MODELING AND RESOLVING LOCK CONTENTION FOR MULTI-THREADED SYSTEMS

YANG ZHANG\textsuperscript{1,2} AND SHAHNAWAZ TALPUR\textsuperscript{1}

\textsuperscript{1}High Performance Embedded Computing Lab
School of Computer Science and Technology
Beijing Institute of Technology
No. 5, South Zhongguancun Street, Haidian Dist., Beijing 100081, P. R. China
zhangyang@hebust.edu.cn; talpur@bit.edu.cn

\textsuperscript{2}School of Information Science and Engineering
Hebei University of Science and Technology
No. 186, Yuhua East Rd., Shijiazhuang 050018, P. R. China

Received July 2011; accepted October 2011

Abstract. Locks are efficient concurrent control mechanisms to ensure that shared resources are accessed by only a single thread in multi-threaded applications. However, for multi-core systems, lock usage leads to lock contention, which degrades program performance, increases system response time, and negatively affects scalability. This paper initially models lock contention from two aspects: static structure and dynamic behavior. Static constraints of lock contention that describe the contention among threads and the dynamic constraints that define the lock request behavior during contended process are presented. In addition, a contention-aware lock (CA-Lock) is proposed. The CA-Lock is an ordinary lock when there is no contention; it improves the concurrency by transforming into software transactional memory (STM) when the contention is serious. For those situations that are not fitted with STM, the CA-Lock can transform back into an ordinary lock. Primary experimental results show that the CA-Lock can track the best world of lock or STM.

Keywords: Lock contention, Model, Multi-threaded systems, Contention-aware lock

1. Introduction. In multi-threaded systems, locks are frequently used as concurrent control mechanisms to ensure that shared resources are accessed by only a single thread. For those applications running on a mono-core processor, locks have been proven to be extremely efficient. Even if lock contention occurs, the overhead mainly includes acquiring and releasing the lock, which has little influence on the performance of the whole program. However, when transplanting these applications to a multi-core platform, shared resources are occupied by only a single thread running on a single core, while threads running on other cores are suspended, thereby leading to lock contention. As a result, lock contention degrades program performance, increases system response time, and negatively affects scalability.

To solve this problem, software transaction memory (STM) and lock-free algorithms are proposed in the current work to increase the concurrency of the critical section. However, these technologies have some disadvantages. For example, STM suffers from high overhead, and some irrevocable operations such as I/O operation. Lock-free algorithms are too difficult for a programmer to master. Although many alternate technologies have already been proposed, it seems that locks are here to stay, enjoying continued use well into the future. A large number of lock-based legacy codes will run on the multi-core machine. Therefore, it is necessary to focus on the problem of lock contention.

Some researchers have proposed various technologies, including closed queue network [1], birth-death process for Markov chains [2], Colored Petri Net [3], and so on. These
technologies are mainly concerned with the performance model of lock contention, and few studies have been conducted on modeling lock contention itself. Early lock contention has been modeled for a database system, which considers the mean blocking time of different database access requests [1]. However, the mean value method is not fit to model lock contention on multi-core processors, because the mean value covers up the number of contented threads. For example, the mean blocking time of 100 threads might be equal to that of two threads, although it is apparent that the lock that blocked 100 threads is more seriously contended than the lock with two blocked threads.

This paper initially models lock contention from two aspects: static structure and dynamic behavior. The static constraints of lock contention describe the contention among threads, and the dynamic constraints define the lock request behavior during contended process. In addition, a contention-aware lock (CA-Lock) is proposed, which encapsulates the existing lock mechanism. In combination with the STM, the proposed lock can make a dynamical choice between the lock and STM. Compared with a single mode like lock or STM, the CA-Lock can select the best performance of the lock or STM. Experimental results show that when the CA-Lock is applied to two benchmarks of STAMP benchmark suite [4], it can track the best world of lock and STM.

2. Lock Contention Model. To model lock contention, a closed system with a fixed number of threads and a fixed number of locks is introduced. The reasons for using a closed system are given in [5]. The static structure of a closed system is first defined as a set of threads and locks.

