ICIC Express Letters
Volume **10**, Number **10**, October **2016**
ICIC International ©2016 ISSN 1881-803X
pp. **2391–2396**

EFFICIENT SELF-TUNING STRATEGY FOR HARDWARE TRANSACTIONAL MEMORY

Ping Wu
College of Information and Control Engineering
Weifang University
No. 5147, Dongfeng Street, Weifang 261061, P. R. China
wping1980@163.com

Received February 2016; accepted May 2016

Abstract. *Transactional Memory offers a neat programming interface for concurrent software developing. Previous Software Transactional Memory system shows great advantages over locking on synchronizing concurrent accesses but it is trapped by its low performance. Recent hardware support from Intel and IBM draws new lights on reducing the huge overhead of transactional processing. However, certain hardware limitations have made it a difficult task to use hardware transactions efficiently. A fall-back strategy is needed to avoid constant abortion of hardware transactions due to the hardware limitations. We show in this paper that different fall-back strategies would hugely affect the performance of different applications and we introduce an efficient dynamic adjustment strategy to achieve optimal performance in all situations. Our key observation is that the atomic blocks in the same execution context are likely to fit the same strategy. Based on this observation, we show in experiments that our dynamic method could achieve an average speedup of 1.4X over static method and 1.05X over TUNER.*
Keywords: Hardware Transactional Memory, Fall-back strategy, Dynamic adjustment

1. **Introduction.** The prevalence of multi-core architecture has made multi-threaded programming the mainstream method for software developing. Multi-threaded programs require proper synchronization mechanism on accessing shared data. Fine-grained locking [1], in many years, has been the main and the most efficient way to achieve this. However, programming with locking has been proven to be difficult and error-prone [1]. Debugging this kind of programs is also a torment [1]. The concept of Transactional Memory (TM) [2] was introduced in this scenario to offer a neat programming interface while guaranteeing the correctness of multi-threaded programs. It has shown great potential to simplify software developing and debugging.

In the last decades, Software Transactional Memory (STM) [3] has demonstrated the feasibility of this way but its efficiency and performance are not comparable to that of fine-grained locking, even after a lot of improvements [4]. The performance problem has been hindering the wide adoption of TM for a long time. Recent hardware support from Intel [5] and IBM drew new lights on this. However, it is not an easy task to achieve high efficiency with current hardware support because of the limitations of hardware [6] (as current HTM is based on cache coherence protocol, it would be affected by the small cache capacity, interrupt, context switch, etc.). Hardware Transactional Memory (HTM) requires a fall-back strategy to tackle the limitations and to avoid constant abortion of hardware transactions. As we will show in this paper, different fall-back strategies would hugely affect the performance and, most importantly, there is no one strategy that can fit all the situations. Thus dynamic adjustment is essential.

This paper presents an efficient way to dynamically adjust the fall-back strategy to achieve optimal performance for HTM. Our dynamic adjustment mechanism is based on an important observation: atomic blocks in the same execution context (the same function or the same loop) fit the same call-back strategy. Based on this observation, we can dynamically profile the execution and choose the optimal strategy for each atomic block and thus achieve better performance. Experiments show our method could achieve an average speedup of 1.4X over static method.

