Multi-Level Execution Trace Based Lock Contention Analysis

Majid Rezazadeh *, Naser Ezzati-Jivan †, Evan Galea †, Michel R. Dagenais *
{majid.rezazadeh,michel.dagenais}@polymtl.ca
{nezzati, eg18gz}@brocku.ca
*Polytechnique Montreal, Quebec H3T 1J4
†Brock University, Ontario L2V 5Y6

***Abstract*—multi-threaded programming is a near-universal architecture in modern computer systems. Thread based programs usually utilize locks to coordinate access to shared resources. However, contention for locks can reduce parallel efficiency and degrade scalability.**

In this paper, we propose an execution-trace based method to analyze lock contention problems, without requiring an application's source code. Our methodology uses dynamic analysis through execution tracing, running in several levels of the system to collect detailed runtime data. We combine it with an extended critical path algorithm which allows us to identify locking issues occurring in userspace. The result is a framework that is able to diagnose all contention issues while adding minimal impact on the system. We propose new views and structures to model and visualize collected data, giving programmers powerful comprehension tools to address contention issues.

I. Introduction

Monitoring and analyzing performance in multi-threaded applications has been always a big challenge for developers. Unexpected latency and performance degradation arising from the interactions of threads can be challenging to resolve as diagnosing it requires detailed and precise runtime data collection while imposing limited latency on the system.

Many run-time performance problems arise from contention for shared resources. Understanding the contention issues can be a difficult task using the existing tools and debuggers. Most of these tools are based on prior knowledge about source code that is almost impossible to apply to large multi-threaded applications. For instance, DTrace provides summary statistics from events as they are generated [1]. Although it can reduce the amount of recorded data, it is difficult to predict the needed statistics ahead of time. Later work proposed strategies for gaining insight into performance losses due to lock contention [2]. Huang et al. [3] proposed a Hardware-assisted Lock profiling mechanism (HaLock). Bin Nisar et al. [4] presented a novel-scheduling technique, shuffling, as an efficient approach to mitigating the loss of performance in the applications employing high lock-contention. There exists other work with a focus on high-level performance evaluation of multi-threaded applications that measure some performance parameters in user space. Anderson et al. [5] showed that simple methods of spin-waiting for mutual exclusion to facilitate the access to shared resources degrade overall performance when the number of spinning processors increases. In addition, they demonstrated that performance can be improved by handling spinlock requests.

Although these solutions can detect some performance issues, they cannot be easily used for large multi-threaded applications without accessing their source code. In addition, previous approaches are usually applicable only in either user level or kernel level, and cannot investigate a problem occuring in various levels together, which is the case in many contention related problems.

Dynamic analysis is a mechanism to extract multiple statistics about the runtime executions, usually without modifying or recompiling source code. Dynamic analysis tools usually provide ways to describe what data to gather, how to manipulate, model, and present data to the user. This technique can be used to diagnose performance issues in multi-threaded applications without having knowledge of the application's source code [6]. Since dynamic analysis (through execution traces) can provide comprehensive insight into various system levels from kernel level [6] to user level [7] and consequently leads to the detection of many performance bottlenecks and degradation, it is broadly used among developers and analysts [8–10].

In this paper, we propose a multi-level execution trace-based approach for lock contention analysis. Using this dynamic analysis method, we can precisely detect the time of locking access to shared objects, the time of waiting for getting a lock to access the critical sections, and the critical section execution time. To do so, thread dependencies need to be caught. Giraldeau et al. [11] presented a critical path algorithm to catch the waiting dependencies between different running threads. But, their solution works only on kernel level based on execution trace data collected from the operating system execution and does not support other layers. There are, however, lock methods working only in userspace level which can not benefit from that work. In this paper, we propose a multi-level critical path algorithm to deploy our analysis for different levels of system supporting various types of locking methods, regardless of whether they work in the kernel level or userspace level. Using the proposed method, several studies can be made for performance problems, fairness, and waiting analysis of the different types of user level or kernel level locks. While we analyze lock-contention of applications written in C++ and PHP in this paper, the methodology is not limited to those languages. This is because data collection is specified using environment variables to generate wrappers, allowing us to trace pre-compiled libraries of any language.

