Is the Optimism in Optimistic Concurrency Warranted?

Donald E. Porter, Owen S. Hofmann, and Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{porterde, osh, witchel}@cs.utexas.edu

Abstract

Optimistic synchronization allows concurrent execution of critical sections while performing dynamic conflict detection and recovery. Optimistic synchronization will increase performance only if critical regions are *data independent*—concurrent critical sections access disjoint data most of the time. Optimistic synchronization primitives, such as transactional memory, will improve the performance of complex systems like an operating system kernel only if the kernel's critical regions have reasonably high rates of data independence.

This paper introduces a novel method and a tool called *syncchar* for exploring the potential benefit of optimistic synchronization by measuring data independence of potentially concurrent critical sections. Experimental data indicate that the Linux kernel has enough data independent critical sections to benefit from optimistic concurrency on smaller multiprocessors. Achieving further scalability will require data structure reorganization to increase data independence.

1 Introduction

As CPU manufacturers have shifted from scaling clock frequency to placing multiple simple processor cores on one chip, there has been a renewed interest in concurrent programming. The end-user benefit of these systems will be limited unless software developers can effectively leverage the parallel hardware provided by these new processors.

Concurrent programming in a shared memory system requires primitives such as locks to synchronize threads of execution. Locks have many known problems, including deadlock, convoying, and priority inversion that make concurrent programming in a shared memory model difficult. In addition, locks are a *conservative* synchronization primitive—they always assure mutual exclusion, regardless of whether threads actually need to execute a critical section sequentially for correctness. Consider modifying elements in a binary search tree. If the tree is large and the modifications are evenly distributed, most modifications can safely occur in parallel. A lock protecting the entire tree will needlessly serialize modifications.

One solution for the problem of conservative locking is to synchronize data accesses at a finer granularity–rather than lock an entire binary search tree, lock only the individual nodes being modified. This presents two problems. First, data structure invariants enforce a lower bound on the locking granularity. In some data structures, this bound may be too high to fully realize the available data parallelism. Second, breaking coarse-grained locks into many fine-grained locks significantly increases code complexity. As the locking scheme becomes more complicated, long-term correctness and maintainability are jeopardized.

An alternative to conservative locking is *optimistic concurrency*. In an optimistic system, concurrent accesses to shared data are allowed to proceed, dynamically detecting and recovering from conflicting accesses. A specialized form of optimistic concurrency is lock-free data structures (including many variants like wait-free and obstruction-free data structures) [4, 5]. Lock-free data structures, while optimistic, are not a general purpose solution. Lock-free data structures require that each data structure's implementation meets certain non-trivial correctness conditions. There is also no general method to atomically move data among different lock-free data structures.

Transactional memory [6] provides hardware or software support for designating arbitrary regions of code to appear to execute with atomicity, isolation and consistency. Transactions provide a generic mechanism for optimistic concurrency by allowing critical sections to execute concurrently and automatically roll-back their effects on a data conflict. Coarse-grained transactions are able to reduce code complexity while retaining the concurrency of fine-grained locks.

To benefit from optimistic concurrency, however, critical sections must have a substantial amount of *data independence*—data written by one critical section must be disjoint from data read or written by critical sections of concurrently executing threads. If critical sections concurrently modify the same data, or have *data con-*

Critical Section 1	Critical Section 2	Critical Section 3

begin critical section;
node = root→right;
node→left = root→left→right;
end critical section;
	begin critical section;
	node = root→left;
	node→left = root→left→right;
	end critical section;
		begin critical section;
		node = root;
		node→right = node→left;
		end critical section;

r		w	r		w	r	w
0x1000	0x2064	0x3032	0x1000	0x2064	0x2032	0x1000	0x1032*
0x1032*	0x3000		0x1032*	0x3000		0x1064	
0x1064	0x3064		0x1064	0x3064			
0x2000			0x2000				

Table 1: Three critical sections that could execute on the tree in Figure 1 and their address sets. The read entries marked with an asterisk (*) are conflicting with the write in Critical Section 3.

flicts[1], optimistic concurrency control will serialize access to critical sections. In this case optimistic control can perform much worse than conservative locking due to the overhead required to detect and resolve conflicts.

In this paper, we present novel techniques (Section 2) and a tool (Section 3) for investigating the limits of optimistic concurrency by measuring the data independence of critical regions that would be protected by the same lock. We apply the methodology and the tool to the Linux kernel, and present results (Section 4).

