No such thing as a bad workload; only unsuitable system calls?

Arjun Balasubramanian

Department of Computer Sciences, University of Wisconsin-Madison

Abstract

This work presents an insight into how subtle intricacies between system calls in UNIX-based systems play an important role in the performance of an application. We compare two sets of aspects - (I) Buffered I/O vs Direct I/O (II) Mutexes vs Spin-locks. Through carefully crafted workloads that emulate actual applications, we compare the performance of these OS techniques. Our experiments show huge differences in performance depending on the kind of workload. As a contribution, we provide a set of learnings to developers on the appropriate use of these seemingly similar OS techniques.

1 Introduction

UNIX-based systems provide system calls as an interface to interact with the operating system. The POSIX standard has ensured a uniformity in interfaces and helped in the portability of user applications across different operating systems. The Linux man page provides comprehensive documentation on the semantics of these system calls. In this way, UNIX-based systems provide a huge win on the ease of programmability.

While the ease of programmability is greatly appreciated by application program developers, a difference between different implementations of the same OS aspect often confuse even the most experienced developers. This arises primarily due to the below factors -

(I) Layering in the API call path - Programming in a low-level language like C or C++ gives developers the illusion that they are directly interacting with the kernel. However, applications usually interface with the GNU C Library (glibc), which in turn interacts with the kernel. The presence of the glibc layer between the application and the kernel has some implications with respect to certain system calls. For instance, vDSO (Virtual Dynamic Shared Object) is a small library that the kernel automatically maps into the address space of a process. With vDSO, certain system calls like gettimeofday() are optimized to avoid the context-switch overhead of switching from user mode to kernel mode. If a developer tries to use a utility like strace to measure the number of times that gettimeofday() is called by an application, he/she would be unsuccessful due to vDSO.

(II) Simply too many options - Courtesy of being open-source, Linux must cater to a wide audience and consequently has multiple variations of the same feature that significantly vary from each other. These implementations usually come in the different flags that developers can specify through the system call. For instance, even the simple mount() system call comes with four different flag options. Hence, while using the API is simple in UNIX-based systems, the variations offered by these flags often make it difficult and time-consuming to get a full understanding. As a result, many of these options can be misunderstood or not utilized appropriately, which might result in severe performance-related issues.

While the first point might impact only advanced users of UNIX-based systems, the second point could potentially impact developers who are not very well acquainted with the internals of the system. Consequently, in this work we show how different implementations of the same OS feature can have different performance based on the workload generated by the application. In this work, we focus on two sets of frequently used OS features – (I) File I/O and (II) Locking Mechanisms.

File I/O is used extensively by almost all applications. Though there are many facets, we focus on comparing Buffered I/O and Direct I/O in this work. With Buffered I/O, the data written is buffered in the page cache and the

actual I/O to the disk is done much later. This allows the filesystem to perform larger writes and fully exploit the available disk write bandwidth. In contrast, Direct I/O immediately has the data flushed out to the disk without doing any kind of buffering. It is easy to notice that the two approaches are completely orthogonal to each other and consequently one performs poorly while the other performs well in certain scenarios. Section 2 presents more theory on the difference and benchmarks the two approaches under different workloads.

Mutli-threaded programs extensively utilize locking mechanisms to synchronize access to shared structures among different threads. Linux offers two flavors of locking – Mutexes and Spin-locks. A spin-lock is a lock that causes a thread trying to acquire it to continuously spin or check in a loop for the availability of the lock. On the other hand, a mutex would first try to acquire the lock. If the lock cannot be acquired, the thread is put to sleep and is woken up by the OS when the lock becomes available. As in the case of file I/O these two approaches offer different performance under different scenarios. Section 3 covers more on the internals of these two types of locks and benchmarks the two approaches under different circumstances.

All our experiments are performed on a c220g5 node in CloudLab having two Intel Xeon Silver 4114 10-core CPUs. We utilize a 1TB SAS Hard Drive mounted using ext4 file system. We verified that journaling was enabled, and the block size was set to the default of 4kB. Our benchmarks are open-source and can be viewed at https://github.com/Arjunbala/Advanced-OS-MiniProject.

