Effective Synchronization on Linux/NUMA Systems

Gelato Conference 2005

May 20th, 2005

by

Christoph Lameter, Ph.D.
christoph@lameter.com

© 2005 Silicon Graphics, Inc. All rights reserved.

Effective locking is necessary for satisfactory performance on large Itanium based NUMA systems. Synchronization of parallel executing streams on NUMA machines is currently realized in the Linux kernel through a variety of mechanisms which include atomic operations, locking and ordering of memory accesses. Various synchronization methods may also be combined in order to increase performance. The talk presents the realization of basic synchronization in Linux on Itanium and then investigates more complex locking schemes.

The current Linux locking mechanisms rely heavily on a simple spinlock implementation that may be fitting for systems of up to 8 processors. However, spinlocks cause excessive cache line bouncing if more processors are contending for a lock. Some approaches that have so far been made to solve the contention issue are presented and it is then suggested to use an implementation for Linux of the approach first proposed by Zoran Radovic which he called "Hierarchical Backoff Locks".

Table of Contents

1 Introduction

It seems that the computing industry has hit a wall improving the speed of processors through increasing the clock frequency of the processor. Current technology leads to temperature problems at high clock rates. Heat problems are now effectively limiting the ability to increase the number of cycles a processor can process. The way to further increase the computing speed of processors is by parallelizing processing. The Itanium instruction set was already designed with such parallelization in mind. However, mainstream computers use IA32 compatible hardware and are designed for an instruction set not optimized for parallelization. The best solution to parallelize processing is to put multiple processors on one chip. Intel and AMD have now begun to ship multi-core processors. Itanium multi-core chips are also expected later in 2005.

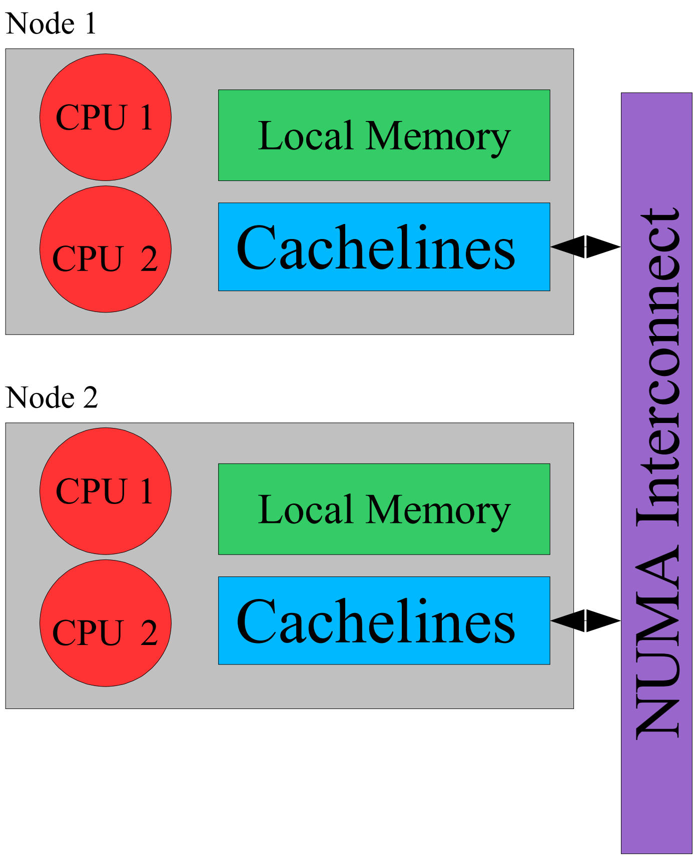
Multi-core processors have significant implications for operating software design since the overwhelming number of systems sold in the future can be reasonably expected to be equipped with multi-core processors. Multi-core processors typically have the characteristics of a NUMA system since communication between the cores on the processor chip (and maybe memory directly connected to one core) is more efficient than off-chip communication to other processors and otherwise connected memory. The AMD multi-core design uses core specific memory and I/O controllers leading to varying memory access times depending on which core in a processor is used to access memory.[1] These characteristics are typical for NUMA systems. It is likely that the number of processors on a chip will increase rapidly. IBM already has a chip with 9 (sort of) processors and other vendors seem to be planning processors with 4 and 8 cores. Effective synchronization algorithms that function well on NUMA systems may become an important core requirement in operating systems. So the issues that SGI has been dealing with all along are likely to occur in the standard hardware of the future.

Here we will discuss the existing synchronization mechanism for Itanium processors and their use to implement the various locking mechanisms available in Linux. This will first include a discussion of the nature of atomic operations on Itanium systems, then an investigation of the various primitives used to realize locking, a description of the implementation of Linux kernel locking mechanisms and a discussion of advanced combinations of various locking techniques in order to increase performance. The performance implications of the current locking schemes will be discussed based on results of performance benchmarks done with the page fault handler. The result is that current algorithms are effective only for up to 4 processors. Spinlock contention becomes a scalability problem for larger configurations.

In the final section we investigate a new way to handle locks first proposed by Zoran Radovic called HBO locks. Performance tests show that his algorithm can address the contention issues arising in a NUMA system with a large number of processors in an more effective way while preserving the efficiency of the existing code for the uncontended case.

1 AMD, "What is Multi-Core". Internet http://multicore.amd.com/WhatisMC (accessed April 28th, 2005).

2 Basic Atomicity



Drawing 1 NUMA System concept

NUMA systems are basically multiprocessor systems with a hardware cache consistency scheme.[2] Access to non local devices and memory is provided via a NUMA interlink designed for high speed inter node communication. However, the speed with which memory may be accessed varies according to the communication distance via the NUMA interlink and the particular hardware characteristics of the device or memory, hence NUMA which means *Non-Uniform Memory Architecture*.

The NUMA interlink uses a hardware cache consistency protocol to provide a coherent view of memory in the system as a whole to all processors. The hardware consistency protocol allows access to memory in chunks of a *cache line*. On the Itanium this cache line size is typically 128 byte. The following diagram shows the concept of a multiprocessor system with a MESI type hardware cache consistency protocol.[3] Note that this model is simplified so that we can focus on the characteristics important for locking. A variety of implementations of protocols to obtain consistency over NUMA exists but all NUMA systems that I am currently aware of follow the basic principles that we are using here.