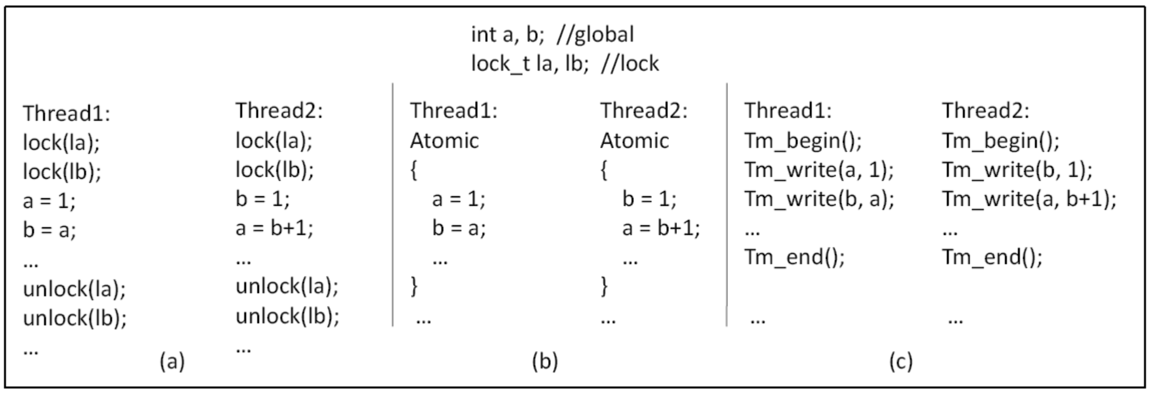
The rest of the paper is organized as follows. We give a brief introduction to TM programming and the problem in HTM in Section 2. Section 3 gives the design and implementation of our dynamic adjustment method. We show experimental results in Section 4 and conclude in Section 5.

2. **Background and Motivation.**

In this section we firstly give a brief introduction of TM and then we discuss the key problem of current HTM.

2.1. **Programming with Transactional Memory interface.**

Figure 1 shows the sample code of programming with locking and TM interface. In this scenario there are two shared variables *a* and *b*. In Figure 1(a), the two variables are protected by the lock *la* and *lb*, respectively. If one thread wants to update the two variables at the same time, it should hold the two corresponding locks. As shown in Figure 1(a), programming with locking in this scenario is complex and may introduce bugs such as deadlock. Alternatively, Figure 1(b) shows the code with Transactional Memory interface. To express the same semantic, programmers only have to put the code into an atomic block. The compiler will then instrument all the memory accesses in the atomic blocks as shown in Figure 1(c) [7]. Then the TM runtime is able to execute each atomic block (could be also referred to as transaction) in isolation (all the updates to shared variables will be buffered as thread-local and submitted to the global version at commit time). If one thread detects conflict (e.g. two threads modify the same shared variable) before submitting the updates to the global version, it has to abort the current transaction and restart executing the atomic block again. In all, the TM runtime will guarantee the execution seems like all the atomic blocks are executed in certain serial order.



```
                              int a, b;  //global
                              lock_t la, lb;  //lock

Thread1:        Thread2:       | Thread1:     Thread2:    | Thread1:           Thread2:
lock(la);       lock(la);      | Atomic       Atomic      | Tm_begin();        Tm_begin();
lock(lb);       lock(lb);      | {            {           | Tm_write(a, 1);    Tm_write(b, 1);
a = 1;          b = 1;         |   a = 1;       b = 1;    | Tm_write(b, a);    Tm_write(a, b+1);
b = a;          a = b+1;       |   b = a;       a = b+1;  | ...                ...
...             ...            |   ...          ...       | Tm_end();          Tm_end();
unlock(la);     unlock(la);    | }            }           |
unlock(lb);     unlock(lb);    | ...          ...         | ...                ...
...        (a)  ...                      (b)                          (c)
```

FIGURE 1. Comparison between locking and TM

2.2. **Hardware Transactional Memory.**

A key process in transactional processing is conflict detection, which is very time-consuming in Software Transactional Memory (STM) [3,4]. Current hardware support for this is based on cache coherence protocol: every core has its private cache. If other cores modify the shared data that has already been cached by a core, the certain private cache line will be invalidated. From this event we can detect the conflict among threads on accessing shared variables and this introduces ignorable overhead. However, the hardware does not guarantee that a hardware transaction will always successfully commit even when there are no conflicts. For example, a cache line will be evicted due to space pressure, leading to the abortion of the corresponding hardware transactions. Thus a fall-back strategy for hardware transaction is essential to avoid constant abortion of transactions. Figure 2 shows the pseudo-code of using hardware transaction.