2 Buffered I/O vs Direct I/O

2.1 Theoretical Comparison

By default, UNIX utilizes Buffered I/O to serve file I/O requests. Application developers can use Direct I/O by passing in the O_DIRECT flag during the open() system call. The buffering offered by Buffered I/O gives benefits during both reads and writes. Buffered I/O offers good write performance since the write system call would simply copy the data from the userspace buffer into the page cache and return. The filesystem flushes dirty pages to the disk much later. This results in larger write requests to the disk which results in better utilization of the disk's write bandwidth. With the presence of page caches, Buffered I/O increases the probability of read requests being served out of cache rather than having to make a read request to the disk. Further, modern filesystems exploit the locality of access to perform read-ahead and prefetch data into the page cache before it is even accessed.

In contrast, Direct I/O directly issues I/O requests to the disk. Direct I/O transfers are initiated between the userspace and the disk controller, completely bypassing the kernel buffers. This behavior reduces the overheads of copying data to/from kernel buffers. Since copying is avoided, it also reduces the CPU utilization. On the flip side, using Direct I/O takes away the benefits of caching. As a result, Direct I/O results in worse write and read performance in most circumstances. However, there are a few applications that maintain their own caching logic in userspace and use Direct I/O to establish fine-grained control over what is cached.

2.2 Experiments

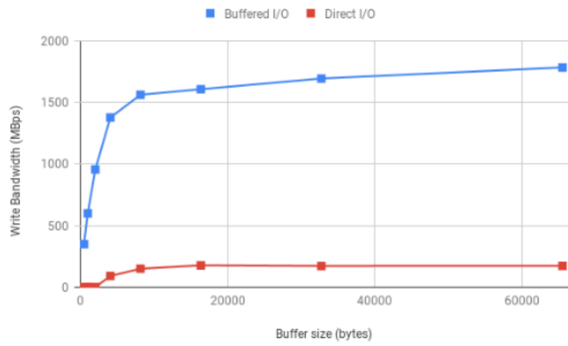

Figure 1: Variation of sequential write bandwidth with buffer size

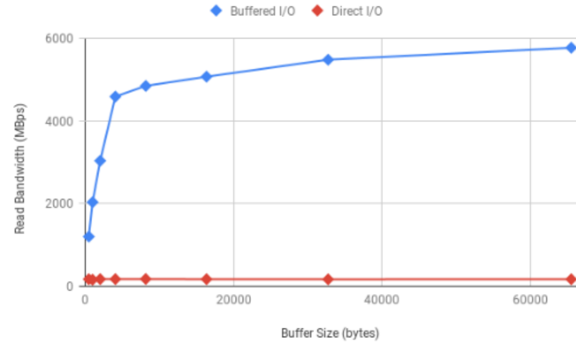

Figure 2: Variation of sequential read bandwidth with buffer size

To start out with, we benchmark the relative performance of sequential reads and writes using Buffered I/O and Direct I/O. In our experiment, we have a synthetic workload that spawns a child process and writes out a 100 GB file using Buffered I/O. Once the child process has finished writing, the parent process reads the same file using Buffered I/O. We also configure the same workload to use Direct I/O instead of Buffered I/O. This kind of workload would occur in frameworks like MapReduce when a mapper writes out intermediate data to the disk and a reducer on the same machine reads in the intermediate data. We also wish to capture the impact of the size of reads and writes and hence vary this setting in the workload as well. The results are presented in Figure 1 and Figure 2. For both Buffered and Direct I/O, the throughput increases with an increase in the block size. This largely boils down to the reduced cost of copying in Buffered I/O. For Direct I/O, we notice that the bandwidth approaches a steady state, which can be attributed to storage hardware configurations like the block size. We observe that writes have a larger sequential bandwidth with Buffered I/O than Direct I/O, which follows from the fact that writes are done to the faster page cache in case of Buffered I/O. The extent to which Buffered I/O performs better varies with the buffer size. It varies from around 200x for a buffer size of 512 bytes to about 6x for a buffer size of 65536 bytes. The difference reduces because the benefits of buffering (accumulating writes before performing disk I/O) are more pronounced when writes are smaller. With respect to the read bandwidth, we observe that it is more efficient to do larger reads. Buffered I/O performs at least 200x times better than Direct I/O for the same block size. This happens because the workload performs sequential accesses. Both page caching and read-aheads contribute to the higher read bandwidth observed for Buffered I/O.

Next, we consider another workload which is similar to the one presented above expect for the fact that an fsync() is issued after each write when Buffered I/O is used. Applications like SQLite or PostgreSQL would exhibit this kind of a workload because they use write-ahead-logging for consistency. The write and read bandwidths observed are presented below -

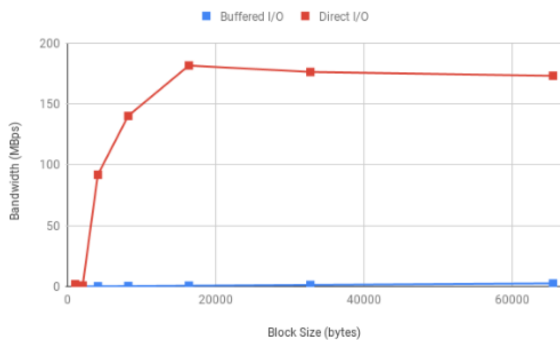

Figure 3: Variation of write bandwidth with buffer size when an fsync is issued after every write

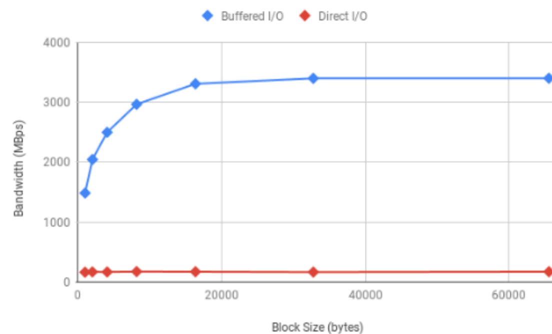

Figure 4 : Variation of Read Bandwidth with buffer size when every write was accompanied by an fsync call

For the above workload, we observe that Direct I/O has a much better performance than Buffered I/O when the writes are accompanied by fsync() calls. This is due to the involvement of the CPU, the additional overheads incurred in copying data to/from kernel buffers, and possibly due to other barriers imposed by the fsync() call. This

overhead is so large that it cumulatively leads to Direct I/O being about 100x better on an average compared to Buffered I/O. Sequential reads performed immediately after the write perform better using Buffered I/O due to page caching and read-aheads.

Next, we shift our focus to the performance comparison of random writes. For this benchmark, we perform overwrites at randomly generated offsets in a file using Buffered and Direct I/O. We use the lseek() system call to move the file pointer to a given position and then we issue a write. Initially, we had planned to do this benchmark for a file size of 1GB and a file size of 100GB. For files whose pages could entirely fit in memory, we thought that Buffered I/O would do better since the writes would be done only to the buffer. For files larger than the available memory, we thought that a high number of page faults would make Buffered I/O perform worse than Direct I/O. However, the empirical results did not reflect this thought and we observed similar write bandwidths for both the 1GB and 100GB file. For sake of brevity, we have omitted the 1GB results in our graph below.

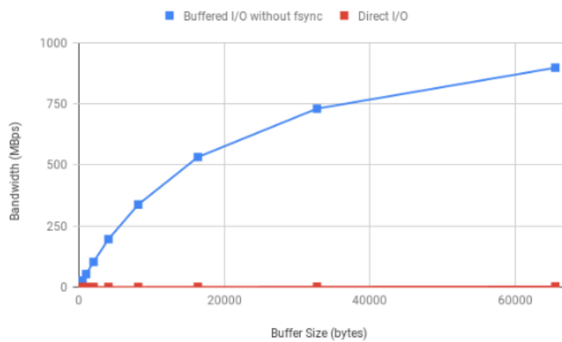

Figure 5 : Variation of write bandwidth with buffer size for random writes

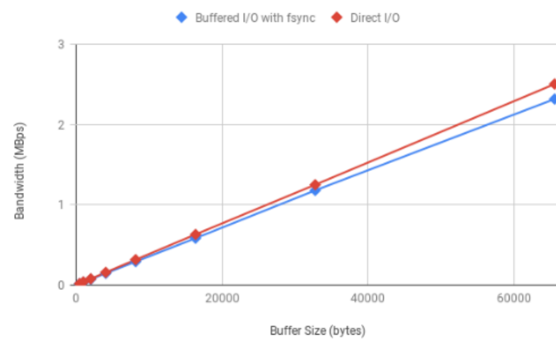

Figure 6 : Variation of write bandwidth with buffer size for random writes accompanied by fsync calls

Figure 5 compares random write bandwidths between Buffered I/O and Direct I/O. It is surprising to observe that on an average Buffered I/O performs about 1000x times better than Direct I/O for random writes. To understand this behavior better, we measured the time taken for each write call. We expected the timings to have a bimodal distribution since writes to pages already in cache would be served a whole lot faster than writes to pages not in the cache due to page faults. We noticed that our workload had resulted in about 70% cache hits which explains the huge bandwidth difference observed. Figure 6 shows the results when random writes are issued along with fsync() calls. As expected, Direct I/O performs better than Buffered I/O due to overheads of CPU involvement and copying incurred in Buffered I/O.

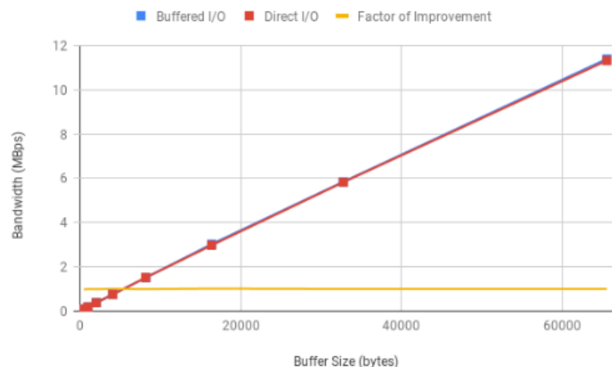

Figure 7: Variation of bandwidth for random reads with buffer size

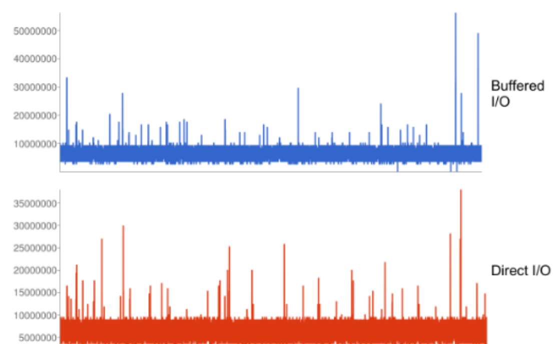

Figure 8: Individual read call timings in nanoseconds for random reads issued using Buffered I/O and Random I/O

On similar lines, we benchmark the performance of random reads with Buffered I/O and Direct I/O. From Figure 7, we observe that the bandwidths are almost the same. Direct I/O slightly edges it as indicated by the factor of improvement line. Though Direct I/O would avoid the overheads of copying data between userspace and kernel buffers, we were initially expecting Buffered I/O to perform better. Even though the requests are random, we were thinking that there would still be enough cache hits for Buffered I/O to outperform Direct I/O. To get further insight, we recorded the individual read times as shown in Figure 8. From the graph, we observe similar access times for both Buffered I/O and Direct I/O indicating that cache hits were not as frequent as we thought.

2.3 Learnings

(I) For workloads involving writes followed by reads to the same offsets, it would be better to use Buffered I/O over Direct I/O due to benefits of page caching. The benefit is amplified if the reads are sequential due to the read-ahead logic that filesystems use.

(II) For workloads requiring perfect ordering or durability guarantees after every write, it would be better to use Direct I/O since fsync() performs very poorly with Buffered I/O from the above experiments.

(III) For random workloads on huge files, there is not much difference between using Buffered I/O and Direct I/O.

3 Mutex vs Spin-lock

3.1 Theoretical Comparison

The Linux OS provides two variations of locking mechanisms – Mutexes and Spin-locks. Both mutexes and spin-locks allow competing threads to obtain exclusive access to shared resources and execute their critical sections. Linux provides the pthread_mutex_lock() and pthread_mutex_unlock() primitives for mutexes and the pthread_spin_lock() and pthread_spin_unlock() primitives for spin-locks.

The exact mechanisms by which locking happens vary considerably. If a thread is unable to acquire a mutex at a given moment, the thread gets added to a mutex queue and other threads can execute on the same hardware thread until this thread is granted access to the mutex lock. On the other hand, if a thread is using spin-locks, the thread will not context switch, but would instead continuously spin (check whether the lock can be acquired) until the lock is acquired.