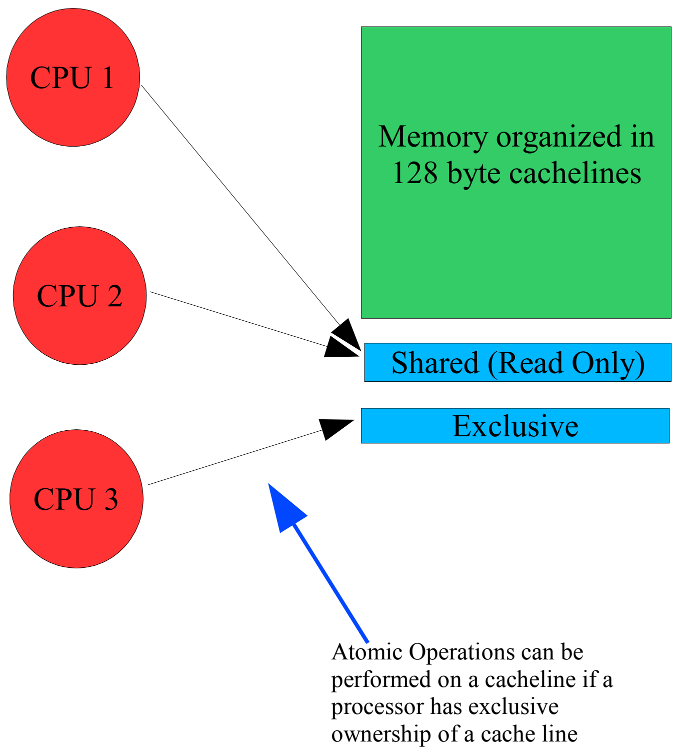
The typical NUMA system contains nodes with a variety of resources. A node usually contains a few processors, some memory and maybe I/O devices. Access from the local processors to local memory is very fast. Access to memory in other nodes is possible through the NUMA interlink but is slower since traffic has to flow across a bus. All memory accesses are managed through the hardware cache consistency protocol to insure a coherent view of memory for all processors in the system.

A program running on a NUMA system can be optimized by insuring that memory used is *local memory*. However, if programs are larger than the memory available on a single node or if a program uses more processors than are available on one node then operation over the interlink becomes necessary. The more use a program makes of the interconnect the more important the speed and the efficient use of the NUMA interlink becomes for the overall performance of the system.

2 Curt Schimmel, *Unix Systems for Modern Architectures*, 305-
3 Schimmel, 314.

2.1 Cache lines

Each processor can obtain access to a selection of cache lines via the cache coherency protocol. The contents of the cache line are then locally available through the cache of cache lines in the node. A cache line may be either be acquired as a *shared cache line* that only allows read access or as an *exclusive cache line.* An exclusive cache line is resource intensive because it must be guaranteed that no other accesses occur to this cache line while one processor has ownership of the cache line. Write access is only allowed on a cache line that is held with exclusive access otherwise multiple updates could be committed simultaneously to memory. This guarantees a cache line level atomicity of writes across the hardware consistency domain.


CPU 1
CPU 2
CPU 3
Memory organized in 128 byte cachelines
Shared (Read Only)
Exclusive
Atomic Operations can be performed on a cacheline if a processor has exclusive ownership of a cache line

Drawing 2 MESI type cache control algorithm

It is therefore useful to organize data structures in such a way that they fit into a cache line. One cache line can then be acquired by a processor and all related data can be processed from the same cache line without any additional memory operations.

It is advantageous to insure that code uses as many shared cache lines as possible because then multiple nodes may cache in the data locally allowing simultaneous access to the same data. Exclusive cache lines may require negotiations and a transfer across the NUMA link before data may be accessed.

Conversion of cache lines between shared and exclusive modes are expensive since a shared line may be held by multiple processors. The copies that other processors are holding must be invalidated if one processor wants to hold the line as exclusive. This means that the other processors will have to perform additional operations to reacquire the cache line if they need to access data in the cache line again.

If two processors keep reading and writing the same cache line then the exclusive access to the cache line has to be renegotiated again and again. The ownership of the cache line and the content of the cache line seem to *bounce* back and forth between multiple processors. This is called a *bouncing cache line.* The constant renegotiation of the ownership of the cache line may cause lots of traffic across the NUMA link which may become a performance bottleneck.

An atomic read/write cycle of a cache line can potentially be used for atomic changes to memory since a cache line must be held for exclusive access before any write can take place. However, the processor must intentionally perform such an atomic read modify write cycle which requires the use of special atomic semaphore instructions. These bypass most of the typical optimizations performed by a processor. Without the atomic semaphore operations the cache lines may change at any time during the normal flow of processing.

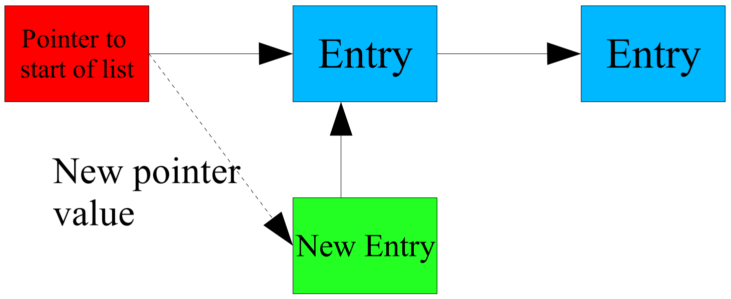
2.2 Atomic nature of processor operations

The processor interfaces with the cache in a way that guarantees the atomicity of certain operations. For the Itanium the guarantee is that operations up to 64 bit—to properly aligned memory locations—are atomic without any other special measures.[4] This means that if multiple processors attempt to store a 64 bit value to a properly aligned 64 bit memory location then the memory location will later contain the value stored into that location by one or the other processor but will not contain a few bits from one processor and a few bits from another processor. Again note that this behavior is only guaranteed if the 64 bit value is *aligned on an 8 byte boundary*. A misaligned 64 bit store is not guaranteed to be atomic. Concurrent stores may yield some bytes set by one processor and some by another.

Note that the Itanium is capable of handling data structures that are longer than 64 bit (like for example 10 byte floating point numbers or 16 byte stores). These are typically also atomic but special considerations (f.e. Atomicity may not be guaranteed if the write back cache is enabled) may apply for these under some circumstances.[5] If these restrictions are not followed then storing two 10 byte floating point numbers concurrently from multiple CPUs may (in some rare cases) yield a mixture of the two.

2.3 Utilizing operational atomicity for list processing

Loads and stores of 64 bit values are atomic as explained above. Addresses are 64 bit entities on Itanium. This means that addresses can be safely replaced or read from memory without additional measures (which may not be a surprising at all but there are processors around that are challenged in this area). The Linux kernel contains a set of RCU list functions[6] that utilize the atomicity of loads and stores to allow the lock-less management of lists.



Drawing 3 Lockless insertion of a list element via an atomic store

To illustrate how this works: Here is an insertion of an element into a linear list accomplished in a SMP safe way just by utilizing the atomic nature of 64 bit operations.

Consider the following list with two entries. The entries are linked by pointer to the next entry and there is a global variable that contains a pointer to the start of the list. It must be safe at all times to scan the list for a lock-less implementation to be considered viable.

The writer first prepares a new element to be put in the list and points the next pointer to

4 Intel. *Intel Itanium Architecture Software Developer's Manual: Volume 2,* 2:235
5 Ibid., 2: 77
6 See include/linux/list.h

the first element of the list. At this point scans through the list will not reach that new element.

The writer then stores the address of the new element into the global variable pointing to the start of the list. This operation is a 64 bit store and atomic and therefore the other processors scanning the list will either scan two list element if the other processors see the old start pointer or three list elements after the other processors see the new start pointer. The list itself will never be inconsistent from the view of any processor scanning the list.

Note that this only works for one writer and multiple readers. Some mechanism must exist to insure that multiple writers do not manipulate the list at the same time.

