ICIC Express Letters
Part B: Applications
Volume **7**, Number **11**, November **2016**

ICIC International ©2016 ISSN 2185-2766
pp. **2405–2410**

AUTOMATIC AND EFFICIENT FALSE SHARING AVOIDER FOR MULTI-THREADED PROGRAMS

Dongying Zheng

Innovation and Entrepreneurship College
Weifang University of Science and Technology
No. 1299, Gold Street, Shouguang 262700, P. R. China
Zdy7817@163.com

Received May 2016; accepted August 2016

Abstract. *False sharing is a representative performance bug in multi-threaded programs that causes severe slowdown. It implicitly impacts performance and it is very hard to detect. Unlike previous methods which are focused on detecting it and helping programmers to fix it manually, this paper proposes a runtime system that can eliminate false sharing transparently at runtime. The key idea of this paper is to give each thread a private copy of shared data and make them just operate on their private data as long as possible. Thus threads will not conflict on accessing shared data. Experimental results show that our method can achieve speedup of 1-600%, showing great potential to tackle the performance problem introduced by false sharing.*
Keywords: False sharing, Multi-threaded program, Eliminate, Private copy

1. **Introduction.** Multi-threaded program is becoming increasingly prevalent to fully leverage the hardware resources in today's fast-developing multi-core architectures. However, it is a challenge to develop multi-threaded programs. Not only is the program error-prone and hard to debug [1], but also it is very difficult to scale [1] on many cores. A lot of studies [2] have been proposed to try to eliminate bugs and thus enhance the reliability of multi-threaded programs with bug-detection methods. However, there is another kind of bug which attracts less attention. This kind of bug will not cause program to crash but will hugely hurt performance and thus we call this kind of bug the *performance bug* [3].

False sharing [1] is a representative *performance bug* in multi-threaded programs. It will cause severe contention among threads and thus limit the scalability and performance. False sharing happens in the situation that, for example, if two threads A and B have their data a and b settled in the same cache line, then thread A's accesses to data a will possibly invalidate the cache line in thread B's (the core thread B is running on) private cache and cause performance loss to thread B on further accessing data b. Well, thread B's further accesses on data b will in turn invalidate the cache line in thread A's private cache and cause performance loss to thread A on accessing data a. In this situation, the two threads are accessing different data but they cause severe contention in cache, leading to unnecessary performance loss. It has been demonstrated in previous work [1] that in some severe situations, the false sharing problem may cause multi-thread programs even slower than its single-threaded counterparts.

Previous work [4,9] to tackle false sharing mainly focuses on detecting false sharing at runtime and producing reports to help programmers fix the bug manually, which involves considerable human effort. Unlike previous work [9], we propose a total automatic runtime system that avoids or eliminates false sharing directly at runtime. Any multi-threaded program can run on our system without any modification to gain the performance benefit of running with less false sharing. Our key idea is based on the software transactional

memory (STM) system that we give each thread a private copy of global shared data and commit the updates to the global data at certain points. Threads do not conflict with each other when operating on their private copy and thus we avoid false sharing totally. Based on the initial idea, two problems remain to be settled: (1) when to commit the update of private copy to global shared data and (2) how to reduce the overhead of making private copy and redirect every access to the private copy. We will deal with the two problems in Section 3.

Some dynamic memory allocators [5] for multi-threaded programs tried to achieve the same goal with us. They usually give every thread a private heap to make dynamic memory allocation and thus the shared data allocated by different threads do not settle in the same cache line. However, there may be some data that are allocated by a thread and used by other threads and this situation may still cause false sharing. Moreover, we argue that the memory allocator work is compatible with our work and can be combined with our work to achieve better result.

The rest of this paper is organized as follows. Section 2 introduces the problem of false sharing. We give our dynamic false sharing avoider in Section 3. We show experimental results in Section 4 and we make conclusion in Section 5.

2. Performance Slowdown Caused by False Sharing.

Figure 1(a) shows an example of code that introduces false sharing. The global variables a and b are shared by threads and they are in the same cache line. Then the two threads perform a serial of calculations on the two variables respectively. Thread 1 only operates on a and thread 2 only operates on b. However, this will introduce great slowdown due to the severe invalidation of the cache lines in the cores for thread 1 and thread 2.

