Fast Asymmetric Thread Synchronization

JIMMY CLEARY, Trinity College Dublin
OWEN CALLANAN, IBM Research, Dublin
MARK PURCELL, IBM Research, Dublin
DAVID GREGG, Lero, Trinity College Dublin

For most multi-threaded applications, data structures must be shared between threads. Ensuring thread safety on these data structures incurs overhead in the form of locking and other synchronization mechanisms. Where data is shared among multiple threads these costs are unavoidable. However, a common access pattern is that data is accessed primarily by one dominant thread, and only very rarely by the other, non-dominant threads. Previous research has proposed biased locks, which are optimized for a single dominant thread, at the cost of greater overheads for non-dominant threads. In this paper we propose a new family of biased synchronization mechanisms that, using a modified interface, push accesses to shared data from the non-dominant threads to the dominant one, via a novel set of message passing mechanisms. We present mechanisms for protecting critical sections, for queueing work, for caching shared data in registers where it is safe to do so, and for asynchronous critical section accesses. We present results for the conventional Intel® Sandy Bridge processor and for the emerging network-optimized many-core IBM® PowerEN™ processor. We find that our algorithms compete well with existing biased locking algorithms, and, in particular, perform better than existing algorithms as accesses from non-dominant threads increase.

Categories and Subject Descriptors: C.1.2 [Computer Systems Organization]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Synchronization, Biased Locking

1. INTRODUCTION

Thread safety incurs overhead. In a serial execution scenario, the running thread simply accesses and modifies data in memory as needed, with the only overhead being the time it takes the processor to access the data. In a concurrent scenario, access to shared resources incurs two overheads. Firstly, where there is contention among threads for shared resources, threads have to wait until the shared resource is available. Secondly, access to the shared resource must be protected by locks or other synchronization mechanisms. The cost of locking must be paid regardless of whether there is contention for the shared resource. So locks can involve a significant overhead even when there is no contention.

One important case of data sharing is where almost all of the accesses are from a single thread, and other threads access the shared data very rarely. We refer to this as asymmetric thread synchronization. The thread that does most of the accesses to
the shared data is known as the *dominant thread*. The other threads that access the shared data only rarely are the *non-dominant threads*.

Special *biased locks* have been developed which are optimized for asymmetric synchronization. In general, biased locks are optimized to reduce overhead for the dominant thread, at the expense of increased overhead for non-dominant threads. Such a scenario is common in many libraries, where for safety sake, potentially shared resources must be protected even if for many applications they will be accessed by one dominant thread. A biased lock will accelerate these libraries in their most common usage scenarios.

The ideal biased locking algorithm would impose no overhead on the dominant thread. No mechanism to achieve this is known. Attiya *et al.* [2011] demonstrated, that for modern processors with relaxed memory models, synchronisation in concurrent algorithms is impossible, without some use of expensive mechanisms such as memory fences or atomic compare and swap instructions. As a result work on biased locking algorithms is focused on finding the best balance of locking cost on the dominant and non-dominant threads, typically by optimising for the dominant thread, at the expense of the non-dominant threads.

The Linux Futex library [Franke *et al.* 2002] uses spin locks to protect shared resources, falling back on operating system mechanisms when a lock is contended for a significant time. An uncontended spin-lock may require only a single atomic machine instruction - orders of magnitude faster than an operating system kernel lock. Atomic instructions still impose a significant overhead on the dominant thread however.

Burrows [2004] introduced the concept of an unnecessary, or biased, mutex, where accesses by the dominant thread require no atomic operations. Burrows’s technique associates the dominant thread ID with the lock, and provides a mechanism for the non-dominant threads to suspend the dominant thread, access the protected resource and then resume the dominant thread. This technique carries a high overhead for non-dominant threads however, and the process for safely suspending and resuming the dominant thread is non-trivial.

Dice *et al.* proposed a system of *Quickly Re-acquirable Locks*, or biased locks, similar to Burrows’ scheme, except instead of the non-dominant threads suspending the dominant thread, instead they set flags in the lock structure to request access to the lock [Dice *et al.* 2003]. The dominant thread checks these flags on each access, granting access if it is requested. Dice’s scheme exploits the total-store-ordered memory model of the Sun SPARC architecture to eliminate the need for atomic or memory-fence instructions [Weaver and Germond 1992]. This technique will not work for more relaxed memory models however.

Some published work has focused on biased lock mechanisms for the Java Virtual Machine [Russell and Detlefs 2006] [Kawachiya *et al.* 2002], however it is specialised to the requirements of the Java Virtual Machine. Vasudevan *et al.* built on this prior work to propose a generalised system of biased locking useful for any native code environment. Their approach eliminates memory fences and atomic instructions for the dominant thread, and includes variants that allow bias-transfer between threads, as well as variants that allow multiple-readers or a single writer to the shared resource.

A significant contribution of this work is asymmetric biased locks, where the dominant thread grants access permission to non-dominant threads, instead of a two tiered locking system. The advantage of Vasudevan’s asymmetric biased lock is that the dominant thread only performs memory fences or atomic instructions when it grants access to a non-dominant thread, but the non-dominant thread must wait for service from the dominant thread, which adversely affects non-dominant thread performance.