**Definition 2.1.** Let the static structure of a closed system be $S_{\text{com}} = (T, L, \theta, \delta)$ where $T$ is a set of threads, $L$ is a set of locks, and $\theta$ is a mapping: $\theta : L \rightarrow \{0, 1\}$. In $\forall l \in L$, if $\theta(l) = 1$, a lock $l$ is locked; if $\theta(l) = 0$, a lock $l$ is unlocked. Then, $\delta$ is a mapping: $\delta : T \times L \rightarrow \{-1, 0, 1\}$. In $\forall t \in T$, $l \in L$, $\delta(t, l) = 1$, it represents that a thread $t$ is requesting for a lock $l$, but the thread $t$ is not holding the lock $l$; if $\delta(t, l) = -1$, then the thread $t$ is not requesting for the lock $l$; if $\delta(t, l) = 0$, then the thread $t$ is requesting for the lock $l$ and the thread $t$ is holding the lock $l$.

**Definition 2.2.** Let $S_{\text{com}} = (T, L, \theta, \delta)$ be a static structure of a closed system. In $\forall t \in T$, $l \in L$, $<t, l>$ denotes that a thread $t$ is sending a request to try to hold a lock $l$. If $R \subseteq T \times L$ meets the following condition: $\forall r \in (T \times L)((r = <t, l>) \land \delta(t, l) \neq -1) \rightarrow r \in R)$, then $R$ is a set of lock requests, and the element in $R$ is a lock request.

To show which element of a set $R$ requests for which lock, $r \rightarrow l$ is used to denote that the request $r$ is requesting for the lock $l$. Furthermore, if $\forall r \in (T \times L)((r = <t, l>) \land \delta(t, l) = 1) \rightarrow r \in R)$ holds, then $r \mapsto l$ is used to denote that the request $r$ is waiting to hold the lock $l$.

**Theorem 2.1.** The number of lock requests is fixed for a closed system.

**Proof:** Due to page limitation, the proof is hereby abbreviated.

**Definition 2.3.** Let $R$ be a set of lock requests. Different scenarios exist in $\forall r_i \in R$, $1 \leq i \leq n$. X1) if there exists a sequence $O_c = r_1r_2r_3 \cdots r_n$, then $O_c$ is an occurrence sequence of lock request; X2) if there exists a sequence $O_q = r_1|r_2|r_3| \cdots |r_n$, then $O_q$ is a request sequence of lock request; X3) if there exists a sequence $O_e = r_1 \preceq r_2 \preceq r_3 \preceq \cdots \preceq r_n$, then $O_e$ is an execution sequence of lock request.

For X1, an occurrence sequence that denotes the former request in $O_c$ is generated not later than the latter request. For X2, $O_q$ represents that these requests are sent to an unlocked lock at the same time. For X3, $O_e$ represents that these requests can make up a specific sequence after the critical section is executed. The former request holds the lock and executes the critical section earlier than the latter request.
2.1. Static constraints of lock contention. The static constraints of lock contention describe the contended relationship between threads. Let $S_{com} = (T, L, \theta, \delta)$ be a static structure of a closed system. The contention between threads is defined as a mapping $LC : T \times T \rightarrow \{0, 1\}$. In $\forall t_1 \in T$, $t_2 \in T$, if $LC(t_1, t_2) = 1$, a lock contention exists between the threads $t_1$ and $t_2$; if $LC(t_1, t_2) = 0$, then there is no lock contention between the threads $t_1$ and $t_2$. If $LC(t_1, t_2) = 1$, $l \in L$, and suppose that $\delta(t_1, l) = 1$, the following conclusions are derived. C1) $\delta(l) = 1$; C2) $\theta(t_2, l) \neq -1$; C3) If $\forall t_i \in T$, $\delta(t_i, l) = 0$, then $\exists 1 \leq j \leq n(t_j \in T \land t_i \neq t_j \rightarrow \delta(t_j, l) = 0)$.

2.2. Dynamic constraints of lock contention. Dynamic constraints of lock contention are defined by the occurrence and request sequence of lock request.

Definition 2.4. Let a closed system be $CS = (S_{com}, R, O)$, where $S_{com}$ is the static structure of a closed system, $R$ is a set of lock requests, and $O$ is the request sequence, which has different values at different stages ($O_c$ or $O_e$ or $O_q$).