Figure 1(b) shows a simple fix of this problem where we introduce an array to separate a and b to make them located in different cache lines. The accesses to a and b from thread 1 and thread 2 will then never conflict with each other in cache. Through this simple fix, we can achieve an obvious speedup of nearly 3X (FIX-manually shown in Figure 2), which well demonstrates that false sharing is a serious performance problem in multi-threaded programs. Furthermore, we extend this example to 4 or 8 threads and Figure 2 shows the performance slowdown caused by false sharing.

```
int a;      //shared by all threads, a and b in the
int b;      //same cache line

int main(){
   Int loop_count = 10000000000;
   for_thread1:{  //thread 1 does this
      for(i=0; i<=loop_count; i++){
         a++;
      }
   }
   for_thread2:{  // thread2 does this
      for(i=0; i<=loop_count; i++){
         b++;
      }
   }
}
```

(a)

FIX

```
int a;
int space[100]  //separate a and b, make them
int b;              //in different cache line

int main(){
   Int loop_count = 10000000000;
   for_thread1:{  //thread 1 does this
      for(i=0; i<=loop_count; i++){
         a++;
      }
   }
   for_thread2:{  // thread2 does this
      for(i=0; i<=loop_count; i++){
         b++;
      }
   }
}
```

(b)

FIGURE 1. Sample of false sharing and fixed version

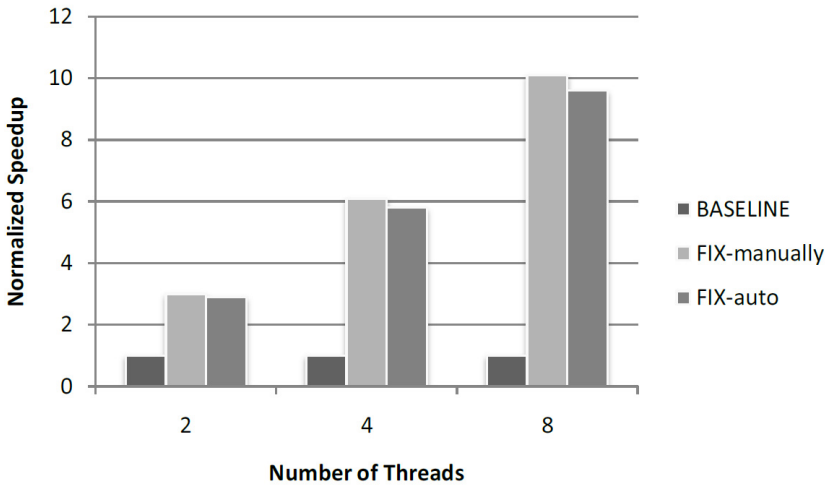

FIGURE 2. Performance comparison of false sharing and fixed versions (baseline is the normal execution. Fix-manually is to fix the false sharing problem manually all by programmers. Fix-auto is the proposed method in this paper.)