In this paper we propose a new general scheme for asymmetric biased locks that builds on the system introduced by Vasudevan *et al.* We propose that when a non-
dominant thread accesses a critical section of code, it should push its work to the dominant thread, instead of waiting to be granted access to the shared resource. To permit this, our mechanisms require an altered interface that asks the programmer to enclose critical sections in specific functions, instead of explicitly calling lock() and unlock() primitives. This interface is shown in Figure 1. This allows us to greatly simplify the code for the dominant thread to access the critical section, reducing the cost in the common (that is biased) case. As long as the critical section is short, and the great majority of accesses are from the dominant thread (ideally more than 99%) there is likely to be an overall speedup. The reduced cost in the common case outweighs any additional cost to the dominant thread in the uncommon case.

This paper makes the following contributions:

— We present a new, general scheme for asymmetric thread synchronization, with altered semantics based on protecting critical sections of code instead of using lock() and unlock() primitives.
— We present algorithms that implement this general scheme using (a) locks, (b) passing work via function pointers, and (c) passing work via integer tokens (Section 4).
— To reduce contention among non-biased threads we propose a queue mechanism for outstanding requests (Section 5).
— We demonstrate how our work-passing algorithms can facilitate asynchronous critical accesses by non-dominant threads (Section 6).
— We show how dominance can easily be switched from one thread to another with almost no extra overhead (Section 8).
— We show how our scheme allows dominant threads to cache small amounts of shared data which they own in their CPUs’ registers, allowing for much faster access in the common case (Section 9).

We demonstrate results for a variety of benchmarks that analyse the performance of our algorithms and compare to the existing biased locking algorithms. In most cases our algorithms are comparable to the existing algorithms, and are frequently superior. We show how using queues for non-dominant thread access improves performance for most levels of dominance and we show that allowing non-dominant threads to make asynchronous accesses to the critical sections allows significantly improved performance with decreasing dominance. We also demonstrate that for certain usage scenarios with multiple critical sections, each with a unique dominant thread, our algorithms can deliver nearly twice the performance improvement of existing biased locks for a range of dominance levels.

Finally to demonstrate the benefit of our proposed mechanisms to real-world workloads and applications, we have applied it to a fiber scheduling library and benchmarked the resulting performance increase. In common with many emerging workloads for multi-core and many-core architectures, the fiber scheduling library needs to schedule a large number of parallel jobs onto a pool of workers. Each worker maintains its own workqueue but, to balance the work between workers, each worker attempts to steal work from a neighbour’s queue when its own queue is empty. Biased locks are an ideal way to reduce the locking overhead for this mechanism, without using complex lock-free algorithms.

2. BIASED LOCKING CONCEPTS

The goal of biased locking is to reduce the cost of locking for the dominant thread, even if this results in greater costs when a non-dominant thread acquires the lock. The main cost of acquiring an uncontended lock is the execution time of atomic compare-and-swap operations, so most biased locking algorithms allow the dominant thread to acquire the lock without atomic operations. Provided only the dominant thread accesses the lock, this works well. The main problem is how to deal with the rare case when a non-dominant thread accesses the lock.

2.1. Tiered Locking

One solution to this problem is to use locks such as Dekker’s [1968] algorithm which, on a uniprocessor, can synchronize two threads only without atomic operations. The Dekker algorithm requires memory fence operations on most real multiprocessors however, because such processors dynamically reorder memory operations to improve performance. As a result the order in which memory instructions are executed on processor \( A \) become visible on processor \( B \) is not necessarily in program order [Gharachorloo 1995].

In order to make Dekker’s algorithm more general, Onodera et al. [2004] presents a two-tiered locking algorithm. The dominant thread is in the first tier. In order to access the shared resource which it owns, it competes for a two process Dekker spin lock with one other thread. All other threads are in the second tier. To acquire the Dekker lock, they must first compete with one another for a conventional spin lock. Once they have obtained this lock, they may compete with the dominant thread for the two process Dekker lock, which grants mutually exclusive access to the shared resource.

For Dekker’s algorithm to work correctly memory fences are needed to place an ordering on (or serialize) memory operations at certain points. The level of memory fencing, and type, required depends on the architecture. For example, the Intel x86 architecture has a stricter memory model than Sun SPARC\textsuperscript{TM} or IBM Power\textsuperscript{TM} architectures, and for many algorithms requires less memory-fencing to guarantee correctness [McKenney 2005]. Dekker’s algorithm still requires fences on x86 however, because x86, in common with most other widely-used modern architectures allows stores to be reordered after loads. These memory fence operations can be almost as slow as atomic operations. The exact cost of memory fences is highly architecture dependent. For example, on Power the full sync instruction must be used, which orders all instructions, since the lightweight memory sync instruction does not provide sufficient ordering. Meanwhile x86 provides instructions that order only the memory operations.