Definition 2.5. Let $CS = (S_{com}, R, O)$ be a closed system; if the following condition is met: $\forall l \in L((\theta(l) = 1) \rightarrow \exists r \in R(r \xrightarrow{w} l))$, the closed system is going through lock contention on lock $l$.

Definition 2.6. Let $CS = (S_{com}, R, O)$ be a closed system, where $S_{com}$ is the static structure of the closed system, $R$ is a set of lock requests, and $O$ is a request sequence. In $\forall r_i, r_j \in R$, $r_i = <t_i, l>$, $r_j = <t_j, l>$, $r_i$ contends with $r_j$, if and only if the following conditions are met: W1) $((\theta(l) = 0) \land (r_i \rightarrow l) \land LC(t_i, t_j) = 1) \rightarrow ((r_j \rightarrow l) \land O_q = r_i | r_j)$; W2) $((\theta(l) = 1) \land (r_i \rightarrow l) \land LC(t_i, t_j) = 1) \rightarrow ((r_j \rightarrow l))$.

For W1, if the thread $t_i$ requests for the lock $l$ and there is a lock contention between the threads $t_i$ and $t_j$ in unlocked state, then the thread $t_j$ is requesting for the lock $l$. Two requests are sent out at the same time. For W2, if the thread $t_i$ requests for the lock $l$ and there is a lock contention between the threads $t_i$ and $t_j$ in locked state, then the thread $t_j$ is requesting for the lock $l$, and two requests that sent out by the threads $t_i$ and $t_j$ are requesting to acquire the lock.

The duration of each lock request occurs from generation up to the end. The dynamic constraints of lock contention are described by considering the time property. However, due to page limitation, the discussion is not presented here. The static structure and dynamic behavior constitute an intact model of lock contention. A basis is provided as a guide in creating the performance model of lock contention.

3. Contention-Aware Lock. In this section, the solution to lock contention, called contention-aware lock (CA-Lock), is presented. The aim of the CA-Lock is to improve the scalability of multi-threaded systems, and to ensure correct concurrent control.

Ideally, the CA-Lock is an ordinary lock when threads are not in contention for locks. Once contention occurs, the CA-Lock rapidly sets up a blocked request queue for the contended threads. The elements of the blocked request queue record some information that would be operated by the CA-Lock, such as contended thread identifier, and identify which lock is contended for. When the contention is serious, the CA-Lock increases the concurrency of these requests by transforming from the lock mode into the STM mode. For those situations that do not fit to the STM mode, such as I/O and thread communication, or when the efficiency of STM is lower than that of the lock mode, the CA-Lock transforms back into an ordinary lock. Frequent state transition is avoided by limiting the transitional count (less than 3).

Algorithm 1 shows how to acquire the CA-Lock. A compare-and-swap (CAS) operation (line 8) is used to acquire the CA-Lock. If acquisition is successful and other threads want to acquire the CA-Lock, these threads can be inserted into the blocked queue, after which
their recorded flag is set (lines 2-4). If the thread in the blocked queue acquires the CA-Lock successfully, the thread is deleted from the blocked queue. Note that the deletion is not protected by the lock operation, because only one thread obtains the lock and executes the deletion. If the CA-Lock is in the STM mode (lines 5-7), it starts a transaction and sets a roll-back point. In the STM mode, threads that are not in the queue continue to be inserted into the blocked queue. This process identifies the threads are going to be notified to roll back at the releasing stage.

Algorithm 1. Algorithm of acquiring the CA-Lock

Algorithm 2. Algorithm of releasing the CA-Lock

Algorithm 2 shows the process of releasing the CA-Lock. In releasing the CA-Lock in lock mode, the CA-Lock judges whether or not the number of blocked threads is greater than the limited number (line 10). If so, the CA-Lock transforms from the lock mode into the STM mode. Otherwise, the CA-Lock sets the value of calock->lockHolder to 0 (line 14) to indicate an unlocked state and continue in the lock mode. If the current state is in the STM mode, the CA-Lock commits the transaction. For a successful transaction, the
CA-Lock deletes the current thread from the blocked queue and clears their record flag. Should the transaction fail, the CA-Lock jumps to the roll-back point to restart. The retry times of transactions are checked to determine whether it should return to the lock mode. Note that it may be an inaccurate opportunity to modify the transition; it is not related to the correctness of the program, but to the mode that executed by the program. When exiting the transactional mode, the CA-Lock must notify all the threads in the blocked queue to roll back. A signal mechanism is used to implement the notification. When all the threads finish the transition, the lock mode starts. The value \texttt{calock->blockingLimit} is not always a small number (such as 1 or 2). Instead, its value can be tuned to control the time of transition.