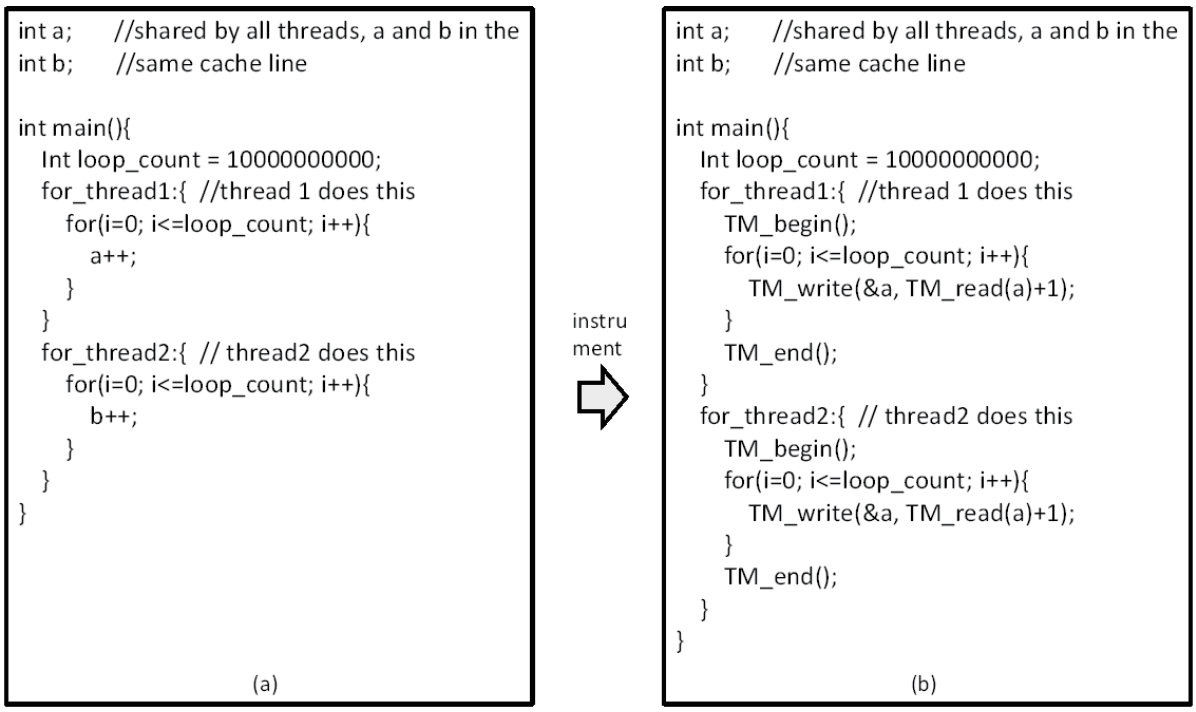

FIGURE 3. Instrumentation

3. **False Sharing Avoider.** We base our work on the idea of giving each thread a private copy of shared data and let them operate on their private copy with minimum inter-thread communications. Thus threads get less chance to conflict with each other. First, we rely on compiler instrumentation [6] to instrument every access at compile time. Figure 3 shows our method. Figure 3(a) is the original code with false sharing while Figure 3(b) is the optimized code. The optimized code is automatically generated by our compiler framework (we leverage the open source compiler framework LLVM [6,13] to insert instructions automatically into the code, see details later. Also, there are many compiler tools such as the Intel C++ STM [10,11] that supports automatic software transactional memory that can be used to achieve this).

In the optimized code shown in Figure 3, we mainly inserted some function calls which are the same as traditional STM (software transactional memory) interface [7]. Then our system executes as follows (which is pretty much like traditional STM system [7]). When thread 1 meets TM_begin(), it will initialize a private buffer used as private copy space. Then the TM_read() will create a private copy of the variable a in the private buffer and serve all the subsequent accesses (TM_read() and TM_write()) for thread 1. Then at TM_end() time the thread 1 will commit the private copy a to the global shared variable a. Thread 2 performs the same way. In this way we actually separate the variables a and b by introducing private copies for them. False sharing will only happen once at the commit time. The performance is shown in Figure 2 as FIX-auto. The overhead of our mechanism is the memory access redirection on every TM_read() or TM_write() but for this we can automatically eliminate false sharing.

The above example shows the basic idea of our method. However, two problems remain: (1) where to insert TM_begin() and especially TM_end(), which determines when to update the global shared data; (2) instrumenting (redirecting) all memory accesses [6] will introduce huge overhead, which may shadow the benefit we gain from eliminating false sharing.

In order to solve the two problems, we propose to focus our method on smaller scope: the loops, instead of the whole program. The main reason for this is that intense memory accesses usually happen in loops [12], which means intense false sharing will only happen in loops. Thus in order to get best performance, we just need to focus on loops. Moreover, instrumenting only memory accesses in loops will reduce a lot of overhead.

Figure 4 shows our method. Firstly, we insert TM_begin() and TM_end() for each loop. We leverage the LLVM tool [6] for loop identification at compile time. Thus basically, each loop forms a transaction for each thread. Moreover, thread may perform synchronization operations (hold lock or release lock) to access certain shared data. At synchronization time, a thread may want to get the latest value of a shared data which is being updated by another thread. Thus, in order to guarantee this correctness, we insert TM_end() at every synchronization point to make thread commit the latest value to the global shared data. By doing this we can guarantee the correctness of the program and this has been discussed in a lot of previous studies [8].