The main procedure of using hardware transaction (as shown in Figure 2) is that we try several times of hardware transactions before falling back to serial execution. There is a trade-off in this situation: on one hand, the hardware transaction is fast and could leverage the multi-core to gain performance benefit but the hardware cannot guarantee the successful commit of the transactions, even in the absence of conflict. On the other hand, falling back to serial execution would lose the benefit of concurrency but it guarantees the proceeding of transactions. Thus a key problem here is to determine when to fall-back to serial execution (to determine the line 1 and line 6 in the sample code in Figure 2). For example, if the abortion of transaction is due to conflict, we can try it for more times because it has chances to succeed. However, if the abortion is due to the cache capacity (the transaction accesses data that could not be fit into a single cache line), we will know that this transaction would never successfully commit and thus a better way is to fall-back to serial execution immediately.

Figure 3 shows examples that different strategies would hugely affect the overall performance in different applications. For example, for the *bayes* in 8 threads, the $attempts = 2$ is the best setting and it outperforms $attempts = 1$ by more than 2X. While for *kmeans*, the $attempts = 8$ is the best setting. The benchmarks *bayes* and *kmeans* are from the benchmark suit STAMP [8]. We can see that there is no fit strategy which can always achieve the best performance. This reveals the needs for dynamic adjustment.

```
1: Int attempt = n;  //try n times of hardware transaction
2: Retry:
3: Endcode = Try_transactions();  //this tries to run atomic block using hardware transaction
4:                                 //(XBEGIN and XEND).
5: If(Endcode == failed){
6:          attempt = attempt -1;  //this is the key of fall-back strategy.
7:          if(attempt > 0){
8:                   goto Retry;
9:          }else{
10:                  //we have to fall back to serial execution
11:                  //get a global lock, run atomic block, and release the global lock
12:         }
13: }
```

FIGURE 2. Sample code of using hardware transaction

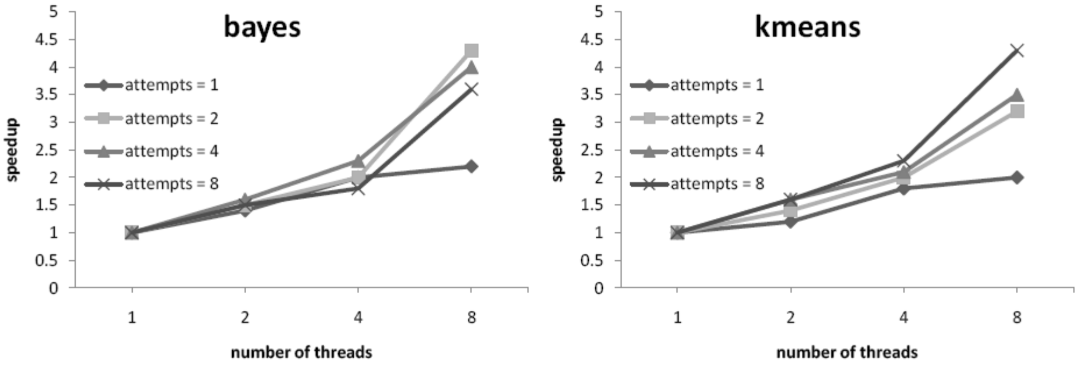


FIGURE 3. Performance under different strategies

3. **Efficient Self-tuning.** As discussed in Section 2, our goal is to dynamically adjust the fall-back strategy for each atomic block in an application. Specifically, we have to determine the initial number of attempts and the decrement (line 1 and line 6 in Figure 2) to get optimal performance. Our method is based on two observations or principles and we discuss them in the next two subsections.

3.1. **The size of atomic blocks.** If one atomic block is small (contains small number of instructions), it has more chances to commit because it is less likely to conflict with other threads or to be interrupted by other events such as cache line eviction. Thus for this kind of atomic block we can set the number of attempts to be larger to try it for more times.