From the mechanisms, it is obvious that spin-locks consume more CPU cycles compared to mutexes and block the execution of other threads. Hence, it wouldn't make sense to use spinlocks in uniprocessor systems. The real benefit of spin-locks come in multi-processor environments. Spinlocks use a two-instruction loop consisting of an atomic instruction that tries to set a particular memory location to a particular value until it succeeds. Mutexes utilize a lot of CPU instructions to add the thread to a queue and wake it up. In fact, if the wait time is very small, the overheads in mutexes can themselves exceed the actual wait time. In such situations, spinlocks are preferred. However, if the wait time is longer, mutexes are preferred since spinning would waste the CPU.

3.2 Experiments

To start out with, we have a simple program consisting of two threads contending for the same lock. We have a synthetic critical section that executes keeps the CPU busy for a fixed time. To issue a sleep of x µs, we simply run a sleep of 1µs x times in a loop. The performance for the above workload for different values of x using both spin-locks and mutexes is captured below -

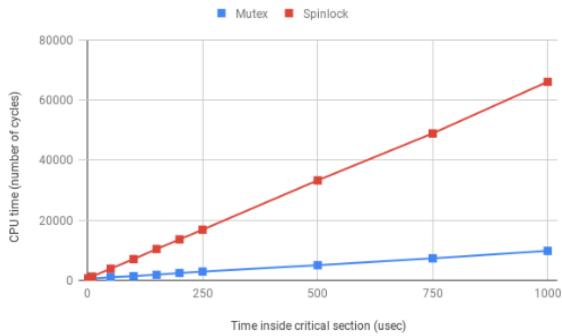

Figure 9: Comparison of CPU times for mutexes vs spinlocks with different critical section times

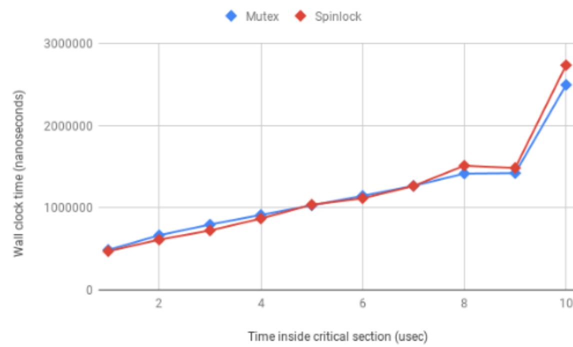

Figure 10: Comparison of wall clock times for mutexes vs spinlocks with different critical section times

From Figure 9, we see that spinlocks have a drastically larger CPU time than mutexes which is due to the CPU cycles spent on spinning. However, from Figure 10 we notice that for small critical section times (< 6 µs), spinning helps reduce the wall clock time. This directly correlates with the benefit we had expected from theory.

Looking at the large difference in CPU time between spin-locks and mutexes, we believe that it is a better decision to use spinlocks when the threads are not competing for CPU cores. To validate this, we construct a synthetic workload where we spawn N threads, numbered 0 to N-1. Each thread is tied to a particular processor in a round-robin fashion. For the purpose of this benchmark, we assume that there are 8 cores, numbered 0 to 7. Since we use round-robin, thread I will be set to execute on CPU number I modulo 8. Each thread first performs some work for t1 µs, tries to acquire a lock using a mutex or a spin-lock, performs some work for t2 µs and then releases the acquired lock.

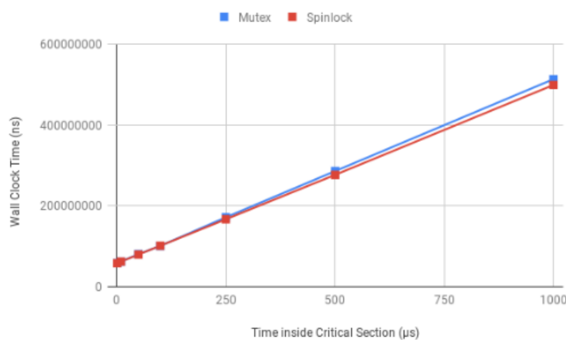

Figure 11: Variation of wall clock time with time spent inside critical section for 8 threads running on 8 different CPUs