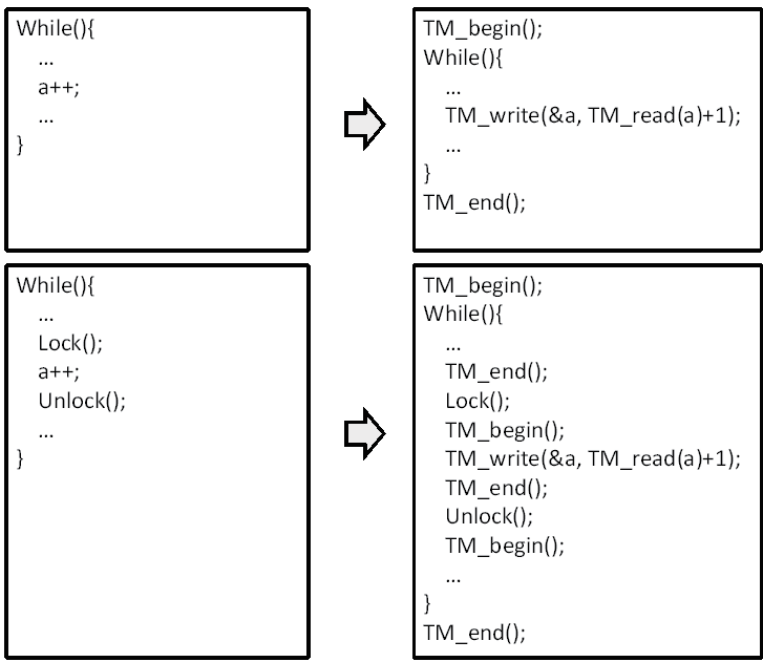

FIGURE 4. Loop-based and lock-based update

As shown in Figure 4, if a loop contains synchronization operations (e.g., lock), we have to insert additional TM_begin() and TM_end() and force the thread to commit its latest value. This may introduce huge overhead. We have made an optimization on this that we analyze each loop at compile time to see if it contains lock operation or not. If a loop contains lock operation, we prefer ignoring the loop for our false sharing avoiding. This may leave some false sharing in the loop un-eliminated but we could achieve better performance on avoiding tacking synchronization operations (e.g., lock).

4. **Experimental Results.** This paper proposes a system that could eliminate the false sharing bugs in multi-threaded programs transparently at runtime and thus achieve better performance. In experiments we mainly test and show its ability to achieve speedup compared with the normal execution. Moreover, we also show that our work can be combined with previous memory-allocator-based method (here we choose to use Hoard [5] which is a memory allocator specialized for multi-threaded programs) to achieve further improvement.

The benchmarks we choose are from the benchmark suit Parsec and Phoenix which have been reported to have false sharing problems (see Table 1). We also choose some benchmarks that contain no false sharing problems to show the pure overhead that we will introduce. Our experimental platform is a 4-core Intel server running Linux 3.11 with 16GB of physical memory. The LLVM compiler framework we used is of version 2.9. All benchmarks are executed with 4 threads with default input. We execute each benchmark for 5 times and report the mean value for each test.

Table 1. Benchmarks

Benchmark	Previously reported to have false sharing [1,9]
kmeans	*N*
dedup	*N*
swaption	*N*
linear_regression	Y
reverse index	Y
streamcluster	Y
word count	Y

The experimental results are shown in Figure 5. First for the benchmark *kmeans*, *dedup*, and *swaption* which have no false sharing problem, our method (FIX-auto) introduces pure overhead of 5-10%, which is due to its redirecting accesses in loops. For the benchmark *linear_regression*, our method achieves a huge speedup of more than 6X, which means the benchmark is greatly bothered by false sharing problem. For other benchmarks, our method can gain 2-9% performance benefit. Moreover, when our method is combined with previous memory-allocator-based method (here we use Hoard [5]), we can achieve further improvement for all benchmarks. The above results show the great potential of our work to tackle false sharing problem in multi-threaded programs.

5. **Conclusion.** This paper introduces a runtime system that can eliminate false sharing in multi-threaded programs to achieve better performance. Unlike previous studies which are focused on detecting false sharing, our runtime system eliminates it automatically and transparently at runtime. The key idea is to give each thread a private copy of shared data so that threads could operate on their private copies without any contention on cache lines. Experimental results show that our method could gain performance benefit of 1-600%, showing great potential to speedup multi-threaded programs. Future work mainly includes reducing the overhead of instrumenting and redirecting memory accesses.

