Partition-Based Hardware Transactional Memory for Many-Core Processors

Yi Liu, Xinwei Zhang, Yonghui Wang, Depei Qian, Yali Chen, Jin Wu

Partition-based Hardware Transactional Memory for Many-core Processors

Yi Liu[1], Xinwei Zhang[1], Yonghui Wang[1], Depei Qian[1], Yali Chen[2], Jin Wu[2]

[1] Sino-German Joint Software Institute, Beihang University, Beijing 100191, China
[2] Huawei Technologies Co., Ltd, Shenzhen 518129, China

yi.liu@jsi.buaa.edu.cn

Abstract. Transactional memory is an appealing technology which frees programmer from lock-based programming. However, most of current hardware transactional memory systems are proposed for multi-core processors, and may face some challenges with the increasing of processor cores in many-core systems, such as inefficient utilization of transactional buffers, unsolved problem of transactional buffer overflow, etc. This paper proposes PM_TM, a hardware transactional memory for many-core processors. The system turns transactional buffers that are traditionally private to processor cores into shared by moving them from L1-level to L2-level, and uses partition mechanism to provide logically independent and dynamically expandable transactional buffers to transactional threads. As the result, the solution can utilize transactional buffers more efficient and moderate the problem of transactional buffer overflow. The system is simulated and evaluated using gems and simics simulator with STAMP benchmarks. Evaluation results show that the system achieves better performance and scalability than traditional solutions in many-core processors.

Keywords: Key words: Transactional Memory, Partition, Many-core

1 Introduction

Among works to improve programmability of parallel systems, transactional memory is an attractive one. Compared to traditional lock-based programming models, transactional memory can improve programmability, avoid deadlock and furthermore, promote performance of concurrent programs.

Most of current hardware transactional memory (HTM[1]) systems are proposed for multi-core processors, and may face some challenges with the increasing of processor cores in many-core systems: firstly, utilization of transactional buffers are inefficient since those buffers are private to processor cores while generally only part of cores execute transactions simultaneously in many-core processors; secondly, the on-going challenge of transactional buffer overflow for HTMs is still unsolved.

In this paper, we propose PM_TM, an architecture of hardware transactional memory for many-core processors. The main idea consists of two points: firstly, turns private transactional buffer into shared by moving them from L1-level to L2-level; secondly, uses partition mechanism to provide logically independent and dynamically

expandable transactional buffers to transactional threads, and furthermore, to isolate access-interferences among large number of processor cores. As the result, the system can utilize transactional buffers more efficient and moderates the problem of transactional buffer overflow in many-core processors.

The rest of this paper is organized as follows. Section 2 analyzes problems of traditional hardware transactional memory in many-core environment and then gives an introduction to our solution. Section 3 presents the architecture of our proposed system. Section 4 evaluates the system with benchmarks. Section 5 introduces related works. And section 6 concludes the paper.

2 Challenges in many-core processors and our solution

2.1 Problem analysis

Most of current hardware transactional memory systems are proposed for multi-core processors, and may face some challenges in many-core systems.

Firstly, transactional buffers are inefficiently utilized and resources are wasted. Traditionally, transactional buffer is located inside processor cores in parallel with L1 data cache, which means it's private for the core. Processor core accesses transactional buffer only in transactional state, i.e. on executing transactions. However, in many-core processors, there will be a large number of processor cores, and generally only small part of them execute transactions simultaneously, while most of transactional buffers are not used at all.

Secondly, Problem of transactional buffer overflow is still unsolved. Generally, size of transactional buffer is fixed in each processor cores, when a transaction reads/writes too many data, buffer overflow will occur. This "buffer overflow problem" is one of ongoing challenges for hardware transactional memory. Despite some solutions have been proposed, most of them rely on co-working between cache-level transactional buffer and main memory or virtual memory, and need complex hardware/software operations.

This problem will even cause some kind of contradictions in many-core processors. On one hand, transactional buffers are inadequate in some processor cores that cause transactional buffer overflows due to some "long transactions", while on the other hand, transactional buffers in other processor cores may not be used at all.