2.4 Barriers and acquire / release

The example given above will only work if some *barriers* are put into place to insure that changes to memory become visible in the proper order. The order in which the Itanium processor reads from memory or writes to memory is undetermined to allow the processor to optimize memory accesses. In the example here we need to be sure that the new entry (which contains the old pointer to the start of the list!) becomes visible to other processors *before* the new pointer to the start of the list becomes visible. We need a *write barrier* between the setup of the new entry and the update of the pointer. The write barrier insures that all writes before the barrier become visible to other processors before any writes *after* the barrier.

Without write ordering the new start pointer value could become visible to other processors before the content of the new entry. The structure may appear to another processor to contain completely invalid data, the pointer from the new entry to the next entry may appear to be NULL (in which case the processor scanning the list will only find one element on the list and not three!) or garbage which may lead to invalid memory accesses.

There is a similar mechanism for read called a *read barrier.* It insures that data read after the barrier are actually retrieved after the read operations before the barrier. Both types of barriers require the compiler to generate additional code to either reload data from memory or to insure that data is written to memory. The read and write barriers are Linux functions that are mapped to the same Itanium *memory fence* instruction. However, their function may be different on other platforms and therefore we need to keep the distinction although the Itanium processor does not do anything different.

Some of the barrier behavior can be encoded in a memory reference on Itanium. In that case a load or store is said to have *acquire* or *release* semantics. A memory operation with acquire semantics will insure that the access is visible before all subsequent accesses. Release semantics imply that all prior memory accesses are made visible before the memory access in the instruction.[7] These semantics will obviously not affect values cached in registers by the code generated through the compiler. This means that the compilers must also in these cases cooperate to have the proper effect. The Linux barriers insure that this occurs. If assembly is used then one needs to make sure that the compiler gets somehow informed about how code has to be generated.

7 Intel. *Intel Itanium Architecture Developer's Manual: Volume 1*, 1:64.

```
void __list_add_rcu(struct list_head * new,
     struct list_head * prev, struct list_head * next)
{
       new->next = next;
       new->prev = prev;
       smp_wmb();
       next->prev = new;
       prev->next = new;
}

void list_add_rcu(struct list_head *new,
          struct list_head *head)
{
       __list_add_rcu(new, head, head->next);
}
```

Drawing 4 RCU add_list implementation

Linux provides a series of list operations that allow lock less handling of double-linked lists called RCU lists (defined in **include/linux/list.h**). Drawing 4 shows the implementation of **list_add_rcu** using barriers and atomic stores. The **prev** and **next** pointer of the new element are set to point to the previous and next element. With that it is ensured that any scan of the list that encounters the new element can continue.

Then a write barrier follows to insure that the pointers in the new element are visible before any changes to the list. The list is changed by pointing the next and prev element pointers to the new item. Other readers of the list will either scan the list without the element or with the element. If the new element is encountered by another processor then it is guaranteed that the pointers of the new item are also visible.

There are additional operations that allow the lock less removal of elements from the list (**list_del_rcu**) and the scanning of the lists(**list_for_each_entry_rcu**). The lock less removal from the list is not without difficulties since there is no way to guarantee that all processors no longer see links to the list element to be removed. The real freeing of an element has to be deferred until it is known that no one is browsing the list anymore. For that purpose two functions exist to track concurrent scans of the lists:

rcu_read_lock()

rcu_read_unlock()

These are not real lock operations. The functions are used to maintain a counter of the number of active readers. If the number of readers reaches zero then it is guaranteed that no link to the list element exists anymore and the freed list element can be finally deallocated.

Only one writer can be active at any one time if RCU type lists are used. To insure a single writer additional locking is required.

Barriers and the regular loads and stores (which are atomic) are the most efficient means for synchronization in a NUMA system since they do not require slow atomic semaphore instructions. However, the elements to control atomicity discussed so far cannot guarantee that a single processor knows that only itself caused a state change in a memory location. For example it is not possible to insure that one processor is the only one that replaces a zero in one memory location by a 1. We have no way to insure that only one processor writes to a memory location.[8]

8 Dekker's algorithm can provide exlusion for two processors but we need to have a general solution for an arbitrary number of processors.

3 Atomic semaphore instructions

The Itanium processor provides a number of atomic read modify write operations called *Semaphore Instructions*.[9] Semaphore instructions are expensive because they acquire an exclusive cache line and then do a read modify write cycle on a cache line atomically. The processor cannot optimize memory access in the same way as done for other Itanium instructions. Semaphore instruction are always *non-speculative.* This means that atomic semaphore instructions result in pipeline stalls.[10] All semaphore operations require the full amount of cycles necessary to access memory (which may be quite a large number for distant memory!) plus 5 clocks. A non semaphore instruction referencing memory may only consume 1 clock or even be parallelizable with another instruction.

However, atomic semaphore operations are necessary in order to effect state transitions by a single processor and these state changes are an essential element to realize locks. Without semaphore instructions multiple processors may change a memory location but there is no way for the processor to tell that it was the unique processor whose store effected the change.

Semaphore instructions must have either release or acquire semantics and always do the read before the write. There will be no other accesses to the same memory region between the read and the write of a semaphore instruction.[11]

3.1 Compare and exchange

The compare and exchange instruction allows one to specify the content that a memory location is expected to have and a new value that is to be placed into that memory location. The atomic operation is performed in the following way. First an exclusive cache line is acquired and all status changes to the cache line are stopped. Then the contents of the memory location are compared with the expected value. If the memory location has the expected value then the new value is written to the memory location. Only then is the cache line allowed to change state or be acquired by another processor. This allows an atomic state transition for a memory location.

The CMPXCHG operation returns the value of the memory location before the new value was stored in it. The operation is successful, if the value returned is the expected old value. If the expected value was returned then we can be sure that no other process raced with this process and only this processor effected the state transition from the old value to the new value.

The uniqueness of the state transition for one processor is typically used for locking because we can insure that only one processor successfully accomplishes the state transition on a certain memory location. If the other processors wait until they can do the same state transition then it can be made certain that only one processor obtains exclusive access to some resources and others are excluded from a resource protected by this mechanism.

9 Ibid.,1:51.

10 Intel. *Intel Itanium Processor Reference for Software Optimization,* Intel Corporation: November 2001, 27.

11 Intel. *Intel Itanium Architecture Developer's Manual: Volume 1*, 1:64.

3.2 Fetchadd

Fetchadd adds a value to a memory location atomically and returns the result. Fetchadd is one way to realize counters without having counter values protected by other locking mechanisms. Normal counter updates are subject to race conditions since incrementing a counter implies loading the counter value, incrementing a register value and then writing the result back to memory.

Another use of fetchadd is to realize usage counters. Usage counters are incremented for each user of the structure. If a user no longer needs the structure then the usage counter is decremented via fetchadd. We can check if this was the last user of the structure since fetchadd also returns the result. If the result is zero then the structure can be freed.

3.3 Xchg

The xchg instruction is rarely used today. Historically it was the first atomic instruction that became available for synchronization in multi-processing environments since it was already implemented for single processor systems. Creative uses of xchg led to the initial locking algorithms. Xchg can be used to atomically replace a value and check the value later. This is useful if a state change needs to be made and if the code can then deal with the prior condition encoded in the state variable.