Algorithms where the dominant thread must execute memory-fence instructions each time it acquires or releases the lock may not be much faster than conventional locking algorithms. To achieve greater speed, some mechanism is needed to remove both atomic operations and memory-fences from the common path, that is where the dominant thread acquires the lock repeatedly.

2.2. Asymmetric Locking

Kawachia \textit{et al.} [2002] solve this problem with \textit{lock reservation}, where the dominant thread accesses the lock using normal loads and stores. Non-dominant threads that access the lock must explicitly stop the dominant thread using an operating system signal. Once the dominant thread is stopped, a signal handler is used to resolve the lock access. This mechanism is slow and complex, but provided that accesses by non-dominant threads are very rare, the trade-off can be worthwhile. This kind of approach to biased locking is referred to as asymmetric locking, which means the actions of another thread are always necessary for non-owner threads to access the critical section.
Dice et al. [2001] generalize the idea of lock reservation by introducing a primitive which they call “Serialize”. SERIALIZE() is an operation called by the non-dominant thread on the dominant thread, which forces the memory operations in the non-dominant thread to be serialized, that is to become visible to other threads in program order. Thus, rather than the dominant thread having to serialize its memory operations on every lock access, it only needs to do so when explicitly signalled by the non-dominant thread. SERIALIZE() is effectively a memory-fence remotely invoked on another thread. No current architecture implements the SERIALIZE() primitive. Instead it must be implemented by some mechanism such as interrupts, operating system signals, or waiting for the dominant thread to perform a context switch. All these mechanisms are slow, and require significant interaction with the operating system.

2.3. Asymmetric Tiered Locking

Vasudevan et al. [2010] combine the ideas of tiered locks and SERIALIZE() into a single scheme. Like Onodera et al. [2004] they use a two-tier locking scheme. They observe that any N-process lock can be used by the non-dominant threads for the second tier lock, and that any two-process lock can be used for the first level lock, giving rise to the simple algorithm shown in Figure 2.

The advantage of a tiered scheme is that the two-process locks, such as Dekker’s algorithm, do not require costly atomic read-modify-write operations. Furthermore, Vasudevan et al. implement a scheme similar to SERIALIZE() in order to save the dominant thread from having to use memory-fence operations when the lock is uncontended. They implement a 2 process synchronization protocol involving request and grant variables. A non-dominant thread setting the request variable is roughly analogous to calling SERIALIZE(). The dominant thread polls the request variable periodically (for example, each time it exits the critical section), and when it is set, it executes a memory barrier, and sets the grant variable, which signals to the non-dominant thread that it may enter the critical section, shown in Figure 3.
This is used in conjunction with tiered locking, as explained above. N non-dominant threads compete for an N process lock before they can set the request variable and enter the critical section when grant is set. When the non-dominant thread releases the lock, it sets grant to false, allowing the dominant thread to enter the critical section again. Note that the Vasudevan et al. algorithm does not eliminate the need for memory-fences entirely. The dominant thread must execute memory-fence operations when it grants access to the non-dominant thread. These memory fences are needed to ensure that writes to memory by the dominant thread become visible to other threads in the correct order. However, in the common case where the great majority of accesses are from the dominant thread, the dominant thread can repeatedly acquire and release the lock without serializing its memory operations.

3. OVERVIEW OF DESIGN VARIANTS
This paper presents a general asymmetric, or biased, locking scheme, together with a number of different algorithms and implementation variations. In this section we attempt to classify the different features of our asymmetric synchronization algorithms into a taxonomy of design choices. In general, there are two mechanisms for protecting access to a critical section - locks and message passing. Previous work on biased synchronization has all been focused on locks [Onodera et al. 2004; Kawachiya et al. 2002; Bacon et al. 1998; Dice et al. 2001]. That is, non-dominant threads gain access to shared resources and interact with them directly.

Our work focuses on message passing. That is, non-dominant threads interact with shared resources indirectly, by passing a message to the dominant thread, which interacts with the shared resource on their behalf. We outline two forms of message passing, namely (1) using a function pointer and (2) using an integer token and a switch statement. Thus all our algorithms are based on message passing and require the dominant thread to be active and processing requests to work correctly. In comparison the locks described in previous work do not require that the dominant thread be active; the non-dominant threads can proceed regardless of the dominant thread’s status.

The starting point for our designs is a synchronous function pointer-based biased lock, described in section 4. To this basic lock we then added various mechanisms to improve its performance for certain scenarios.

— Queues to allow multiple non-dominant threads to wait at a time (section 5).
— Asynchronous variants to allow the non-dominant thread to continue whilst waiting for a request to be serviced (section 6.1).
— Asynchronous queueing variants that combine the first two mechanisms (section 6.2).
— An integrated spinlock variant, based on the standard function pointer-based lock. In this variant the spinlock — which is used prevent simultaneous accesses from non-dominant threads — is also used to store the function pointer (section 6.3).

Instead of using a function pointer as the message between non-dominant and dominant threads, it is also possible to use a token (see section 7). The dominant thread can then decide, via a switch statement, what action to take. This allows any function called to be in-lined, providing a potential performance advantage, depending on the length of the function and the number of cases in the switch statement. All the variants above are applicable to the token-based system as well.

