On Performance Debugging of Unnecessary Lock Contentions on Multicore Processors: A Replay-based Approach

Long Zheng Xiaofei Liao *
Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China
{longzh, xfliao}@hust.edu.cn

Bingsheng He
School of Computer Engineering Nanyang Technological University Singapore
bshe@ntu.edu.sg

Song Wu Hai Jin
Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, China
{wusong, hjin}@hust.edu.cn

Abstract

Locks have been widely used as an effective synchronization mechanism among processes and threads. However, we observe that a large number of false inter-thread dependencies (i.e., unnecessary lock contentions) exist during the program execution on multicore processors, thereby incurring significant performance overhead. This paper presents a performance debugging framework, PERFPLAY, to facilitate a comprehensive and in-depth understanding of the performance impact of unnecessary lock contentions. The core technique of our debugging framework is trace replay. Specifically, PERFPLAY records the program execution trace, on the basis of which the unnecessary lock contentions can be identified through trace analysis. We then propose a novel technique of trace transformation to transform these identified unnecessary lock contentions in the original trace into the correct pattern as a new trace free of unnecessary lock contentions. Through replaying both traces, PERFPLAY can quantify the performance impact of unnecessary lock contentions. To demonstrate the effectiveness of our debugging framework, we study five real-world programs and PARSEC benchmarks. Our experimental results demonstrate the significant performance overhead of unnecessary lock contentions, and the effectiveness of PERFPLAY in identifying the performance critical unnecessary lock contentions in real applications.

1. Introduction

In the era of multi-core processors, parallel programming is prevalent. The efficiency of process/thread communication is very important for the overall performance of parallel executions. In multi-threaded applications, locks are widely-used to ensure mutual accesses to shared data within critical sections. A thread has to acquire a lock until the lock release if this lock is held by another thread. However, multiple critical sections protected by the same lock do not necessarily conflict at runtime. Therefore, a program may produce false inter-thread dependency (i.e., unnecessary lock contention). Such unnecessary lock contentions serialize the access, leading to the severe performance loss of programs [22, 23]. In this paper, we study whether and how we can help the programmer identify the unnecessary lock contention and further understand their performance impact.

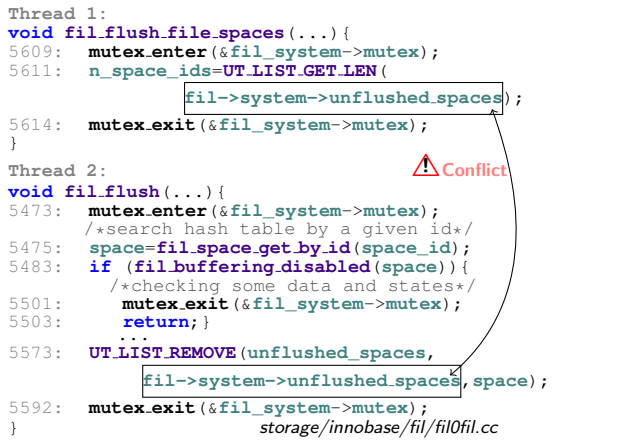


Figure 1. An example of the potential parallelism serialized by the unnecessary lock contention from mysql in the dynamic execution

Figure 1 is a real example from mysql−5.6.11 [11]. We depict how the unnecessary lock contention occurs in the dynamic execution. Both threads use the same shared lock `fil_system->mutex` to coordinate the shared access to `fil->system->unflushed->spaces`. However, in the dynamic execution, the thread always does not update it, if the buffer is explicitly disabled by the user (i.e., `fil_buffering_disabled(space)=TRUE`). In this case, two threads do not conflict, and the lock unnecessarily serializes the function `UT_LIST_GET_LEN` and the function `fil_space_get_by_id`, thereby leading to the performance degradation. In practice, we need to identify and generalize *Unnecessary Lock Contention Pair* (ULCP). A ULCP consists of two critical sections which are protected by the same lock and access the *parallelizable* code regions.

* Corresponding Author

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00

Due to the significant overhead of ULCPs at runtime, a volume of runtime research [22–24] attempts to eliminate the performance impact of ULCPs by speculatively executing critical sections without actually acquiring the lock. The lock is taken only when a data conflict needs to be resolved. The major advantage of those approaches is that they are transparent to programmers. However, they incur many problems in practice [1, 28] and there is still a long way before their practical and wide adoptions. First, they are prone to trigger false aborts due to the hardware limitations [28]. Second, a few transaction aborts (including false aborts) may lead to a large number of rollbacks [1].

Instead of relying on complicated dynamic approaches, this paper argues that the programmer should play a proactive role in eliminating the overhead of ULCPs. If the programmer can fix the performance problem caused by ULCPs, the side-effect problems of existing ULCP tools [3, 10, 22–24] can be avoided. We perform five real-world programs and PARSEC benchmarks to study the explicit characteristics of ULCPs. Based on our observations, we get an important finding: the root cause of many ULCPs lies in the problematic synchronization implementation. Thus, ULCPs can be fixed by programmers. It is necessary to detect them and further assist the programmers to understand and correct them, rather than take tolerant attitudes in the previous work. However, it is a nontrivial task to identify the source of ULCPs as well as figure out their performance impact. In fact, in a multi-threaded program, there may be so many ULCPs that it is difficult, or even impossible, to check all the ULCPs manually. Even worse, they are interwined with each other in the source code.

To help the programmer address the ULCP problems, this paper presents a performance debugging framework (namely PERFPLAY) to *understand* the performance impact of ULCPs in the lock-based programs. The core idea of PERFPLAY is based on record/replay. Under this framework, the ULCP analysis is performed as the following steps. PERFPLAY first records the program execution into a trace. Through analyzing the original trace, PERFPLAY can identify all ULCPs in the original execution. Then we propose a novel technique of trace transformation formalized by four rules to transform these ULCPs in the original trace into the correct pattern as a new trace free from ULCPs. We ensure that the new ULCP-free trace can be executed with the correct program semantics. By replaying both the original trace and ULCP-free one, PERFPLAY gets the performance impact of each ULCP. Finally we group the ULCPs generated by the same code regions and summarize the overall performance per code-site. We can recommend the programmer to fix the identified code region with the highest performance impact.

Our experimental results demonstrate the performance fidelity (including performance stability and precision) and the efficiency ($< 4.3\%$ lockset overhead) of PERFPLAY. With the most beneficial code regions recommended by PERFPLAY, our case studies verify the effectiveness of PERFPLAY in identifying the performance critical ULCPs.

Applications	LOC	Size	# Locks	# ULCPs			
				NL.	RR.	DW.	Bengin
openldap	392K	6M	1,851	75	1,414	473	15
mysql	1,132K	22M	2,109	125	9,822	2,924	194
pbzip2	5K	1M	1,281	2	1047	838	51
transmissionBT	79K	4M	352	15	111	123	29
handbrake	1,070K	3M	18,316	10	1,536	1,143	189
blackscholes	812	204K	0	0	0	0	0
bodytrack	10K	9.0M	32,642	0	1,322	321	43
canneal	4K	628K	34	0	0	0	0
dedup	3.6K	156K	19,352	231	2,421	1,952	164
facesim	29K	4.8K	14,541	102	871	819	12
ferret	9.7K	316K	6,231	11	101	231	343
fluidanimate	1.4K	72K	82,142	2	10,501	6,694	197
streamcluster	1.3K	44K	191	0	0	0	0
swaptions	1.5K	152K	23	0	0	0	0
vips	3.2K	17M	33,586	142	4,512	1,142	26
x264	40.3K	2.4M	16,767	941	3,841	412	84

Table 1. Breakdown of ULCPs in real-world programs and PARSEC benchmarks. $Size$ is denoted to the code size of programs, $\#Locks$ the number of lock protections generated in dynamic execution. $NL.$ refers to the null-locks, $RR.$ the read-read pattern, $DW.$ the pattern of disjoint-write.

The rest of this paper proceeds as follows. We provide the introduction on ULCP, the motivation and overview of our work in Section 2. Section 3 elaborates how to transform a recorded program execution trace into a new ULCP-free trace. Section 4 describes how to assess the ULCP performance impact from two replayed results. Section 5 further presents the implementation details. Section 6 presents the experimental results. We review the related work in Section 7 and Section 8 concludes the work.

2. ULCP: The Classification, A Brief Study, and Motivation

We start with a motivation study on ULCPs. We have observed that ULCPs are a very common problem in many multi-threaded programs. Next, we give another concrete example to show the performance impact of ULCPs. Motivated by the study and examples, we develop a debugging framework to address the ULCP problem.

2.1 A Motivation Study

We have surveyed the number of each category of ULCPs in five real-world programs (including two server applications– *openldap* [16], *mysql* [11]; three desktop applications– *pbzip2* [19], *transmissionBT* [26] and *handBrake* [8]) and PARSEC benchmarks [17]. The detailed experimental setup can be found in Section 6.

Table 1 lists the quantitative distribution of ULCPs of all applications with two threads. In our study, we consider the ULCP in the format of pairs, because pairs are the most basic representation and can be used to represent other complex combinations beyond pairs. For instance, three sequential critical sections can be encoded as two pairs. Our study has observed the following four major kinds of ULCPs.

(1) *Null-Lock* refers to the synchronization pair where there exists no shared-memory access in the critical sections. ULCP problems of this type are usually relatively easy to understand and identify. Null-locks usually come from *if-branch* of the program [21].

(2) *Read-Read* pattern indicates that only read operations on shared data exist between two critical sections protected by the same lock. The performance problem of this type mainly stems from the serial access to the shared data, especially for memory-intensive applications. Figure 2 demonstrates such a ULCP problem from OpenLDAP [16].

(3) *Disjoint-Write* pattern occurs in the scenario where two critical sections protected by the same lock update different shared addresses, and at least one of them is the write operation. One common example of disjoint-write is that program uses the uniform reference (e.g., pointer alias) protected by the same lock to update different shared objects.

(4) *Benign* pattern represents the benign feature of some *false* conflicting ULCPs. Specifically, two critical sections indeed access the same shared data concurrently but they do not constitute a conflicting pair, such as redundant writes, disjoint bit operation, and ad-hoc synchronization [4, 13].

According to Table 1, we find that ULCPs are pervasive. Meanwhile, different applications generally show different characteristics of ULCPs. Moreover, if we increase the number of threads in the application, the number of ULCPs increases dramatically. This phenomenon emerges due to the reason that the occurrence of ULCPs, in most cases, is interconnected rather than isolated. The ULCP interconnection can be embodied in the fact that they are produced by some common codes that will be repeatedly executed in most threads.

The four classified categories of ULCPs facilitate the achievement of two goals: 1) ULCP identification: different patterns may involve different detection techniques; and 2) ULCP transformation (i.e., trace-level ULCP elimination): after ULCP identification, we need to transform the trace into a ULCP-free execution, but different patterns may require different transformation strategies.

2.2 Another Motivating Example

Figure 2 depicts a source code snippet protected by the lock `dbmp->mutex` from OpenLDAP, a lightweight directory access protocol server [16]. This piece of code may affect the CPU utilization of system when a large number of threads call this code simultaneously. That is because it produces a large number of lock/unlock pairs (i.e., critical sections, CSs) where no effective execution statement exists if `dbmfp->ref` is always **FALSE**. In fact, these shared reads can be operated simultaneously unless `dbmfp->ref` is set to **TRUE**. Figure 2(a) illustrates many ULCPs (i.e., a two-tuple consisting of two critical sections $\langle CS, CS \rangle$), such as $\langle CS_1, CS_2 \rangle$ and $\langle CS_2, CS_3 \rangle$. Each ULCP introduces subtle performance impact due to the lock protection serializing two critical sections. We can further group ULCPs based on their code-site, which introduces a profitable accumulated performance gain. For instance, $\langle CS_1, CS_2 \rangle$ and $\langle CS_2, CS_3 \rangle$ are both generated by the pair of above-depicted source code, therefore their performance benefits should be accumulated up when we evaluate the ULCP performance impact per code-site.

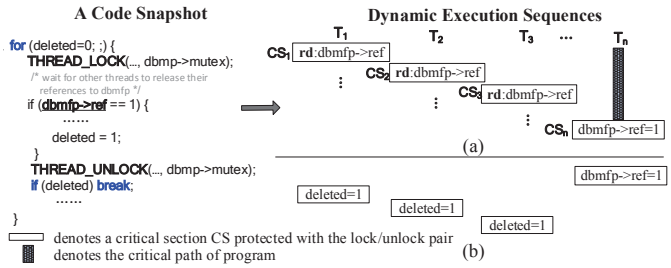


Figure 2. A code snippet with problematic synchronization implementation from OpenLDAP and its possible dynamic execution sequences when many threads call this code simultaneously. (a) A great deal of CPU time is wasted due to the spin-waits of threads $T_0, \cdots, T_{n-1}$ for the release of `dbmfp->ref` if the critical thread T_n runs slowly. (b) Little CPU time is wasted if T_n is finished fast.

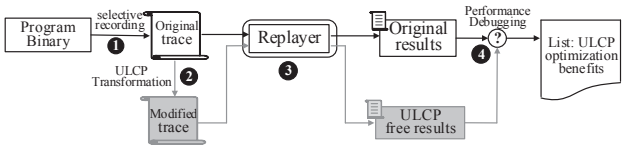


Figure 3. Overview of PERFPLAY

Lock Elision (LE) [22] is a technique that dynamically eliminates the inter-thread ULCP dependencies. Previous studies based on LE [1, 22–24] resolve ULCPs at runtime, which do not offer debugging information to programmers. For the example in Figure 2, they remove the lock acquisition and release operations of the critical sections (i.e., $CS_1, \cdots, CS_{n-1}$) completely before CS_n is executed. As a result, $CS_1, \cdots, CS_{n-1}$ are performed in parallel. LE cannot precisely track the impact of system resource waste for ULCPs. In fact, the programmer is able to fix them. The root cause of the problem in this example can be attributed to the imperfect synchronization implementation (according to our categorization in Section 2.1). To understand and fix this problem, it is necessary to detect the code regions producing ULCPs for the programmers and help them further understand and correct them. In fact, this source snapshot performs the same function as `pthread_mutex_barrier` primitive. Programmers can use barrier primitive to fix the problem and obtain better CPU utilization.

2.3 Overview of Our Approach

From the aforementioned study and real-world example, we can at least get two important implications:

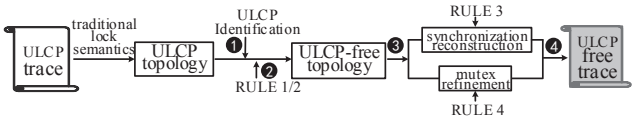


Figure 4. The process of ULCP transformation

- ULCP is a diverse program behavior. It is ubiquitous in the multi-threaded program and scattered in the program execution;
- It is difficult, or even impossible, to manually figure out which code-site incurs the highest performance impact due to ULCPs.

Therefore, a performance debugging tool is needed to assist the programmer in addressing the problem of ULCPs in their code. Particularly, we propose PERFPLAY, a replay framework to help programmers understand ULCPs in two aspects. First, the replay system records the program execution into a trace, based on which we therefore can know the explicit characteristic of each ULCP and further group them according to their code-site. Second, the replay system provides the possibility of reproducing the program execution, so that we can assess the performance impact of ULCPs for the performance comparison before and after optimization to further determine the most beneficial ULCP to fix.

Figure 3 depicts the overview of PERFPLAY. PERFPLAY operates on application binaries, and reports a list of the potential optimization benefits. This list is used to assist programmers to understand the ULCP performance problems. The first step of PERFPLAY is to record the intervals of a program execution trace. After the generation of original recording trace, the second step of PERFPLAY is to transform the original trace with ULCPs into a new trace without ULCPs. Next, PERFPLAY replays the original trace and the modified one. By comparing these two replayed results, PERFPLAY finally evaluates the potential performance impact of the aggregated ULCPs per code-site.

Using record/reply as the key technique, we have addressed the following two major challenges. First, the ULCP transformation may change the synchronization structure of program, thus possibly incurring the incorrect program semantics. There lacks a mechanisms in record/replay to ensure program correctness. PERFPLAY develops novel rule-based trace transformation techniques to preserve program semantics (Section 3). Second, we assess the performance of a ULCP, and further determine the code-site which produces the highest performance impact due to ULCPs. A new performance model is further proposed to tackle this problem. (Section 4).

3. ULCP Transformation

This section presents the detailed procedure of transforming the original trace with ULCPs into a new trace without ULCPs. The ULCP transformation may involve a change of the synchronization structure, thus making it a major threat to the program semantics. To cope with this problem, we propose a novel technique of trace transformation. We model the trace transformation problem into the graph analysis by means of topological graph theory [5]. Since topological graph theory has been studied for decades, the ULCP problem can be solved easily by analyzing the graph.

Algorithm 1: ULCP Identification

Input : $\langle C_1, C_2 \rangle$, two critical sections in the sequential order;
Output: A type, indicating the ULCP type between C_1 and C_2

```
if C1.Srd = ∅ and C1.Swr = ∅ or C2.Srd = ∅ and C2.Swr = ∅ then
    return NULL_LOCK;
else if C1.Swr = ∅ and C2.Swr = ∅ then
    return READ_READ;
else if C1.Srd ∩ C2.Swr = ∅ and C1.Swr ∩ C2.Srd = ∅ and
  C1.Swr ∩ C2.Swr = ∅ then
    return DISJOINT_WRITE;
else
    return FALSE;
```

The basic idea is as follows. We first build a topological graph which contains the original ULCP problems. Through some technical graph analyses, we then can easily identify the ULCPs and further eliminate them based on this graph as a new topological graph exclusive of ULCPs. As the topological graph can not be recognized to perform a program execution by computers, it is necessary to re-construct the ULCP-free program structure the new topological graph represents so that the computer can perform the new ULCP-free program execution. Figure 4 depicts the detailed process of our trace transformation. It is a rule-based approach. Based on the four rules proposed, the new ULCP-free trace is performed with the correct program semantics in most cases. If not, it would report the data races. Next, we present the details of each step in the trace transformation. To facilitate the description, we make the definitions:

- ***Causal-order topology*:** a topological graph of the cause and effect of an execution trace. If there is no special instruction, the causal-order topology can be also referred to as topology for short.
- ***Node*:** a critical section in the topology.
- ***Causal-edge*:** a specific causality between two nodes.

3.1 Building ULCP-free Topology

Following the traditional lock dependencies, we first build the causal-order topology of original execution (*abbr.* original topology). The original topology involves many causal-edges caused by ULCPs. Thus we then transform original ULCP topology into a new topology which does not contain causal-edges caused by ULCPs (*abbr.* ULCP-free topology).

Prior to building ULCP-free topology, we need to identify ULCPs. We use shadow memory [14] to store the state information about critical section. Shadow memory state refers to the information about each critical section *C* of the running program, which mainly consists of two sets:

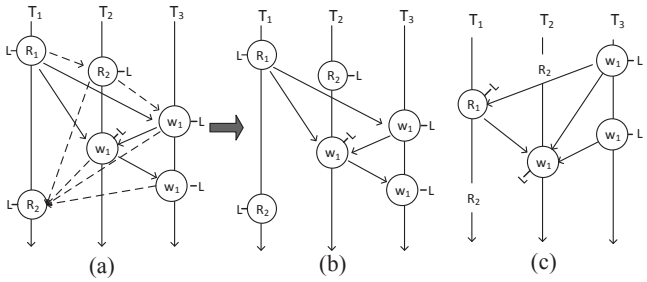


Figure 5. The causal dependencies for an example. ◯ represents the critical section, while L attached to ◯ means this critical section is protected by lock L. R_1 indicates a read on shared data 1 and the dotted arrow shows a ULCP.

- $C.S_{rd}$. a set of all shared reads in the critical section C.
- $C.S_{wr}$. a set of all shared writes in the critical section C.

We identify ULCPs in different categories. As shown in Algorithm 1, null-lock, read-read, and disjoint-write can be easily identified by intersecting the read-write sets of critical sections as line 1, 3, 5 indicate. But both benign ULCPs and true lock contention pairs (TLCPs) involve the conflicting access. In this case, Algorithm 1 does not work. To further distinguish the false conflict of benign ULCPs from the real conflict of TLCPs, we extend the reversed replay execution [13] for the distinction between benign ULCPs and TLCPs by additionally replaying the execution trace with a reversed order of two critical section for a given ULCP. If the two replays produce the same result, then this ULCP can be classified as a benign pattern.

In the original topology, we know the timing relationship with respect to all critical sections in the original execution. For a certain critical section CS, in order to search another CS' in other threads, which comprises the TLCP with CS, we define the operations:

- ***Sequential searching*** refers to searching such CS' in a given thread in the order from the timing index of CS to largest timing index of that thread.
- If we find such a CS' in a given thread, it is called ***matched***.

Afterwards, we define the first rule to facilitate the building of ULCP-free topology from an original ULCP topology.

RULE 1. *A causal edge is established only when the* current *critical section and its first matched critical section in every other thread constitute a TLCP during the sequential searching.*

Figure 5(a) depicts an example of the building process of the ULCP free causal-order topology. To begin with, we denote the critical section R_1 in thread T_1 as the *current* critical section. Then it is matched with R_2 in T_2. R_1 and R_2 consist of a Read-Read ULCP. We use the dotted arrow to denote the non-causal edge relation between them. R_1 in T_1 is successively matched with W_1 in T_2, in which case there establishes a causal edge between them due to the TLCP relation, denoted as the solid arrow. When the first causal edge with W_1 in T_2 for T_2 is established, R_1 in T_1 starts to do the similar traverse in T_2, establishing another causal edge with the first W_1 in T_3. After the first round of causal edge building, R_2 in T_2, subsequent to R_1 in T_1, becomes new *current* critical section, and repeats the previous procedure.

Figure 5(b) illustrates the ULCP-free topology built according to Rule 1. Following the program semantics of ULCP-free topology in Figure 5(b), we may get the program execution as shown in Figure 5(c) which affects the performance fidelity for the multiple replays (detailed discussion about this will be presented in Section 5). In order to observe the *stable* performance impact of ULCPs, we then put forward Rule 2.

RULE 2. *All causal-edge nodes protected by the same lock in the ULCP free topology are guaranteed with the same partial order as the original topology.*

In the original topology, the partial order of the nodes R_1 in T_1, W_1 in T_2 and two W_1 in T_3 in Figure 5(a) is $\{R_1(T_1) \prec W_1^{1st}(T_3) \prec W_1(T_2) \prec W_1^{2nd}(T_3)\}$. According to Rule 2, the nodes R_1 in T_1, W_1 in T_2 and two W_1 in T_3 of ULCP-free topology in Figure 5(b) should be restricted to the same partial order with the original topology as $\{R_1(T_1) \prec W_1^{1st}(T_3) \prec W_1(T_2) \prec W_1^{2nd}(T_3)\}$.

In summary, we apply RULE 1 and 2 to build the ULCP-free topology, which will be refined by RULE 3 and 4.

3.2 Re-establishing the Program Structure of the ULCP-free Topology

We eliminate the false inter-thread dependencies caused by different categories of ULCPs. First, in absence of conflict with any critical section, PERFPLAY removes lock/unlock events of all null-locks and all standalone nodes in the topology, such as R_2 in T_1 and R_2 in T_2 as shown in Figure 6(a). Second, to ensure true inter-thread dependencies between two critical sections, we use lockset [25] to protect the critical sections in the topology. Lockset is a software component comprising multiple locks, which is generally used as a fine-grained lock synchronization. Consequently, PERFPLAY uses many distinct auxiliary synchronization locks instead of the original locks to reconstruct the ULCP-free causal dependencies. It should be noted that all these auxiliary synchronization locks provided by PERFPLAY are written with a prefix @L for the sake of the discrimination from the original one.

Now, the question is how to assign these ad-hoc locks onto each node in the ULCP-free topology while ensuring the program correctness. We perform the re-synchronization procedure as RULE 3 describes.

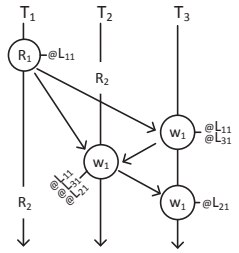


Figure 6. The re-synchronization of the ULCP free causal dependencies. @L indicates auxiliary locks.

RULE 3. *Each node with the outdegree in the topology will be given a new auxiliary lock. While each node with the indegree should be synchronized by the given lock of its source node.*

Figure 6 shows the outcome of the example in Figure 5 according to RULE 3. According to RULE 3, the nodes with outdegrees, namely R_1 in T_1, W_1 in T_2 and W_1 in T_3, are given with new auxiliary $@L_{11}$, $@L_{21}$ and $@L_{31}$, respectively. While the node with the example of W_1 in T_3 has the indegree from the given source node R_1 in T_1, thus it needs to be synchronized with the additional lock of the source node R_1 in T_1, i.e., $@L_{11}$. Ultimately, W_1 in thread T_3 has the lock-set LS=$\{@L_{11}, @L_{31}\}$. Each critical section will maintain a lock-set. We further refine the mutex relation for the ULCP-free trace execution. Therefore a new mutex relationship can be described as follow:

RULE 4. *Two critical sections are mutually-exclusive if the intersection of their lockset LS is empty-set.*

Theorem 1 gives the correctness of our transformation. The detailed proof can be found in our technical report [29].

Theorem 1 (Correctness). *The transformed ULCP free trace is performed with a guarantee of either the program correctness or reporting the data races.*

One implementation detail is worthy of being further discussed. After applying RULE 3, a node in the topology may suffer from the overhead of maintaining the large-scale lock-sets. For instance, the lockset of the critical section C in Figure 7(a) is $\bigcup_{i=0}^{N} L_{1i}$. To reduce runtime overhead of the large lock-sets, we propose a dynamic locking strategy, as shown in Figure 7, the main idea of which is that the synchronization of the targeted node C depends on the runtime state (i.e., `END`) of each source node $C_1, \cdots, C_N$. For instance, if C_1.`END`=`TRUE`, it means that the critical section C_1 is already finished. If the node C_1 is finished before the execution of the node C at runtime, the lockset LS of the node C can exclude the lock of one of its source nodes C_1, i.e., L_{11}. Based on the dynamic locking strategy, PERFPLAY saves much overhead for the maintenance of the locksets and is able to deal with any thread interleaving as shown in Figure 7(b).

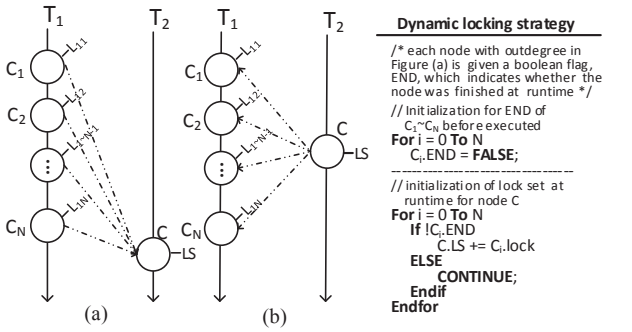


Figure 7. Dynamic locking strategy

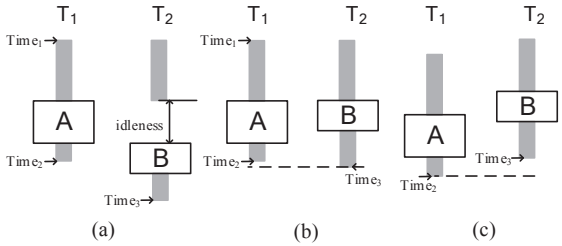


Figure 8. Two different performance measurements

4. ULCP Performance Debugging

After the phase of ULCP transformation, we obtain a set of ULCPs. For an effective debugging framework, we still face one major problem. There may be many ULCPs, and some of them are even from the same code-site. An effective debugging tool should point out the succinct code-site for distinctive ULCPs, and also locate the most performance critical ULCP for programmers. Thus, we propose ULCP fusion and performance accumulation based on their code regions in the source code level (Section 4.1), and point out the most performance critical ULCP to programmers (Section 4.2).

4.1 ULCP Fusions

We model the potential runtime overhead of a ULCP. Figure 8 illustrates a detailed diagrammatic representation of the performance metrics, where A and B constitute a ULCP. We label the start point of precursor segment of the first critical section A using $Time_1$; the end point of successor segment of A using $Time_2$; the end point of successor segment of the second critical section B using $Time_3$. When the ULCP free trace is executed, the replayed program may perform the traces in two possible ways, as shown in Figure 8(b) and Figure 8(c). We consider both cases: for case (b), the improved performance of ULCP is $\Delta Time_3 - \Delta Time_1$; for case (c), the result is $\Delta Time_2 - \Delta Time_1$. Consequently, we define the performance improvement of each ULCP as follows.

$$\Delta T_{ULCP} = \Delta MAX\{Time_2, Time_3\} - \Delta Time_1 \quad (1)$$

Algorithm 2: ULCP Fusion and Performance Accumulation

Input : $\langle \mathsf{ULCP}_1, \mathsf{ULCP}_2 \rangle$, two standalone ULCPs;
Output: $\mathsf{ULCP}_{\mathsf{new}}$, a new synthetic ULCP;
NULL, two standalone ULCPs that can not be merged

```
/* Handle the same code regions or nested locks */
if ULCP1.CR1 ⊓ ULCP2.CR1 ≠ ∅ and ULCP1.CR2 ⊓ ULCP2.CR2 ≠ ∅ then
    ULCPnew.CR1 ← ULCP1.CR1 ⊔ ULCP2.CR1;
    ULCPnew.CR2 ← ULCP1.CR2 ⊔ ULCP2.CR2;
    ΔTULCPnew ← ΔTULCP1+ΔTULCP2;
else if ULCP1.CR1 ⊓ ULCP2.CR2 ≠ ∅ and ULCP1.CR2 ⊓ ULCP2.CR1 ≠ ∅ then
    ULCPnew.CR1 ← ULCP1.CR1 ⊔ ULCP2.CR2;
    ULCPnew.CR2 ← ULCP1.CR2 ⊔ ULCP2.CR1;
    ΔTULCPnew ← ΔTULCP1+ΔTULCP2;
else
    ULCPnew ← NULL;
```

where $Time_{label}$ indicates the current timestamp of application when the program is executed at the location of $label$, and MAX is denoted as the maximum value. Δ is denoted as an operation that calculates D-value (difference value) before and after the optimization.

After the process of Algorithm 1, PERFPLAY collects a large number of ULCPs, denoted as $\{ULCP_1, ULCP_2, \cdots, ULCP_n\}$, each consisting of two critical sections $\langle C_1, C_2 \rangle$. To facilitate the description, we define the operator $\cdot$ to obtain the attribute or component of a ULCP, such as $ULCP_1.C_1$. However, some ULCPs are possibly caused by the same code region (CR). Thus, we propose ULCP fusion to merge two ULCPs into the unique ULCP per code region in the source code level. Then, we can report the accumulated performance impact of ULCPs at the CR level to the programmers. Particularly, we accumulate up the performance improvement of ULCPs generated by the same code regions according to Algorithm 2. In Algorithm 2, $\langle CR_1, CR_2 \rangle$ is denoted as the code regions incurring two critical sections $\langle C_1, C_2 \rangle$ of a ULCP. The binary operator $\sqcup$ means whether two CRs involve the shared region of the code; while $\sqcap$ indicates the conflated code region of two CRs. Through Algorithm 2, the final state of ULCP group is that any two ULCPs can not be fused further.

4.2 ULCP Recommendations

After ULCP fusion and performance accumulation by Algorithm 2, we obtain a group of unique ULCPs, denoted as $\{ULCP_1, ULCP_2, \cdots, ULCP_m\}$, and its corresponding performance improvement $\{\Delta T_{ULCP_1}, \Delta T_{ULCP_2}, \cdots, \Delta T_{ULCP_m}\}$. For the effectiveness of a debugging tool, it is desirable to prioritize the most beneficial ULCPs to programmers, since there may be many ULCPs in the program. We denote P as the relatively optimizable value of a ULCP among the total ULCP group:

$$P = \frac{\Delta T_{ULCP}}{\sum_{j=1}^{m} \Delta T_{ULCP_j}} \quad (2)$$

which refers to the relative optimization opportunity of a corresponding ULCP. Each ULCP in $\{ULCP_1, ULCP_2, \cdots, ULCP_m\}$ has its own P, and $\sum_{i=1}^{m} ULCP_i.P = 1$. To further ascertain the most beneficial ULCPs, we resort $\{ULCP_1, ULCP_2, \cdots, ULCP_m\}$ by P in a descending order, i.e., $\forall i > j, ULCP_i.P < ULCP_j.P$. Then we can pinpoint the most performance critical code regions from that order list. Thus, our tool can recommend the most performance critical ULCP as $ULCP_1$.

5. Implementation Issues

We implement PERFPLAY for the parallel replay based on Pin [20], an underlying framework that enables programmers to perform the program analysis at runtime without source codes. Particularly, we remove and insert the auxiliary locks in the trace level, instead of modifying the compiler or binary. Modifying trace with the lock mechanisms can provide an easy implementation for that objective, which has the same effect to provide useful debugging hints for ULCPs as modifying the binary or compiler.

In the following, we briefly discuss two implementation details: i) what information should be recorded for the performance analysis using replay technique in the recording phase; ii) how to perform the faithful replay for each run upon the given trace so that the performance impact of examined problems can be evaluated precisely in the replay phase. Due to the space limitation, we refer the readers for more implementation details in our technical report [29].

What and How to Record: To evaluate the performance impact of ULCP, we should record all information of ULCP to perform its performance. Consequently, it is necessary to record all instructions and memory accesses between the lock and unlock operations. Moreover, if a certain critical section is invoked N times, each critical region execution in the trace is recorded N times and then will be executed N times when the trace is replayed.

For other events, the recording strategy of PERFPLAY is quite flexible, ranging from complete recording to selective recording. Thus, PERFPLAY chooses selective recording whenever appropriate. Specifically, for the non-mutual exclusive semaphore, PERFPLAY only ensures the correctness of the partial order in the sense that it is the same as the original ordering.

Performance Fidelity: There has been significant amount of work on building record/replay systems [12, 13, 18, 27] for understanding the correctness of bugs in programs, but not much effort has gone into leveraging them to study performance issues. Based upon a given trace, the determined information contains: the path branches each thread performs, synchronization operations, and the instructions or events performed by each thread. Therefore, suppose we perform the same trace twice, performance fluctuation of the program largely depends on the lock synchronization interleaving. As shown in Figure 9, if two critical sections

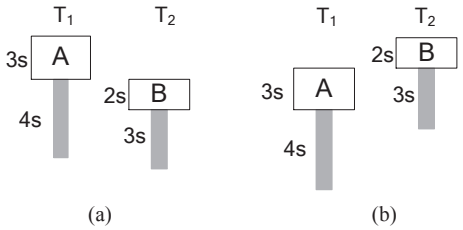


Figure 9. An example of the lock mutual exclusion for the performance fluctuation with different sequences, where the digits indicate the time cost of the program segment. (a) If A precedes B, the program costs 8s; (b) If B precedes A, the program costs 9s.

coequally contend for the lock resource, the program may perform different performance due to the potential different time cost of subsequent program segments.

To enable performance analysis using replay technique (*abbr*. performance replay) for the parallel execution, we propose an enforced locking serialization constraint (ELSC) which enforces the total order of the dynamic lock synchronizations for the replayed trace according to the schedule order of these locks at runtime. That is, ELSC schedules the same lock order as the scheduled order of these locks when the program runs at runtime. As shown in Figure 9, if the program runs as Figure 9(a) shows when the trace is being recorded, ELSC sets down this order of $A \rightarrow B$ in the recording phase and then enforces ALL subsequent replays for this trace with *hard* ordering of A happening before B in the replay phase. ELSC ensures the performance fidelity of replay execution for the multiple replays based upon the same given trace. We have formally proved the property of performance fidelity, and further compared the differences between PERFPLAY and some other previous work [15, 18] in our technical report [29].

6. Evaluation

6.1 Experimental Setup

System configuration: All experiments are performed on a machine with two Intel quadcore Xeon E5310 1.60Ghz processors, 8GB memory, one 250GB SATA hard disk, and 1Gbit Ethernet interface. The running operating system is CentOS 5.6 (X86_64) with Linux kernel 3.0.0-12.

Benchmark test configuration: We evaluate PERFPLAY with five real-world applications and PARSEC benchmarks (used in Section 2.1). The detailed setup of individual applications are presented as follows.

1) *openldap*: a lightweight directory access protocol server. In our test, we use the default thread pool mode for *openldap* server, and use the professional tool DirectoryMark by MindCraft[1] to benchmark it with the option of searching 2000 entries.

[1] http://www.mindcraft.com/directorymark/

2) *mysql*: an open source database system which is widely-used in the world. We use the test tool mysqlslap released in *mysql* software package to test *mysql* with 1000 queries, and 2 iterations.

3) *pbzip2*: a parallel implementation of the bzip2 compressor. We test the benchmark by compressing a 256M file with the option of two processors.

4) *transmissionBT*: a BitTorrent client. We only perform its download function by downloading a local 300M file.

5) *handBrake*: a video transcoder. We test the benchmark by conversing a 256M DVD format file into MP4 format with the options of H.264 codec and 30 FPS.

6) *PARSEC Benchmarks*: a benchmark suite with 12 multi-threaded programs. We test all PARSEC benchmarks (except *freqmine*) with *simlarge* input. PERFPLAY is implemented currently based on pthread library. As *freqmine* benchmark is an openMP program, PERFPLAY can not identify its synchronization.

Methodology: To demonstrate the performance fidelity of PERFPLAY, we perform the replay execution with the following four schemes:

1. Memory-based schedule (MEM-S) [18], which enforces a deterministic execution sequence of all shared memory accesses.

2. Synchronization-based schedule (SYNC-S) [15], which enforces the total order of the lock synchronizations for the same input.

3. ELSC-based schedule (ELSC-S), which enforces the total order of the lock synchronization for the same schedule.

4. Parallel replay for the original execution without any enforcement strategy for the events (ORIG-S).

We focus on the key part of dynamic executions in the trace replay. In our implementation, we have decoupled the replayed execution time of program from the other time-consuming manipulations, such as the loading of trace from disk into memory, and the format transformation of trace from the string-style into the instruction-style.

6.2 Performance Fidelity of PERFPLAY

To evaluate performance fidelity, two aspects require to be assessed, including performance stability and performance precision. Stability represents whether PERFPLAY shows the same performance across the multiple replays with the same trace. The precision means whether PERFPLAY strictly adheres to the original execution. If our debugging framework has a high precision, we can determine that the performance improvement of ULCP-free replayed execution comes entirely from the optimization of ULCPs.

We record all PARSEC benchmarks with *simlarge* input, and we replay the trace of each application ten times using different replay schemes (i.e., MEM-S, SYNC-S, ELSC-S, and ORIG-S). Figure 10 shows the final replayed execution time using these schemes. From the small error bars, we can see that MEM-S, SYNC-S, and ELSC-S all enforce the deterministic program execution for the multiple

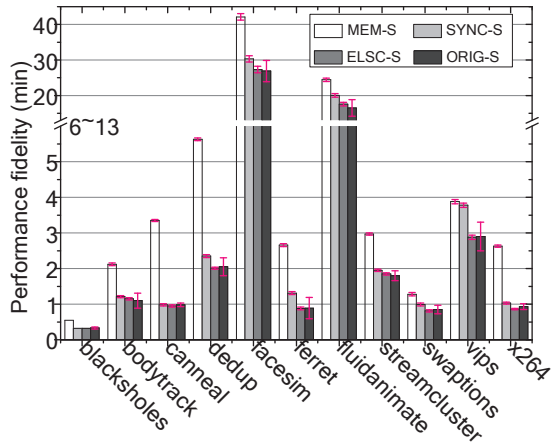


Figure 10. Performance fidelity comparison between different execution schemes for the replay

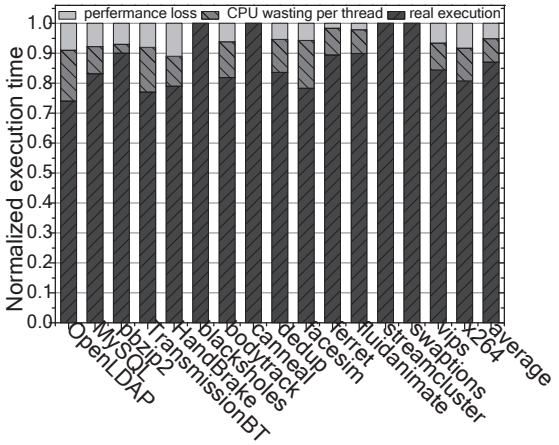


Figure 11. The normalized execution time through replaying the traces with and without ULCPs

times, thus providing the *stable* performance analysis. Nevertheless, ORIG-S shows the indeterminate (i.e., large error bars) program execution due to the inter-thread lock interleaving. Except the nature of enforcement scheme itself, both MEM-S and SYNC-S manifest themselves with the additional performance introduction compared with ORIG-S. While ELSC-S eliminates the waiting time of SYNC-S for lock acquisition by only enforcing the synchronization order based on the scheduled synchronization order for the same schedule. As a result, we can see that ELSC-S almost produces the same program performance with ORIG-S. This yields the conclusion that PERFPLAY with ELSC scheme strictly schedules the replay execution as the original scheduled execution without introducing any additional performance overhead, thus providing the *precise* performance analysis. From the above-discussed results, it is revealed that only ELSC-S provides both the performance stability and performance precision, thus ensuring the performance fidelity of replay execution.

6.3 Performance Impact Evaluation of ULCPs

Performance impact of ULCPs in this work includes:

- **Performance degradation** (T_{pd}): The performance improvement of program before and after the optimization;
- **Resource wasting** (T_{rw}): In our test, resource wasting mainly refers to the wasting of CPU resource, which makes the useless ULCP computation (e.g., spin-lock) on the non-critical path.

where T_{pd} can be directly quantified by replaying ULCP trace (T_{ut}) and ULCP-free trace (T_{uft}), i.e., $T_{pd} = T_{ut} - T_{uft}$. With Equation 1, T_{rw} can be indirectly calculated as $\sum \Delta ULCP - T_{pd}$. To quantify them in the following experiments, we evaluate them with the metric of the normalized performance impact (i.e., T_{pd}/T_{real}) and CPU-time wasting per thread on average (T_{rw}/N_{thread}), respectively. All tests are executed with two threads.

Performance impact of ULCPs: Figure 11 illustrates the normalized performance impact and normalized CPU-time wasting of ULCPs from 5 real world programs and PARSEC benchmarks. In our tests, PERFPLAY produces different opportunities of performance impact for different applications. For example, *blacksholes*, *canneal*, *streamcluster*, and *swaptions* hardly obtain any performance impact due to the correct use of lock or exclusive use of lock. While for other applications, such as *openldqp*, *mysql*, *pbzip2*, the program has a significant percent for the improvement of performance (1.6%–11%) and CPU time per thread (1.1%–16.7%) due to the ULCPs. On average, the performance of these applications can be improved by 5.1% and the resource utilization per thread by 7.85%. Usually, a program with more ULCPs has larger performance improvement, which indicates the benefits of removing ULCPs by our performance debugging framework. One exception is that, *fluidanimate* has a larger number of ULCPs than *facesim*, but produces a lower speedup. That is because ULCPs in *facesim* have the larger-scale critical sections.

Performance gain from the most beneficial ULCPs: Table 2 reports the number of the exploited ULCP code regions and corresponding performance gain of the most beneficial one. Column *grouped ULCPs* counts total number of the unique ULCPs after the fusion and performance accumulation of ULCPs. Column $ULCP_1.P$ (discussed in Section 4.2) shows the relative optimization portion of the most beneficial ULCP code regions among the total ULCP group set. From Table 2, we find that different applications show different optimization opportunities. For instance, *openldap* has 18 grouped ULCP code regions while its most beneficial one takes up 30.1% of optimization gain among the total ULCP set. *mysql* produces a larger number (57), but the most beneficial one exhibits only 12.5% of performance benefit. The performance gain of the beneficial ULCPs for other applications is in Table 2.

Applications	Grouped ULCPs	$ULCP_1$.P
openldap	18	30.1%
mysql	57	12.5%
pbzip2	4	59.4%
transmissionBT	2	53.5%
handbrake	29	15.4%
blackscholes	0	0
bodytrack	5	20.9%
facesim	11	31.2%
fluidanimate	3	26.5%
swaptions	0	0

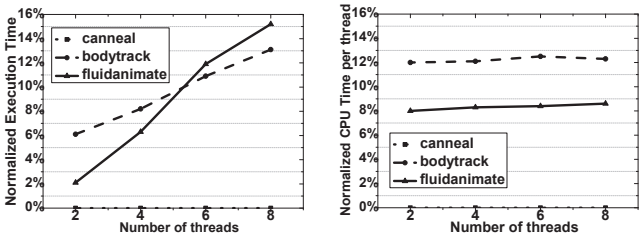
Table 2. # Grouped ULCP code regions and opportunity of the most beneficial one

Applications	w/o DSL	w/ DSL
blackscholes	0	0
bodytrack	5.3%	0.5%
canneal	0.2%	0.2%
dedup	4.6%	0.7%
facesim	7.8%	1.2%
ferret	10.7%	3.6%
fluidanimate	14.1%	4.3%
streamcluster	2.9%	0.6%
swaptions	0.4%	0.4%
vips	7.6%	2.4%
x264	5.0%	1.9%

Table 3. Runtime overhead of locksets with/without dynamic locking strategy

6.4 Overhead Reduction via Dynamic Locking Strategy

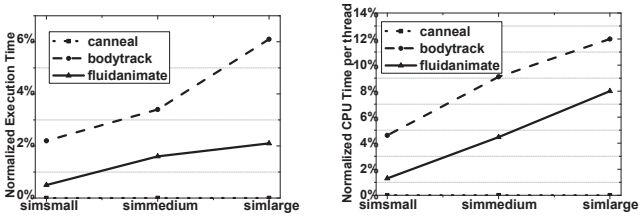
Lockset is introduced to transform ULCPs into the parallel pattern. However, it also introduces the significant overhead for the determination of mutex relationship by intersecting two locksets in RULE 4, especially for the lock intensive programs. To quantify lockset (LS) overhead, we replay PARSEC benchmarks with and without dynamic locking strategy (DLS), respectively. Table 3 compares runtime overhead of locksets with and without DLS. When not using DSL, lockset maintenance incurs significant $(0.2\% - 14.1\%)$ amount of runtime overhead. In contrast, lockset with DLS further reduces performance impact of lockset into a negligible level, only incurring 4.3% overhead even for the lock intensive application *fluidanimate* which makes extensive use of locks.



(a) The performance loss with the increasing number of threads

(b) The CPU wasting with the increasing number of threads

Figure 12. Impact with the increasing number of threads



(a) The performance loss with the varying input size

(b) The CPU wasting with the varying input size

Figure 13. ULCP impact with the varying input size

```
int Query_cache::try_lock(bool){
  mysql_mutex_lock(&structure_guard_mutex);
  while(1){
     set_timespec_nsec(waittime,(ulong)5000000L);
     int res=mysql_cond_timedwait(
            &COND_cache_status_changed,
            &structure_gurad_mutex,&waittime);
     if(res==EITMEOUT){
        ...
       break;
     }
  }
  mysql_mutex_unlock(&structure_guard_mutex);
}
```

Figure 14. A verified ULCP problem from mysql-5.6.11

```
void *consumer(void *q){
2109:  pthread_mutex_lock(&mu);
2122:  if(fifo->empty&&syncGetProducerDone()==1)
2124:       pthread_mutex_unlock(&mu);
   }
int syncGetProducerDone(){
 533:    int ret;
 534:    pthread_mutex_lock(&muDone);
 535:    ret=producerDone;
 536:    pthread_mutex_unlock(&muDone);
 537:    return ret;
 538: }
```

Figure 15. A ULCP problem from pbzip2

6.5 Sensitivity Study of ULCPs

To evaluate the evolution of ULCP impact, we study the ULCP sensitivity to the varying thread number and input size. We select *canneal*, *bodytrack*, *fluidanimate* from PARSEC benchmarks with the different numbers (i.e., a few, medium, large) of ULCPs.

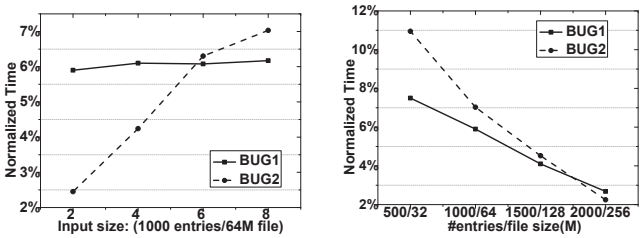
Figure 12 depicts the sensitivity of ULCPs to the thread number. We can find that ULCPs lead to the increasing performance loss as the number of threads increases while the resource wasting per thread stays the same. Figure 13 depicts the sensitivity of ULCPs to the input size. It can be observed that both performance loss and resource wasting increase as the input size increases. The explanation for both figures is: in those applications 1) all threads reuse the same code (e.g., functions) to perform the program execution; 2) more input sizes merely mean the number of executions on some code segments is increasing.

It should be noted that *canneal* still does not show any potential opportunity for both the increasing thread number and input size. Combining the results from *bodytrack* with *fluidanimate*, we seems to reveal that in most cases the ULCP code-sites are not affected by the thread numbers and input sizes of these applications. ULCPs can manifest themselves in two threads, and more thread numbers may only change their performance impact.

6.6 Case Study

To evaluate the effectiveness of PERFPLAY, we have checked some ULCP bugs that have already been verified by official bug system under PERFPLAY framework.

MySQL #68573. Figure 14 depicts the code snippet of this case from mysql version-5.6.11. The real designed intention of the programmers is that "a 50ms timeout for a SELECT statement waiting for the query cache lock is



(a) Case study with the varying number of threads

(b) Case study with the varying input size

Figure 16. Sensitivity study of #BUG 1 and #BUG 2

set. If the timeout expires, the statement executes without using the query cache"—mysql official documents. However, the ULCP performance problem "increases" this timeout threshold unwittingly when multiple threads invoke this code, thus severely degrading the efficiency of SELECT statement. Other examples in mysql include #37844, #60951 and #69276.