4 Spinlocks

4.1 Purpose

Spinlocks are implemented in the Linux kernel to *protect data structures* and allow the holders of the lock *exclusive access* to the structures that are protected by a spinlock. Spinlocks are designed to be fast and simple. Only limited nesting of locks is allowed (any nesting needs to be properly documented!), there is no deadlock prevention mechanism and no explicit management of contention. A variety of spinlock types exists to deal with concurrent interrupts or bottom handlers. These specialized versions of the spinlocks are not discussed here since they would complicate the descriptions too much.

The two spinlock functions mainly used to delimit critical sections are:

```
spin_lock(spinlock_t *lock);
spin_unlock(spinlock_t *lock);
```

A typical code example showing the use of a spinlock:

```
spin_lock(&mmlist_lock);
list_add(&dst_mm->mmlist, &src_mm->mmlist);
spin_unlock(&mmlist_lock);
```

The **mmlist_lock** is acquired that protects the **mmlist**. Then an operation on mmlist is performed (another list element is added)--this is the critical section that has cannot be run concurrently—and the lock is released again (note that the list discussed here is not an

RCU list!). The spinlock insures that there are no concurrent operations on the list. This includes concurrent writers and readers which the lock less list operation discussed earlier could not provide.

4.2 Implementation

Please keep in mind in the following discussion that spinlocks protect *data structures.* There is typically an implied understanding as to which data elements a spinlock protects which should be documented somewhere in comments near the declaration of the spinlock. For example here is a snippet from the definition of the task_struct:

```
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
        spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
        spinlock_t proc_lock;
/* context-switch lock */
        spinlock_t switch_lock;
```

A frequent misunderstanding is to think that spinlocks protect a critical section. Multiple critical section may exist that may manipulate the same data and therefore use the same lock to obtain exclusive access to the data structure for a variety of purposes.

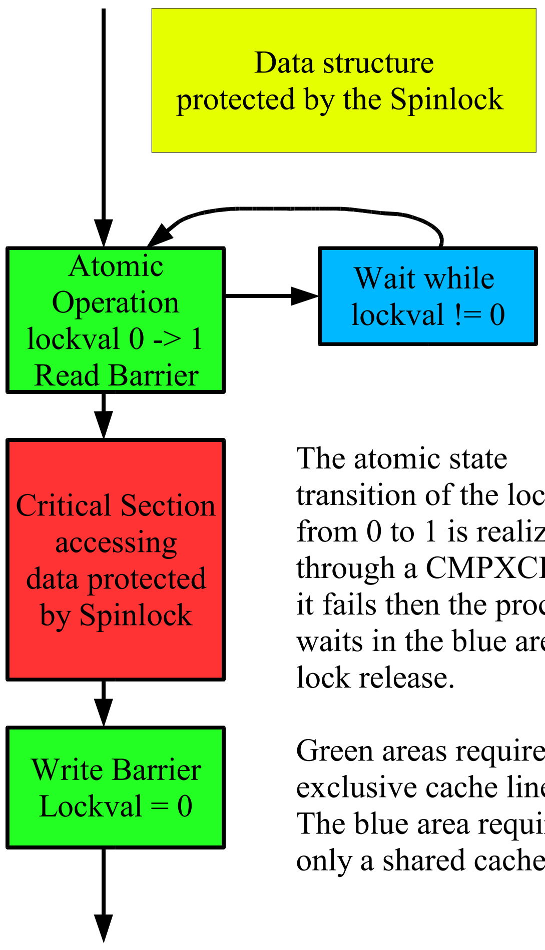
Spinlocks are realized on Itanium using a single 32 bit value that is either 0 (*unlocked*) or 1 (*locked*). The state change from 0 -> 1 is done using a CMPXCHG. Then a read barrier is needed to insure that data protected by the lock—which may have changed while the lock was acquired—is reread by the processor so that the state after the possible completion of another critical section can be obtained.

Unlocking is simply a write barrier to insure that modifications done within the lock are visible to others before the lock appears to be available. Then a zero is written to the lock.

If an attempt to acquire the lock using CMPXCHG fails then we enter into a wait loop. A CMPXCHG requires the processor to acquire the cache line containing the lock for exclusive access. If the operation fails then it is likely that the other processor holding the lock will read or write to data in the cache line and thus exclusive access must be transferred back to the processor holding the lock leading to the cache line bouncing back and forth.

If the CMPXCHG would simply be retried on failure then there is a high likelihood that the cache line will continually bounce between the processor holding the lock and the processor (actually processors because multiple processors may want the lock!) trying to acquire the lock until the lock is acquired by another processor.

In order to limit cache line bouncing, retries simply read the lock value and wait using regular load instruction while the contents of the lock value is not zero. If the contents are zero then another CMPXCHG is attempted to acquire the lock. Reading the lock value is possible through a shared cache line. Multiple processors can then simultaneously wait for changes to the lock state using a shared cache line. The idea is to avoid any additional


Data structure protected by the Spinlock
Atomic Operation lockval 0 -> 1 Read Barrier
Wait while lockval != 0
Critical Section accessing data protected by Spinlock
Write Barrier Lockval = 0

The atomic state transition of the lockval from 0 to 1 is realized through a CMPXCHG. If it fails then the processor waits in the blue area for lock release.

Green areas require exclusive cache lines The blue area requires only a shared cache line.

Drawing 5 Spinlock implementation

cache line bouncing after the first CMPXCHG.

However, any write to the cache line by the holder of the spinlock requires the acquisition of the exclusive access to the cache line again. Then the processors attempting to acquire the lock will immediately force the cache line back into shared mode since they are all spinning in a read loop. So a different kind of cache line bounce is incurred at each write to the cache line. In heavily contended environments it may be better to locate the lock in a different cache line from the data protected by the lock.

If a heavily contended lock is released then it is likely that multiple nodes will see the lock becoming zero while doing simple reads. The nodes will acquire the same shared cache line with the lock value being zero. All processors will then try a CMPXCHG simultaneously to acquire the lock which will result in multiple cache line bounces because each processor requires exclusive access to the cache line to perform the atomic semaphore operation before being able to start spinning using a read. After each CMPXCHG each processor then needs to acquire the cache line again in shared mode.

4.3 Effectiveness

There are many advantages to the existing spinlock implementation. Spinlocks are very effective due to their simplicity. A single instruction is typically sufficient to acquire and release the lock. The scheme is known to work well for a limited number of processors.

However, spinlocks are usually only held for very brief periods. Release and reacquisition is frequent. The acquisition and release will always require all participating processors to acquire exclusive access to the cache line. A large number of processors contending for the same lock increases the traffic on the NUMA interconnect until the cache line negotiation activity saturates the link which will lead to a significant drop in performance of the whole system.