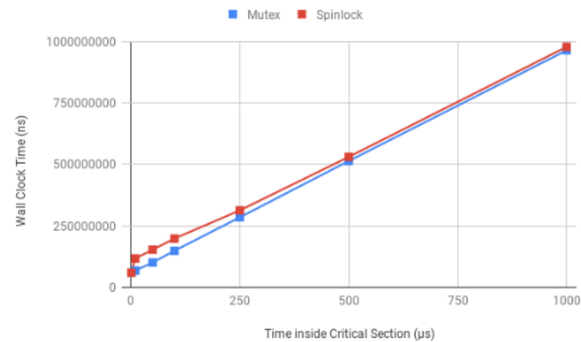

Figure 12: Variation of wall clock time with time spent inside critical section for 16 threads running on 8 different CPUs tied in a round-robin fashion

In our experiment, we first set N=8, so that each thread has its own dedicated CPU to run on. In the second setup, we set N=16, so that two threads compete for execution on the same CPU. We set t1=1000µs and vary t2. Figure 11 and Figure 12 show the wall clock times with these two setups. As expected, increasing the number of threads results in an increase in wall time times for both mutexes and spin-locks. When there is no competition for running on threads, we notice that spin-locks perform better since spinning results in smaller lock acquisition times. However, when threads compete for the CPU, spinning performs worse. This is because spinning prevents the competing thread from completing work while it is waiting for the lock. We do not see much change in the relative performance of spin-locks and mutexes by varying the value of t1.

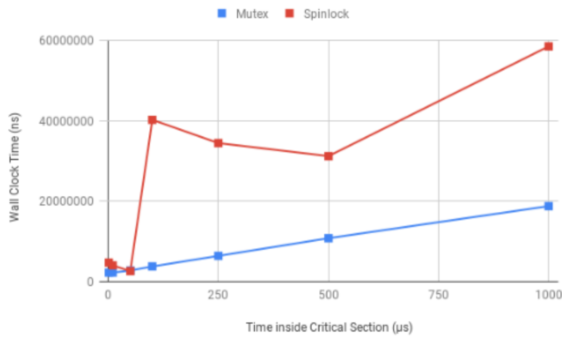

Figure 13: Variation of wall clock time with time inside critical section for 16 threads performing non CPU work of 1µs outside the critical section

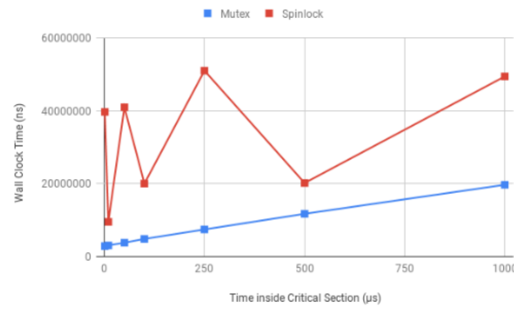

Figure 14: Variation of wall clock time with time inside critical section for 16 threads performing non CPU work of 1000µs outside the critical section

For our next workload, we make a slight tweak - Instead of keeping the CPU busy outside the critical section, we do a sleep to ensure that the CPU is not occupied. This is meant to represent an I/O bound task that is performed before entering the critical section. For instance, there might exist application workloads that read a file and then update shared data-structures in the critical section. From Figure 13 and Figure 14, we notice that spin-locks perform a lot worse that mutexes. The reason is that the spinlock keeps the CPU busy and steals CPU cycles from the other thread wanting to do the file I/O. Hence, even though the other thread spends most of its time in I/O, it is unable to start the I/O due to the spinlock occupying the CPU for long periods.

3.3 Learnings

(I) For applications that extensively utilize multi-threading for performance, the threads would contend for execution on CPU cores. Our results show that it is better to use mutexes for such workloads.

(II) When contending threads wish to perform I/O bound work outside the critical section, mutexes offer better performance over spin-locks.

(III) Spinlocks are generally useful for smaller waits, while mutexes work better for longer waits. It is due to this reason that spinlock implementations today do not spin forever but enter a sleep after some time. Similarly, mutex implementations initially spin and then enter into a sleep.

4 Conclusion

This paper shows that application developers need to have a good understanding various implementations of OS features in order to extract the best performance out of the OS. We illustrate the above by looking at two specific OS features – File I/O and Locking. In File I/O, we show how different workloads can get huge benefits from either using Direct I/O or Buffered I/O. We have summarized our learnings in Section 2.3. In Locking, we show situations where spin-locks perform better than mutexes and vice versa. These learnings have been summarized in Section 3.3

Bibliography