2.2 Our solution: an overview

Based on the above discussions, we propose an architecture of hardware transactional memory, called PM_TM, for many-core processors. In our proposed solution, the transactional buffer is "logically independent" because that the transactional cache is partitioned into multiple partitions, each of them corresponds to one transactional thread and can only be accessed by it. The transactional buffer is "dynamically expandable" because that each partition is initially allocated a buffer with basic size, and can be expanded if the corresponding thread accesses excessive data speculatively in a transaction.

The advantages of the proposed solution include:

(a) Transactional buffers can be utilized more efficiently. In many-core environment, generally only part of processor cores execute transactions at the same time, that is, most of transactional buffers will be idle if they are private for cores. By turning them from private to shared, all of transactional buffers can be utilized by ongoing transactions, and the waste of resources can be reduced greatly.

(b) The problem of transactional buffer overflow is moderated. Since the transactional buffers are shared by all of the processor cores, and generally only part of cores execute transactions simultanously, by managing transactional buffers with partition mechanism and expanding partitions when necessary, a transaction can have much bigger transactional buffer than traditional private buffer. As the result, the possibilities of transactional buffer overflow are smaller.

In addition, from the implementation point of view, it is easier to integrate much bigger L2-level transactional buffer into processors than L1-level.

(c) Context switch and migration of transactional threads are easier to implement. For some long-transactions or transactions with system-calls, the operating system will suspend the transactional thread inside a transaction and schedule other threads to run on the core. After a while, the original transactional thread will be re-scheduled to run on either the same or a different core. Traditionally, this is a problem for HTMs since the transactional thread may face either a damaged or a totally new transaction context. In our proposed solution, the transactional buffer are shared by all of the cores and bound to transactional threads instead of cores, so the context switch and migration of transactional threads can be easily supported.

3 Partition-based hardware transactional memory architecture

3.1 System architecture

Fig.1 shows the architecture of tile-based[2],[12] many-core processors with support of our proposed hardware transactional memory. The system is composed of three types of tiles: the first type is tiles of processor cores plus private L1 cache and routing mechanism; the second is tiles of L2 cache banks; and the third is tiles of transactional cache (TC). All of the tiles are connected with an on-chip network, and both L2 cache and transactional cache are shared by all of the processor cores.

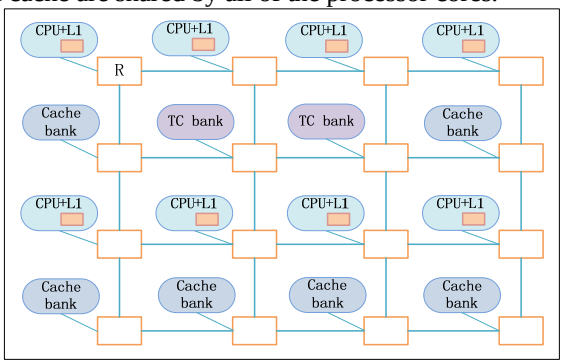


Fig. 1. System architecture

The transactional cache is used to buffer the data accessed by transactions speculatively, and as shown in Fig.2, its structure is similar to L2 cache and data is also stored by line. The difference is that the transactional cache holds both *old* and *new* version of data for each line, where the *old* version is the data that the transaction started with, and the *new* version is the current version updated by the transaction speculatively. Detailed introduction to consistency and conflict detection are given in section 3.3.

In addition, to support efficient nesting of transactions, the system uses a partial rollback mechanism which was proposed in our previous works [11]. The mechanism uses n-bits read and write vector for each line to indicate whether the line has been read or written speculatively, with each bit in the vector corresponding to one level of nested transactions. By adding limited hardware, conditional partial rollback can be implemented, that is, when a transaction needs to roll back due to a conflict, instead of rolling back to the outermost transaction as in commonly-used flattening model, the system can just rolls back to the conflicted transaction itself or one of its outer-level transactions if given conditions are satisfied.

Tag	State	Data		R	W
		old	new		
A					
B					

Fig. 2. Structure of transaction cache

In architecture level, transactional cache is in parallel with L2 cache. Commonly a processor core accesses L2 cache, and once a transaction is started, it switches to access transactional cache in order to guarantee that all the speculative accessed data are buffered and not valid until commit of the transaction.

