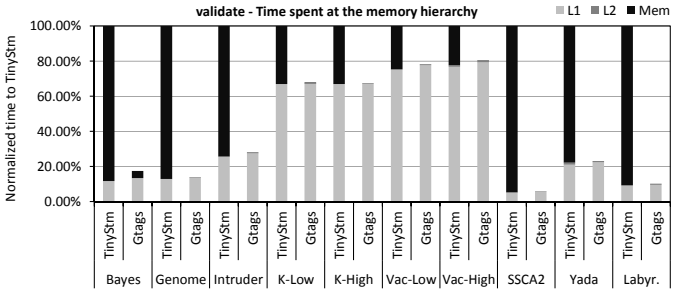


Fig. 9. `validate` - Time spent in memory hierarchy. Time is normalized to TinySTM.

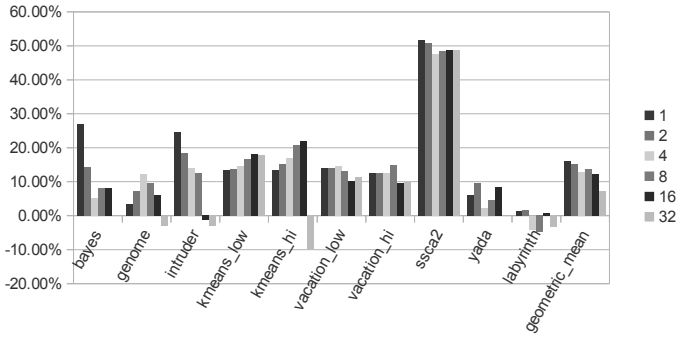


Fig. 10. Percentage speedup of TagTM over TinySTM for committed transactions

Cache type	Regular L1	L1 extended with GTags
Size	64KB	64KB + 8KB for GTags
Cache line size	64 bytes	64 bytes + 8 bytes for GTags
Other	4-way, 1 bank, 45nm	4-way, 1 bank, 45nm
Total area (mm2)	0.841	0.892 (6.05% increase)

TABLE III
CACHE PARAMETERS CALCULATED WITH CACTI 5.3.

of 503.52% (Figure 1). Kmeans (low and high) and Vacation (low and high) have similar transactional overheads and exhibit the similar speedup with GTags of about 13.28% (geometric mean). Intruder has a performance gain of about 17.2% while running from 1 to 8 threads, and has a performance loss for 16 and 32 threads. The rest of transactional applications (Genome, Bayes, Yada, and Labyrinth) have small transactional overhead, and show that the benefit of GTags can be positive (for Bayes and Yada) or negative (for Labyrinth). The performance benefit happens when the (transactional) data and the lock are accessed in transaction. In the applications with small transaction overhead, the performance loss happens when the non-transactional data pays the price of fetching tags that will not be used. This is the one of the reasons why Labyrinth experiences performance degradation. Overall, we have an average speedup of 13.05% in committed transaction execution time while running on 1-32 cores.

C. GTags - L1 cache overhead

We use CACTI 5.3 [14] to evaluate the increase in L1 cache area caused by adding GTags to the data part of the cache lines (see Table III). GTags increase the L1 cache area by 6.05%. This is a small overhead compared to the total cache size, because GTags utilize the already existing lookup logic.

V. RELATED WORK

Saha et al. proposed HaSTM [13] for improving the performance of STM systems by using additional bits per cache line. Every 16 bytes of an L1 cache line is protected with 1 bit. HaSTM uses these bits to eliminate redundant logging and to eliminate validation overhead altogether for transactions whose transaction records fit in the cache. In the case of GTags, the tags speed-up the logging by reducing the time necessary to accesses to global version table.

Ming el al. proposed SigTM [11] that uses hardware signatures to track the read-set and write-set for pending transactions and to perform conflict detection between concurrent threads. All other transactional functionality, including data versioning, is implemented in software. However, the biggest performance benefit of SigTM is the elimination of read-set logging. GTagTM do not have the problem of false aborts because the version information stored in tags is exact.

Hammond et al. introduced Transactional Memory Coherence and Consistency (TCC) [7] to execute transactions in hardware. Their system changes the coherence hardware and require that all the code executes simple transactions. On the other hand, GTags requires no changes to existing coherence protocols and can be used to speedup existing software TM systems.

Harris et al. proposed Dynamic Filtering (DF) [9], a multi-purpose architecture support for language runtime systems, to speedup software TM systems and to reduce the overheads in language base security systems. DF reduces the metadata accesses in eager STM systems by reducing unnecessary transactional logging. GTags reduce the metadata accesses by reducing the overheads associated with accesses to cache-line locks/timestamps.

Adl-Tabatabai et al. designed compiler and runtime support for software TM [1] which is able to reduce the overheads of STM. Their implementation should benefit from the uses of GTags because, atomic GTags' instructions can enable some further optimizations that would reduce the STM overheads even more.

Baugh et al. [2] used fine-grained protection mechanism

to isolate transactional data in an implementation of TM with strong atomicity, and to separate hardware-managed and software-managed transactional data in hybrid systems. This approach could be used to extend use of GTags for hardware HTM systems.

Several other proposals add hardware for fast meta-data acesses and application monitoring. Zeldovich et al. [17] implementad Loki, a tagged memory architecture, to enforce appilcation security policies in hardware. Venkataramani et al. [15] propose hardware support for memory access monitoring tasks. Chen et al. [6] use hardware extensions to accelerate metadata accesses.

VI. Conclusion and future work

In this paper we introduce TagTM, a software TM system augmented with new hardware mechanism that we call GTags. GTags are new hardware cache coherent tags used for fast meta-data access. TagTM use GTags to reduce the cost associated with accesses to the transactional data and corresponding metadata. For the evaluation of TagTM, we use STAMP benchmark suite. In the average case TagTM provide the speedup of 7-15% (across all STAMP applications), and in the best case shows up to 52% speedup of committed transaction execution time (for SSCA2).

Acknowledgment

We would like to thank Nehir Sonmez, Vesna Smiljkovic, Vladimir Gajinov, and all anonymous reviewers for their comments and valuable feedback. This work is supported by the agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union under contracts TIN2007-60625 and TIN2008-02055-E, and by the European HiPEAC Network of Excellence.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, 41:26–37, June 2006.

[2] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. 11th IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, September 2008.

[6] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *SIGARCH Comput. Archit. News*, 36:377–388, June 2008.

[7] 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*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2nd edition, June 2010.

[9] T. Harris, S. Tomic, A. Cristal, and O. Unsal. Dynamic filtering: multi-purpose architecture support for language runtime systems. *SIGARCH Comput. Archit. News*, 38:39–52, March 2010.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[11] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proc. 34th International Symposium on Computer architecture*, pages 69–80, June 2007.

[12] T. Riegel, P. Felber, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, February 2008.

[13] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proc. 39th IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, December 2006.

[14] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[15] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[16] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *SPAA '08: Proc. 20th annual symposium on parallelism in algorithms and architectures*, pages 265–274, June 2008.

[17] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

[18] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 285–294, New York, NY, USA, 2010. ACM.