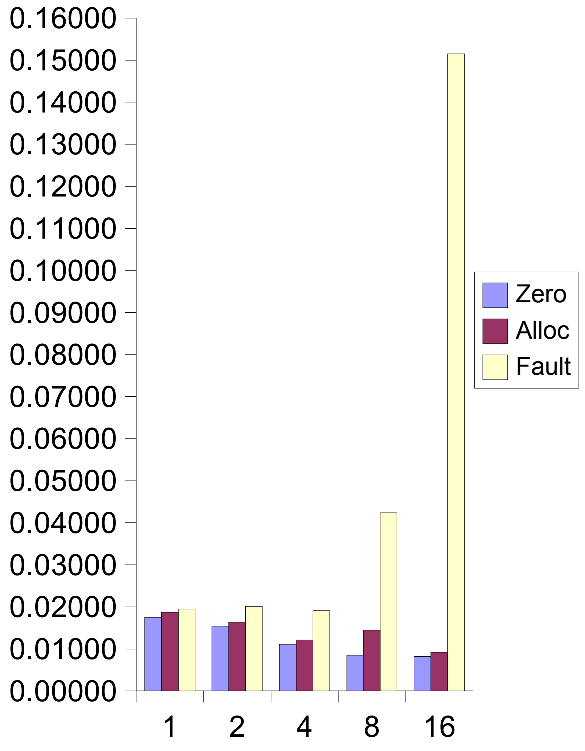


Illustration 1 Time spent in Page fault handler in ms with an increasing number of processors.

Illustration 1 shows the time spent in the page fault handler for anonymous page faults. The page fault handler typically acquires the page_table_lock twice. Yellow is the total time spent in the fault handler per fault. Red is the time spend allocating a page (which may include acquiring yet another spinlock that we will not consider for this discussion) and blue is the time spend zeroing the page. For one and two processors the time spent in the fault handler is dominated by the necessity to zero a page before providing the application access to it. The situation slightly changes for 4 processors when the time spent apart from zeroing and allocating increases. This is the time spend trying to acquire the page_table_lock.

If 8 processors are contending for the lock then more than 50% of processing time is spent acquiring the lock, meaning the processors are busy causing cache lines to bounce back and forth without making much progress in doing the work that they are expected to do. The time spent on lock acquisition increases exponentially as the number of processors increase. The diagram does not contain bars for more than 16 processors because they would no longer fit onto the page.

Spinlocks are only efficient up to 4 processors. Beyond 4 processors the system will spend significant resources on lock acquisition. Most of that time will be spend bouncing the cache line containing the lock around. This includes acquisition of exclusive access to the cache line as well as converting the cache line to shared mode.

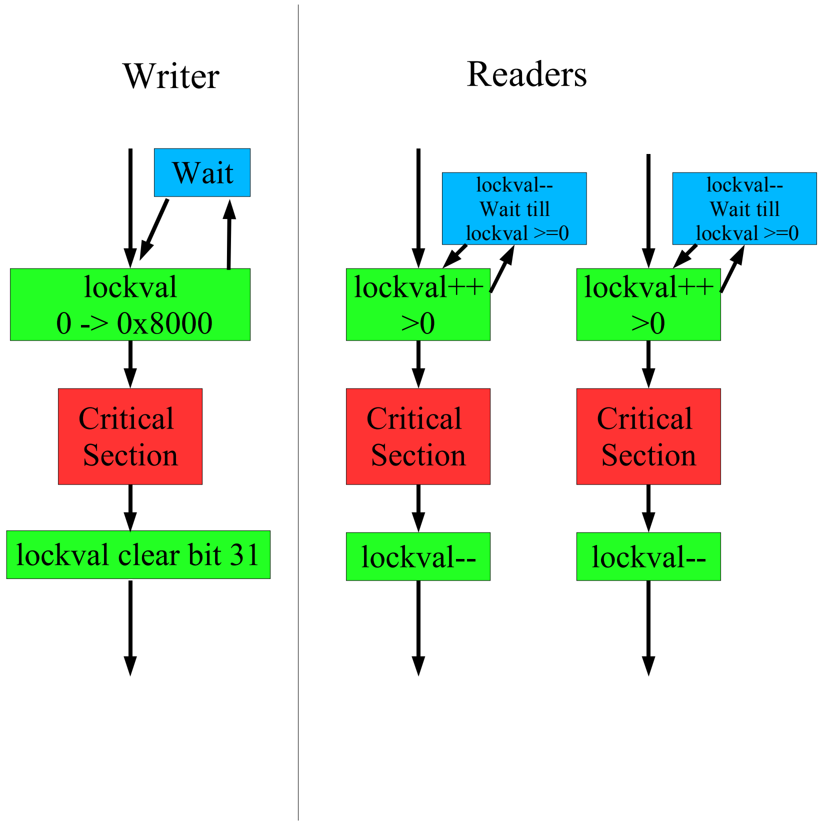
5 Reader/Writer Spinlocks

5.1 Purpose

Reader-Writer spinlocks allow *multiple readers* on the same data structure or one s*ingle writer*. This is useful if a protected structure is frequently read and only rarely written. For example the Linux task list is protected by a rwlock. Multiple processes may be scanning the task list or a single process may modify the task list.

5.2 Implementation

The implementation of rwlocks for Itanium also uses a single 32 bit value. The lock value is incremented for each reader. If the lock value is positive then the lock value counts the number of active readers. If a writer is active then bit 31 is set and the 32 bit signed value is negative. The lock is unused if the lock value is zero.



Drawing 6 Reader Writer Spinlock Implementation

Lock operations for the writer can be realized like the regular spinlock lock and unlock operations. However, 0x8000 is written into the counter instead of 1 setting bit 31. The write unlock simply resets this bit by using a CMPXCHG. It cannot write zero to the lock value since the readers may do some incrementing and decrementing with the lock value —even while the writer has bit 31 set—and the writer should not disturb the reader count. The CMPXCHG adds another atomic semaphore operation to the lock.[12] This means that the performance for the write lock is worse than regular spinlocks since two CMPXCHG operations are performed instead of one.

Readers increment the lock value and then check if the lock value is negative. If it is not negative then the read lock was successfully acquired. The unlock is simply a decrement of the lock value. However, both the increment and decrement are expensive semaphore operations which make a reader lock more expensive than a regular spinlock even in the uncontended case.

If the lock value is negative after increment then bit 31 is set and we know that a writer is holding the lock. The reader undoes the increment by decrementing the lock value and

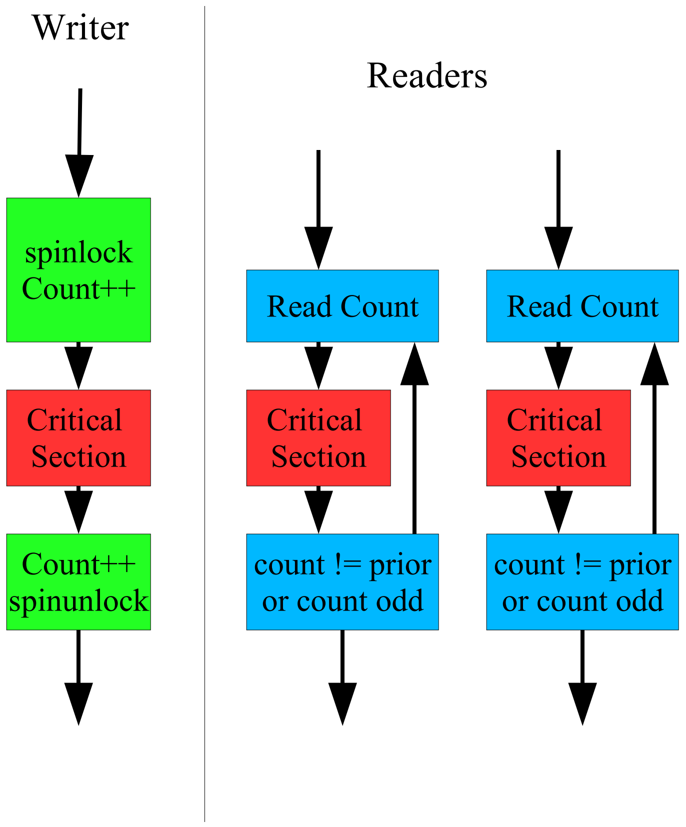
12 This may have been replaced by a simple byte store if my patch is accepted by the time this presentation is given.