3.2 Partition mechanism

(1) Overview

Partition mechanism is a method to manage hardware resources in multi-/many-core processors. In our proposed system, partition mechanism is used to establish multiple logically independent transactional buffers, i.e. partitions, in shared transactional cache, furthermore, to make these transactional buffers dynamically expandable. In the system, transactional caches are allocated to partitions in partition-unit (PU) which corresponds to multiple successive lines in the transactional cache. Each partition corresponds to one transactional thread, and is created with one PU initially that can expand to multiple successive PUs if it is needed.

Partitions are created only for threads that execute transactions, called transactional thread. Once a thread executes a transaction at the first time, a partition is created with only one PU, and along with the increasing of read/write data set of a transaction, more PUs can be allocated to the partition dynamically if its transactional buffer overflows. After the commit of the first transaction, the partition is reserved and used by subsequent transactions of the thread, and finally released when the thread is finished.

It is noted that the owner of a partition is transactional thread rather than processor core. The reason is that transactional threads may be suspended during its execution

and scheduled to run in another core after it is waked up later. In other words, migration of transactional threads can be supported by binding partitions with transactional threads instead of cores.

(2) Partition access

When a thread starts to execute a transaction, it switches to access its transactional buffer (i.e. partition) instead of L2 cache. As Fig.3 shows, an associative buffer is used to store information of partitions including starting address, partition size and owner thread, with each entry corresponding to one partition. Based on this hardware infrastructure, the transactional buffer of a thread can be located quickly by means of the thread ID.

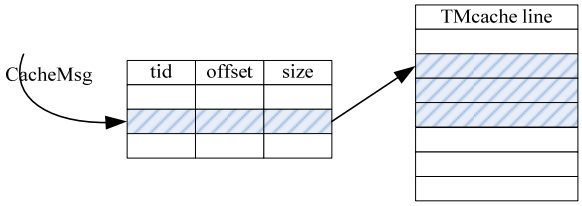


Fig. 3. Structure of transaction cache

(3) Partition management

As mentioned above, a partition starts with one PU and may expand along with the execution of the transactional thread if excessive data are accessed in a transaction speculatively, and to simplify the management of partitions, it is limited that all of the PUs of a partition must be successive in transactional cache. In order to leave spaces at the end of partitions for potential expansions in the future, it's better to allocate partitions dispersedly in transactional cache.

According to above discussions, the system allocates the first partition from the beginning of transactional cache, and subsequent partitions are allocated in the following policy: searching for the biggest free area in the transactional cache, and allocating the PU in the middle of the area to the new partition.

Table.1 shows the addresses that will be allocated to partitions one by one, where B is the total size of transactional cache in lines, N is the number of processor cores, and the size of partition unit PU=B/N.

Table 1. Address allocation of partitions

Seq. of creation	Start address	Initial end address
0	*0*	*PU - 1*
1	*B/2*	*B/2 + PU - 1*
2	*B/4*	*B/4 + PU - 1*
3	*3B/4*	*3B/4 + PU - 1*
4	*B/8*	*B/8 + PU - 1*
...	...	...

A partitioning example is shown in Fig.4. In Fig.4(a), four partitions are created one by one for transactional thread T0--T3, and T0 has successfully expanded its partition in 1 PU; in Fig.4(b), thread T0 and T3 finish their execution and partitions are released, after that, a new partition is created for thread T4.

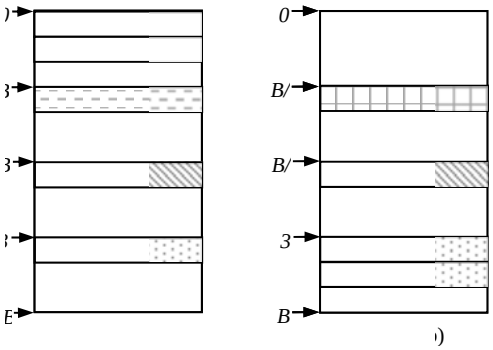


Fig. 4. Partitioning example

In addition, it is necessary to consider some extreme situations. Since partitions are created and maintained for transactional threads instead of processor cores, in some cases, there may be too many transactional threads in the system, or there may be some long-running transactional threads that occupy large amount of transactional cache permanently, so the associative table or transactional cache may be used out. At this time, a LRU-like (Least Recently Used) discard policy can be used to discard and release the partition that was not accessed for the longest time, the difference with the LRU policy is that, to make things simple, the partition is discarded instead of swap to main memory as in LRU. If the owner thread of the discarded partition re-accesses its partition later, a new partition will be created for it.