Our main contributions in this paper are: **First**, we propose a multi-level critical path analysis algorithm to capture threads dependencies causing lock contentions in order to show which thread holds the lock and which threads are waiting to get the

978-1-7281-7735-9/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSREW51248.2020.00068

lock, at any moment of the execution. **Second**, we present tools and visual views alongside with the proposed algorithm in order to visualize lock contention problems from different aspects.

The rest of this paper is organized as follows: Section II presents a summary of other existing approaches in the performance evaluation of multi-threaded applications. In section III we explain the motivation and methodology of our analysis. In section IV, we evaluate the efficiency and usefulness of our approach for a unified contention analysis. Section V concludes the paper and presents our future plan.

II. Related Work

Because contention on shared resources can cause degradation in throughput of a multi-threaded program, identifying the specific parts of the program which make large latency of whole execution is noteworthy. Without contention detection of a multi-threaded program, it is not feasible to completely exploit the advantages of parallel processing. Most multi-threaded programs often lose performance benefits due to the inefficiency of source code that is not detectable for programmers.

Pusukuri et al. [12] showed that shuffling can improve the performance of multi-threaded applications with high-lock contention. Shuffling aims to reduce the variance in the lock arrival times of the threads which are scheduled on the same socket. However, it shows a minor performance improvement in systems running several multi-threaded applications with high lock-contention.

Hence, Bin Nisar et al. [4] proposed a novel-scheduling technique as an improvement to shuffling [12]. They proposed per-application thread grouping to reduce the lock contention due to imbalance thread mapping in parallel applications. The authors addressed the performance degradation of multi-threaded applications due to the high penalty cost of cache misses.

Tallent et al. [2] proposed and evaluated three approaches for obtaining information about performance losses due to lock contention. Their approaches move from blaming lock contention on victims, then to suspects, and finally to perpetrators.

In another work, Pusukuri et al. [13] presented ADAPT, a scheduling framework that continuously monitors the resource usage of multi-threaded programs. They developed ADAPT as a platform for effective scheduling of multi-threaded programs on multi-core machines.

Lozi et al. [14] proposed a new lock algorithm, Remote Core Locking (RCL), which aims to improve the performance of critical sections in legacy applications on multi-core architectures. RCL replaces lock acquisitions by optimized remote procedure calls to a dedicated core. RCL accommodates blocking within critical sections and nested critical sections as well. The design of RCL addresses both access contention and locality.

These solutions, while effective, usually need to access the source code, and often require prior knowledge about the underlying problem. Furthermore, some of the existing techniques ignore the interaction of threads at run time which potentially leads to performance bottlenecks due to lock contention. In this paper, we address solutions to analyze all lock contentions in software programs without having any knowledge about them, without having their source code, and without modifying the original program.

We use a method based on the critical (active) path of the lock-contention scenarios. To extract the critical path of each thread, we rely on the algorithm introduced by Giraldeau et al. [11]. They have proposed an algorithm to extract the active execution path across threads running possibly on distant machines. Their method extracts various execution states for each thread, including running, interrupt handling, waiting for disk, waiting for a network, waiting for a timer, and waiting for another task. The algorithm first builds an execution graph showing all interactions between threads. Although interesting, they only support execution graphs for the trace events collected from the operating system level. This does not support contentions or dependencies (e.g. in spinlock) which happens totally in userspace level where there is no notion of their execution in the kernel. In this paper, we extend the method introduced by Giraldeau et al. [11] to capture any dependency between running threads at various levels of the system including the userspace, kernel, and network, etc.

III. Contention Analysis

A. Motivation

Contention analysis is necessary to obtain a comprehensive view of large multi-threaded applications when troubleshooting for performance. The problem is that performance counters (in their current form, as used in Perf [1]) present incomplete information. For example, the performance counters in a large multi-threaded application can be used to examine total wait time over a fixed period of time, across all mutex instances, but they cannot discern which threads were affected most, or which specific mutex instances resulted in large wait time. Therefore, a new approach that collects and aggregates relevant information for use in contention analysis and therefore in performance evaluation and debugging is required.

We have analyzed the performance of two releases of TraceCompass [15], a large multi-threaded application which is used for viewing and analyzing trace/log data. Using our method, we perceived a lock contention problem that we did not expect initially. Looking at the source code, we could find the cause of the contention problem and solve it in a new release. This motivated us to propose a unified method to handle contention analysis using execution tracing.