then waits until the counter is greater than zero. This means that the reader uses two atomic semaphore operations for a failed lock under contention which creates more opportunities for cache line bouncing.

In general the performance behavior of reader writer locks will be worse than the performance of regular spinlocks.

6 Seqlocks

Seqlocks are the most scalable form of locks that is useful if there are a large number of readers. The readers do not need to write to memory at all to handle this type of "lock". Maybe one should not talk about acquiring a lock at all. Readers compare a counter before and after a critical section. If the counter did not change in the critical section then no writer was active and the outcome produced by the critical section is valid. If it has changed then the results of the critical section is discarded and it is run again.



Drawing 7 SeqLock

A seqlock contains a spinlock which is used for mutual exclusion between writers. Writers increment the counter once on lock acquisition resulting in an odd count value. The readers know that the writer is in the critical section if they encounter an odd value and rerun the critical section. Writers increment the counter again on unlock so that it becomes even. The lock acquisition needs two atomic semaphore operations. One to acquire the spinlock and another to increment the counter. The write unlock requires one semaphore operation to increment the counter and a store to unlock the spinlock. This means that the seqlock writer is less effective than regular spinlocks or rwlocks.

The readers can defer execution by repeatedly executing the critical section. This is good if writers are rare. In an uncontended case all that is involved in the lock are two memory read operations and two memory barriers. If there is or was a writer active while the reader critical section was executed then we know that the result is invalid and we need to repeat the critical section.

The seqlock is mainly used for time retrieval in Linux. For the Itanium I was able to avoid any writes during the reading of time information clock so that clock access became highly scalable. All processors may maintain their own copies of cache lines containing the relevant information.

The problem with seqlocks is that the reader cannot really do anything in the critical section apart from reading values since the critical section must be repeatable. The critical section for read may race with the critical section for write. It is therefore problematic for the writer or reader to perform updates of pointer structures, allocate memory etc.

Performance wise this is an ideal type of lock since readers never require an exclusive cache line. Ideally the writer acquires an exclusive cache line which contains the spinlock and the counter as well as the data managed once in a while and updates the information.

7 Atomic variables and usage counters

Linux provides a facilty to define variables of type **atomic_t**. Variables of that type cannot be handled using regular C operators but need to be manipulated using accessors. The idea is that these variables can be safely manipulated in a multiprocessing environment without needing the protection of a spinlock.

Here is a list of the most frequently used macros to mainpulate atomic variables.

```
atomic_t x;
ATOMIC_INIT(x);
y = atomic_read(x);
atomic_set(x, 789);
atomic_add(35, x);
if (atomic_dec_and_test(x)) { ... }
```

An *atomic_t* works just like a regular integer but the operations are guaranteed to be atomic in a multiprocessing environment. The cost of the operations on Itanium varies. Initialization, read and store of atomic variables have the same cost as a regular variable since they use the atomicity of loads and stores. However, adding and incrementing an atomic variable uses a *fetchadd* instruction which is much slower than a regular increment as discussed earlier. The same is valid for *atomic_dec_and_test* . Atomic operations may need memory barriers to convince the compiler to write variables to memory or refetch them and to insure that other processors see the right values when checking an atomic variable. These issues make the handlig of atomic variables much more complex than spinlocks.

There are other atomic operations that can be performed on arbitrary variables. These are atomic bit operations **set_bit , clear_bit, change_bit, test_and_set_bit, test_and_clear_bit, test_and_change_bit**. All of these do a loop around a load and a cmpxchg on the same cache line. A cache line may be first acquired in shared mode and then converted to exclusive. A regular cmpxchg would be more efficient since the cache line is only acquired once as an exclusive cacheline.

One frequent technique used in the Linux kernel to find out when to free an object is to keep a usage counter in the object. An atomic value is defined in the structure and initialized to 1. When an additional pointer is set up to the object the counter is incremented using **atomic_inc.**

```
/*
 * Decrement the use count and release all resources for an mm.
 */
void mmput(struct mm_struct *mm)
{
	if (atomic_dec_and_test(&mm->mm_users)) {
		exit_aio(mm);
		exit_mmap(mm);
		if (!list_empty(&mm->mmlist)) {
			spin_lock(&mmlist_lock);
			list_del(&mm->mmlist);
			spin_unlock(&mmlist_lock);
		}
		put_swap_token(mm);
		mmdrop(mm);
	}
}
EXPORT_SYMBOL_GPL(mmput);
```

Drawing 8 Example of atomic_dec_and_test to free a struct from the code for the removal of the reference to a memory descriptor.

When a reference is removed then the reference counter is decremented while checking if it reaches zero via the **atomic_dec_and_test** instruction. Dec and test is performed using the fetchadd semaphore operation on Itanium so that it is guaranteed that only one processor sees the reference counter become zero. That processor then knows that it is the only one still holding a reference to the structure and can safely free it.

The example in drawing 8 shows the use of atomic_dec_and_test to free a memory descriptor for a process. It may be necessary to acquire another spinlock during removal in order to safely free this element from a list.

The use of fetchadd for increment and decrement is an expensive operation since fetchadd requires exclusive cache lines and causes pipeline stalls. If references to objects are frequently established and then removed again then these cache lines may start bouncing and will become a performance bottleneck. This is known for example to happen for the routing information of the Linux IP stack and in the page fault handler if more than 64 processors are simultaneously allocating memory.

8 Disabling interrupts, preemption and split counters

A simple form of guaranteeing “atomicity” of operations can be had by disabling interrupts or preemption if the variables in use are only accessible from a single processor. Each processor has a special section of variables that are reserved for its own use exclusively called the *per cpu variables*. Variables defined per cpu are placed in this area. It is safe to assume that no other processor accesses these.

One way to avoid the overhead of using semaphore instructions to increment and decrement a counter is to split a counter into per cpu variables. Individual counters can then be incremented using regular load and store instructions to cache lines that are not shared between processors. However, in order to obtain a global count, one then needs to loop over all per cpu areas and add up all the processor specific counters. This method may also be used in some situations to avoid cache line bouncing on usage counters. One runs into issues with the atomicity of the check for zero references. Since no locking is available there is also no protection against races. The results of adding up all the counters may only be approximate.