If an algorithm is asynchronous, it allows for more progress and fairness for non-dominant threads. Asynchronous algorithms can often not be used, however, as a non-dominant thread’s execution might be dependent on its critical section access completing.

Queue-based algorithms also improve fairness and progress for non-dominant threads, even if synchronous accesses are required, and especially if the algorithm is
also asynchronous. The overhead involved in pushing work to the dominant thread and the dominant thread polling for work may however be slightly higher than non queue-based algorithms depending on what queue is used. Queue choice is an important factor in both the speed and space efficiency of the algorithm. A lock free multiple-reader, single-reader queue eliminates the need for an N-process lock, for example, while a queue with no bound on its size could use up a lot of memory in some cases.

Finally we discuss caching shared variables in memory, which can provide considerable speed-ups in cases with high dominance and a small amount of shared data. It is not applicable to larger shared structures, potentially results in a less elegant interface, and might not provide such a large speed-up at slightly lower levels of dominance.

4. FASTER ASYMMETRIC THREAD SYNCHRONIZATION

Vasudevan’s asymmetric algorithm, shown in Figure 3, is fast and simple. When uncontended the dominant thread only checks two Boolean variables at the start and end of the critical section. When contended memory barriers are required to transfer control to a non-dominant thread, and again to return to the the dominant thread. The lock variables must also be moved between the caches of the dominant and non-dominant threads causing cache misses. Building on this, we explore methods of reducing locking overhead further, particularly for slightly lower levels of dominance, when the overhead of transferring control to non-dominant threads becomes more significant.

In this section we present: a general scheme for asymmetric synchronization, a variation of Vasudevan et al.’s asymmetric locking algorithm which fits into our scheme, and a number of algorithms which use an alternative approach to biased synchronization.

4.1. Our General Scheme

Our generalized asymmetric locking scheme is shown in Figure 4. Our scheme provides a mechanism to implement critical sections, which only one thread can access at a time, instead of providing lock/unlock primitives. The critical section mechanism has the disadvantage of being different, and conceptually slightly more complex than the standard lock-unlock pairing, impacting programmer productivity to an extent. However it allows implementation of a much wider variety of lock mechanisms, and once the code is written it becomes highly portable between this variety of mechanisms. Potentially this would allow the programmer to experiment and find the most suitable mechanism. Once within the critical section, the dominant thread executes the critical section immediately, in this case invoking a function pointer, and then polls to see whether any non-dominant threads are waiting for their critical sections to be executed. Meanwhile non-dominant threads call a push_work() function, notifying the dominant thread that they want access. Typically push_work() modifies a field of the lock structure to achieve this.

After analysis we found that for Vasudevan’s asymmetric locking algorithm much of the overhead in the dominant thread comes from checking if the lock is granted to any non-dominant thread (see line 2 of the biased_lock code in Figure 3). This check is made every time the dominant thread acquires the lock. However a biased lock is rarely granted to a non-dominant thread. Therefore, it is typically faster for the dominant thread to grant the lock and then wait for the non-dominant thread to release the lock before continuing. This avoids the initial check on whether the lock has been granted to a non-dominant thread. In other words it removes a check from the most frequently executed path of the dominant thread, and moves it to a less frequently executed part of the code. Figure 5 shows this mechanism in terms of our scheme.
4.2. Passing Messages

Consider what happens in the algorithm shown in Figure 5. To access the critical section a non-dominant thread sends a request signal to the dominant thread, and the dominant thread returns a grant signal, allowing the non-dominant thread to proceed. Thus when the dominant thread polls the lock, and thus receives any request signal, it then spins until the non-dominant thread releases the lock. The signal is sent via a shared memory location polled by the dominant thread, so it need not be a boolean value; it could be any type. Effectively the request variable is a message container, allowing the message itself to be a request to dominant thread asking it to perform work on behalf of the non-dominant thread. In this work we propose two different message types; function pointers, discussed in the next section, and tokens, discussed in section 7.

4.3. Function Pointer Passing

For this approach the non-dominant thread writes a pointer value into a variable shared with the dominant thread. The dominant thread then calls that function, executing the non-dominant thread's work on its behalf. Figure 6 shows the interface to this mechanism, including the poll and push_work functions. The non-dominant thread spins whilst waiting for the dominant thread to complete.

Every time the dominant thread enters the critical section, it checks if the shared variable is set; if so then it calls the function pointed to by the shared variable, signalling the non-dominant thread when function has completed. Note that this approach could be expressed more elegantly in a language which supports closures or continuations, by pushing the closure to the dominant thread, rather than a function pointer and separate parameters. Harris et al. [2011] describe the use of a similar technique, but for a different application.

We use memory fences to guarantee the order in which writes to memory by one thread become visible to other threads. Recall that in most modern processors different writes to memory do not necessarily become visible to other threads in program
Fast Asymmetric Thread Synchronization

```
0 struct Lock{
1     LockN n;
2     func_ptr * func;
3     Params * params;
4 }

0 poll (Lock * l, Data * d)
1     if(l->func != NULL) { //poll
2         l->func (l, d, l->params);
3         fence();
4         l->func = NULL;
5     }

0 push_work(Lock * l, func_ptr * fp, Params * p)
1     lockN(l->n);
2     l->params = p;
3     fence();
4     l->func = fp; // push work
5     while(l->func != NULL) /*spin*/;
6     unlockN(l->n);
7 }
```

Fig. 6. Asymmetric Function Pointer Passing

order, unless a memory fence or other memory sequentializing instruction is used to guarantee the order. A memory-fence is needed in `push_work()` to ensure that the parameters are visible to the dominant thread before the function's address becomes visible.

A fence is also needed after the function implementing the critical section has been executed by the dominant thread (see line 3 of the `poll` function in Figure 6). However, if the dominant thread is the sole maintainer of the shared data structure, it may be possible to further optimize the code by omitting this fence if no other thread ever interacts with the shared data. However, it will probably make the code more maintainable to leave the fence in place, as it may be difficult to guarantee that there will be no future changes to the program that affect the shared data.

Note that as with many other locking algorithms, we assume that a memory write from one thread will eventually become visible to other threads. We do not insert extra fences an attempt to force visibility of memory writes.

5. QUEUE-BASED VERSION

Even though non-dominant accesses to biased locks are rare, multiple non-dominant threads may still attempt simultaneous access to the lock. In this case, the non-dominant threads will stall, waiting for access to the lock. By en-queueing the requests instead, the dominant thread can execute all the requests on the queue each time it polls, rather than just a single request per poll, reducing the overhead of dealing with work requests. It may also be faster for the non-dominant threads, because they are not forced to wait for the dominant thread to poll the lock multiple times. Finally a queuing mechanism is fairer and reduces the chances of a non-dominant thread being starved.

An outline of the code for a queuing mechanism is shown in Figure 7. Note that the code in Figure 7 assumes a single-reader single-writer queue which is protected by an N-process lock. An alternative would be to use a lock-free single-reader multiple-writer queue. A technique that is used very successfully by MCS locks [Mellor-Crummey and Scott 1991] is to build the queue out of nodes which are local variables on the stack of waiting threads. This technique is less suitable for the proposed mechanism however. The main reason is that single-reader multiple-writer queues are much more complicated than the single-reader single-writer queues used in the proposed method. In particular both the dominant and non-dominant threads would need to use atomic compare-and-swap operations to manage the multiple-reader queue, whereas the proposed implementation places all the atomic operations in the non-dominant thread, improving performance for the dominant thread.


6. ASYNCHRONOUS VARIATIONS

In the algorithms presented in sections 4 and 5, the non-dominant thread spins while its work is executed by the dominant thread. However, in some applications it may not be necessary for the non-dominant thread to wait for the critical section to complete. For example, if the non-dominant thread’s variables are not modified in the critical section, then the non-dominant thread can, in theory, continue execution instead spinning. We refer to this as an asynchronous variation of our scheme. We consider two cases: first, the case where the dominant thread does not maintain a queue of waiting requests from non-dominant threads, and second where the dominant thread maintains such a queue.

6.1. Asynchronous Variant without a queue

In the case where there is no queue of waiting requests, a non-dominant thread can simply pass its request to the dominant thread and then continue executing. However to continue execution, however, the non-dominant thread must first release the N process lock protecting the shared function pointer variable, potentially allowing another non-dominant thread to overwrite the function pointer before the first thread’s function has completed. To prevent this we force all non-dominant threads to wait for the dominant thread to complete execution of any current function, signalled by the dominant thread setting the function pointer to NULL. All that needs to be modified in Figure 6 is to move line 5 to just above line 2 of the push_work function. As no variables local to the non-dominant thread are modified, the memory fence in line 3 of poll() can also be eliminated.

6.2. Asynchronous Variant with a Queue

The asynchronous algorithm in Figure 6 reduces blocking by non-dominant threads; if the lock is heavily contended blocking can still occur. By adding a queue to the asynchronous variant we can reduce blocking even further. Unlike the synchronous queuing algorithm, no done array is needed, simplifying the design somewhat. Also no fence is required in poll(), the function passed to the dominant thread is not permitted to change any variables local to the non-dominant thread for this queuing version. Code
for an implementation of this algorithm assuming a single-reader single-writer queue is given in Figure 8.

Note that we are not the first to build queues of work requests to perform critical sections. Shalev and Shavit [2006] proposed a scheme where threads add data structure updates to a queue, and use the information on the queue to predict the outcome of their request. More closely related, Hendler et al. [2010] use work queues in the design of concurrent data structures such as stacks and lists to reduce the cost of synchronization in a scheme called flat combining. They use a single lock to protect the entire data structure. When the lock is contended waiting threads put their requests onto a work queue and spin until their request is completed. The thread that acquires the lock becomes a combiner and is responsible for carrying out all outstanding requests. This allows them to combine multiple data structure updates into a single operation on the data structure, reducing the cost of updates. Where the lock is not contended, the acquiring thread pays the full cost of a normal lock. In contrast our approach is aimed at allowing the dominant thread to acquire access to the critical section without paying the full cost of a conventional lock.