4. **Experimentation.** Measurements were taken on Intel Xeon E5602 CPU (two processors with eight cores on each processor, and each core ran at 2.4GHZ) with 8G of RAM running 64-bit Linux (kernel 2.6.35). The GCC version was 4.4.5 and version 0.9.6 of TL2 was used for the CA-Lock. The STAMP benchmark suite [4] version 0.9.10 was also used. The \texttt{labyrinth} benchmark was evaluated with the input file “random-x512-y512-z7-n512.txt” and \texttt{kmeans} benchmark was evaluated with the input file “random-n2048-d16-c16.txt”. The experimental results are shown in Figure 1. All the measurements are medians of five runs. The figure plots the execution time, and in this case, lower is better.

The execute time of CA-Lock was compared with those of the lock and STM. A \texttt{mutex} lock in \texttt{pthread} library was used for the lock mode, and TL2(x86)-based STM implementation was also used for the STM mode. In these two benchmarks, the value of variant \texttt{calock->blockingLimit} was set to 1 for the \texttt{labyrinth} benchmark and 2 for the \texttt{kmeans} benchmark. For the \texttt{labyrinth} benchmark running with the CA-Lock, execution process indicates that it runs in lock mode first, and then changes its mode to STM when lock contention occurs. Note that the execution time of the \texttt{labyrinth} benchmark in the lock mode is very high and does not change much, as shown in Figure 1(a). This is because the core function is locked by the \texttt{mutex} lock. In the STM mode, the execution time decreases with the increase of the number of threads, and the execution of the CA-Lock closely monitors that of the STM. Moreover, the \texttt{kmeans} benchmark runs in lock mode for most situations when using the CA-Lock. Due to low contention, the benchmark does not change into STM mode. Therefore, the execution of the CA-Lock tracks the execution time of the lock mode.

![Figure 1. Execution time of labyrinth benchmark and kmeans benchmark](image)

5. **Related Works.** There are many distinguished studies on modeling lock contention [1, 2, 3] and resolving it [6, 7, 8]. Thin locks [6] allow lock and unlock operations to be performed with only a few machine instruction words per object. Thin locks become fat locks with a blocked queue when contention occurs. Both thin locks and the CA-Lock use
CAS operation and build up the blocked queue. However, there are two key points that distinguish the CA-Lock from thin locks. First, thin locks judge the contention in locked operation while the CA-Lock is in unlocked operation. Second, once thin locks become fat locks, they never transform back into thin locks, whereas the CA-Lock allows mode transition several times (less than 3).

Adaptive lock [7] dynamically observes whether a critical section is best executed transactional or while holding a lock. Different from the adaptive lock, the CA-Lock ensures that the transition occurs at the unlocked stage; in addition, it does not use a state machine. Another difference between the CA-Lock and the adaptive lock is that the CA-Lock cannot judge quickly which mode it should be when it encounters lock contention. However, with the execution of the critical section first, the CA-Lock can obtain some information about the execution environment, such as execution time of critical section, the number of contended threads, and the existence of thread communication and I/O operation, among others. If there are some conditions that do not fit into the STM mode, the transition does not occur at all.

Some studies [9, 10] have discussed the lock protocol in different applications.

6. Conclusions. This paper presents a formal model of lock contention via static structure constraints and dynamic behavior constraints. This model is concerned mainly with describing lock contention itself. Solutions to lock contention are introduced by presenting algorithms to acquire and release the CA-Lock. Based on the experimental results, the CA-Lock can track the best of the world of lock or STM. Further studies may include providing a performance model of lock contention based on the proposed lock contention model and testing the CA-Lock using more benchmarks.

Acknowledgment. This work is partially supported by National Nature Science Foundation of China under grant No. 60973010. The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES