A Performance Debugging Framework for Unnecessary Lock Contentions with Record/Replay Techniques

Xiaofei Liao, Long Zheng, Bingsheng He, Song Wu, Hai Jin

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, incurring significant performance overhead. This paper presents a performance debugging framework, PERFPLAY, to facilitate the identification of unnecessary lock contentions and to guide programmers to improve the program performance by eliminating the unnecessary lock contentions. Since the performance debugging of unnecessary lock contentions is input-sensitive, we first identify the representative inputs for performance debugging. Next, PERFPLAY quantifies the performance impact of unnecessary lock contention code regions for each candidate input. Taking into account conflicting attribute of performance impact and input coverage in the real world, we finally make the tradeoff between performance impact and input coverage to recommend the optimal unnecessary lock contention code regions. Our final results on five real-world programs and PARSEC benchmarks demonstrate the significant performance overhead of unnecessary lock contentions, and the effectiveness of PERFPLAY in troubleshooting the target unnecessary lock contention code regions with the consideration of both performance impact and input coverage.

Index Terms—record/replay, unnecessary lock contention, performance impact, multiple input, program debugging

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 will not acquire a lock until the lock releases 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 [1], [2]. In this paper, we study whether and how we can help the programmer identify the unnecessary lock contention and further understand their performance impact.

Let us illustrate the problem of unnecessary lock contentions with a real example from mysql−5.6.11 [3] as shown in Figure 1. It illustrates 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 disabled by the user. 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 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

```
Thread 1:
void fil_flush_file_spaces(...){
5609:   mutex_enter(&fil_system->mutex);
5611:   n_space_ids=UT_LIST_GET_LEN(
                fil->system->unflushed_spaces);
5614:   mutex_exit(&fil_system->mutex);
}
Thread 2:                                  Conflict
void fil_flush(...){
5473:   mutex_enter(&fil_system->mutex);
       /*search hash table by a given id*/
5475:   space=fil_space_get_by_id(space_id);
5483:   if (fil_buffering_disabled(space)){
          /*checking some data and states*/
5501:     mutex_exit(&fil_system->mutex);
5503:     return;}
          ...
5573:   UT_LIST_REMOVE(unflushed_spaces,
            fil->system->unflushed_spaces,space);
5592:   mutex_exit(&fil_system->mutex);
}                          storage/innobase/fil/fil0fil.cc
```

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

• *X. Liao, L. Zheng, S. Wu, and H. Jin are with 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. E-mail: {xfliao, longzh, wusong, hjin}@hust.edu.cn.*
B. He is with School of Computer Engineering, Nanyang Technological University, 639798, Singapore. E-mail: BSHE@ntu.edu.sg.

parallelizable code regions.

Due to the significant overhead of ULCPs at runtime, a volume of runtime research [1], [4] 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 [5], [6] 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 [5]. Second, a few transaction aborts may lead to a large number of rollbacks [6].

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 [1], [4], [7], [8] 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 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 ULCPs manually. Even worse, they are intertwined 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 identify the common performance critical ULCP code regions in the lock-based programs. The core idea of PERFPLAY is based on replay technique. To find common ULCP code regions across inputs, we have to take the multiple inputs to test all possible paths of program. To enable the high-coverage debugging, one intuitive method is to test the program with all potential inputs, but this work is very laborious and impractical for the real programs. Fortunately, our observations in the ULCP study motivate us that such massive inputs, in fact, can be dramatically reduced into a few representative inputs in practice. Consequently, we are able to reduce the number of inputs in order to resolve the programmer burden when facing too many inputs. Taking the representative inputs, we test the program with them one by one. To be specific, PERFPLAY records the program execution with a given candidate input into a trace. Through analyzing this 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. Next, we group the ULCPs generated by the same code regions and summarize the overall performance per code-site. Finally we get a list of ULCP code regions for this input.

Afterwards, we test the program with another candidate input and repeat the same process. We therefore collect a full list of UCLP code regions of the program. We also find that, the performance impact and input coverage of ULCP code regions show an interesting tradeoff in performance debugging of ULCP. To troubleshoot the optimal ULCP code regions with high performance impact and input coverage, we formulate this issue into a mathematical isse, i.e., finding the pareto-optimal points that have the largest possible number of performance improvements and input coverage. As a result, we use the multi-objective optimization [9] to identify the final tradeoffs.

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 presents input reduction. Section 4 elaborates how to transform a recorded program execution trace into a new ULCP-free trace. Section 5 describes how to recommend the optimal ULCP code regions with the high performance impact and coverage. Section 6 further presents the implementation details. Section 7 presents the experimental results. We review the related work in Section 8 and Section 9 concludes the work.

2 ULCP: A MOTIVATION STUDY, ILLUSTRATIVE EXAMPLE, AND FINDINGS

We start with a motivation study on ULCPs. Then, a concrete example is given to illustrate dynamic behavior of ULCPs. Learning from our study, we next summary several novel findings regarding ULCPs. Motivated by the study and implied by findings above, we ultimately develop a debugging framework to address the ULCP performance problem.

2.1 A Motivation Study

We have surveyed the number of each category of ULCPs in five real-world programs (including *openldap* [10], *mysql* [3], *pbzip2* [11], *transmissionBT* [12] and *handBrake* [13]) and PARSEC benchmarks [14]. The detailed experimental setup and discussion can be found in Section 7.1.

(1) *Null-Lock* refers to the synchronization pair where there exists no shared-memory access in the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

Table 1
Breakdown of ULCPs in real-world programs and PARSEC benchmarks. $Size$ indicates the total size of all source files (e.g., *.h, *.c, *.cpp). $\#Locks$ means the number of lock/unlock pairs during the dynamic execution. $NL.$ refers to the null-locks, $RR.$ the read-read pattern, $DW.$ the pattern of disjoint-write.

Applications	LOC	Size	# Locks	# ULCPs				# TLCPs
				NL.	RR.	DW.	Benign	
openldap	392K	6M	1,851	75	1,414	473	15	411
mysql	1,132K	22M	2,109	125	9,822	2,924	194	638
pbzip2	5K	1M	1,281	2	1047	838	51	2,06
transmissionBT	79K	4M	352	15	111	123	29	95
handbrake	1,070K	3M	18,316	10	1,536	1,143	189	2,311
blackscholes	812	204K	0	0	0	0	0	0
bodytrack	10K	9.0M	32,642	0	1,322	321	43	1,751
canneal	4K	628K	34	0	0	0	0	29
dedup	3.6K	156K	19,352	231	2,421	1,952	164	2,091
facesim	29K	4.8K	14,541	102	871	819	12	1,099
ferret	9.7K	316K	6,231	11	101	231	343	465
fluidanimate	1.4K	72K	82,142	2	10,501	6,694	197	2,591
streamcluster	1.3K	44K	191	0	0	0	0	36
swaptions	1.5K	152K	23	0	0	0	0	16
vips	3.2K	17M	33,586	142	4,512	1,142	26	1,279
x264	40.3K	2.4M	16,767	941	3,841	412	84	817

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 [15].

(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 [10].

(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 [16], [17].

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 An Illustrative Example

Figure 2 depicts a source code snippet protected by `dbmp->mutex` from OpenLDAP [10]. This code may affect the CPU utilization of system when a large number of threads call this code simultaneously. That

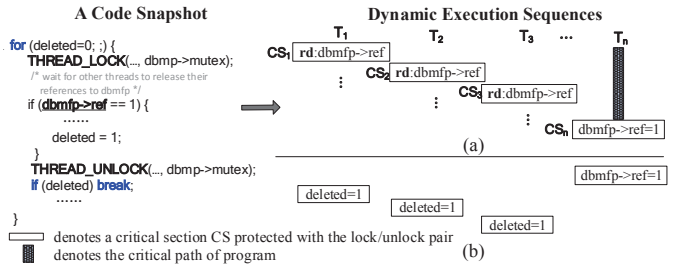


Figure 2. A code snippet with problematic synchronization implementation from OpenLDAP. (a) A great deal of CPU time is wasted due to the spin-waits of threads $T_0, \cdots, T_{n-1}$ for the release of `dbmfp->ref` if the critical thread T_n runs slowly. (b) Little CPU time is wasted if T_n is finished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

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 (Motivated from Implication (1)).

Lock Elision (LE) [1], [2], [4], [6] is a technique that dynamically eliminates the inter-thread ULCP dependencies. 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. However, LE cannot precisely track the impact of system resource wasting caused by ULCPs. For instance, if the ULCPs incur on the non-critical path of program, e.g., $T_1, T_2, \cdots, T_{n-1}$, the optimization of LE-based work just performs more CSs as ineffective execution. It does not notice this impact. As a debugging tool, both performance degradation and system throughput loss should be concerned as the ULCP performance impact. 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). Nevertheless, if the program is scheduled as shown in Figure 2(b), this ULCP performance problem will disappear.

2.3 Findings and Implications

From the studies on real-world and benchmark programs, we have obtained a number of interesting

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

Table 2
The number of ULCP code regions with the varying number of threads (2/4/6/8)

Applications	# ULCP code regions			
	2	4	6	8
openldap	18	18	19	21
mysql	57	59	60	60
pbzip2	4	4	4	4
transmissionBT	2	2	2	2
handbrake	29	30	30	32
blackscholes	0	0	0	0
bodytrack	5	5	5	5
facesim	11	12	12	12
fluidanimate	3	3	3	3
swaptions	0	0	0	0

findings, which have significant implications in the design of our performance debugging framework.

Finding (1): ULCP is a diverse program behavior. It is ubiquitous in the multi-threaded program and scattered in the program execution.
Implication (1): It is time-consuming and tedious to manually figure out which code-site incurs the highest performance impact due to ULCPs.

Explanation: We count the quantitative distribution of ULCPs of all applications with two threads in Table 1. 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 ULCPs, in most cases, are interconnected rather than isolated. The ULCP interconnections may be embodied since they are produced by some common codes that will be repeatedly executed in most threads.

Finding (2): We need to test massive inputs to cover all statements of program in the real world.
Implication (2): It is more reasonable to identify the relatively important or representative inputs..

Explanation: We make an attempt to find a set of representative inputs to cover all program statements in the real world as much as possible. More details are discussed in Section 3 motivated from this finding.

Finding (3): The number of the ULCP code-sites is usually orthogonal to the thread numbers in the real-world. The majority of ULCPs can manifest themselves with two threads only.
Implication (3): We can use only two threads to expose ULCP problems, thus significantly reducing the complexity of debugging.

Explanation: We investigate the number of final ULCP code regions in real applications and PARSEC benchmarks with the varying number of threads as shown in Table 2. We also find that the number of ULCP code regions almost has not changed as the number of threads increases. The main reason is: in those applications all threads reuse the same code (e.g., functions) to perform the program execution. More thread numbers may only change their performance impact. On the other hand, the overhead of test and analysis, in general, would be increased in an exponential order to the number of threads. Therefore, we can significantly reduce this overhead with only two threads while ensuring that the final concerned results are not missing.

Finding (4): In our tested programs, the program with more ULCPs, in general, can be improved much more from the ULCP performance influence.
Implication (4): The number of ULCPs in a trace can be used as an important metric to indicate the value of trace.

Explanation: We survey the performance impact of ULCPs with the different number of ULCPs (more results in our technical report [18]). Finding (4) implies that we can use the number of ULCPs in the trace to measure whether we can get the valuable ULCP code regions from this trace before the prohibitive replay execution. Following this implication, in practice, we can use it to distinguish a few traces from a large number of execution traces with all inputs to better represent the ULCP performance behavior of program. Furthermore, the cost of trace analysis will be further reduced if only a few traces need to be analyzed.

Finding (5): The conflicting variation is shown between performance impact and input coverage of the ULCP code regions in the real world.
Implication (5): We need to make a tradeoff between performance impact and input coverage to point out the common performance critical ULCP code regions.

Explanation: We observe the relationship between performance impact and input coverage of all identified ULCP code regions in the real world. Finding (5) tells us that performance impact of a ULCP code region does not have a variation of direct proportion with input coverage. One common reason in our observation is that the majority of performance critical ULCP code regions are exclusive in some functional units which are triggered with some specific inputs. For instance, downloading function for transmissionBT application is only triggered by download-related options of input, not by uploading option. Hence, a high performance impact ULCP code region in downloading function will not take place under the uploading option of input.

2.4 Overview of Our Approach

From the aforementioned study and real-world example, we develop a performance debugging frame-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

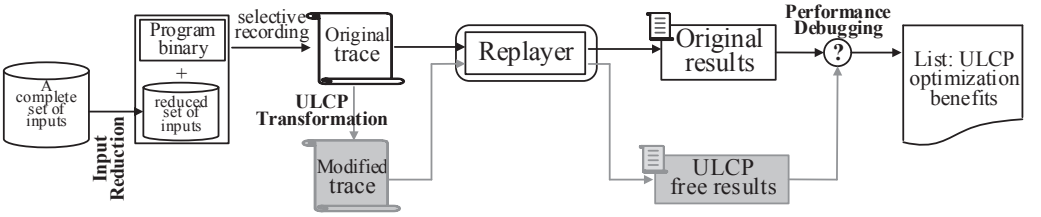


Figure 3. Overview of PERFPLAY

work, namely PERFPLAY, to assist the programmer in addressing the problem of ULCPs in their code. In this work, we use the record/replay technique as our core solution for two reasons. First, the replay system, in general, 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 beneficial ULCP to fix.

Figure 3 depicts the overview of PERFPLAY. PERFPLAY completely 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. To be specific, we first use an existing coverage tool [19] to generate a complete set of inputs for the program. From Implication (4), the complete set of inputs are further reduced into a few representative inputs for the effective ULCP debugging (Section 3). PERFPLAY takes the representative inputs as the input base and tests them one by one. For each given input, PERFPLAY records the intervals of a program execution trace. After the generation of original recording trace, PERFPLAY then transforms the original trace with ULCPs into a new trace without ULCPs, and replays the original trace and the modified one. By comparing these two replayed results, PERFPLAY evaluates the potential performance impact of the aggregated ULCPs per code-site for this given input, but may be not helpful for the ULCP debugging with other inputs. Consequently, we perform the program traces with all candidate inputs. After collecting all ULCP results from all candidate inputs, we identify those pareto-optimal ones via multi-objective optimization.

3 INPUT REDUCTION

In this work, we use the trace-driven model to tackle the ULCP problems. One big problem of this model is that it only analyzes the specific trace with a given input. To enhance the input sensitivity of trace-driven model and find common problems across inputs, an intuitive way is to test the program with all potential inputs generated by the input generators (e.g., Microsoft PEX [19]). However, this is not realistic in practice, since the number of generated inputs is too large for programmers to perform performance debugging. Table 3 shows the total number of functions, if-branches and while-loops in five real programs. *handbrake* has the smallest number (only 26) of functions, but it still involves around 8741 if-branches and 360 while-loops (i.e., program paths). It is impractical to cover all these paths, even for this "small" program. More seriously, *mysql* even has more than 18898 functions, 70228 if-branches and 5556 while-loops, which makes the problem more complicated.

Table 3
The total number of functions, if branches and while loops in the real programs

Applitions	openldap	mysql	pbzip2	transmissionBT	handbrake
#Functions	425	18898	92	683	26
#If	29922	70228	496	8086	8741
#While	916	5556	25	403	360

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
1 **if** $C_1.S_{rd} = \emptyset$ **and** $C_1.S_{wr} = \emptyset$ **or** $C_2.S_{rd} = \emptyset$ **and** $C_2.S_{wr} = \emptyset$ **then**
2 **return** NULL_LOCK;
3 **else if** $C_1.S_{wr} = \emptyset$ **and** $C_2.S_{wr} = \emptyset$ **then**
4 **return** READ_READ;
5 **else if** $C_1.S_{rd} \cap C_2.S_{wr} = \emptyset$ **and** $C_1.S_{wr} \cap C_2.S_{rd} = \emptyset$ **and** $C_1.S_{wr} \cap C_2.S_{wr} = \emptyset$ **then**
6 **return** DISJOINT_WRITE;
7 **else**
8 **return** FALSE;

In order to select out the representative inputs, we take the following series of cooperative approaches, which are specifically designed for ULCPs.

ULCP Identification: After collecting the traces performed with all potential inputs, we first identify the four-classified ULCPs in the traces so as to evaluate the value of trace related to ULCP performance problems. To be specific, we use shadow memory [20] 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:

- $C.S_{rd}$: a set of all shared reads in C.
- $C.S_{wr}$: a set of all shared writes in 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. The complexity of Algorithm 1 is up to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

distribution of lock-protected code regions. If these regions are performed by the same thread, Algorithm 1 takes the best complexity with O(1). However, if they are alternated between different threads, Algorithm 1 will perform the worst complexity with O(N-1). In practice, 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 [16] for the distinction of 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.

For each trace, Algorithm 1 is used to quickly filter out the majority of simple ULCPs (i.e., Null-Lock, read-read ULCPs, and disjoint-write pattern). Afterwards, each pair remaining will be replayed once by the reversed replay execution for the distinction of benign ULCP and TLCP. This means that the scalability of our ULCP analysis has the linear relationship to the total number of benign ULCPs and TLCPs. Still, in practice it does not affect our tool as an effective tool to debug the ULCP performance problems due to two reasons. First, as shown in Table 1, benign ULCPs and TLCPs usually take a small fraction of all lock pairs, even in the lock-intensive applications, such as *fluidanimate*. Second, motivated by Finding (3), we test the program with only two threads.

After the *prior* ULCP identification in the trace, next we propose two input reduction techniques as follow:

Similarity reduction: In the actual test, a large number of inputs, in fact, have the similar options. For instance, for pbzip2, there are two potential variable options of thread number and data set when we test its compression and decompression functions. Interestingly, more thread numbers and data sets always go through the same set of functions, thus the variation of these two options does not affect the exposure of final ULCP code-site recommendations. In the following, we refer to input similarity as those inputs that will trigger the program executions with the same or the similar function set.

After identifying ULCPs, we then locate the code sites that produce the individual ULCPs in the trace. We therefore can sort those inputs that trigger the traces with the same or similar ULCP code site set. Among these sorted inputs, we recommend the *minimum configuration* of input as the potential candidate input on behalf of the inputs that have the similarity, e.g., two threads implied in Finding (3) instead of using more than two threads.

Quantity reduction: Fortunately, we find that the majority of functions are exclusive of the lock/unlock operations and ULCPs. Motivated by this and Implication (4), the inputs thus can be greatly reduced by only exploiting ULCP-intensive candidate inputs. For the example in Figure 2(b), when `main` function (i.e., T_n) encounters an error, the program will trap into the error-handler function. In this case, the program is finished fast as usual but few ULCPs take place. The input incurring this case can be neglected.

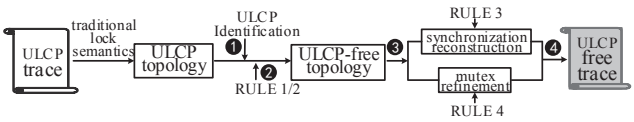


Figure 4. The process of ULCP transformation

We next follow Implication (4) to further reduce the candidate inputs to be analyzed via the quantity of ULCPs in the trace. We first count the number of ULCPs in the trace. According to the counted ULCP number, we finally can recommend those inputs that trigger the traces with the *high magnitude order* of ULCP number as the potential candidate inputs. For example, suppose we have two inputs—Input#1 and Input#2. Input#1 induces the ULCPs with the 10^5 magnitude order while Input#2 is with the 10^3 magnitude order. In this case, we finally will choose Input#1 as the candidate input.

In practice, similarity reduction and quantity reduction can co-work to dramatically reduce the program inputs that are technically selected out and specially designed for the ULCP performance problems. With two input reduction techniques, we guarantee that our candidate inputs will have the capacity of exposing the performance critical ULCPs instead of the untested inputs. That is, our tool does not miss any critical ULCP code sites that may be triggered under an untested input.

4 ULCP Transformation

This section presents the detailed procedure of transforming the original trace with ULCPs into a new trace without ULCPs. To solve 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 [21]. Since topological graph theory has been studied for decades, the ULCP problem can be solved easily by analyzing the graph.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

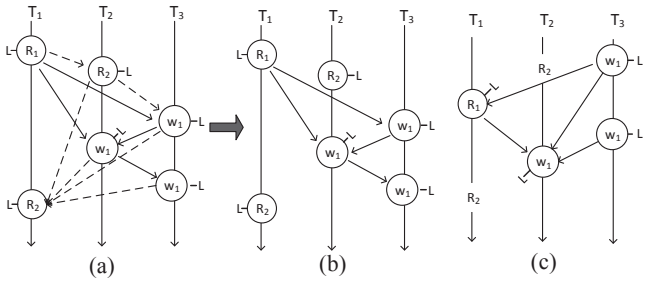


Figure 5. The causal dependencies for an example. ◯ represents the critical section, while L attached to ◯ means this critical section is protected by lock L. R_1 indicates a read on shared data 1 and the dotted arrow shows a ULCP.

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.
- ***Node*:** a critical section in the topology.
- ***Causal-edge*:** the causality between two nodes.

4.1 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).

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, 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 6). 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 $\{R_1(T_1) \prec W_1^{1st}(T_3) \prec W_1(T_2) \prec W_1^{2nd}(T_3)\}$.

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

We eliminate the false inter-thread dependencies by different ULCP categories. 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 [22] 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.

RULE 3. *Each node with the outdegree in the topology will be given a new auxiliary lock. While each node with*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

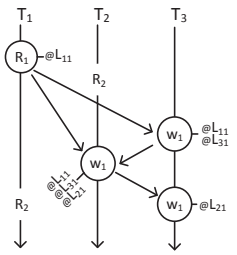


Figure 6. The re-synchronization of the ULCP free causal dependencies. @L indicates auxiliary locks.

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 (**Correctness**). *The ULCP free trace is performed without introducing new deadlocks while ensuring the program correctness or reporting the data races.*

Proof. **Deadlock Condition:** The fine grained locks in this paper are mainly introduced to construct the ULCP free execution, the program semantics of which is represented by the ULCP free topology in Section 4.1. Recall that, in the ULCP free topology, we just remove the dotted directed edges arising from ULCPs while still preserving the solid directed arrows caused by TLCPs. This procedure implies that:

Cond-1 We do not add any new directed arrows in the topological graph

Cond-2 We do not change the direction of any directed arrows in the topological graph

According to the above properties of ULCP free topological graph, we can conclude that:

- We do not introduce new deadlock conditions as we do not introduce new arrows directly (i.e., **Cond-1**) or indirectly (i.e., **Cond-2**). As a consequence, our approach definitely does not introduce new deadlocks.
- For a program that may already have some deadlocks, our removal of dotted arrows may destroy the conditions of the original deadlocks. As a consequence, our approach may suppress the deadlocks that already exist in the original ULCP program.

Program Correctness: We reduce program correctness into the trace correctness due to the trace-based transformation. Consider such a typical model for the execution trace: two threads have the execution sequences $\{SG_1, A, SG_2\}$ and $\{SG'_1, B, SG'_2, C, SG'_3\}$, respectively. A, B and C are the critical sections protected by the same lock, and SG denotes the program segment. It is assumed that A precedes B in one observation, with $\langle A, B\rangle$ consisting of an ULCP, and $\langle A, C\rangle$ being non-ULCP. According to RULE 1, C is the first non-ULCP for A, a causal edge pointing between A and C should be established. Therefore SG_1 and SG'_3 maintain the original semantics based on RULE 2. Only difference of ULCP-free trace from the original one lies in the parallelism between SG_1 and SG'_2 due to RULE 3.

- If the segments SG_1 and SG'_2 involve the conflict free memory accesses, the ULCP free trace will be performed with the same result as the original one. Therefore, the correctness of the ULCP free trace is guaranteed in the sense that it produces the same program semantics as the original one.
- Otherwise, our transformation possibly produces diverse results due to the problematic interleavings of shared accesses, i.e., data races between $\langle SG_1, B\rangle$, $\langle SG_1, SG'_2\rangle$, or $\langle A, SG'_2\rangle$. This case may present the correct program semantics of other executions, but it produces the same value of data races as ULCP performance problem. It further enables PERFPLAY to help developers understand the correctness of the original trace. □

5 ULCP PERFORMANCE DEBUGGING

After the phase of ULCP transformation, we obtain a set of ULCPs from each trace. However, we still face two major problems. i) There may be many ULCPs, and some of them are even from the same code-site. ii) The ULCP debugging with a specific input may not be helpful for the one with other inputs. An effective debugging tool should point out the succinct code-site for distinctive ULCPs, and also locate the common performance critical ULCP code regions across inputs. Thus, we propose ULCP fusion and performance accumulation based on their code regions (Section 5.1), and recommend the ULCP code regions with high performance impact and input coverage (Section 5.2).

5.1 ULCP Fusions

For a specific trace replay, we model the potential runtime overhead of a ULCP. Figure 7 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2472412, IEEE Transactions on Parallel and Distributed Systems

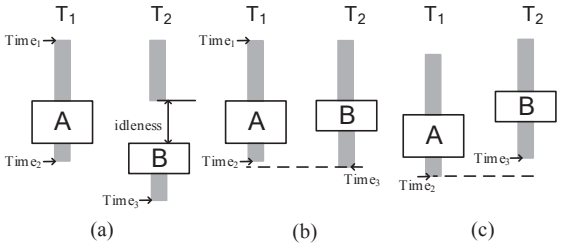


Figure 7. Two different performance measurements

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 7(b) and Figure 7(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 $\Delta Time_2 - \Delta Time_1$. Thus, we define the performance improvement of each ULCP:

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

where $Time_{label}$ indicates the current timestamp of program when it 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. Algorithm 2 takes the sequential checking for any two consecutive ULCPs in the given ULCP set. This means Algorithm 2 has the complexity with O(N-1).

Algorithm 2: ULCP Fusion and Performance Accumulation

Input : $\langle ULCP_1, ULCP_2\rangle$, two standalone ULCPs;
Output: $ULCP_{new}$, 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;
```

Algorithm 3: Unique UCR Fusion Across Inputs

Input : $I_i = \{ULC_1, ULC_2, \cdots, ULC_{k_i}\}, (1 \le i \le n)$
Output: $\mathbb{S} = \{S_1, S_2, \cdots, S_m\}$

```
/* The initialization */
1  S ← ∅;
2  m ← 0;
3  for i ← 1 to n do
4      for j ← 1 to |Ii| do
5          if Ii.ULCj ∈ S then
               /* Assume Ii.ULCj = St (1 ≤ t ≤ |S|) */
6              St.x++;
7              St.ΔT ← St.ΔT + Ii.ULCj.ΔT;
8          else
9              m ← m + 1;
10             Sm ← Ii.ULCj;
11             Sm.x ← 1;
12             Sm.y ← Ii.ULCj.ΔT;
```

5.2 Pareto-optimal ULCP Recommendations

Through the trace replays with a reduced set of candidate inputs $\{I_1, I_2, \cdots, I_n\}$, and ULCP fusion and performance accumulation by Algorithm 2, we obtain a group of ULCP code-regions (UCRs) for each input I_i $(1 < i < n)$, and its corresponding performance improvement ΔT. As a ULCP code region may take place under different inputs, we further propose Algorithm 3 to merge these ULCP code-sites within different inputs into a set of unique ULCP code-regions, denoted as $\mathbb{S} = \{S_1, S_2, \cdots, S_m\}$. Each S in $\mathbb{S}$ is consisted of two attributes x and y, i.e., (x, y) where x refers to how many inputs this merged UCR spans, and y indicates the total accumulated performance improvement of this UCR. In Algorithm 3, each input has the fixed number (i.e., k_i) of ULCs reported by our debugging tool. In the case of N inputs, as a consequence, Algorithm 3 takes the complexity with O(N). Finding (5) tells us that x and y may show the conflicting variation in the real world. As a result, to troubleshoot the optimal UCRs with the high performance impact and input coverage, we transform the problem into the following mathematical problem: finding the pareto-optimal points in $\mathbb{S}$ with the largest possible number of both x and y due to the tradeoff between performance impact and input coverage.

We leverage multi-objective optimization [9] (also known as pareto optimization) to solve this kind of problem. In short, it takes an optimal decision with the tradeoffs between two or more conflicting objectives. Towards our problem, the two conflicting objectives are x and y, respectively. We denote the preferred relation between two ULCPs as the binary operation $\prec$. If two ULCP code-sites (e.g., UCR and

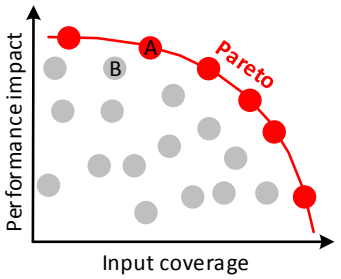


Figure 8. The Pareto frontier where nodes together with the line are the pareto optimal points.

UCR') satisfy the conditions of $UCR.x \geq UCR'.x$ and $UCR.y \geq UCR'.y$, we define that UCR is preferred to (strictly dominating) UCR', written as $UCR' \prec UCR$, which means that UCR has the higher optimization value of being recommended. The pareto-optimal ULCP code-sites thus can be formalized as follows:

$$P(\mathbb{S}) = \{UCR \in \mathbb{S} : \{UCR' \in \mathbb{S} : UCR \prec UCR', UCR \neq UCR'\} = \emptyset\} \quad (2)$$

Proof. Given the set $\mathbb{S}$, a pareto optimal point has the following property:

> *"If a certain UCR is pareto optimal, among a given $\mathbb{S}$ there is no such element UCR'_k, which has greater x and y than UCR_k."*

The sub-equation $\{UCR' \in \mathbb{S} : UCR \prec UCR', UCR \neq UCR'\} = \emptyset$ exactly expresses the very meaning of "among a given $\mathbb{S}$ there is no such element UCR', which has greater x and y than UCR" (i.e., $UCR \prec UCR'$ in the paper). Consequently, the given UCR is one of the pareto optimal results. With the purpose of finding all pareto optimal points, Equation 2 is provided to further traverse all elements in $\mathbb{S}$. □

Finally, we recommend the pareto-optimal set $P(\mathbb{S})$ as the final ULCP code regions across inputs to programmers for the performance debugging of ULCP. With Equation 2, the final pareto-optimal results will be manifested themselves as the line-crossed points as shown in Figure 8. For instance, the point A has the larger number of both x (input coverage) and y (performance impact) than the point B, i.e., $B \prec A$, thus A is recommended whilst B is ignored.

6 Implementation Issues

We implement PERFPLAY for the parallel replay based on Pin [23], an underlying framework that enables programmers to perform the program analysis at runtime without source codes. In the following, we briefly discuss one implementation detail: 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.

Performance Fidelity: There has been significant amount of work on building record/replay systems [16], [24], [25], [26] 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 [27].

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. ELSC ensures the performance fidelity of replay execution for the multiple replays based upon the same given trace.

Theorem 2 (Performance Fidelity). *ELSC guarantees the performance fidelity of the parallel replay for the same given trace.*

We provide the detailed proof regarding performance fidelity in our technical report [18].

7 Evaluation

7.1 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 each individual applications is as follow.

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 to benchmark it with the option of searching 2000 entries.

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 test it 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.

Table 4
Input selections and pareto-optimal URCs in the real programs. *#Input* is the number of candidate inputs, *Magn.* the magnitude order of ULCPs in the dynamic executions. $|P(\mathbb{S})|$ refers to the number of final pareto-optimal URC recommendations, $\overline{x}$ the average number of appearance times in the candidate inputs.

APP.	#Total	#Reduced Input		Magn.	$\lvert P(\mathbb{S})\rvert$	$\overline{x}$	Coverage
		Similarity	Quantity				
openldap	120	43	29	$> 10^3$	11	24	82.8%
mysql	3,080	898	100*	$> 10^6$	26	87.5	87.5%
pbzip2	68	18	6	$> 10^3$	4	4.5	75.0%
transBT	180	37	14	$> 10^2$	2	13	92.9%
handbrake	340	76	18	$> 10^4$	8	14.5	80.6%

* Through the input reduction discussed in Section 3, *mysql* still has 388 candidate inputs. In our test, we randomly sample them into 100 reduced candidate inputs.

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

1. Memory-based schedule (MEM-S) [25], which enforces a deterministic execution sequence of all shared memory accesses.

2. Synchronization-based schedule (SYNC-S) [28], which enforces the same total order of the lock synchronizations for the same input.

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

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

Input Configuration: Basically, we select the inputs that will cover all functionalities of the program. For *openldap*, *mysql*, and *handbrake*, they have the well-designed test inputs written by their developers to cover all possible core functionalities. For *transmissionBT* and *pbzip2*, they do not have available test inputs in public. We have designed their inputs based on their provided options of input. For example, we test the *transmissionBT* application by covering the downloading, uploading, suspending, resuming and troubleshooting etc. *pbzip2* is driven with the main input options of compression, decompression, compression integrity testing and error message suppressing etc. To evaluate the quality of our generated inputs, we use `gcov` [29] to help us improve the statement coverage of all designed inputs.

7.2 Pareto-optimal ULCP Recommendations

According to our input configuration, we count the number of all collected or designed inputs for each program. We then perform the similarity and quantity reduction scheme, respectively. Table 4 reports the final results, and concludes the criteria of our selection and pareto-optimal UCR recommendations.

For *openldap*, we collect 120 inputs in total. Through similarity and quantity reduction, we reduce these inputs into a few candidate inputs by 43 and 29, respectively. Each execution trace has $> 10^3$ ULCP magnitude scale. Through analyzing 29 traces, PERFPLAY recommends 11 pareto-optimal (input insensitive) ULCP code regions across inputs. Each recommended pareto-optimal ULCP code region covers 24 inputs on average, which takes 82.8% proportion of the original inputs. The relevant results for other applications are shown in Table 4. Among them, *mysql* has the highest magnitude order of ULCPs, but only 26 pareto-optimal UCRs are reported. For the *pbzip2* applications, only 6 representative inputs are generated for the subsequent trace analyses. This significantly reduces the overhead of test and analysis. In summary, we test the program with a dramatically reduced number $(6 - 100)$ of candidate inputs and recommend only a few $(2 - 26)$ pareto-optimal UCRs in the real programs. Besides, the final pareto-optimal UCRs cover a high proportion $(75\% - 92.9\%)$ of the given candidate inputs.

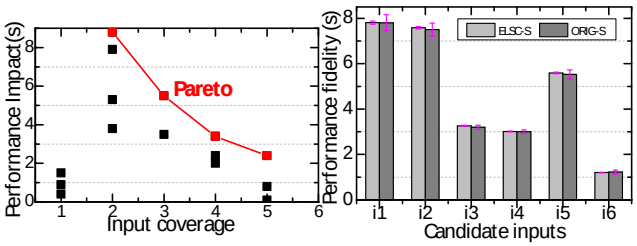


Figure 9. The pareto optimal points for pbzip2

Figure 10. Performance fidelity across inputs

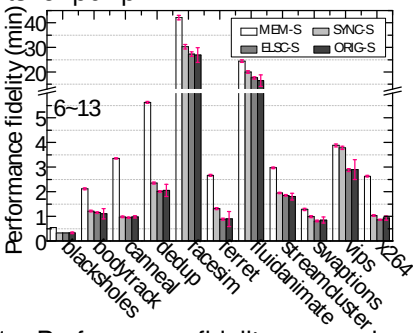


Figure 11. Performance fidelity comparison between different schemes for the replay execution

Figure 9 illustrates 4 pareto-optimal UCR recommendations out of all UCRs in the *pbzip2* application with six candidate inputs in Table 4. We can see that 1) no UCR covers all (6) candidate inputs; 2) some UCRs with high performance impact are with the limited input coverage; 3) the performance impact of the pareto-optimal UCR has the downward trend as the input coverage increases. This observation further verifies our Finding (5) in Section 2.1. The full list of UCRs and more details about pareto-optimal UCRs in *pbzip2* can be found in the supplementary material.

7.3 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 11 illustrates the final replayed execution time of PARSEC benchmarks with the specific input (given in Section 7.1) 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 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.

Figure 10 depicts the performance fidelity of PERFPLAY across all candidate inputs of *pbzip2* in Table 4. Our detailed candidate inputs for *pbzip2* include: compression with/without verbose model, decompression with/without verbose model, compression integrity testing and error message suppressing, labeled as i1, i2, i3, i4, i5, i6, respectively. The final results also demonstrate the performance fidelity of PERFPLAY under different inputs.

7.4 Case Study

To evaluate the effectiveness of PERFPLAY, we have checked some pareto-optimal ULCP code regions reported by PERFPLAY framework. More real examples of this work regarding ULCP problem can be found in our technical report [18].

mysql. With the InnoDB storage engine, the block read rate of *mysql* server is severely limited by the unnecessary lock contentions on the lock `fil_system->mutex`. The problem is that for each block read operation, the hash table lookup operation `fil_space_get_by_id` will be invoked four times at least by the functions `fil_space_get_version`, `fil_inc_pending_ops`, `fil_decr_pending_ops` and `fil_space_get_size`. All of them have the code pattern as follow:

```
lock(file_system->mutex);
fil_space_get_by_id(id);
do some simple work;
unlock(file_system->mutex);
```

Among these four functions, actually, only the `inc` and `decr` operations are conflicting with each other. Figure 12 illustrates the code snippets of both `fil_inc_pending_ops` and `fil_space_get_size` from *mysql* version-5.6.11. They can be executed in parallel. Suppose the multiple threads perform a large number of read-only transactions, this type of ULCP problem serializes many lookups of the hash table with a slowdown of 4X at least. A mitigation practice is to do the hash table lookup once for each block read with the reference count and pointer. After the first lookup, we save the pointer to the table space. As a result, we can use the pointer to operate the table directly without the extra lookups by other callers, and make the `inc/decr` callers increase/reduce the reference count on the space instead of deleting it. The proposed method would significantly reduce the number of ULCPs in this case.

```
Thread 1:
bool fil_inc_pending_ops(){
  mutex_enter(&fil_system->mutex);
  space=fil_space_get_by_id(id);
   ...
  space->n_pending_ops++;
  mutex_exit(&fil_system->mutex);
}
Thread 2:
ulint fil_space_get_size(...){
  mutex_enter(&fil_system->mutex);
  space=fil_space_get_by_id(id);
  size=space?space->size:0;
  mutex_exit(&fil_system->mutex);
}
```

Figure 12. A ULCP code region problem from mysql

```
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 13. A ULCP code region problem from pbzip2

pbzip2. Figure 13 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 can 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.

8 Related Work

Multiple Input Testing. A complete set of input testing makes the software testing costly [19], [30], [31], [32]. Some more advanced techniques have been proposed to reduce this cost, such as automated software testing [33] and sampling test [34]. However, they are still hard to be used for the ULCP test because 1) ULCP analysis persecutes the automation of software testing on account of complex characteristics of ULCPs (i.e., Finding (1)); 2) sampling technique may miss some critical UCLPs. As a result, inspired by Finding (3) and Finding (4) in our work, we propose input reduction techniques specific to ULCPs.

Unnecessary Lock Contention. Lock Elision (LE) [1], [4] removes the lock speculatively during the dynamic execution. The lock is acquired only when a conflict occurs. However, LE-based work is still challenging. A few transaction aborts may cause excessive rollbacks, which severely limits the exposed concurrency [6]. Also, it is prone to trigger false aborts due to the hardware limitations [5]. 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 [27]. This work is exactly such work to help the programmers understand the ULCPs.

Performance Tools. The code snippet with a lock/unlock pair running simultaneously by multiple threads may unroll into two execution cases as ULCPs and TLCPs. Thus, it is hard for static tools [35] to obtain the characteristics of ULCPs. As for the existing dynamic tools [36], 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. PERFPLAY is the very performance tool to solve this problem.

Record/Replay System. Plentiful replay systems are proposed in the past several decades. For instance, deterministic replay systems [25], [26] reproduce the bug debugging by enforcing the order of the execution events. Modified replay debugging [16], [24] 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.

9 Conclusion

In this paper, we develop a debugging tool to identify the performance critical ULCP code regions across inputs on multi-core processors. First, we extract out a dramatically reduced set of inputs as the final candidate. Taking each candidate input, we then 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 via novel transformation rules. Finally, PERFPLAY replays two traces. Based on two replayed results, we evaluate performance impact of each ULCP and then group all ULCPs into the unique UCRs per their code-site. After the collections with all candidate inputs, we figure out the optimal ULCP code regions using multi-objective optimization. Our experimental results demonstrate the performance fidelity of PERFPLAY. With case studies, we also demonstrate its effectiveness in troubleshooting the pareto-optimal UCRs across inputs.

References

[1] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proc. of the 34th ACM/IEEE International Symposium on Microarchitecture*, 2001, pp. 294–305.

[2] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 5–17.

[3] MySQL. [Online]. Available: http://www.mysql.com

[4] A. Roy, S. Hand, and T. Harris, "A runtime system for software lock elision," in *Proc. of the 4th ACM European Conference on Computer Systems*, 2009, pp. 261–274.

[5] T. N. Viktor Leis, Alfons Kemper, "Exploiting hardware transactional memory in main-memory databases," in *Proc. of the International Conference on Data Engineering*, 2014, pp. 580–591.

[6] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *Proc. of the ACM Symposium on Principles of Distributed Computing*, 2014, pp. 212–221.

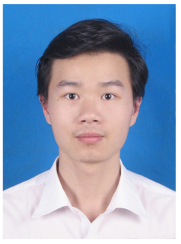
[7] Intel Corporation, "Intel architecture instruction set extensions programming reference," 2013. [Online]. Available: http://software.intel.com/file/41417

[8] 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 *Proc. of the International Symposium on Computer Architecture*, 2013, pp. 225–236.

[9] K. Deb and D. Kalyanmoy, *Multi-Objective Optimization Using Evolutionary Algorithms*. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[10] OpenLDAP. [Online]. Available: http://www.openldap.org
[11] pbzip2. [Online]. Available: http://compression.ca/pbzip2
[12] TransmissionBT. [Online]. Available: www.transmissionbt.com
[13] Handbrake. [Online]. Available: http://handbrake.fr
[14] PARSEC. [Online]. Available: http://parsec.cs.princeton.edu
[15] H. Qi, A. A. Muzahid, W. Ahn, and J. Torrellas, "Dynamically detecting and tolerating if-condition data races," in *Proc. of the 20th IEEE International Symposium on High Performance Computer Architecture*, 2014, pp. 120–131.
[16] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 22–31.
[17] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 1–16.
[18] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin, "On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach," *CoRR*, vol. abs/1412.4480, 2014. [Online]. Available: http://arxiv.org/abs/1412.4480
[19] N. Tillmann and J. Halleux, "Pex: White box test generation for .net," in *Proc. of Tests and Proofs*, 2008, pp. 134–153.
[20] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proc. of the 3rd International Conference on Virtual Execution Environments*, 2007, pp. 65–74.
[21] J. L. Gross and T. W. Tucker, *Topological Graph Theory*. New York, NY, USA: Wiley-Interscience, 1987.
[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
[23] Pin Tool. [Online]. Available: http://www.pintool.org
[24] N. Viennot, S. Nair, and J. Nieh, "Transparent mutable replay for multicore debugging and patch validation," in *Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 127–138.
[25] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proc. of the Symposium on Code Generation and Optimization*, 2010, pp. 2–11.
[26] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *Proc. of the 32nd International Symposium on Computer Architecture*, 2005, pp. 284–295.
[27] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin, "on performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach," in *Proc. of the Symposium on Code Generation and Optimization*, 2015, pp. 56–67.
[28] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in *Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 97–108.
[29] Gcov. [Online]. Available: gcc.gnu.org/onlinedocs/gcc/Gcov.html
[30] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," in *Proc. of the Symposium on Software Testing and Analysis*, 2004, pp. 97–107.
[31] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proc. of the Conference on Automated Software Engineering*, 2011, pp. 73–82.
[32] T. Y. Chen, F. C. Kuo, D. Towey, and Z. Zhou, "A revisit of three studies related to random testing," *Science China Information Sciences*, vol. 58, no. 5, pp. 1–9, 2015.
[33] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *Proc. of the Workshop on Automation of Software Test*, 2013, pp. 111–117.
[34] M. R. Lyu, *Handbook of software reliability engineering*. Hightstown, NJ, USA: McGraw-Hill, Inc, 1996.
[35] 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 *Proc. of the Conference on Software Engineering*, 2011, pp. 1066–1071.
[36] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. of the International Conference on Software Engineering*, 2012, pp. 145–155.

Xiaofei Liao received his Ph.D degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now a professor in the school of Computer Science and Technology at HUST. He has served as a reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P system, cluster computing and streaming services. He is a member of the IEEE and the IEEE Computer Society.



Long Zheng is a Ph.D candidate in the school of Computer Science and Technology at Huazhong University of Science and Technology (HUST) in China. He received his B.S. degree at HUST in 2010. His research interests focus on program analysis and software reliability.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems. He has been awarded with the IBM Ph.D. fellowship (2007-2008) and with NVIDIA Academic Partnership (2010-2011).

Song Wu is a professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He received his Ph.D. from HUST in 2003. He is now the director of Parallel and Distributed Computing Institute at HUST. He is also served as the vice director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST. His current research interests include grid/cloud computing and virtualization technology.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is a senior member of the IEEE and a member of the ACM. He has co-authored 15 books and published over 600 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.