Concurrently with the work described in this paper Lozi et al. [2012] developed another scheme called remote core locking that offloads the work of critical sections to a single server thread. When a given thread wants to perform a critical section, it does not execute the critical section itself. Instead the thread sets a flag in an array, and provides a function pointer to the function to be invoked to execute the critical section and a structure containing the parameters. The server thread spends its entire time spinning over the entries in this array, checking the flags. When the server thread discovers a set flag, it executes the corresponding critical section. Thus, all critical sections are executed by a single dedicated thread. Lozi et al. found that remote call locking can give significant speedups on some very heavily contended locks.

6.3. Integrating Function Pointer Passing with a Spinlock

This variant is conceptually similar to the function pointer passing variant, except that the function pointer and function parameters are contained in a specific C structure. Then the non-dominant thread attempts an atomic compare-and-swap on the lock variable using a pointer to this struct. This is shown in Figure 9. The push_work function is analogous to an optimistic spinlock using a spin count, although any other similar spinlock (such as a test-and-test-and-spin lock [Kruskal et al. 1988]) could be adopted to behave similarly. The poll() function used is the same as that used in Figure 6.
struct qnode{
    func_ptr * func;
    Params * params;
}

void push_work(Lock * l, func_ptr * fp, Params * p, int threadid)
{
    static qnode qns[NUM_THREADS];
    qn[threadid].func = fp;
    qn[threadid].params = p
    bool success = compare_and_swap (&l->qn, NULL, &qn[threadid]);
    while(!success) {
        for(int i = 0; i < SPIN_COUNT; i++) /*spin*/;
        success = compare_and_swap (&l->qn, NULL, &qn[threadid]);
    }
}

message_handler(int token, Lock * l, Data * d, Params * p){
    switch(token)
    {
        case 1: ...; break;
        case 2: ...; break;
        ...
    }
}

void critical_section (int threadid, Lock * l, Data * d,
        int msg, Params * params)
{
    if (threadid == l.owner) {
        message_handler (msg, l, d, params);
        poll (l, d);
    }
    else
        push_work (l, d, msg, params);
}

7. MESSAGE HANDLERS AND TOKENS

Our use of function pointers is flexible and dynamic, allowing the non-dominant threads to push any work they please to the dominant thread. There are some disadvantages to this approach however. These function pointer calls are not easily inlined, since the function pointer changes non-deterministically during program execution, so the function call overhead is unavoidable.

To avoid this overhead, we developed a system that passes tokens (e.g. integer values) instead of function pointers. The dominant thread then chooses one of a a set of inlined, predefined handlers, based on the token it is passed. This eliminates the function call overhead, at the cost of executing a switch statement. In our experiments GCC often generated more efficient code for the switch statement than for a function call. However, the interface is less clean than when using the function calls.

An example of a handler function is shown in Figure 10. The general scheme must change slightly to use a message handler, see Figure 11. Message handling can be used in all algorithm variations presented here, simply by replacing function pointer calls with a message_handler call.
8. BIAS TRANSFER

For many applications the dominant thread is not static; over time another thread may come to access the lock more frequently than the original dominant thread. By allowing lock ownership to change, we can improve performance for such applications. To achieve this the owner variable within the lock structure must be protected by the synchronization mechanism, ensuring ownership does not change while another thread is in the critical section, which would obviously lead to a violation of mutual exclusion.

In Onedara or Vasudevan’s schemes where non-dominant threads access the shared resource directly, ownership cannot be changed if the initial owner is about to enter the critical section, i.e. it is spinning and about to enter as soon as the lock is released, since this could allow the new dominant thread to enter at the same time.

For all our schemes all work on a shared resource is performed by the owner thread, making bias transfer much simpler. A thread requesting ownership passes a message, e.g. a function pointer such as the one in Figure 12 to the owner instructing it to modify the owner variable of the lock to the ID of the requesting thread. The initial owner makes this change in the poll() function and returns from critical_section(), switching ownership of the lock. Note that in asynchronous algorithms, the thread requesting ownership must make a one off synchronous critical section access; otherwise it could enter the critical section before the dominant thread has executed its message, meaning it would still be non-dominant, leading to deadlock.

9. CACHING SHARED DATA IN CPU REGISTERS

In our algorithms memory fences are necessary to ensure that any modifications, made by the dominant thread, to the non-dominant thread’s local variables become visible to the non-dominant thread before it resumes execution. No fencing is required for the dominant thread’s local variables, including any variables accesses inside the critical section, since they are never viewed outside that thread. Thus if the dataset is small enough, then the dominant thread’s shared data can stay cached for the entire duration of a program’s execution. If the dataset is even smaller, then the dataset can stay in the processors registers. This is not possible for Vasudevan’s algorithms since the shared data is touched by the non-dominant threads, causing it to be spilled from registers and/or cache.