9 Combination of locking techniques

One may combine several of the locking techniques. The lock less list operation using the RCU mechanism described needs spinlocks to insure exclusivity for writers. The seqlocks use a spinlock in the same way.

Another example for combining locking techniques are the modifications that I have proposed in order to make atomic operations on page table entries possible. This is accomplished by changing the locking from only relying on the *page_table_lock* to a combination of atomic operations plus the use of the page table spinlock.

In the Linux kernel the page_table_spinlock is acquired for any operation on the page table in order to serialize updates to the page table. However, the system also acquires a read lock on *mmap_sem* when processes makes small changes to memory mappings which includes changing individual entries. Mmap_sem is acquired for larger changes to the memory maps as a write lock. We can therefore rely on the mmap_sem for protection against large scale remapping of page table entries. The role of the page_table_lock is therefore only essential for small modifications of page table entries.

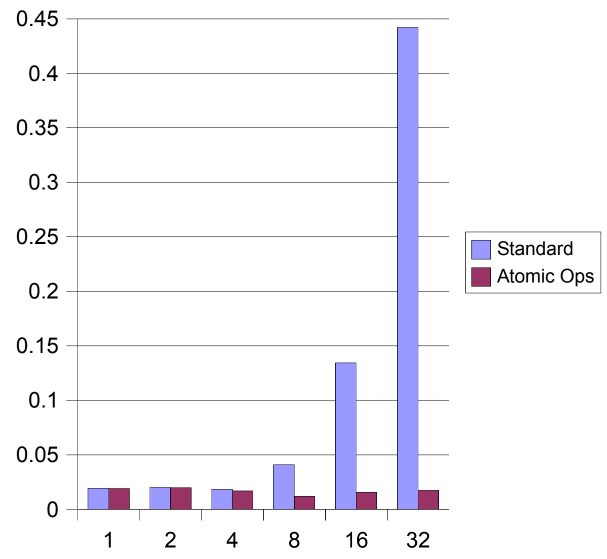


Illustration 2 Page fault time in ms per page with an increasing number of processors.

One can now redefine the role of the page_table_lock to only offer protection against modifications to a page table entry in the sense of replacing a valid entry with another but the replacement of an empty page table entry with a valid entry without the page_table_lock. Then it must also be guaranteed that an empty page table entry can always be populated even when a read lock of the mmap_sem and the page_table_lock is held.

This means that all code using the page_table_lock must now insure that a page table entry is never sporadically set to empty. Also if a page table entry is empty then the code using the page_table_lock must not assume that the page table entry will stay empty but must use atomic operations to replace values in order to guard against other processors concurrently changing the value without obtaining the lock. Holding the page_table_lock now only insures that a valid page entry is not changed. It no longer protects from empty page table entries becoming populated.

This solution to the contention problem allows for the removal of the use of the page_table_lock from the anonymous fault handler which will then make the page fault handler scale linearly as seen on the diagram.

Modifying lock semantics requires a change in the way an important lock is handled in the memory subsystem and is typically seen as an invasive change. The patch that does the modifications proposed here has so far not been accepted for inclusion into the Linux kernel.

10 Locking approaches not used in Linux

There are a variety of approaches that have been proposed over time to solve the contention issues. These range from simple back off algorithms to complex queuing logic.

The obvious first measure has always been a back off algorithm since the main issue encountered in large NUMA systems is contention on the NUMA interlink. A back off algorithm keeps the interlink free and allows useful work to go forward. The interlink is a type of a network and the obvious choice here is to use an exponential ethernet style back off algorithm. However, that carries the risk of long waiting periods. So there needs to be a cap on the maximum allowed back off period.

For an exhaustive list of locking approaches have a look at Radovic's writings mentioned in the bibliography. We only discuss two approaches here. MCS queue locks because they are often mentioned and hardware specific locks because they may hold some promise.

10.1 Queue locks

Queue locks are often mentioned because they are seen to address the issue of contention by serializing the lock acquisition. Queue locks allow the proper ordering and prioritization of processes trying to acquire the lock. The simple spinlock implementation favors the fastest. Whoever grabs the lock first can proceed first. Remote nodes may be at a disadvantage. Queue locks can insure that the lock is acquired in the proper sequence to insure that all processes obtain the lock in order. Queue locks allow *fair* lock acquisition. Queue locks can also insure that each processor spins on a different cache line reducing cache line bouncing significantly.

However, queue locks typically require much more effort than simple spinlocks and will slow down the system for the uncontended case. Efforts have been made to combine queue locks with a spinlock mechanism in order to simplify the uncontended case.

The most widely known of the queue based locking approaches are the MCS locks. John Stultz has done an implementation for Linux in 2002.[13]

The problem with queue locks is that lists of processors have to be managed. This hurts the uncontended lock case significantly and also generally leads to lower performance during contention due to complex list handling. We do not want to hurt the uncontended case with our modifications nor can we afford to do complex list processing which in turn may require its own locking scheme to synchronize the lists between multiple processors.

10.2 Hardware specific locks

The main performance limitations for spinlocks result from the use of the MESI scheme to negotiate cache lines. Cache lines contain much more information than necessary for the spinlock itself and thus some optimized hardware based solution could be more efficient if smaller size entities could be handled by specialized hardware without having to deal with bouncing cache lines. Hardware logic may also use a different algorithm from the cache line based approach of the hardware coherency protocols.

13 http://www.gelato.unsw.edu.au/linux-ia64/0202/3009.html

However, the performance of those operations must be able to compete with the atomic primitives available in the Itanium chip which is difficult to do with an I/O mapped hardware device given the speed advantage that instructions of the processor enjoy since they are realized in the processor core.

11 An HBO implementation for Linux

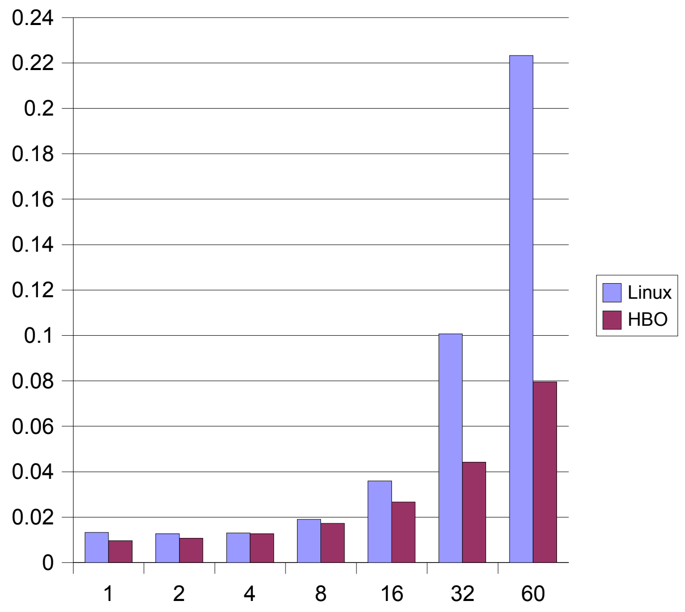


Illustration 3 Average Fault Time in ms using HBO locking

If there is lock contention limiting the scalability of Linux then the approach used so far has been to change the way the spinlocks are used in the Linux kernel. Locks can be broken up into multiple locks, the algorithm may be changed to avoid locks (like the approach taken for the atomic page table entry operations discussed above) and so on. While these are all valid approaches: Why not see a general problem with spinlocks under contention for systems with more than 4 processors and seek a modification to the spinlock algorithm itself to address contention? What we need is a back off algorithm that results in processors staying off the NUMA interlink for awhile to allow other processors to finish their work avoiding useless cache line bouncing while not hurting the uncontended case.