3.3 Consistency and conflict detection

The system uses lazy data version management and eager conflict detection policy, that is, all of the data modified in a transaction are buffered in the transactional buffer and invisible to other processor cores until commit of the transaction; and each memory access of a transaction is checked to identify if there is a conflict among transactions.

(1) Consistency and directory

When a processor core starts to execute a transaction, it flushes data in L1 data cache and L2 cache to lower level and main memory in order to guarantee data consistency. In addition, write-through policy is used for L1 cache to write data directly into transactional buffer.

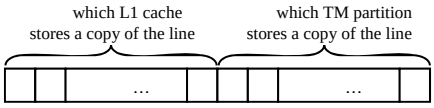
To achieve scalability in many-core processors, the consistency among multiple L1 data cache and transactional buffers(partitions) is maintained by distributed cache directory. As Fig.5(a) shows, structure of the directory is the same with ordinary cache directory except that the sharer list in each entry is extended to record not only which L1 cache but also which transactional buffer(partition) stores a copy of the line, as shown in Fig.5(b).

Once a line of L1 cache is updated, an invalidate message will be sent to each directory that stores a copy of the updated line. If such an invalidate message is received by directory of transactional cache, one or more conflicts will be triggered, depending on the number of partitions that store copy of the line. As the result, the

corresponding transactions will abort their execution and roll back. Similarly, there also is a directory with the same structure in transactional cache. Once a line of transactional buffer is updated, an invalidate message is also sent to other sharers.

Tag	State	Sharer List
addr1	...	010..10
addr2	...	100..00
...	...	...

(a) cache directory



(a) sharer list

Fig. 5. Structure of cache directory

(2) Conflict Detection

Method to detect conflicts among transactions is: when a processor P reads/writes address A in a transaction, the cache controller sends a share-/exclusive-request to transactional cache directory, once the reply is received, it sets status of the transactional cache line to shared or exclusive; meanwhile, if another processor Q accesses address A too, the request is forwarded to processor P to identify if there is a write-write or read-write conflict, and consequently, to approve or reject the request.

Fig.6 shows examples of conflict detection:

(a) Transaction startup: processor P starts a transaction and switches to access transactional buffer.

(b) Writing data: P writes address A0 which is not in its transactional buffer, firstly it sends a get-exclusive request to directory, which is approved with the requested data, then P stores the data to its transactional buffer and sets write-flag, finally it replies an ACK to directory.

(c) Reading data: P reads address A1 which is not in its transactional buffer, the procedure is the similar to (b) except that the request is get-shared instead of get-exclusive.

(d) Transaction conflict: processor Q reads the address A0 which was just written by P, firstly it sends a get-shared request to directory, which is forwarded to P and identified as a read-write conflict, then a NACK is sent back to Q; Q deals with the conflict after receiving the NACK and replies a NACK to directory.

(e) Successful shared reading: processor Q reads the address A1 which was just read by P, the request is identified as conflict-free and approved.

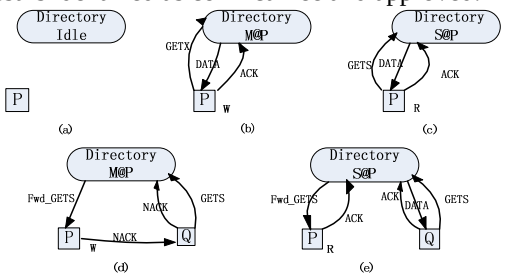


Fig. 6. Examples of Conflict Detection

3.4 Execution of Transactions

When a processor core starts to execute a transaction, it flushes data in L1 data cache and L2 cache to lower level and main memory, and switches to access transactional buffer instead of L2 cache.

During execution of a transaction, all of the data accessed by the transaction are buffered temporarily in its transactional buffer(partition). When a data is accessed for the first time in the transaction, it is loaded to both old and new version of the line in transactional buffer, subsequent updates to the data are just stored to new version, and R/W status are set at the same time.