10. DEADLOCK CONSIDERATIONS WITH MULTIPLE DOMINANT THREADS

The algorithms presented in this paper so far have all assumed one dominant thread protecting one shared resource. In a more realistic scenario, there might be several shared resources with different owner threads, each of which might access another’s resource occasionally. Deadlock is a possibility in this scenario, with threads stuck in a loop waiting for each other’s resources. The solution is simple. When threads which own shared data are spinning in a busy wait loop within push_work(), they must also call poll(), to ensure that they service requests from other threads waiting on an action from them.

Note that this solution to deadlock does not work for the variation of Vasudevan et al.’s algorithm, since if two threads are waiting on a grant signal from each other, and each enter their own poll() function to signal grant to the other, both might get stuck spinning waiting for their grant variable to be reset by the other. This so-
olution requires the passing of the thread’s own shared data and associated lock to the `critical_section` function and the `push_work` function. An object-based approach, where a thread’s shared data and locks were attributes could provide a cleaner interface for this.

11. EXPERIMENTS

11.1. Target Platforms

Most of our experiments were run on an Intel Core 2 (Sandy Bridge, CPU family 6, model 42) machine with four cores running at 2.10GHz. Each core has two-way simultaneous multi-threading. The machine ran Ubuntu Linux 11.04 (kernel 2.6.38-8-server). Benchmarks were compiled with GCC 4.5.2 using -O3 optimization.

To test the scalability of our algorithms, we also secured time on an early version of the many-core IBM PowerEN processor. The PowerEN Edge of Network processor was recently described by Franke et al. [2010]. PowerEN is an interesting architecture for biased-locking because it is an available highly multi-threaded processor with 64 hardware threads. Also, many of its target application areas, such as network intrusion detection and financial market data feed processing can benefit from biased-locking and asymmetric thread synchronization techniques. PowerEN integrates design features from both the server-type processors and network processors to address the domain of network facing applications.

The PowerEN has sixteen embedded 64-bit Power cores running at frequencies as high as 2.3GHz. Each core is comprised of four concurrent hardware threads that feed a dual issue in-order pipeline (1 Int + 1 AXU/FU instruction from different threads/cycle). There are also a number of dedicated hardware acceleration units: cryptography, data compression, pattern matching, XML processing, and packet processing. The accelerators or co-processors are attached to the chip-internal system bus and are cache coherent. They are synchronously or asynchronously invoked by means of a special accelerator assembly instruction. PowerEN supports the standard embedded IBM Power™ architecture, and runs Linux along with the standard Linux development tools. The large number of cores (16) and hardware threads (64) makes the PowerEN extremely suitable for testing the scalability of locking algorithms.

11.1.1. Number of Threads. Asymmetric synchronization algorithms are most useful in performance critical applications, where, typically, the number of software threads will match the number of hardware threads available on the system. Thus, the number of threads in our benchmarks are equal to the number of hardware threads available. We also ran some experiments using the taskset command to ensure just one thread ran per core, to measure the impact of SMT on our algorithms.

11.1.2. Which Unbiased N-Process Lock?. All our results are described as speed-ups over a baseline unbiased-lock mechanism. We initially used pthread spinlocks, as are used by Vasudevan et al. [2010]. However, after seeing unexpected results for the pthread spinlock, we found that that after a thread fails a number of times to acquire the lock, it suspends itself and thus falls back on the OS scheduler, which is undesirable for performance measurements. Thus, we implemented a simple, optimistic spinlock with a spin count (Figure 13), and used that for the N process lock in our algorithms and as our basis for comparison.

11.2. Performance Results

11.2.1. Incrementing Benchmark. In this benchmark, each thread accesses the critical section a fixed number of times, incrementing the shared resource (an integer) on each access. The relative performance, in terms of overall execution time, of each algorithm
spinlock::lock (int * n)
bool success = compare_and_swap(n, 0, 1);
while(!success) {
    for(int i = 0; i < SPIN_COUNT; i++) /*spin*/;
    success = compare_and_swap(n, 0, 1);
}

Fig. 13. Unbiased spinlock used in benchmarks

compared with an unbiased spinlock is shown in Figures 14 and 15. The speed-ups gained reduce quite linearly as dominance decreases, which is to be expected. Our message passing algorithms perform better than lock-based algorithms as dominance lowers slightly and as the cost of transferring direct access to another thread increases.

Our asynchronous algorithms in Figure 15 have higher speed-ups at lower dominances, when compared to the standard algorithms in Figure 14. Detailed analysis of the performance data showed that, when not using asynchronous non-dominant thread access, the access frequency of the non-dominant threads dropped significantly with falling dominance. This is due to poor handling of contention between the non-dominant threads. In comparison, when asynchronous access was used, relative access frequency for non-dominant threads was much better preserved with decreasing dominance, since the asynchronous critical section access allows the algorithm to handle contention much more efficiently.