We also re-implement a few *easy-to-understand* ULCP cases found by PERFPLAY in a ULCP-free fashion, and further re-quantify its performance impact.

Resource wasting from openldap (#BUG 1). We re-implement the code snippet from OpenLDAP in Figure 2 with `pthread_mutex_barrier`, and re-quantify the CPU utilization of this ULCP problem by testing the program compared with the original code.

Performance degradation from pbzip2 (#BUG 2). Figure 15 depicts the simplified code of ULCP problem from the parallel compression utility pbzip2. It employs the producer-consumer idiom for the parallel compression: the producer produces the blocks by reading file and the multiple consumers consume (compress) these blocks in parallel. When the last file block is dequeued (i.e., `fifo->empty`=1 and `producerDone`=1), the program starts the end stage of thread join. In this case, the example above will incur many read-read ULCPs as follows:

```
lock(mu);
load(fifo->empty);
lock(muDone);load(producerDone);unlock(muDone);
unlock(mu);
```

The joins of all threads are serialized and extra nested lock overhead is added by this read-read ULCP, which causes the performance loss. We fix it via the signal/wait model: we take the producer, rather than the consumer, with the responsibility of checking the state of `fifo->empty` and `producerDone`. If both of them are `TRUE`, the producer will give a signal to inform all consumers of their safe exit without any check when their work is completed.

Results. Figure 16 depicts the sensitivity of two exploited ULCP bugs (i.e., #BUG 1 and #BUG 2). As the number of threads increases, #BUG 1 causes the stable resource wasting per thread while #BUG 2 has an increasing performance loss of program. Whereas, different from the illustration shown in Figure 13(b), the performance impact of both #BUG 1 and #BUG 2 presents a downward trend as the input sizes increases. That is because for a given thread number both #BUG 1 and #BUG 2 have the fixed execution frequency, which increases superior to the input size. Moreover, the increasing input size aggravates the workload of application, thus increasing the program execution time. As a result, the performance impact of both #BUG 1 and #BUG 2 is declining. Both above-depicted results verify that the real ULCPs can be exploited by PERFPLAY.

7. Related Work

Unnecessary Lock Contention. There has been significant amount of work on dynamically eliminating the performance impact of ULCPs. Lock Elision (LE) [22, 24] leverages the hardware assistance and the underlying cache coherence protocol to enable highly concurrent multi-threaded execution by dynamically removing unnecessary lock-induced serialization. The lock is acquired only when a data conflict occurs. However, LE-based work is still challenging in practice. For instance, a few transaction aborts may cause excessive rollbacks and serializations, which severely limits the exposed concurrency of ULCPs [1]. Meanwhile, it is prone to trigger false aborts due to the hardware limitations [28]. We believe that the most effective and efficient manner for ULCPs is that programmers can fix the problem in their code, rather than relying on dynamic tools which may lead to severe runtime overhead. Consequently, we propose a novel framework, PERFPLAY, to evaluate the performance impact of ULCPs and further assist the programmers to identify the most performance critical ULCP.

Performance Tools. It is hard for static exploration tools [2] to obtain the characteristics of ULCPs (e.g., their amounts, categories and the time they cost). Due to the dynamic nature of ULCPs, the major obstacle is that they may produce abundant false ULCPs due to the runtime behaviors of ULCPs. Another obstacle is that the code snippet with a lock-/unlock pair running simultaneously by multiple threads may unroll into two execution cases as ULCPs and TLCPs. Under different runtime (e.g., thread scheduling and input set), both ULCPs and TLCPs manifest themselves in different amounts and performance impact. As for the existing dynamic tools [6, 7], they also bear some limitations in the impact analysis of ULCPs. Still, the majority of them are devoted to performance measurement, but they are not applicable to the performance transformation and further performance comparison before and after optimization. As a result, they cannot be used directly for performance debugging, e.g., how much performance would be improved if the ULCPs were removed. PERFPLAY is the very performance tool to make an attempt to solve this problem.

Record/Replay System. Plentiful replay systems are proposed in the past several decades. For instance, deterministic replay systems [12, 18] reproduce the bug debugging by

enforcing the order of the execution events. Modified replay debugging [13, 27] distinguishes different categories of bugs by comparing the results of the original trace with the modified one. Overall, almost all of them are built for identifying and understanding the correctness of bugs in programs. but not much effort has gone into the study of performance issues. PERFPLAY first (to our best knowledge) has put effort into studying the performance bugs using replay technique.

8. Conclusion and Future Work

We propose a performance debugging framework, PERFPLAY, to evaluate the performance impact of unnecessary lock contention pairs (ULCPs) of multi-threaded applications using reply technique. We first record the multi-threaded program execution trace, based on which we can identify all ULCPs. Then PERFPLAY transforms the original ULCP trace into the new ULCP free one while ascertaining the correctness of program via novel transformation rules. Finally, PERFPLAY replays two traces. Based on two replayed results, we evaluate the potential performance improvement of each ULCP and then group all ULCPs into the unique ULCPs according to their code-site. Our experimental results on five real-world programs and PARSEC benchmarks demonstrate the performance fidelity and efficiency ($< 4.3\%$ lockset overhead) of PERFPLAY. With case studies, we demonstrate its effectiveness to identify the performance critical ULCP. It also shows that the majority of ULCPs can be resolved by taking the most critical code regions. As for future work, we are interested in making PERFPLAY as a pintool in the PIN framework, investigating input sensitivity to our debugging tool and also the applicability of our tool to many-core programs (such as GPU-based applications [9]).

Acknowledgments

This paper is supported by National Natural Science Foundation of China under grant No. 61272408, 61322210, National 973 Fundamental Basic Research Program under grant No. 2014CB340600, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048 and Hubei Funds for Distinguished Young Scientists under grant No. 2012FFA007. Bingsheng's work is partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore.

References

[1] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *PODC'14*.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE'11*.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *ISCA'13*.

[4] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI'10*.

[5] J. L. Gross and T. W. Tucker. *Topological Graph Theory*. Wiley-Interscience, New York, NY, USA, 1987.

[6] R. J. Hall. Call path profiling. In *ICSE'92*.

[7] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE'12*.

[8] Handbrake. http://handbrake.fr/.

[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *PACT'08*.

[10] Intel Corporation. Intel architecture instruction set extensions programming reference. 2013.

[11] MySQL. http://www.mysql.com/.

[12] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA'05*.

[13] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI'07*.

[14] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE'07*.

[15] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS'09*.

[16] OpenLDAP. http://www.openldap.org/.

[17] PARSEC. http://parsec.cs.princeton.edu/.

[18] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO'10*.

[19] pbzip2. http://compression.ca/pbzip2/.

[20] Pin Tool. http://www.pintool.org/.

[21] H. Qi, A. A. Muzahid, W. Ahn, and J. Torrellas. Dynamically detecting and tolerating if-condition data races. In *HPCA'14*.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MIRCO'01*.

[23] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS'02*.

[24] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *EuroSys'09*.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[26] TransmissionBT. http://www.transmissionbt.com/.

[27] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *ASPLOS'13*.

[28] T. N. Viktor Leis, Alfons Kemper. Exploiting hardware transactional memory in main-memory databases. In *ICDE'14*.

[29] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin. Debugging performance impact of unnecessary lock contentions via replay technique. Technical report, Huazhong University of Science and Technology, http://grid.hust.edu.cn/xfliao/CGCL-SYS-TR-2014-05.pdf, 2014.