Reducing Lock Contention on Multi-core Platforms

Haimiao Ding, Xiaofei Liao, Hai Jin, Xinqiao Lv, Rentong Guo
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: xfliao, hjin@hust.edu.cn

***Abstract*—As multi-core platforms with hundreds or even more quantities of cores are popular, system optimization issues, including lock contentions, start to puzzle programmers who work on multi-core platforms. Locks are more convenient and clear than lock-free operations (for example, transactional memory) for multi-core programmers. However, lock contention has been recognized as a typical impediment to the performance of shared-memory parallel programs. This paper mainly discusses two important reasons that cause lock contention, including large critical sections and frequent lock requests. For current solutions, it is hard for programmers to find the locations of large critical sections and good scheme to reduce lock contentions on hot critical sections. This paper proposes FFlocker, a series of runtime solutions that reduce lock contention caused by the two issues. FFlocker includes a profiling algorithm to find the locations of large critical sections. Based on the profiling scheme, it binds the threads acquiring the same locks onto the same core. We evaluate our techniques with three benchmarks. The results show that FFlocker offers better performance than Function Flow and OpenMP.**

Keywords*—Multi-core; parallel programming; runtime; lock contention

I. Introduction

Parallel programming [1] on multi-core platforms [2] leads many new problems that have not been considered on platforms with single core. Lock contentions issues [3], [4] are related with two kinds of parallel programming patterns, thread-oriented pattern or task-oriented pattern. Programmers used to develop parallel programs with thread libraries such as Pthreads [5] and Windows threads [6]. This pattern of parallel programming will fork new threads to execute one parallel task, so it is called thread-oriented pattern of parallel programming. Another pattern creates working threads at the beginning of parallel programming instead of creating threads when we need new parallel tasks. This pattern is task-oriented pattern of parallel programming [7].

Whenever we use thread-oriented pattern or task-oriented pattern, lock contention may always badly hurt system performance. In fact, it is impossible to reduce all the lock contentions, because there are varies of situations that will cause lock contentions. This paper mainly addresses two specific issues of lock contention. One is the lock contention caused by large critical sections, another one is the contention caused by frequent lock request.

To reduce the lock contention caused by large critical sections, Agrawal [8] proposes helper locks to solve this problem. A helper lock is a mechanism that protects large critical sections, There is a helper lock L, and it protects a large critical section called P. Thread T1 firstly gets L, then it starts to execute the region P. Other threads that request L will help T1 to execute the parallel subtasks in P. It is a good solution that reduces the lock contention caused by large critical sections. However, it does not work well when region P is strictly critical. That is, if there are no parallel subtasks, helper lock will become normal lock. In this paper, we propose a better solution, called task-level lock (also called TLock), and based on helper lock. The difference between TLock and helper lock is that TLock will steal the parallel tasks of other threads instead of only helping executing the parallel subtasks created by the large critical sections. One issue we must consider when facing large critical sections is that, as the growth of parallel programs size, it is hard for programmer to find the locations of large critical sections. Addresses this issue, we propose an algorithm to locate the large critical sections by profiling.

The issue of frequent lock request always happens in thread-oriented patterns of parallel programming. That is, different threads may request the same lock frequently. If these threads are located on different cores, these cores will switch between user state and system state frequently. In this situation, most CPUs resources are spent on the context switching. For this issue, we propose a runtime solution that binds the threads that request the same locks onto the same cores. If the threads, which request the same locks, are bound to the same cores, there is only one active thread, which requests the lock at one time. For this reason, CPUs reduce much computing resources on context switching.

As mentioned above, the main contributions of this paper are as follows:

- We evaluate the influences of lock contention in two situations. One is the problem of large critical sections. Another one is frequent lock request.
- We propose an algorithm based on profiling to detect the locations of large critical sections.
- We propose TLock based on helper lock, and based on TLock, we also present a runtime solution to reduce the lock contention caused by the problem of frequent lock request.

The rest of the paper is organized by follows. The next section describes the motivation. Section III presents the design and architecture of the two solutions that reduce lock contention in two typical situations, focusing on detecting the large critical sections and the runtime system of reducing lock contention caused by frequent lock request. Besides this, the improvement of helper locks is also mentioned. Section IV presents experimental results. Section V discusses related work. Finally, we present some concluding remarks.

978-1-4799-7615-7/14/$31.00 ©2014 IEEE

Table I: CPU usage on a sample of frequent lock request

CPU	User mode	System mode
Cpu0	44.2%	55.8%
Cpu1	36.0%	64.0%
Cpu2	36.0%	64.0%
Cpu3	52.9%	47.1%
Cpu4	51.9%	48.1%
Cpu5	33.3%	66.7%
Cpu6	32.7%	67.3%
Cpu7	58.8%	41.2%

```
1.  const int t = 100000;
2.  const int u = 20;
3.  const int bound = 100;
4.  void * fn(void * arg){
5.      for (int i = 0 ; i < t; ++i){
6.          if (i % bound < u){
7.              pthread_mutex_lock(&locks[(int*)arg]);
8.          insert_new_element(queue[(int*)arg]);
9.              pthread_mutex_unlock(&locks[(int*)arg]);
10.     }
```

Figure 1: A simple simulation function simulate the situation that frequent lock request happens

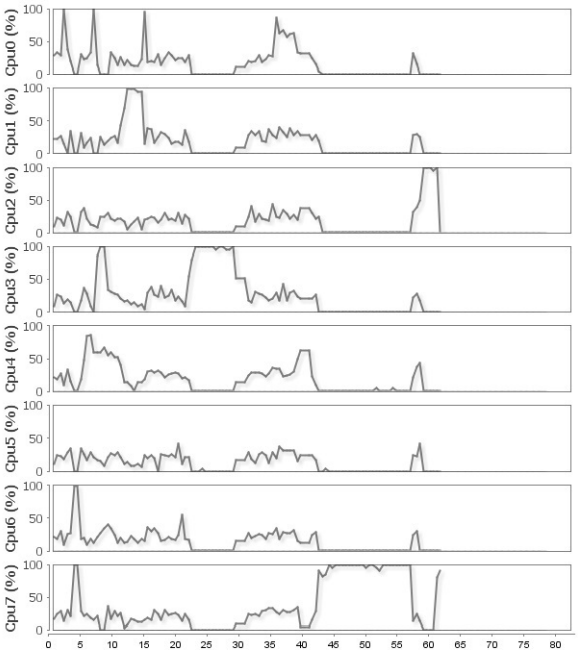

Figure 2: An experiment of parallel hash table insertions on a multi-core platform with 8 cores. There are 8 worker threads (OpenMP worker) inserting elements into a parallel hash table.

II. Motivation

This section presents motivation of this paper. There are two motivating examples about lock contentions that badly hurt the performance of parallel programs. One is about frequent lock requesting, another one is about large critical section. To make it easy to read, we choose two typical parallel programming environments to express the two motivating examples. The first one is based on Pthreads library and the second one is based on OpenMP [9], [10] (others model, including TBB [11], Cilk [12] and Function Flow [13]). Pthreads library is a typical thread-oriented pattern of parallel programming. OpenMP is a typical task-oriented parallel programming environment. It will not create new threads to execute new parallel tasks. Instead of creating new threads, OpenMP creates working threads called worker at the beginning of the parallel programs. The runtime system divides working threads into several groups. Working threads in the same group share parallel tasks. This solution of dispatching is called work-sharing scheme [10].

Frequent lock requesting happens when a number of threads request locks concurrently and frequently. So, different threads request a set of lock frequently. These threads are located on different cores, and CPUs should wake some of the threads. At the same time, the other threads will sleep. As these situations occur, most CPU computing resources are spent on the context switching. Frequent lock requesting often happens on Internet applications. For example, a parallel program on a multi-core platform is serving many clients at the same time. For good corresponding delay, the program creates a thread to connect a client. Some of the clients frequently request the same table in a database. The same table is always protected by one lock. This is a typical situation that frequent lock requesting happens.

To make it clear, we develop a simulation program to simulate frequent lock requesting. As shown in Figure 1, f_n is a function that requests a specific lock with 20% probability.

More specifically, we create 64 threads with function f_n to request 8 locks on a multi-core platform with 8 cores. Table I is the result of CPU usage at a time when frequent lock requesting happens. As shown in Table I, about half of CPUs resources are wasted on context switching or any other system calls. If threads on a same core request their locks successfully at some time while threads on another core fails to get their locks, the resources competitions are intense. Then the CPU cores cannot keep load balance. Operating system may move the threads from heavy cores to light cores. This process may hurt the performance.

In this paper, we propose a solution to reduce the lock contentions, which are caused by frequent lock requesting. A number of threads on different cores will hurt performance when requesting the same locks frequently. However, if we bind the threads, which request the same locks or the same set of locks, onto same cores, the conflict among the threads will extremely reduce, because there is only one active thread request per lock. So, this paper proposes a runtime solution to bind the threads that frequently request the same locks onto the same cores. This solution is simple but effective.

Large critical section is another typical problem, which leads lock contention. If the section protected by lock is very large, any other working threads should wait until the large critical sections finish execution. Agrawal [8] develops a parallel hash table as a benchmark to test the influences of large critical section. They then propose helper locks to reduce the lock contention caused by large critical section. Helper lock is a synchronization mechanism that protects the large critical sections. If working threads fails to acquire a helper lock, it helps to execute parallel subtasks of the critical sections.

Helper lock is a good solution to solve the problem of large

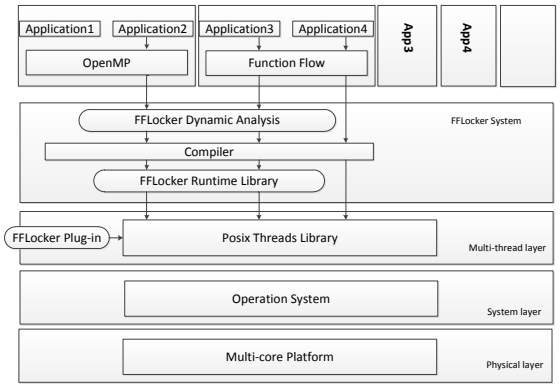

Figure 3: The structure of FFLocker consists of two main parts. The first is to reduce the lock contention caused by large critical section, including dynamic analysis module and runtime library. The second is to reduce the lock contention caused by frequent lock request.

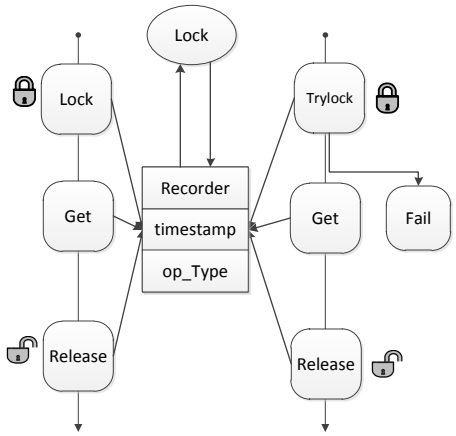

Figure 4: The structure of recorder and the possible execution path of how a working thread acquires lock.

critical section. If there is no parallel subtask created in large critical section, the helper locks will become normal locks.

We extend helper lock in this paper to make the solution works well in most situations. That is called a task-level lock (TLock). A TLock is similar to a helper lock when the large critical sections have parallel subtasks. However, if the large critical sections do not have parallel subtasks or the subtasks finish execution, TLock guides the working threads to execute parallel tasks of other working threads. Another important difference between TLock and helper lock is that runtime system manages the global TLocks. Unified management brings a number of benefits. For example, runtime system can set the priority of the task queues assigned to TLocks.

Another problem of reducing lock contention caused by large critical sections is to detect the locations of large critical sections. As the scale of parallel program grows, it becomes a big challenge for programmers to locate the large critical sections.

We implement the benchmark of parallel hash insertions and test it on a multi-core platform. Figure 2 shows the results of the benchmarks. It is not difficult to locate two large critical sections at least in Figure 2, because the usage of CPUs can clearly show that the working threads are active or not. This paper proposes a profiling solution with CPU usage and instrumentations. The key idea is that the locks keep waiting while most CPUs keep idle, which probably is protecting large critical sections.

III. Design and implementation

FFlocker proposed in this paper is a series of runtime solutions that reduce lock contention caused by large critical sections or frequent lock request.

We provide a series of algorithms to reduce the lock contention, including how to detect large critical sections by profiling, a scheduler based on TLock, global management of TLocks. All these measurements mentioned above are organized to reduce lock contention caused by large critical sections with reasonable completion time bounds and space usage bounds. To reduce the lock contention caused by frequent lock request, FFlocker can bind the threads onto same core.

This section shows the main parts of runtime system within FFLocker. Several key technical points are also described below.

A. System overview

FFLocker system has two main parts. One is to reduce the lock contention caused by large critical sections. The other is to reduce lock contention caused by frequent lock request. For the first goal, FFLocker contains two main steps. As shown in Figure 3, the first step is to detect large critical sections and the second is to reduce lock contention with runtime system. The process of detection is off-line profiling, it gives programmer useful guide on where large critical sections locations are. After detecting the locations of large critical sections, programmer can choose TLock to protect the sections. It is available but not required. Whether to use TLock or not mainly depends on programmer's choice. For the second goal, FFLocker provides a plug-in for Pthreads library to reduce lock contention.

B. Detecting the locations of large critical sections

We use off-line profiling to detect the locations of the large critical sections. The input is the source code while the output is the locations of large critical sections.

Before off-line profiling, we must insert instrumentations into the specific location of source code to record the operations of every lock. As shown in Figure 4, a recorder is a structure to store a series of operations with a specific lock. Every recorder is a proxy for a specific lock. The recorder has main information, includes the timestamp when every operation of the lock happens and the type of operations. The operations are stored in a resizable array. We use vector to store the time stamp and operations.

As shown in Figure 4, locks may face two kinds of requests. The first is lock while the second is try-lock. If the request is to acquire a lock, there are three states on execution path, including acquire, get and release.

No matter the lock is a mutex, spinlock or reader/writer lock, these three states exist on the execution path. However, if the request tries acquiring a lock, the states will be different. If

Algorithm 1 The process of how to record the behaviors of every request for a lock. The algorithm has the computation time complexity of $O(lgN)$.

```
1: Map lock_to_recorder
2: procedure RECORD_LOCK_ACTION(id, op_type)
3:     if !(address∈ lock_to_recoder) then
4:         lock(mapLock)
5:         if !(address∈ lock_to_recoder) then
6:             t_recorder ← new Recoder
7:             lock_to_recorder.insert(address, t_recorder)
8:         end if
9:         unlock(mapLock)
10:    end if
11:    recorder ← lock_to_recorder[id]
12:    lock(recorder)
13:    recoder ← push_back(current_time, op_type)
14:    unlock(recorder)
15: end procedure
```

working threads succeed acquiring lock, the states are the same as those of lock operation. However, if the working threads fail acquiring lock, a new state comes up: fail. The possible execution path is shown in Figure 4. Finally the set of states consist of four states, including request, success, release and fail.

With the set of four states mentioned above, there are two possible execution paths for a thread when acquiring a lock. The first path is the successful path, including request, success and release. The second path is the failed path, which only happens when threads try to acquire locks, including request and fail.

We present the process of how to record the behaviors of every request for a lock as Algorithm 1. The first line is definition of a map from the lock to the recorder. As hash map's computation is not complex, we choose red-black tree to implement the map, because the high efficiency of hash map is based on extra memory space.

The system bottleneck is memory when profiling, so red-black tree is better than hash in this situation. Line 3 to Line 8 is the initialization of recorder. If it is the first time to record the action of a lock, we must build a new recorder to store the action. Multiple threads will access the initialization code block, so we use lock to protect it. Because initialization code block should only be executed once, we use the strategy of double check to ensure the consistency of initialization. After initialization, it is certain that there is a corresponding recorder for the lock. So we store the action and corresponding timestamp into the resizable array of recorder. The whole process has the time complexity O(lgN). This process mentioned above is the basement of profiling.

Then, we propose the algorithm to record the action of every lock based on the function record_lock_action. Now, most parallel programming environment is based on Pthreads library. For example, OpenMP uses Pthreads to create threads. In the implementation of GNU for OpenMP, they straightly use the lock mechanism of Pthreads to keep the consistency of shared-memory programs. More details can be found in the head file mutex.h. So we add the process of record in the library of Pthreads.

```
1.  class proxy_mutex_t{
2.      pthread_mutex_t mutex;
3.      int id;
4.      proxy_mutex_t(int _id):id(_id){
5.          pthread_mutex_init(&mutex, NULL);
6.      }
7.      proxy_mutex_t(proxy_mutex_t &t){
8.          mutex = t.mutex;
9.          id = t.id;
10.     }
11.     ~proxy_mutex_t(){
12.         pthread_mutex_destroy(&mutex);
13.     }
14.     void lock(){
15.         pthread_mutex_lock(&mutex);
16.     }
17.     void unlock(){
18.         pthread_mutex_unlock(&mutex);
19.     }
20. };
```

Figure 5: The sample implementation of proxy_mutex_t, for this solution, we can hook the operations for locks without polluting the Pthreads library and the source code.

To prevent from polluting the formal Pthreads library, we define new structures as proxy for recording the actions of locks. For example, there is a lock L with type of pthread_mutex_t, we redefine the type of L as a new structure called proxy_mutex_t. In the structure of proxy_mutex_t, there is a mutex as a key element. We rewrite the function of lock and unlock for proxy_mutex_t. Then, all the operations of the locks are hooked. We can call the function of record_lock_action before or after the operations to record the operations of corresponding locks. We present the sample implementation of proxy_mutex_t in Figure 5. As shown in Figure 5, the operation of lock is redirected to the variable mutex. The variable id is used to identify different locks. The other types of locks are implemented in a similar manner.

After the new structure proxy_mutex_t has been defined, we insert instrumentation into the specific locations of source codes. The solution is to define new operations of locks instead of the original operations. Algorithm 2 shows the sample implementation of the operations within proxy_mutex_t. As shown in algorithm 2, we redefine three operations of mutex, including locking, trying to lock and releasing. During the execution path, we call function record_lock_action to record the corresponding actions. The final time complexity of these algorithms is $O(lgN)$.

With the solution mentioned above, we can start to do profiling work. First, we locate main function when getting the input of source code. We insert initialization function at the beginning of main function. A head file is also inserted into the function. In the head file, we rewrite the definition of former locks. After the operation is done, we define new lock structures. So the locks are all redirected to our own lock structures. In these locks we record the moment of every lock operation and unlock operation. To avoid the interruption of print logs, we print logs at last.

Second, we compile the source code that has been converted. When the program is executing, FFLocker forks another process to record usage of CPUs per half second.

Algorithm 2 The solution to redirect original operations. We redefine the three operations of $mutex$. The computation time complexity of algorithm is $O(lgN)$.

```
1: procedure PTHREAD_MUTEX_LOCK(proxy_mutex_t * t)
2:     record_lock_action(t→id, StateType.request)
3:     t→lock()
4:     record_lock_action(t→id, StateType.success)
5: end procedure
6: procedure PTHREAD_MUTEX_TRYLOCK(proxy_mutex_t
   * t)
7:     record_lock_action(t→id, StateType.request)
8:     int result→pthread_mutex_trylock(&t→mutex)
9:     if result == 0 then
10:        record_lock_action(t→id, StateType.success)
11:    else
12:        record_lock_action(t→id, StateType.fail)
13:    end if
14:    return result
15: end procedure
16: procedure PTHREAD_MUTEX_UNLOCK(proxy_mutex_t *
   t)
17:    record_lock_action(t→id, StateType.release)
18:    t→unlock()
19: end procedure
```

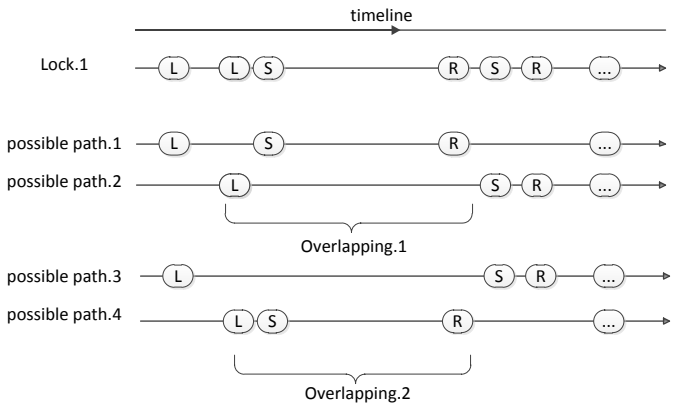

Figure 6: A series of operations for a specific lock on the direction of timeline. L is a request. S is success and R means release. Possible paths are the possible combinations with operations.

Finally, we get two files, including the lock operations and the usage of CPUs. FFLocker will analyze these two log files. Those locks locked when most CPUs keep idle can probably have been used to protect large critical sections. Then, FFLocker profiler gives the surmise of where large critical sections locate.

Another important problem, which should be considered, is how to confirm the combinations of operations. When the steps mentioned above finish, we get a series of discrete points that show the operations of a lock. As shown in Figure 6, there is a series of operations for a lock with the direction of timeline. It is difficult to restore the execution path, as it is almost impossible to enumerate all the combinations of operations. However, there is a breach for us. That is,

Algorithm 3 An algorithm to check if the lock can protect the large critical sections. The computation time complexity is $O(KN)$.

```
1: procedure LOCKCHECK(lock_info, cpu_info)
2:     path1 ← get_first_path
3:     hPointer ← get_next_start
4:     tPointer ← get_next_end
5:     while hPointer != lock_info.tail do
6:         path2 ← build(hPointer, tPointer)
7:         overlap ← get_overlap(path1, path2)
8:         for each sample in overlap do //sampling rate K
9:             if most_cpu_blocks then
10:                ++cnt
11:            end if
12:            if cnt> threshold then
13:                ++gCnt
14:            end if
15:        end for
16:        path1 ← path2
17:        hPointer ← get_next_start // next "L"
18:        tPointer ← get_next_end // next "F" or "R"
19:    end while
20:    if gCnt > gThreshold then return true
21:    elsereturn false
22:    end if
23: end procedure
```

the idleness of thread when acquiring a lock equals to the overlapping of two possible paths. We can find from Figure 6 that the first overlapping equals to second overlapping though the combinations of operations is different. So if we construct a possible combination for all operations, it is feasible to get all the overlapping. Based on this theory, we design an algorithm with $O(KN)$ time complexity to judge if the lock protects large critical sections or not. In the time complexity, N means the total operations for a lock while K is the sampling rate for every overlapping.

As shown in algorithm 3, we use two pointers to find the next possible path. hPointer is the start of a path, corresponding to the operation L. tPointer is the end of a path, corresponding to the operations F and R. As an end is always after a start, there is no need to find path recursively. So the total computation time complexity to find the possible path is linear. After finding the paths, we get the overlap between the two paths. We set a sampling rate K to check if the lock protects large critical sections. We divide the overlap into K parts. During the intervals, if most CPUs are in idle state at the same, we let the statistical value plus 1. If the statistical value exceeds the threshold, we identify the overlap as a large critical overlap and let the global statistical value plus 1. If the global statistical value exceeds the threshold, the lock analyzed may protect large critical sections. The time complexity of a specific lock is $O(KN)$, so the total time complexity of all locks is $O(KNM)$, assuming the total sum of locks is M. The result is a little foggy, but can help programmer locate large critical sections. Especially in programs with thousands of lines code, this solution of profiling is more useful than experienced programmers.

C. Runtime optimization for large critical region

After detecting the locations of large critical sections, the runtime solution is mainly designed to support task-level lock, called TLock. As mentioned is section 2, TLock extends helper lock proposed. Runtime system to support TLock is presented as a dynamic link library [14], including the logic of how TLock makes performance efficienct. We present the process of how a TLock works below.

A parallel subtask called KTask (Key Task) is put into the double-end task queue assigned to the corresponding TLock instead of the task queue of system. We record the double-end task queue as TQueue(L) when it is assigned to a TLock L. WQueue(w) represents the private task queue of worker thread w when work-stealing. GQueue represents the global task queue of system that stores the tasks created by workers when work-sharing. TManager (TLock Manager) is the global manager of all the information of TLock and TQueue (Li).

Initialization function of TLock will be executed at the beginning of runtime system, and it initializes the TManager. Then, worker threads start to execute tasks. When subtasks are created, they are put into WQueue(wi) (when work-stealing) or GQueue (when work-sharing). All the steps are the same as popular parallel programming environment until the worker threads need to acquire TLock. There are two branches after attempting to acquire TLock.

If worker thread succeeds in acquiring TLock, runtime system will call the initialization function of TLock to initialize the status of TLock L and registers L at TManager. Runtime system creates a new task queue of TLock called TQueue(L) and assigns TQueue(L) to L. After these actions have been done, runtime system signals the TManager that the initialization of TLock L is over. Worker thread continues to execute the large critical section. If there are parallel subtasks created, runtime system pushes them to the tail of TQueue(L). Runtime system sets an implicit synchronous barrier at the end of every parallel region. Worker threads start getting tasks from the top of TQueue(L) to execute when facing barrier. After all the tasks have been completed, worker thread continues to run the follow codes. The process mentioned above is how runtime system works when acquires TLock successfully.

If worker thread fails in acquiring TLock L, it does not block to wait for L. Instead, worker thread calls the help function to execute the subtasks created by the large critical section. At the beginning of help function, worker thread attempts to get information of L from TManager. If TLock L is still initializing, worker thread occupies the CPU. Then, worker thread keeps searching until L finishes initializing. After L finishes initializing, worker thread visits the TQueue(L) and gets the subtask from head to execute. If TQueue(L) is empty, worker thread attempts to search other additional task queues from TManager. TManager can decide which additional task queue to be executed first for their different priorities. If additional task queues are not empty, worker thread steals subtasks from the top of TQueue(Li) to execute. If all the TQueue(Li) are empty, worker thread attempts to steal subtasks from the top of WQueue(wi) to execute. Certainly, before any subtask starts to execute, worker thread shall judge if L is released by other worker threads. If L is released, worker thread attempts to roll back to acquire L.

D. Runtime optimization for frequent lock request

We attempt to propose another runtime solution to reduce the contention caused by the problem of frequent lock request. When the issue of frequent lock request happens, different threads may request the same lock frequently. In this situation, most computation ability of CPUs is spent on the context switching. These cores will switch between user state and system state frequently. In fact, if the conflict between two threads is very high, there is no need to put them onto different cores. Because even they are put on different cores, there is only one active thread at a time. Putting them onto different cores brings much more system cost.

o, we propose a runtime solution, which binds the threads, which request the same locks, onto the same cores dynamically. The issue of frequent lock request always happens in thread-oriented patterns of parallel programming. So we present our runtime solution on Pthreads library. To prevent from polluting the Pthreads library, we propose the solution as a plug-in for Pthreads. First, we redefine the structure of locks that may lead frequent lock request. There is a unique ID in the new structure of locks. Then, in the initialization function, we create a table to store the corresponding relationship between the number of core and the lock that leads frequent lock request. Then we redefine the structure of threads and the function for creating new threads. For the new created function we assign a symbol to the specific thread as a pthread_key_t [15] to store the information if this thread has been migrated. After these actions have been done, we redefine the operations of locks. Before the operations of locks have been done, threads will check if it needs to be migrated or not. If it needs to be migrated, the runtime system calls the function sched_setaffinity [16] on Linux platforms to bind the threads to the right cores. After migration, runtime system modifies the value of migration information to prevent from duplicate migration.

If the threads, which request the same locks, are bound to the same cores, there is only one active thread, which requests the lock at a time. For this reason, CPUs can reduce much computation time on context switching. However, this solution only works well when the conflict among the threads is high. If the conflict among the threads is low, there is little possibility for threads to face the problem of frequent lock request.

IV. Performance evaluation

This section reports performance of FFLocker. We have implemented two benchmarks with the situations lead lock contention, including large critical sections and frequent lock request. These experiments run on a multi-core platform with 8 cores (Intel Xeon E5310). More specially, we implement the first benchmark that leads large critical sections with OpenMP and Function Flow [13]. Then we implement the second benchmark that leads frequent lock request on Pthreads library. These three parallel programming environments cover task-oriented and thread-oriented parallel programming environments.