We can detect the size of an atomic block during compiling time. To achieve this we leverage the well-known compiler framework LLVM [9]. LLVM will firstly translate the applications' source code into intermediate code and then translate the intermediate code into the final binary code. The LLVM framework offers convenient interfaces for us to perform analysis on the intermediate code level. Thus we can just count the instructions in each atomic block and then insert instructions to set the initial attempts (line 1 in Figure 2) at the beginning of each atomic block. In our current implementation, we set the initial attempts as: *attempts = 20 – number_of_instructions/100.* The maximum is set to be 20 and the minimum is set to be 1. The larger the atomic block is, the smaller the attempt is set to be. Moreover, for simplicity, we just set the initial number of attempts fixed at compiling time and do not adjust it later at runtime. At runtime we can adjust the decrement (line 6 in Figure 2) to achieve further dynamic adjustment.

3.2. **Atomic blocks in the same execution context.** Atomic blocks in the same execution context (the same function or the same loop) are likely to fit the same strategy. This is because the codes in the same execution context are likely to have the same memory access pattern, which has been proven in previous work [10]. Moreover, programs tend to implement similar features in the same execution context. Thus we can determine the strategy for the latter atomic block according to the execution of the previous atomic block. Specifically, if we find the previous atomic block has aborted for many times and finally fell back to serial execution, we will change the line 6 in Figure 2 from *attempts–* to *attemptes/=2* to accelerate this process for the latter atomic block.

An atomic block is not likely to be just executed once. Thus we will also keep a history of execution record for each atomic block. Thus the strategy of each atomic block is not only affected by the previous atomic block in the same execution context, it is also affected by the previous execution history of its own.

To determine whether two atomic blocks are in the same function, we rely on *backtrace()* in the *libc* library. To determine whether two atomic blocks are in the same loop, we rely on the LLVM [9] compiler framework to identify loops at compiling time. We insert a special function call at the beginning of each loop in which we give each loop an ID at runtime. Identifying loops is a mature tool provided in LLVM framework.

4. **Experimental Results.** This paper introduces efficient method to dynamically tuning the fall-back strategy for hardware transactional memory. In the experiment we mainly test and compare our method with default static method. We also compare our work with a previous relevant tool TUNER [6] which aims to achieve the same goal. TUNER adjusts the strategy only based on the isolated history of each atomic block. It does not discover the relationships among atomic blocks in the same execution context. As we will show in the experiments that our work performs better than TUNER in most cases and our work could be combined with TUNER to achieve better performance.

Our experimental platform is an Intel server with HTM support running Linux 3.11 with 16 GB of physical memory. Our benchmarks are from the transactional memory