Zoran Radovic has developed an algorithm to deal with lock contention in large NUMA systems[14] and I have implemented his algorithm on Itanium. Radovic calls his algorithm a *Hierarchical Back off Algorithm*, short *HBO*.

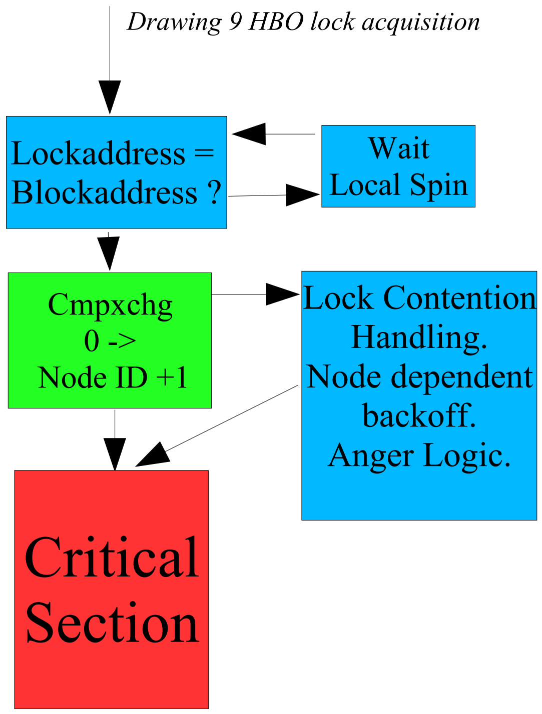
11.1 NUMA awareness

The standard Linux Spinlock implementation uses 0 to signify that a spinlock is unlocked and 1 for a lock in the locked state. The lock value is a 32 bit entity and we could put more information into that lock value. So instead of simply writing 1 to the lock, we write the node id (plus one so that node 0 is not confused with a lock not taken) to the lock. This means that the node number of the processor holding the lock will be available during contention and the processor trying to acquire the lock can adapt its behavior according to the distance to the node of the processor holding the lock. Operations over the NUMA link are more expensive and thus should not be tried as often as intra node lock attempts.

The optimal back off period for local contention was found to be 3 microseconds and around 8 microseconds for remote contentions. The different back off periods favor the local locks over remote locks in order to avoid moving the cache line with the lock around too much. For every failure to obtain the lock the back off period is increased by 50% until a limit is reached.

14 Radovic, 9

11.2 Limiting off node contention


Drawing 9 HBO lock acquisition
Lockaddress = Blockaddress ?
Wait Local Spin
Cmpxchg 0 -> Node ID +1
Lock Contention Handling. Node dependent backoff. Anger Logic.
Critical Section

The lock address will be checked against a node specific *local lock block address* before a CMPXCHG is performed to acquire the lock. If the lock address is equal to the local block value then the processor will spin on that instead of on the lock. The local lock block address is node specific. It will be in the processor cache allowing a comparison with only minimal overhead.

If a processor from one node finds that it cannot acquire a remote lock then it will set the local lock block address. Other processors from the same node will then be hindered to acquire the same off node lock. This limits off node lock acquisition attempts in heavily contended environments.

11.3 Starvation and anger levels

The back off algorithm may lead to starvation. Ownership of the lock may stay on one node if the local processors are continually accessing the same lock. At some point it will become more advantageous to move the lock since processes are stalling on remote nodes for too long. For that purpose off node lock acquisition first goes through a series of back offs. But if the node does not acquire the lock in a certain number of attempts it increases its “anger level”. Finally—when the anger level has reached a predetermined limit (50 attempts in our test cases)—it will remotely access the block address of the remote node and set that to the lock to be acquired. That will make the processors on the remote node spin when trying to acquire the lock the next time and the node will loose ownership of the lock to the angry node which will clear the block address of the remote node after having acquired the lock. This means that the lock ownership will be transferred to another node and then multiple processors waiting for the lock on the new node may proceed. Ideally—if the lock logic is properly tuned—the approach may lead to repeated moves of the lock between nodes. Hopefully all the local pending locks will be processed before moving to the next node thereby minimizing traffic on the NUMA Link.

The uncontended case is inlined in order to make lock acquisition as fast as standard spinlocks. There is an additional load and compare and the need to load the node id which creates overhead but this overhead was not measurable in the test cases that I have tried.

If contention arises then functions will be called to deal with the reason for the contention. These functions may contain more extensive logic and perform the exponential back off and contain the anger logic. The current implementation also contains a /proc interface that allows the tuning of the back off periods as well as a look at statistics regarding lock acquisition. If this ever would need to be implemented in production systems then additional tuning would certainly have to be done.

12 Conclusion

The existing implementation of spinlocks is not very suitable for large scale NUMA machines that may experience heavy spinlock contention. The proposed logic for the HBO locks is more complex than the simple spinlocks in Linux. It adds the overhead of an additional load before the CMPXCHG in the non contented case. However, there is no performance loss because all locking uses the same load address and the load is made from a cached entry.

The performance win for high contention begins to be significant with 16 processors and grows more and more for more and more processors. Using HBO locks may yield a nice performance boost for large scale systems.

13 Bibliography

Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization.* Intel Corporation. May 2004.

Intel. *Intel Itanium Architecture Software Developer's Manual: Volume 1: Application Architecture*. Intel Corporation. Revision 2.1. October 2002.

Intel. *Intel Itanium Architecture Software Developer's Manual: Volume 2: System Architecture*. Intel Corporation. Revision 2.1. October 2002.

Intel. *Intel Itanium Architecture Software Developer's Manual: Volume 3: Instruction Set Reference.* Intel Corporation. Revision 2.1. October 2002.

Intel. *Itanium Processor Microarchitecture Reference for Software Optimization.* Intel Corporation, March 2000.

Love, Robert. *Linux Kernel Development.* Sams Publishing, Indianapolis: Indiana, 2004.

Mosberger, David and Stephane Eranian. *IA64 Linux Kernel: Design and Implementation.* Prentice Hall, 2002.

Radovic, Zoran and Erik Hagersten. "Hierachical Backoff Locks for Nonuniform Communication Architectures" in *Proceedings of the Ninth International Symphosium on High Performance Computer Architecture*, Anaheim, California, February 2003. Internet:http://www.it.uu.se/research/group/uart/projects/nucasynch .

Radovic, Zoran. "Efficient Synchronization in Coherence in Nonuniform Communication Architectures". Licentiate Thesis: Department of Information Technology, Uppsala University, September 2003. Internet:http://www.it.uu.se/research/group/uart/projects/nucasynch .

Schimmel, Kurt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, 1994.