2 The Limits of Optimistic Concurrency

The most general way to determine data independence is to compare the *address sets* of the critical sections. The address set of a critical section is the set of memory locations read or written. If the code in critical sections access disjoint memory locations, the effect of executing them concurrently will be the same as executing them serially. In addition, the address sets need not be entirely disjoint; only the write set of each critical section must be disjoint from the address set of other potentially concurrent critical sections. In other words, it is harmless for multiple critical sections to read the same data as long as that data is not concurrently written by another. This criterion is known as conflict serializability in databases, and it is widely used due to the relative ease of detecting a conflict.

Conflict serializability is a pessimistic model, however, as two critical sections can conflict yet be safely executed in parallel. For example, if two critical sections conflict only on their final write, they can still safely execute concurrently if one finishes before the other issues the conflicting write. Some data structures and algorithms make stronger guarantees that allow critical sections to write concurrently to the same locations safely, but these are relatively rare and beyond the scope of this paper. Such guarantees correspond to view serializability; computing view serializability is NP-complete and hence rarely used in practice [9]. This paper will use conflict serializability exclusively.

[1]We selected the term data conflicts over data dependence to avoid confusion with other meanings.

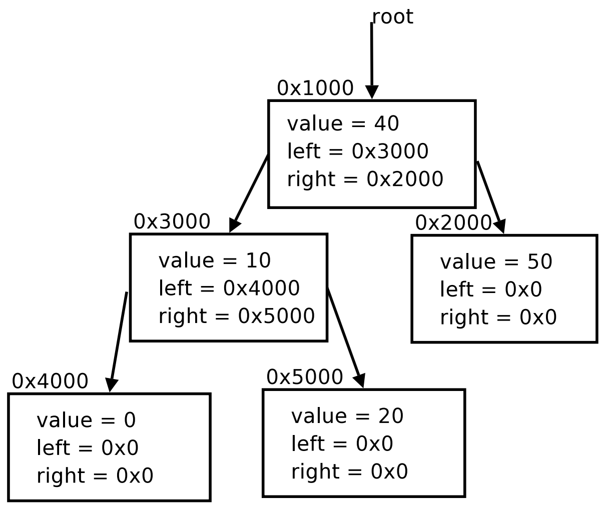

Figure 1: A simple binary tree.

As an example of conflict serializability, consider the simple binary tree in Figure 1. Three different critical sections that operate on this tree, along with their address sets are listed in Table 1. Critical sections 1 and 2 read many of the same memory locations, but only write to locations that are not in the other's address sets. They are, therefore, data independent and could safely execute concurrently. This makes sense intuitively because they modify different branches of the tree. Critical section 3, however, modifies the right pointer in the root of the tree, which cannot execute concurrently with critical sections that operate on the right branch of the tree. This is reflected in the address sets: 0x1032 is in Critical Section 3's write set and in the read sets of Critical Sections 1 and 2.

In our simple example, we can determine the data independence of the critical sections by simple static analysis. In most cases, however, static analysis is insufficient because functions that modify a data structure generally determine what to modify based on an input parameter. Thus, the data independence of critical section executions largely depends on the program's input, requiring investigation of the common case synchronization behavior in order to determine whether to employ optimistic concurrency.

There are cases where the structure of the critical section code can limit concurrency. If most executions of a critical section are not data independent, but only conflict on a small portion of the address set, finer-grained locking or data structure reorganization may remove these conflicts. For instance, the SLOB allocator in the Linux kernel uses a shared pointer to available heap space that is read at the beginning and written at the end of every allocation. Thus, any two allocations will have needless conflicts on this pointer and will never be data independent.

There are also cases where a high degree of data independence can be an argument for consolidation of overlapping critical sections under optimistic concurrency. In a program with fine-grained locking, multiple lock acquires and releases can be nested during overlapping critical sections to minimize the time a lock is held and increase performance. If, in an optimistic model, the outermost critical section is generally data independent, avoiding nested critical sections yields a worthwhile reduction in code complexity and potential increase in performance.

3 Syncchar

To measure the data independence of critical sections, we present a tool called *syncchar* (synchronization characterization) that runs as a module in Virtutech Simics, version 3.0.17 [7]. Simics is a full-system, execution-based simulator. Syncchar tracks the synchronization behavior of the Linux kernel, including each lock acquire and release, contention for a lock acquire, and tracks the address set of memory locations read and written in the critical section.

If a thread busy-waits to acquire a lock in the kernel, syncchar compares the thread's address set when it completes the critical section to the address sets of all critical sections it waited on to determine if they conflict. If two threads that waited for the same lock touch completely disjoint data, then they both could have optimistically executed the critical section concurrently, rather than needlessly waiting on a lock.