In addition, programmers and developers cannot be aware of what is exactly happening in their multi-threaded programs when it leads to performance degradation and latency. This is more remarkable when we encounter some difficulties in large applications like web browsers. Hence, we will show how the program displays the critical section and contention

[1] https://perf.wiki.kernel.org/index.php/Main$_Page$)

using visual graphs and charts to get a deep insight in multi-threaded applications which can inform developers on how the threads actually run in the program.

B. Data Gathering

In this paper, we use version 2.10 of Lttng[16], a powerful low impact tracing tool, and version 4 of TraceCompass[15], an open-source trace analysis and visualization tool.

Some lock methods like mutex and semaphore are only implemented in kernel space *(syscall-futex)* in order to block and wake up the critical section. However, some methods like spinlock use only user-space. Therefore, we decided to record traces in both the kernel and user spaces to support a comprehensive analysis. As some lock methods utilize futex system call to interact with the kernel, we have instrumented futex methods to be able to investigate lock-based system calls to get details of when they are touched. The events that are created/used to collect trace data are reported in Table I.

Table I
THE EVENTS OF A FUTEX SYSTEM CALL

TP provider name	TP name	Description	Instrumented function
lttng_ust_pthread	pthread_mutex_lock_req	The thread requests to get mutex object	pthread_mutex_lock
lttng_ust_pthread	pthread_mutex_lock_acq	The thread acquires the mutex object	pthread_mutex_lock
lttng_ust_pthread	pthread_mutex_unlock	The thread unlocks the mutex object	pthread_mutex_unlock

In the user space level, we have instrumented Pthreads library using LD_PRELOAD technique. LD_PRELOAD is an optional environmental variable that contains one or more paths to shared libraries or shared objects, loaded before any other shared library, including the C runtime library (libc.so). As a result, program behavior can be non-invasively modified, i.e. a recompile is not necessary. Our preloaded events are pthread_mutex_lock_req, pthread_mutex_lock_acq, pthread_mutex_unlock, pthread_spin_lock_req, pthread_spin_lock_acq, pthread_spin_unlock, sem_wait_req, sem_wait_acq, and sem_post. When we preload the lttng-pthread-wrapper shared object, it replaces the functions listed in the Table I by wrappers which contain tracepoints and call the replaced functions.

C. Data Model

Our method is a stateful method for which we collect and maintain state values during the period for which the analysis is undertaken. In this section, we explain how the collected raw trace events are converted to state values. Each state can be defined as the time duration between two events. We store all the states in a state database system [17], an efficient tree-based data structure to store a history of different state values of system attributes (i.e., processes, CPUs, disks, etc.).

Using an event-to-state mapping method, also called the State Provider, we convert the trace events into state values. the state provider checks the type and contents of all events in the trace and updates the state system. As a case in point, we change the value of a thread to "in a futex lock" state value when we see the *syscall_entry_futex* trace event. For each relevant event defined in the event mapping table, the State Provider is called and updates the state value in the state database.

We also define a general term, "attribute", as the fundamental part of our state system, which can include just one state value at any given time within the execution. Each attribute can represent all or part of a system resource (a thread, file, CPU, etc.), and a resource may have several attributes to characterize its different aspects. To manage a huge number of related attributes, we organize them in a tree-shape structure called an attribute tree. The attribute tree is an in-memory data structure of the attributes of the system resources. For complex systems, as the number of attributes increases, putting them in a tree structure leads to a better-structured organization and, depending on the type of queries, better performance. In the attribute tree, there is only one access path for each attribute from the root node. To have a better understanding of the attribute tree, we compare this structure with the files and directories in a file-system. Directories resemble attribute access paths, and filenames correspond to the names of attributes. In fact, the content of the file represents the different state values of that attribute [17].

When the state provider reports a state change, a transient state record is created for the new attribute and stored in memory, in the current state system. Each transient record involves an attribute name, a time, and a state value. If an entry exists for this attribute, the present value is replaced by the new one. However, the previous one is kept and stored in the state history for use in the next inquiries. So, we first set one interval record from the available information which includes an attribute (from the attribute tree), the old status value, the old time value, and the new value of time (interval end time). The completed interval can now enter into the state history to consider a part of the history. This process will be continued until the end of trace, during which we gather a set of completed intervals for each attribute. As mentioned before, the history state values are kept in a tree-based structure which is named "state system" [17].

As an example, we apply the above-mentioned architecture to a multi-threaded program running with 3 threads to query and visualize the state of threads in each time interval. We use thread ID (TID) as an attribute and define different states of threads including running, waiting, blocked, futex, gettid and etc. We store states in the state system in order to facilitate finding thread status in each desired time. Figure 1 depicts state system structure when a thread is obtaining a mutex lock in order to execute its critical section. We define two different states named "wait for lock" and "running critical section" for each thread between events "pthread-mutex-lock-req", "pthread-mutex-lock-acq", "pthread-mutex-unlock", respectively.

D. Multi-level dependency Analysis

In this section, we propose a multi-level critical path algorithm to extract and identify the waiting relationships between the interacting threads in a lock contention scenario. Giraldeau et al. [11] presented a critical path algorithm that relies totally

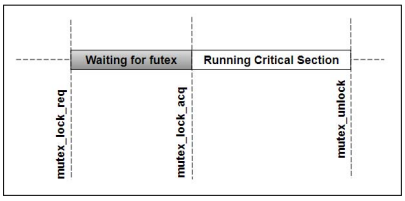

Figure 1. State system structure for a thread which is obtaining a mutex

on the events extracted from the kernel events. Because of this, it is not usable for the lock methods that are implemented and supported totally or partially in user level. Spinlock is one case in which lock Acquire and Release methods are implemented in user level. As shown in Figure 2, the critical path analysis of Giraldeau et al. [11, 18] does not reveal any details regarding the execution of threads having lock contention using the spinlock method. The green color in Figure 2 only shows the threads running in userspace, without revealing the details on when they access the lock or when they wait to acquire the lock.

Figure 2. Spinlock used to manage the shared resource accesses of two threads

Our proposed method performs multilevel critical path analysis using the execution trace events collected from the different levels of the system in order to show all running details about threads' dependencies using various user level or kernel level lock methods.

Algorithm 1 Execution Graph Construction algorithm, for events collected from lock library . **Input Input:** Trace T, Threads = {t1, t2, ..., tn}
,
Set1 = {*_lock_req},
Set2 = {*_lock_acq},
Set3 = {*_unlock}
Output Output: execution graph G

1: $THREADS \leftarrow$ initial set of threads active in the multi-threaded application Declarations
2 : $LOCKHOLDER \leftarrow \emptyset$
3 : **for all** thread t $\in$ THREADS **do**$//Initialization$
4 : Create initial vertex of thread t with timestamp t.begin
5 : **end for**
6 : **for all** event e $\in$ T **do**$//Mainprocedure$
7 : **if** e $\in$ Set1 **then**
8 : new_h_edge(e.tid, e.ts, blocked)
9 : new_v_edge(e.tid, LOCKHOLDER)
10 : **end if**
11 : **if** e $\in$ Set2 **then**
12 : new_v_edge(LOCKHOLDER, e.tid)
13 : new_h_edge(e.tid, e.ts, running)
14 : LOCKHOLDER $\leftarrow e.tid$
15 : **end if**
16 : **if** e $\in$ Set3 **then**
17 : LOCKHOLDER $\leftarrow e.tid$
18 : **end if**
19 : **end for**

We extend the algorithm proposed in Giraldeau et al. [11]in order to support dependencies between threads occurring in various system levels. Algorithm 1 builds the execution graph using data captured from lock-related events from either the user or kernel level. Set1, Set2 and Set3 indicate events for lock request, lock acquire and unlock occurring, respectively. This algorithm either builds an execution graph for the running threads, or adds some horizontal or vertical nodes to an existing execution graph.

The execution graph is a two-dimensional sparse graph with labeled edges. Horizontal edges denote a thread's running state changes, while vertical edges indicate links between threads capturing their blocking dependencies. This graph is able to reconstruct the dependencies and relationships between threads participating in a lock contention scenario.

The execution graph captures all dependencies between running threads, however, we look for critical (active) paths along with this execution graph. The critical path is extracted by recursively replacing the waiting edges of a thread with the edges of the waking thread. To extract the critical path of each thread, we rely on the algorithm introduced by Giraldeau et al. [11]. The input to the algorithm is the execution graph, regardless of whether it is created in the kernel or userspace and the output is an active path within that graph.

Using the execution graph and the extracted critical path, different analyses can be made concerning lock contention problems including possible latency problems, fairness analysis, and blocking behavior of the different types of locks. Our analysis comes along with a few analysis views built on top of Trace Compass, which will be explained in the next section.

E. Visualization

In this section, we present the views to visualize the hierarchical trace data collected from the various levels of the system and its relevant lock-contention analysis. We have implemented several views, including wait-block, time-line, flame graph, and critical flow view as a plugin in the Trace Compass. [15] tool.

Figure 3 demonstrates wait-block timeline for a program run with three threads. The view shows how the threads are blocking each other during a lock contention, and how by entering a thread to the critical section, the other threads are blocked until the mutex is free to be taken.

Flame Graph view allows the user able to distinguish the executed code path. Figure 4 shows the flame graph view for a program with three threads. Each entry in the flame graph shows a combination of all the calls to a function in a specified depth of the call stack with the same caller. Running a simple

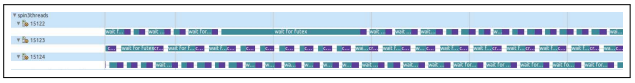

Figure 3. Wait-block analysis view for a multi-threaded program with 3 threads

program with three threads can show us the thread holding the shared object (mutex) while the two other threads wait to obtain the mutex. As shown in Figure 4, thread 15122 takes the mutex at the beginning of program execution, and the other thread wait for obtaining the mutex. Threads 15124 and 15123 take the mutex after running thread 15122. If contention exists in a multi-threaded program, it is easy to detect contention-causing bottlenecks using a flame graph view.

Figure 4. Flame graph view for a multi-threaded program with 3 threads

Figure 5 shows the critical flow view for a program running with three threads. The critical flow view is used to analyze the contention and the communication of participation threads in more detail. It shows execution dependencies between interacting threads to accomplish a task. This gives a detailed execution view of threads while removing unrelated information. In this case, we can see blocking/preemption caused by thread priorities on taking and releasing a lock.

Figure 5. Critical flow view for a multi-threaded program with 3 threads

As shown, our algorithm (as shown in Figure 2 reveals additional data that was not available in Giraldeau's algorithm (Figure 5).

IV. Evaluation and Use Case

A. Performance Evaluation

In the proposed technique, we use both the kernel level and user level trace collection and trace processing for extracting runtime data. For the user level tracing, the overhead is less than .7% of the execution time, as it traces only a few lock-related events (i.e., lock_acquire, lock_request and unlock). This is calculated for a C++ application using the PThreads library to implement spinlock and semaphore lock methods. For the kernel level, each enabled kernel event can cause an overhead up to 10 ns and in total it imposes up to 25% of the execution time depending on how many tracing events are enabled. When all kernel events (which are not used in our method) are enabled, the slowdown goes up to 25%, however, with a minimal tracing (events necessary for conducting lock-contention analysis, the case of our method) the overhead is 7%. This includes tracing the cpu scheduling events, system call events (e.g., futex system call), interrupt and timer events.

On the other hand, the analysis of the collected trace is processed offline. The most time-consuming part is converting the event data to state values. The state values which are stored in the state system are then used to populate the analysis views. Figure 7 shows the time required to process trace events and construct the attribute tree and state database, for different trace sizes. Not surprisingly, the fastest time is for the case where the system only reads the trace events without parsing and analyzing them. The second fastest case is when the system reads the trace, analyses them, extracts states but stores them only in memory and not on disk, therefore, no disk I/O is involved. The worst case is when the state database is fully built and written to disk.

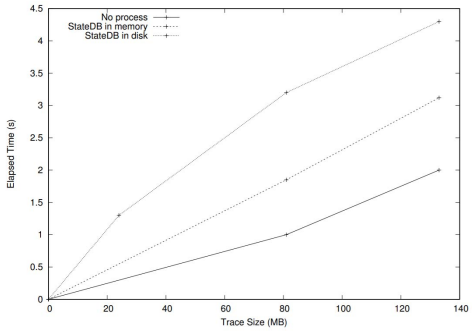

Figure 6. Trace processing and analysis time

B. Usecase: Apache Contention Analysis

Apache is an open-source web server application. Apache pushes jobs to worker threads in order to increase scalability and overall efficiency. As each worker thread exists independent of another, threads avoid communication with each other. Contention is theoretically avoided as each thread is designated its own resources. However, we perceive extra latency when observing the handling of concurrent server requests.