A. Experiment for optimization of large critical region

We develop a parallel program that concurrently inserts 10^8 random numbers into a parallel hash table as our benchmark to test the performance of FFLocker. We similarly implement it with Function Flow and OpenMP. This is a typical implementation of parallel hash table insertions, but not the unique implementation.

Table II: THE computation time cost on the benchmark of parallel hash table insertions

Experiment No.	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10Th
Serial	9.13 (S)	8.91 (S)	9.14 (S)	9.08 (S)	8.82 (S)	9.04 (S)	8.95 (S)	9.14 (S)	9.23 (S)	8. 98 (S)
OpenMP	5.58 (S)	5.59 (S)	5.92 (S)	5.62 (S)	5.52 (S)	5.71 (S)	5.53 (S)	5.63 (S)	5.42 (S)	5.57 (S)
OpenMP with FFLocker	3.11 (S)	3.14 (S)	3.23 (S)	3.13 (S)	3.21 (S)	3.22 (S)	3.15 (S)	3.24 (S)	3.44 (S)	3.13 (S)
Function Flow	5.38 (S)	5.44 (S)	5.21 (S)	5.42 (S)	5.26 (S)	5.52 (S)	5.34 (S)	5.61 (S)	5.59 (S)	5.45 (S)
Function Flow with FFLocker	3.08 (S)	3.21 (S)	3.14 (S)	3.05 (S)	2.93 (S)	3.05 (S)	3.18 (S)	3.24 (S)	3.06 (S)	2.97 (S)

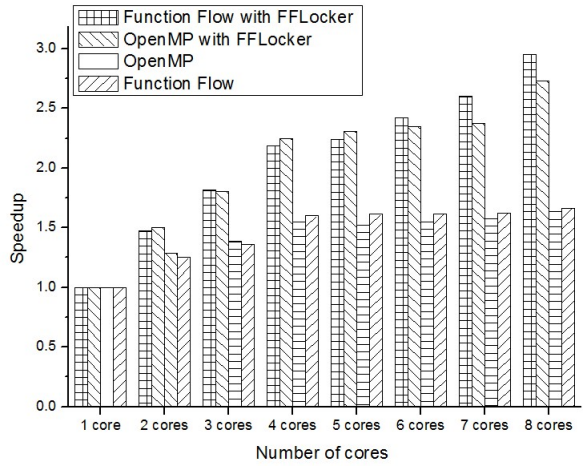

Figure 7: This plot performs parallel hash table insertions on multi-core platforms with different number of cores.

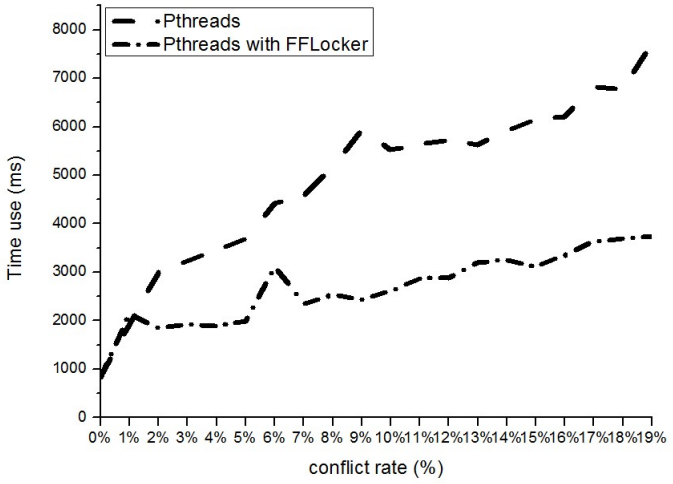

Figure 8: This plot performs the result of our benchmark that leads frequent lock request on a 8 core platform.

Table II shows the results of the series of experiments on the benchmark of parallel hash table insertions. On a 8 core platform, FFLocker does well to help parallel programming environment to reduce the lock contention.

Then we have a series of experiments on multi-core platforms with different cores. We made a horizontal comparison to find the relationship between speedup and the number of cores in FFLocker. The result is shown in Figure 7. It is not difficult for us to find that FFLocker helps parallel programming environments to get much performance gain when facing large critical sections. Besides this, there is another clear conclusion we can get from Figure 7. The system optimization made by FFlocker is positive correlation with the number of cores.

B. Experiment for optimization of frequent lock request

We develop a parallel program that leads the problem of frequent lock request as our benchmark on Pthreads. The program is designed to simulate the process of issuing tickets. Assuming there is a series of threads that keep serving the corresponding clients. Some of clients perhaps keep booking hotels and some perhaps keep ordering tickets of train. This situation makes a corresponding situation on server as follows. Some threads keep acquiring the same locks. Our benchmark is designed to simulate this situation. More specially, we create 64 threads on a 8 core platform. These 64 threads keep acquiring 8 locks. We set the conflict rate among the threads as a variable. The result of experiment is shown in Figure 8.

As shown in Figure 8, as the conflict rate becomes bigger, the optimization result of FFlocker becomes more obvious. Our solution does well if the conflict rate is higher among the threads. As shown in Figure 8, if the conflict among the threads is low, our solution does not work well. For example, if the conflict rate is 1%, our solution is slower than Pthreads solution.

V. Related work

Recent research attempts to learn and analyze lock contention. For example, HPCToolKit [17] is proposed as a series of tools, which analyze the performance of parallel programs. Tallent [18] proposes and evaluates three new strategies to analyze the performance loss caused by lock contention. They have implemented a runtime strategy in HPCToolKit that only incurs $< 5\%$ extra overhead on a benchmark of quantum chemistry application. Huang [19] proposes HaLock as a lock profiling mechanism assisted by hardware to find the performance loss caused by lock contention.

Lock prediction is another solution to analyze lock contention. Brandon [20] discusses a series of use cases for lock prediction. One of the use cases is to reduce the lock contention. For this reason, lock prediction is another solution to make optimization for a parallel program. Then, they also describe a series of multiple prediction algorithms [20]. These papers mentioned above provide useful information about lock contention by analysis or prediction. They pass the problem of system optimization to parallel programmers. It is of course reasonable for HPC. However, because multi-core processors with hundreds or even more quantities of cores become universal, lock contention in HPC applications starts to puzzle normal programmer. Programmers need less programming complexity and better performance. There are a number of papers to discuss new operations instead of lock, such as CAS or Transactional Memory [4]. They may

do well in solving the problem of lock contention in some situation. Tim [21] presents a new form of revocable lock that provides mutual exclusion without blocking threads. It is a good inspiration for us to solve the problem of lock contention.

VI. Conclusion

This paper mainly explores the runtime solution to reduce lock contention on multi-core platforms. We focus on reducing lock contention caused by large critical sections and frequent lock request. In the future, we will merge lots of new solutions into FFLocker for the good scalability of FFLocker. Then, programmers can easily reduce lock contention by FFLocker without considering complex programming logics.

Acknowledgment

This work is supported by National Natural Science Foundation of China under grant Nos. 61272408, 61322210, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048.

References

[1] J. K. N. Farhang and T. Nolte, "Towards migrating legacy real-time systems to multi-core platforms," in *In Proceedings of Emerging Technologies and Factory Automation*, 2008, pp. 717–720.

[2] G. David, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] S. Nir and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[4] V. S. A. B. Robert and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 247–258.

[5] B. Lewis and D. J. Berg, *Multithreaded programming with Pthreads*, 1998.

[6] A. P. Johnson and M. W. S. Macauley, "High precision timing within microsoft windows: threads, scheduling and system interrupts," *Microprocessors and Microsystems*, vol. 25, no. 6, pp. 297–307, 2001.

[7] C. Gene, "Top-c: Task-oriented parallel c for distributed and shared memory," in *In Proceedings of the Workshop on wide area networks and high performance computing*, 1999, pp. 109–117.

[8] C. E. L. A. Kunal and J. Sukha, "Helper locks for fork-join parallel programming," in *In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 245–256.

[9] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[10] B. Christian and M. Brorsson, "Odinmp/ccp -a portable implementation of openmp for c," *Concurrency and Computation: Practice and Experience*, vol. 12, no. 12, pp. 1193–1203, 2000.

[11] E. C. F. Matteo and H. Keith, "The implementation of the cilk-5 multithreaded language," in *In Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1998, pp. 212–223.

[12] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, 2010.

[13] L. Z. X. L. C. Y. X. Fan, H. Jin and X. Tu, "In proceedings of ppopp workshop pmam," in *Function flow: making synchronization easier in task parallelism*, 2012, pp. 74–82.

[14] J. E. Walsh, "U.s. patent no. 5,375, 241," in *Method and system for dynamic-link library*, 20 Dec. 1994.

[15] D. B. N. Bradford and J. Farrell, *O'Reilly Media Inc*, 1996.

[16] *Linux Manual*, 2006.

[17] S. B. L. Adhianto and M. Fagan, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[18] J. M. M.-C. N. R. Tallent and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 269–280.

[19] Z. C. Y. Huang and L. Chen, "Halock: hardware-assisted lock contention detection in multithreaded applications," in *In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 253–262.

[20] J. D. B. Lucia and T. Bergan, "Lock prediction," in *In Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, 2010, pp. 101–101.

[21] T. Harris and K. Fraser, "Revocable locks for non-blocking programming," in *In Proceedings of the tenth Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 72–82.