Comparing the address set of threads that happen to contend for a lock during one execution provides only a limited window into the latent concurrency of a lock-based application. To determine a more general limit on optimistic concurrency, when a lock is released, syncchar compares its address set to the address sets of the previous 128 critical sections protected by that lock. By recording and comparing the address sets for multiple critical sections, syncchar's measurements are less sensitive to the particular thread interleaving of the measured execution.

An ideal analysis would identify and compare all possible concurrent executions, rather than just the last 128. However, this would require tracking events such as thread forks and joins that structure concurrent execution. In addition, the number of possibly concurrent executions could make comparing address sets infeasible for applications of substantial length. Comparing over a window of critical section executions captures many executions that are likely to be concurrent, while minimizing false conflicts that result from serialization through other mechanisms.

Syncchar supports a window of 128 critical sections based on a sensitivity study of the window size. Changing the window size from 50 to 100 affects the data independence results for most locks by less than 5%. As the largest commercial CMP effort to date is an 80 core machine [1], 128 also represents a reasonable upper bound on the size of CMP's likely to be developed in the near future.

Syncchar only compares critical regions across different threads. Determining whether two executions of a critical section in the same thread can be parallelized is a more complex problem than determining whether critical sections from different threads can be executed concurrently. If possible, this would require substantial code rewriting or some sort of thread-level speculation [14]. However, this paper focuses only on the benefits of replacing existing critical sections with transactions.

Syncchar filters a few types of memory accesses from the address sets of critical sections to avoid false conflicts. First, it filters out addresses of lock variables, which are by necessity modified in every critical section. It filters all lock addresses, not just the current lock address, because multiple locks can be held simultaneously and because optimistic synchronization eliminates reads and writes of lock variables. Syncchar also filters stack addresses to avoid conflicts due to reuse of the same stack address in different activation frames.

Some kernel objects, like directory cache entries, are recycled through a cache. The lifetime of locks contained in such objects is bounded by the lifetime of the object itself. When a directory cache entry emerges from the free pool, its lock is initialized and syncchar considers it a new, active lock. The lock is considered inactive when the object is released back to the free pool. If the lock is made active again, its the address set history is

Lock	Total Acquires	Contended Acq	Data Conflicts	Mean Addr Set Bytes	Mean Confl Bytes
zone.lock	14,450	396 (2.74%)	100.00%	161	50
ide_lock	4,669	212 (4.54%)	97.17%	258	24
runqueue_t.lock (0xc1807500)	4,616	143 (3.23%)	84.62%	780	112
zone.lru_lock	16,186	131 (0.81%)	48.09%	134	8
rcu_ctrlblk.lock	10,975	84 (0.77%)	97.62%	27	4
inode.i_data.i_mmap_lock	1,953	69 (3.53%)	89.86%	343	48
runqueue_t.lock (0xc180f500)	3,686	62 (1.74%)	90.32%	745	118
runqueue_t.lock (0xc1847500)	2,814	27 (0.96%)	88.89%	530	74
runqueue_t.lock (0xc1837500)	2,987	24 (0.80%)	95.83%	526	100
runqueue_t.lock (0xc184f500)	3,523	22 (0.68%)	86.36%	416	70
runqueue_t.lock (0xc182f500)	2,902	24 (0.83%)	87.50%	817	103
runqueue_t.lock (0xc1817500)	3,224	17 (0.68%)	94.12%	772	108
runqueue_t.lock (0xc1857500)	2,433	20 (0.86%)	90.00%	624	100
dcache_lock	15,358	21 (0.14%)	0.00%	0	0
files_lock	7,334	20 (0.27%)	70.00%	15	8

Table 2: The fifteen most contended spin locks during the pmake benchmark. Total Acquires is the number of times the lock was acquired by a process. Different instances of the same lock are distinguished by their virtual address. Contended Acq is the number of acquires that required a process had to spin before obtaining the lock, including the percent of total acquires. The Data Conflicts column lists the percentage of Contended Acquires that had a data conflict. Mean Addr Set Bytes is the average address set size of conflicting critical sections, whereas Mean Confl Bytes is the average number of conflicting bytes.

cleared, even though it resides at the same address as in its previous incarnation.

Before scheduling a new process, the kernel acquires a lock protecting one of the runqueues. This lock is held across the context switch and released by a different process. As this lock is actually held by two processes during one critical section, Syncchar avoids comparing its address set to prior address sets from either process.

Finally, some spinlocks protect against concurrent attempts to execute I/O operations, but do not actually conflict on non-I/O memory addresses. Syncchar cannot currently detect I/O, so results for locks that serialize I/O may falsely appear data independent.

4 Experimental Results

We run our experiments on a simulated machine with 15 1 GHz Pentium 4 CPUs and 1 GB RAM. We use 15 CPUs because that is the maximum supported by the Intel 440-bx chipset simulated by Simics. For simplicity, our simulations have an IPC of 1 and a fixed disk access latency of 5.5ms. Each processor has a 16KB instruction cache and a 16KB data cache with a 0 cycle access latency. There is a shared 4MB L2 cache with an access time of 16 cycles. Main memory has an access time of 200 cycles.

We use version 2.6.16.1 of the Linux kernel. To simulate a realistic software development environment, our experiments run the pmake benchmark, which executes `make -j 30` to compile 27 source files from the libFLAC 1.1.2 source tree in parallel.

4.1 Data Independence of Contended Locks

The data independence of the most contended kernel locks during the pmake benchmark is presented in Table 2. Columns 2 and 3 show the total number of times each lock was acquired and the percentage of those acquires which were contended. There are low levels of lock contention in the kernel for the pmake workload. For all locks, at least 95% of acquires are uncontended. Linux developers have invested heavy engineering effort to make kernel locks fine-grained.

Each time two or more threads contend for the same lock, syncchar determines whether the critical sections they executed have a data conflict (shown in column 4). These locks have a high rate of data conflict, indicating that 80–100% of the critical sections cannot be executed safely in parallel. Many of the locks in this list protect the process runqueues, which are linked lists. The linked list data structure is particularly ill-suited for optimistic concurrency because modification of an element near the front will conflict with all accesses of elements after it in the list.

There is one noteworthy lock that protects critical regions that are highly data independent—`dcache_lock`, a global, coarse-grained lock. The `dcache_lock` protects the data structures associated with the cache of directory entries for the virtual filesystem. Directory entries represent all types of files, enabling quick resolution of path names. This lock is held during a wide range of filesystem operations that often access disjoint regions of the directory entry cache.

In the cases where the critical sections have data conflicts, we measured the number of bytes in each address set and the number of conflicting bytes, listed in Columns 5 and 6. A particularly small address set can indicate that the lock is already fine grained and has little potential for optimistic execution. For example, the `rcu_ctrlblk.lock`, protects a small, global control structure used to manage work for a Read-copy update (RCU). When this lock is held, only a subset of the structure's 4 integers and per-CPU bitmap are modified and the lock is immediately released. Our experimental data closely follows this pattern, with the lock's critical sections accessing an average of 7 bytes, 5 of which conflict.

A larger working set with only a small set of conflicting bytes can indicate an opportunity for reorganizing the data structure to be more amenable to optimistic synchronization. The `zone.lru_lock` protects two linked lists of pages that are searched to identify page frames that can be reclaimed. Replacing the linked list with a data structure that avoided conflicts on traversal could substantially increase the level of data independence.

4.2 The Limits of Kernel Data Independence

To investigate the limits of data independence in the kernel, syncchar compares the address set of each critical section to the address sets of the last 128 address sets for the same lock, as discussed in Section 3. The purpose of comparing against multiple address sets is to investigate how much inherent concurrency is present in the kernel.

Amdahl's law governs the speedup gained by exploiting the concurrency inherent within Linux's critical sections. Locks that are held for the longest period of time contribute the most to any parallel speedup, so the results in Table 3 are presented for the ten longest held locks. Conflicting acquires vary from 21%–100%, though the average data independence across all spinlocks, weighted by the number of cycles they are held, is 24.1%. That level of data independence will keep an average of 4 processors busy. Data structure reorganization can increase the amount of data independence.

One interesting lock in Table 3 is the `seqlock_t.-lock`. Seqlocks are a kernel synchronization primitive that provides a form of optimistic concurrency by allowing readers to speculatively read a data structure. Readers detect concurrent modification by checking a sequence number before and after reading. If the data structure was modified, the readers retry. Seqlocks use a spinlock internally to protect against concurrent increments of the sequence number by writers, effectively serializing writes to the protected data. Because the same sequence number is modified every time the lock is held,

Lock	Confl Acq	Mean Confl Bytes	
zone.lock	100.00%	32.49	20.40%
zone.lru_lock	65.82%	6.63	20.10%
ide_lock	70.92%	6.93	19.03%
runqueue_t.lock (0xc1807500)	27.41%	24.06	24.50%
runqueue_t.lock (0xc180f500)	29.14%	26.10	23.20%
inode.i_data.i_mmap_lock	46.03%	22.81	16.53%
runqueue_t.lock (0xc1837500)	27.30%	27.83	19.15%
seqlock_t.lock	100.00%	56.01	48.41%
runqueue_t.lock (0xc1847500)	23.32%	26.64	19.55%
runqueue_t.lock (0xc183f500)	21.03%	28.21	18.37%

Table 3: Limit study data from the ten longest held kernel spin locks during the pmake benchmark. This data is taken from comparing each address set to the last 128 address sets for that lock, rather than contended acquires, as in Table 2. Conflicting Acquires are the percent of acquires that have conflicting data accesses. Mean Conflicting Bytes shows the average number of conflicting bytes and their percentage of the total address set size.

this lock's critical section will never be data independent. Although this seems to limit concurrency at first blush, there remains a great deal of parallelism in the construct because an arbitrarily large number of processes can read the data protected by the seqlock in parallel. Sometimes locks that protect conflicting data sets are not indicators of limited concurrency because they enable concurrency at a higher level of abstraction.

The data conflicts of fine-grained locks can distract attention from the ability to concurrently access the larger data structure. The locks protecting the per-CPU data structure `tvec_base_t` from an occasional access by another CPU tend to have an extremely low degree of data independence because they are fine grained. In the common case, however, the data these locks protect is only accessed by one CPU, which represents a large degree of overall concurrency. We leave extending this methodology to multiple levels of abstraction for future work.

4.3 Transactional Memory

To provide a comparison between the performance of lock-based synchronization and optimistic synchronization, we ran the pmake benchmark under syncchar on both Linux and on TxLinux with the MetaTM hardware transactional memory model [13]. TxLinux is a version of Linux that has several key synchronization primitives converted to use hardware transactions. Syncchar measures the time spent acquiring spinlocks and the time lost to restarted transactions. For this workload, TxLinux converts 32% of the spinlock acquires in Linux to transactions, reducing the time lost to synchronization overhead by 8%. This reduction is largely attributable to removing cache misses for the spinlock lock variable. The syncchar data indicates that converting more Linux locks to transactions will yield speedups that can be exploited by 4–8 processors. However, gaining even greater concurrency requires data structure redesign.

5 Related work

This paper employs techniques from parallel programming tools to evaluate the limits of optimistic concurrency. There is a large body of previous work on debugging and performance tuning tools for programs [3, 15]. Our work is different from other tools because it is concerned more with identifying upper bounds on performance rather than performance tuning or verifying correctness.

Lock-free (and modern variants like obstruction-free) data structures are data-structure specific approaches to optimistic concurrency [4, 5]. Lock-free data structures attempt to change a data structure optimistically, dynamically detecting and recovering from conflicting accesses.

Transactional memory is a more general form of optimistic concurrency that allows arbitrary code to be executed atomically. Herlihy and Moss [6] introduced one of the earliest Transactional Memory systems. More recently, Speculative Lock Elision [10] and Transactional Lock Removal [11] optimistically execute lock regions transactionally. Several designs for full-function transactional memory hardware have been proposed [2,8,12].

6 Conclusion

This paper introduces a new methodology and tool for examining the potential benefits of optimistic concurrency, based on measuring the data independence of critical section executions. Early results indicate that there is sufficient data independence to warrant the use of optimistic concurrency on smaller multiprocessors, but data structure reorganization will be necessary to realize greater scalability. The paper motivates more detailed study of the relationship of concurrency enabled at different levels of abstraction, and the ease and efficacy of data structure redesign.

7 Acknowledgements

We would like to thank the anonymous reviewers, Hany Ramadan, Christopher Rossbach, and Virtutech AB. This research is supported in part by NSF CISE Research Infrastructure Grant EIA-0303609 and NSF Career Award 0644205.

References

[1] Michael Feldman. Getting serious about transactional memory. *HPCWire*, 2007.

[2] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[3] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, pages 331–342, 2000.

[4] M. Herlihy. Wait-free synchronization. In *TOPLAS*, January 1991.

[5] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, 2003.

[6] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[7] P.S. Magnusson, M. Christianson, and J. Eskilson et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, 2006.

[9] C. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[10] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[11] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[12] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*. 2005.

[13] H.E. Ramadan, C.J. Rossbach, D.E. Porter, O.S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[14] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, 2000.

[15] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.