As before, we are able to capture the system's state and activity in threads, including the time each threads spend waiting for lock contention. This enables us to confirm and isolate latency occurring from contention. To avoid race conditions, Apache uses filelock, a locking mechanism implemented in user space. With our multi-level critical path method, we are able to capture the required user-space events and identify intervals of processes waiting to acquire a lock.

As shown in Figure 5, we are able to isolate and locate areas of contention and identify the source. In this case, we determine that the cause of the anomalous latency stems from its engine source code. Apache employs OPcache, a mechanism to cache compiled scripts to improve performance. When a script is submitted for compilation, OPcache checks to see if the bytecode already exists in the cache. This allows a process to run scripts while avoiding costly recompilation. However, other processes must wait before writing into the

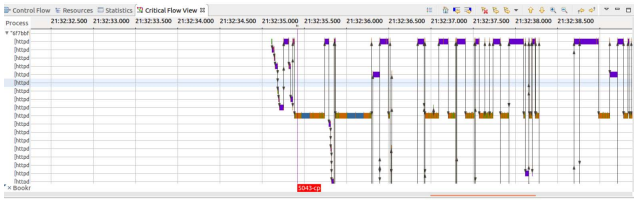

Figure 7. Analysis of cache write contention in Apache Web servers

shared cache space, leading to contention for cache memory. Because of this, one process is able to hold up other processes during concurrent script execution.

V. Conclusion

In this paper, parallel efficiency and the bottlenecks consisting of lock contention problems was explored. The source of information is an execution trace collected from several levels in a system to give a more comprehensive understanding of runtime in user space combined with a critical path analysis that pinpoint the lock contention problem. The research explains a new vertical view of a software stack by using dynamic tools. Uses cases described show that the proposed method is working, and that the performance penalty is in reasonable level; around 7% for the lock contention problem.

Acknowledgement

We would like to thank Ciena, the Natural Sciences and Engineering Research Council of Canada (NSERC), EfficiOS, Ericsson and Google for their financial support. We also thank Geneviève Bastien for her help and comments.

References

[1] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.

[2] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. *SIGPLAN Not.*, 45(5):269–280, January 2010.

[3] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. Halock: Hardware-assisted lock contention detection in multithreaded applications. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 253–262, Sept 2012.

[4] U. B. Nisar, M. Aleem, M. A. Iqbal, and N. S. Vo. Jumbler: A lock-contention aware thread scheduler for multi-core parallel machines. In *2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, pages 77–81, Jan 2017.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan 1990.

[6] Naser Ezzati-Jivan and Michel R. Dagenais. A stateful approach to generate synthetic events from kernel traces. *Adv. Soft. Eng.*, 2012:6:6–6:6, January 2012.

[7] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, July 2006.

[8] Christopher LaRosa, Li Xiong, and Ken Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC 08, pages 880–885, New York, NY, USA, 2008. ACM.

[9] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Online system problem detection by mining patterns of console logs. In *2009 Ninth IEEE International Conference on Data Mining*, pages 588–597, Dec 2009.

[10] M. Meyer and L. Wendehals. Selective tracing for dynamic analyses. In *Proc. of the 1st Workshop on Program Comprehension through Dynamic Analysis (PCODA), co-located with the 12th WCRE, Pittsburgh, Pennsylvania, USA*, 2005.

[11] F. Giraldeau and M. Dagenais. Wait analysis of distributed systems using kernel tracing. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2450–2461, 2016.

[12] K. Kumar, P. Rajiv, G. Laxmi, and N. Bhuyan. Shuffling: A framework for lock contention aware thread scheduling for multicore multiprocessor systems. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 289–300, Aug 2014.

[13] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[14] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[15] Trace Compass. https://projects.eclipse.org/projects/tools.tracecompass,, 2018. [Online; accessed 2020-04-12].

[16] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. *OLS (Ottawa Linux Symposium)*, 01 2006.

[17] Alexandre Montplaisir, Naser Ezzati-Jivan, Florian Wininger, and Michel Dagenais. Efficient model to query and visualize the system states extracted from trace data. pages 219–234, 09 2013.

[18] Francis Giraldeau, Michel R Dagenais, and H Boucheneb. Teaching operating systems concepts with execution visualization. In *121st ASEE Annual Conference and Exposition, Indianapolis, IN, USA*, 2014.