Once the read/write-set of a transaction exceeds the partition size, a transactional buffer overflow occurs. At this time, the system tries to expand the partition by allocating one more PU in transactional cache. If the successive PU at the end of the partition is free, the partition can be successfully expanded and memory accesses are continued, otherwise it stalls for a short period of time and tries again. If there is still no free PU, the expanding operation fails and a global lock is set, the overflowed transaction continues exclusively without conflict until its commit. Of couse, the performance will be suffered in this situation.

If a transaction needs to roll back in case of conflict, data in transactional buffer are copied from old to new line by line, at the same time, R/W status are cleared, after that, all of the lines in L1 data cache are set to invalidate.

When a transaction finishes its execution and commit, all of the updated data in transactional buffer(partition) are written to main memory.

3.5 ISA extensions and programming interface

As a hardware transactional memory, PM_TM supports transparent execution of transactions with no restriction on programming languages. Only two instructions are extended to specify the start and end of a transaction, as shown in Table.2. Programmers just need to identify program statements that must be executed atomically in their applications, and define them as transactions by inserting appropriate API at the beginning and the end of each transaction.

Table 2. Address allocation of partitions

Instruction	Description	Programming interface
XB	Trans. start	BEGIN_TRANSACTION()
XC	Trans. end	COMMIT_TRANSACTION()

4 Experiments and Evaluation

4.1 Experimental environment

The proposed system is simulated in GEMS[13] and Simics[14], and by extending the simulator, our partition mechanism and consistency protocol are implemented on SPARC-architecture processors in the simulator.

We evaluate PM_TM system using Stanford STAMP[15] benchmark, and experimental results are compared with LogTM[5] and a native HTM, called NativeTM, in which transactional buffers are in L1-level and private to each processor core.

Table 3 summarizes parameters of the simulated target system.

Table 3. Configuration of target system

Processors	Ultrasparc-iii-plus, 1GHz
Cache size	L1: 64KB L2: 4MB
Size of cache line	64 bytes
Memory	1GB 80-cycle latency
Cache coherence protocol	MESI_CMP_filter_directory
Interconnection network	Tiled NoC; X-latency:1, Y-latency:2
Transactional cache	PM_TM: 1/2/4 MB; NativeTM: 8KB/core
Operation System	Solaris 10

The evaluation uses 4 applications that vary in size of read/write data set, length of transactions and contention degree among transactions, as in Table 4.

Table 4. Applications from STAMP benchmark

Application	R/W Set	Len. of transactions	Contention
intruder	medium	short	high
kmeans	small	short	low
vacation	large	medium	low
bayes	large	long	high

4.2 Results and analysis

(1) Performance

Fig.7 shows average execution time of applications in PM_TM, LogTM and NativeTM with 4--128 processor cores. Each application is executed with number of threads equaling to processor cores.

We can see from Fig.7 that PM_TM behaves not very well in less processor cores. The main reason is that access latency of transactional buffers in PM_TM is longer than others due to its L2-level location. Along with the increasing of processor cores, PM_TM achieves better performance than two other systems, since that less transactional buffer overflow occur in PM_TM, and contentions among transactions are also handled more efficient in PM_TM.

Fig.7 also shows that results of different applications are not quite the same due to their characteristics. Kmeans has not only less transactions but also small read/write data set, so there is few transactional buffer overflows during the execution. As the result, the performance of Kmeans in PM_TM is not improved by the partition mechanism, instead, the performance is influenced by the long access latency of transactional buffers. Compared with kmeans, the intruder application has bigger read/write data set and higher contentions among transaction. So PM_TM achieves better performance along with the increasing of processor cores. Vacation has almost the same size of read/write data set with intruder, but vacation has some long transactions and contention in vacation is lower than intruder. Compared to other applications, the

bayes has bigger read/write data set and longer transactions. Contention is also higher than the others.

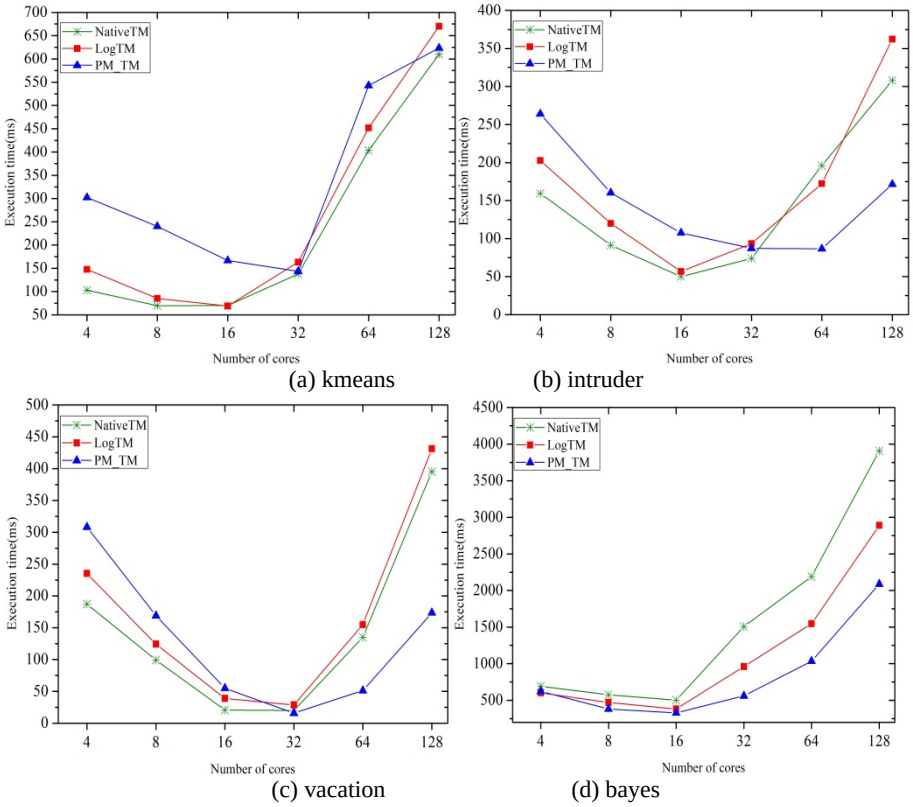


Fig. 7. Average execution time of applications

Table 5. Transaction overflows

Application	System	Transactional buffer size	Number of processor cores					
			4	8	16	32	64	128
kmeans	NativeTM	8KB/core	0	0	0	0	0	0
	PM_TM	1MB	0	0	3	5	6	8
		2MB	0	0	2	3	5	6
		4MB	0	0	0	0	2	5
intruder	NativeTM	8KB/core	12	24	48	96	192	384
	PM_TM	1MB	0	0	4	8	29	52
		2MB	0	0	3	4	9	31
		4MB	0	0	0	0	3	7
vacation	NativeTM	8KB/core	19	33	56	131	263	477
	PM_TM	1MB	0	6	13	18	45	104
		2MB	0	0	7	11	18	47
		4MB	0	0	0	5	9	21
bayes	NativeTM	8KB/core	361	733	1307	1891	3249	4811
	PM_TM	1MB	173	267	661	1081	2033	4795
		2MB	30	181	277	649	1213	2258
		4MB	0	24	190	307	636	1309

(2) Transactional buffer overflows

Table 5 gives transaction overflow statistics of applications in NativeTM and PM_TM. LogTM is not included in this table because transactional data of LogTM is stored in the memory directly. From the table we can see that most applications overflow less in PM_TM than in NativeTM except kmeans, which has not only less transactions but also small read/write data set. Furthermore, with the increasing of L2-level transactional cache, the overflow times reduce significantly. As discussed in section 2.2, from the implementation point of view, it is easier to integrate much bigger L2-level transactional buffer into processors than L1-level buffer.

(3) Conflict and rollback

Fig.8 shows transaction rollbacks of applications. LogTM uses bloom filter[16] to store transaction read/write data set, that may produce false-conflicts, and furthermore, the cost of abort in LogTM is much higher due to its eager version management. Compared to LogTM, PM_TM uses bit-set to record data set of transactions so that there is no false-conflict in it. And due to this reason, the number of conflicts in NativeTM is the same with PM_TM, and omitted in the figure.

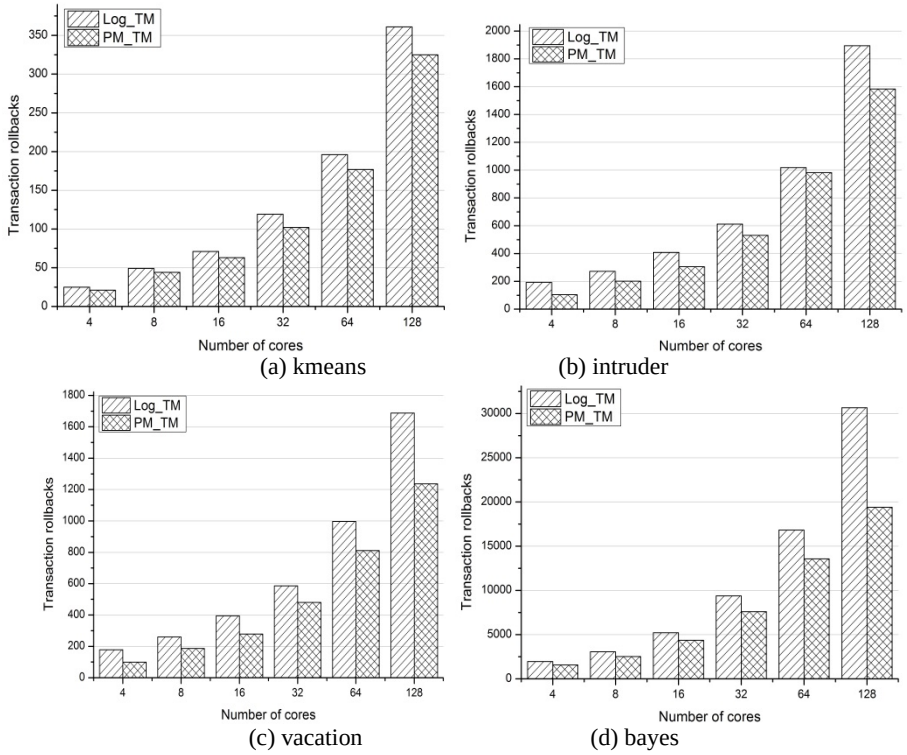


Fig. 8. Transaction rollbacks

In kmeans, transactions are small and shared data among transactions are few, so frequency of conflict is much lower than other programs. Transactions in intruder are slightly larger than kmeans, but frequency of conflict is much higher. Although vacation has some large transactions, competition between transactions in vacation is low-

er than intruder. As for bayes, PM_TM system is much better than LogTM in rollback test. Bayes has the largest R/W set among the four programs, so overflow times of bayes is the most.

5 Related Works

Transactional memory was firstly proposed in [1], since then, lots of hardware transactional memory (HTM) systems have been proposed that support atomicity of transactions by hardware, and achieves high performance. On the other hand, HTMs are often bounded by space and time constraints, i.e. transactional buffer overflow and transaction migration.

Some solutions have been proposed to deal with transactional buffer overflow. The simplest solution is partial-commit or in-place commit which uses a global lock or things like that to prevent other transactions to commit, until commit of the overflowed transaction [2],[7],[8]. Beside partial-committing, some solutions deal with overflows by co-working between transactional buffer and memory[5],[6]; some solutions support unbounded transactions by means of complex hardware mechanism[4],[10]; hybrid transactional memory[9] has also been proposed, which integrates both hardware and software transactional memory, and switches to software mode in case of buffer overflow.

TM systems for many-core processors have also been proposed. TM^2C[17] is a software TM system which provides two services: the application service and the Distributed TM service. The former connects transaction with the application and controls the transactional runtime. The latter grants a data access to the requesting transactions through distributed locking. The main contribution of TM^2C lies in guaranteeing starvation-freedom with low overhead.

LogTM[5] is a log-based HTM system. It saves old values in a log and puts new values in target address. When transaction commits, values in target address become visible and log is abandoned directly. This will accelerate the process of transaction committing. When rollback occurs, it simply copies old values in the Log to the target address. LogTM uses directory-based Cache consistency protocol to guarantee data consistency and eager conflict detection to find conflict between transactions.

6 Conclusion

Transactional memory is an appealing technology to improve programmability of multi-core and many-core processors. However, most of current hardware transactional memory systems are proposed for multi-core processors, and may face some challenges with the increasing of processor cores in many-core systems: firstly, utilization of transactional buffers are inefficient since those buffers are private to processor cores while generally only part of cores execute transactions simultaneously in many-core processors; secondly, the on-going challenge of transactional buffer overflow for HTMs is still unsolved.

This paper proposes an architecture of hardware transactional memory for many-core processors, called PM_TM. The main idea consists of two points: firstly, turns

private transactional buffer into shared by moving them from L1-level to L2-level; secondly, uses partition mechanism to provide logically independent and dynamically expandable transactional buffers to transactional threads, and furthermore, to isolate access-interferences among large number of processor cores. As the result, the system can utilize transactional buffers more efficient and moderates the problem of transactional buffer overflow in many-core processors. The system is simulated and evaluated using gems and simics simulator with STAMP benchmarks. Evaluation results show that the system achieves better performance and scalability than traditional solutions in many-core processors.

7 Acknowledgements

This work was supported by National Science Foundation of China under grant No. 61073011, 61133004, and National Hi-tech R&D program(863 program) under grant No. 2012AA01A302.

8 References

1. Maurice Herlihy, J.Eliot B.Moss, Transactional Memory Architectural Support for Lock-Free Data Structure, 20th International Symposium on Computer Architecture, IEEE, 1993, 289-300.
2. Thomas Moscibroda and Onur Mutlu, A Case for Bufferless Routing in On-Chip Networks, 36th International Symposium on Computer Architecture, IEEE, 2009. 196-207
3. Lance Hammond, Vicky Wong, Mike Chen, et.al, Transactional Memory Coherence and Consistency, 31th International Symposium on Computer Architecture (ISCA2004), IEEE CS Press, 2004. pp. 53-65
4. C. Scott Ananian, Krste Asanovic and Bradley C. Kuszmaul, et.al, Unbounded Transactional Memory, 11th International Symposium on High-Performance Computer Architecture (HPCA2005), IEEE CS Press, 2005. pp. 316-327
5. Moore Kevin E, Bobba Jayaram, and Moravan Michelle J, et.al, LogTM: log-based transactional memory, 12th International Symposium on High-Performance Computer Architecture (HPCA2006), IEEE CS Press, 2006. pp. 258-269
6. Luis Ceze, James Tuck, et al. Bulk Disambiguation of Speculative Threads in Multiprocessors, 33rd International Symposium on Computer Architecture, 2006, 227-238.
7. Arrvindh Shriraman, Michael F. Spear, et al. An Integrated Hardware-Software Approach to Flexible Transactional Memory, 34th Annual International Symposium on Computer Architecture, ACM, 2007, pp. 104-115.
8. Arrvindh Shriraman, Sandhya Dwarkadas, Michael L.Scott, Flexible Decoupled Transactional Memory Support, 35th International Symposium on Computer Architecture, IEEE & ACM, 2008, pp. 139-150.
9. Sanjeev Kumar, Michael Chu, Christopher J. Hughes, et.al, Hybrid Transactional Memory, 11th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM Press, 2006. pp. 209-220
10. Ravi Rajwar, Maurice Herlihy and Konrad Lai, Virtualizing Transactional Memory, 32th International Symposium on Computer Architecture, IEEE CS Press, 2005. pp. 495-505.

11. Yi Liu, Yangming Su, Cui Zhang, et.al. Efficient Transaction Nesting in Hardware Transactional Memory, 23rd International Conference on Architecture of Computing Systems (ARCS), Germany, Feb 2010.
12. Michael Bedford Taylor, Walter Lee, Jason Miller, et.al, Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams, 31st Annual International Symposium on Computer Architecture, 2004. pp. 2-13
13. Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, et.al, Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, SIGARCH computer Architecture News, Nov. 2005. pp. 92-99
14. Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, et.al, Simics: A full system simulation platform, IEEE Computer Society, Feb. 2002, 35(2):50-58
15. Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis,et al., STAMP: Stanford Transactional Applications for Multi-Processing, 2008 IEEE International Symposium on Workload Characterization, IEEE CS Press, 2008. pp. 35-46
16. Burton H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, 1970, pp. 422-426.
17. Vincent Gramoli, Rachid Guerraoui, Vasileios Trigonakis, TM2C: a Software Transactional Memory for Many-Cores, ACM European Conference on Computer Systems (EuroSys2012), 2012, pp. 351-364.