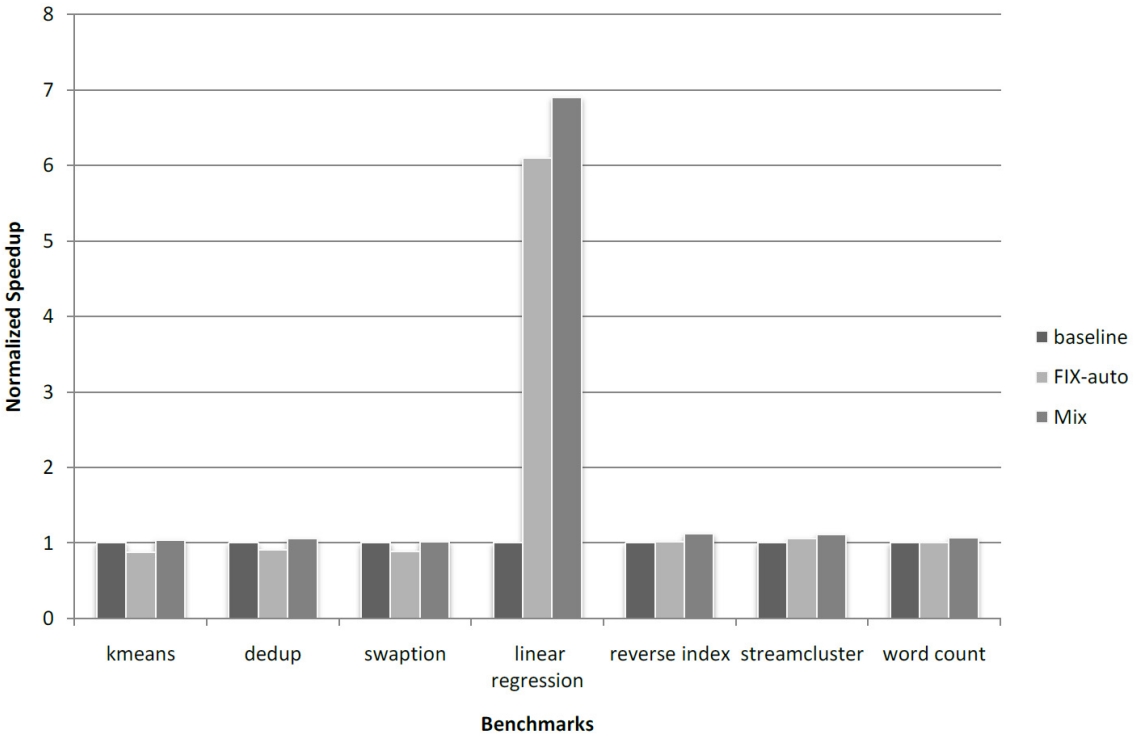

FIGURE 5. Experimental results

REFERENCES

[1] T. Liu and E. D. Berger, SHERIFF: Precise detection and automatic mitigation of false sharing, *ACM SIGPLAN Notices*, vol.46, no.10, pp.3-18, 2011.

[2] Q. Gao and M. Xu, *Detecting Resource Deadlocks in Multi-Threaded Programs by Controlling Scheduling in Replay*, United States Patent 9052967, 2015.

[3] O. Olivo, I. Dillig and C. Lin, Static detection of asymptotic performance bugs in collection traversals, *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[4] M. A. Spear, *Whose Cache Line Is It Anyway: Automated Detection of False Sharing*, 2015.

[5] E. D. Berger et al., Hoard: A scalable memory allocator for multithreaded applications, *ACM SIGPLAN Notices*, vol.34, no.5, pp.117-128, 2000.

[6] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, *IEEE International Symposium on Code Generation and Optimization*, 2004.

[7] A. Ghosh and R. Chaki, Implementing software transactional memory using STM haskell, *Advanced Computing and Systems for Security*, Springer India, pp.235-248, 2016.

[8] T. Liu, C. Curtsinger and E. D. Berger, Dthreads: Efficient deterministic multithreading, *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[9] L. Lyu, FS-D: A high efficient false sharing detector for multi-threaded programs, *ICIC Express Letters*, vol.9, no.8, pp.2205-2210, 2015.

[10] A. Dragojevic, N. Yang and A.-R. Adl-Tabatabai, Optimizing transactions for captured memory, *Proc. of the 21st ACM Annual Symposium on Parallelism in Algorithms and Architectures*, 2009.

[11] V. Ying et al., Dynamic binary translation and optimization of legacy library code in an STM compilation environment, *Proc. of the Workshop on Binary Instrumentation and Applications*, 2006.

[12] E. P. Markatos and T. J. LeBlanc, Using processor affinity in loop scheduling on shared-memory multiprocessors, *IEEE Trans. Parallel and Distributed Systems*, vol.5, no.4, pp.379-400, 1994.

[13] S. Falke, F. Merz and C. Sinz, LLBMC: Improved bounded model checking of c programs using LLVM, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2013.