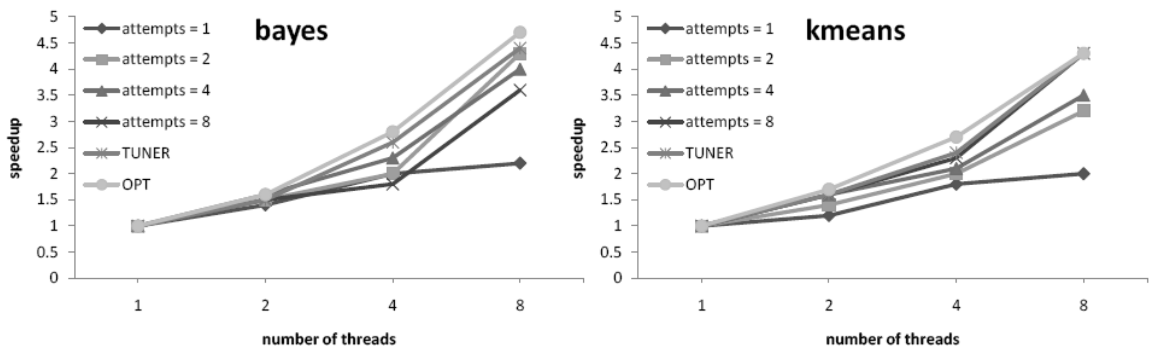


FIGURE 4. Comparison of different strategies

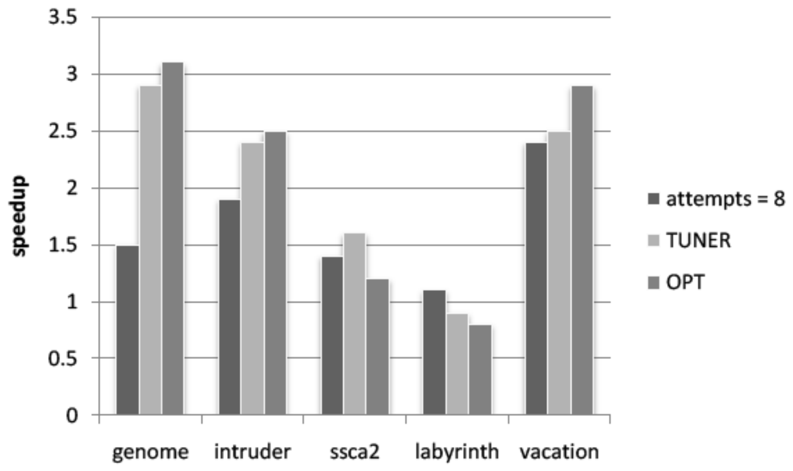


FIGURE 5. Comparison of different strategies with 4 threads

benchmark suit STAMP [8]. The inputs are default native running (we always choose high contention if not specially mentioned).

Figure 4 shows the results in the benchmarks bayes and kmeans. We can see for all the situations our method (OPT) behaves better than TUNER and other static strategies. The largest gap lies on the 4-threads situation because 4 threads running exhibits intermediate contention that calls for fine-grained adjustment. Figure 5 shows other benchmarks with 4 threads running. Averagely, our method could achieve 1.4X speedup over static method and 1.05X speedup over TUNER.

5. Conclusions.

This paper introduces an efficient dynamic fall-back strategy for hardware Transactional Memory. By exploiting the relationships of atomic blocks that are in the same execution context, we can effectively profile the execution and determine the optimal fall-back strategy. Experimental results show that our dynamic method could achieve an average speedup over static method of 1.4X and 1.05X over the previous state-of-the-art dynamic method. Our future work mainly lies on integrating software transactions into the fall-back process of hardware transactions.

Acknowledgment. This work is supported by the Weifang Science and Technology Development Project No. 2014GX031.

REFERENCES

[1] K.Asim, M. J. Renzelmann and M. M. Swift, Fine-grained fault tolerance using device checkpoints, *ACM SIGARCH Computer Architecture News*, vol.41. no.1, 2013.

[2] D. Diego et al., Identifying the optimal level of parallelism in transactional memory applications, *Computing*, vol.97, no.9, pp.939-959, 2015.

[3] W. Jons-Tobias et al., FastLane: Improving performance of software transactional memory for low thread counts, *ACM SIGPLAN Notices*, vol.48, no.8, 2013.

[4] C. Calin et al., Software transactional memory: Why is it only a research toy? *Queue*, vol.6, no.5, pp.40-46, 2008.

[5] M. Y. Richard et al., Performance evaluation of Intel® transactional synchronization extensions for high-performance computing, *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[6] D. Nuno and P. Romano, Self-tuning intel transactional synchronization extensions, *The 11th International Conference on Autonomic Computing*, 2014.

[7] S. Martin et al., Towards transactional memory support for GCC, *The 1st GCC Research Opportunities Workshop*, 2009.

[8] M. C. Cao et al., STAMP: Stanford transactional applications for multi-processing, *IEEE International Symposium on Workload Characterization*, 2008.

[9] L. Chris and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, *International Symposium on Code Generation and Optimization*, 2004.

[10] R. Guo et al., NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems, *USENIX Annual Technical Conference*, 2015.