Our queue-based asynchronous algorithms shown in Figure 15 have higher speed-ups at lower dominances, when compared to the standard algorithms shown in Figure 14. We examined the performance in detail and found that at lower levels of dominance, the non-dominant threads using synchronous locking algorithms become starved because they are spending large amounts of their time waiting for the dominant thread to poll the lock. However, the asynchronous algorithms that place work on a queue continue to perform well at lower levels of dominance because the non-dominant thread can proceed without waiting for the dominant thread to complete a polling operation. The exceptions are the integrated spinlock schemes, which use a queue of only one item to reduce the cost of polling the list. Our results suggest that although these schemes look appealing in principle, in practice the performance is not good except at very high levels of dominance, where synchronous algorithms also perform well.

11.3. Performance with Bias Transfer

For this benchmark we measured the performance of our bias transfer mechanism. After a certain number of critical section accesses, the initially dominant thread signals to other threads that it is about to become less dominant, at which point another thread switches to being dominant and accesses the critical section much more frequently. One bias switch is performed per run. The results are shown in Figure 16. The speed-ups are more or less identical to Figure 14, demonstrating that allowing for bias transfer in our asymmetric synchronization algorithms comes with virtually no extra overhead.

11.4. Performance on a highly parallel architecture - PowerEN

Figure 17 shows the results of running the benchmarks shown in Figure 14 on the emerging IBM network optimized processor architecture, PowerEN. PowerEN’s many-core architecture supporting 64 hardware threads, makes it an ideal platform to test the performance of our asymmetric thread synchronization algorithms on a highly parallel processor. Furthermore many of the potential applications of PowerEN, such
as network intrusion detection, financial market data stream processing and optimized network processing for cloud data centres can make use of the techniques described in the current paper [OPRA 2011] [Mukherjee et al. 1994] [Golander et al. 2010].

The benchmarks were run using 64 threads on prototype PowerEN hardware. At high dominance levels, the techniques described in this paper deliver significant speed-ups over standard spin-locks, with asynchronous message passing being a particularly successful technique. The speed-ups degrade quickly with decreasing dominance how-
ever. This most likely due to the large number of threads on PowerEN competing over a single lock. The high speed-ups achieved for high dominance rates suggest that further analysis and optimization of these algorithms for many-threaded architectures such as PowerEN is promising area for future work.
11.5. Multiple Owners of Multiple Shared Resources

This benchmark measures the performance of our algorithms in a scenario closer to a realistic application, such as a packet processor for a network intrusion detection system [Mukherjee et al. 1994], or a financial market data feed processing system [OPRA 2011]. Each thread owns a shared resource and accesses it frequently. Each thread also accesses other threads’ data at a much lower frequency. We measured the impact of our algorithms on this scenario at different access frequencies by non-dominant threads. Critical sections consist of incrementing a variable.
A graph of our results is show in Figures 18 and 19. By “Non-dominant Access Period”, we mean that for every n accesses to a dominant thread’s own shared resource, it makes one access to another thread’s shared resource.

We note very high speed-ups for low access frequencies to non-owned shared data. We also note the performance of our message passing algorithms degrading less than Vasudevan et al.’s asymmetric lock as dominance lessens. These results are significant since even for quite low dominance levels, our techniques show substantial speed-ups for the most common application area of our asymmetric thread synchronization techniques.

Finally to evaluate the advantages of these techniques for a realistic application we tested them in the scheduler for a lightweight fiber library. A fiber is a program execution context including a program counter and stack that can be paused, stored as a continuation, and subsequently resumed. Fibers can be used to implement user-level threads, where hundreds or thousands of fibers can run on a single operating system (OS) thread, using cooperative switching to switch between fibers in a thread.

We evaluated our techniques in the context of a multi-threaded fiber library, where a very large number of fibers run on a much smaller number of OS worker threads. Each worker thread has a scheduler which maintains its own queue of ready fibers. When the currently-running fiber is suspended (or completes) the scheduler on that worker thread takes a fiber from its ready queue. If, however, the worker thread’s ready queue is empty, the scheduler instead tries to steal work from another worker thread’s ready queue. Given that multiple worker threads may attempt to access the same ready queue at the same time, it is necessary to protect the ready queue with some sort of synchronization mechanism. However, it is important to note that almost all of the accesses to a ready queue will be from the worker thread that owns the queue, and that work-stealing accesses from other worker threads will usually be rare.
suggests that asymmetric thread synchronization mechanisms would be very suitable for this problem\(^1\) [Frigo et al. 1998].

Note that in any fiber or other lightweight threading system the scheduler consumes only a very small proportion of total time. One of the major advantages of fibers is that they minimize thread switching time even when there are tens of thousands of fibers, and therefore most of the time is spent doing the work in the running fiber, not switching between them. Nonetheless, it is essential that the scheduler is as efficient as possible because we want to minimize overheads from the fiber system. In order to make the execution time of the scheduler more visible, we ran very large numbers of fibers on the system that each performs only a little work before yielding control back to the scheduler. This allows us to measure the overhead of the scheduler much more precisely, and allows us to optimize it to minimize overhead using our techniques.

Figure 20 shows benchmark results for the fiber library using pthread spinlocks, Vasudevan-style locks, and our function-pointer and token passing locks. The fibers are running a simple task; each fiber immediately yields after loading. This is not a real-world workload, but we are interested in the effect the different locks have on the work-scheduling algorithm and having the fiber perform real-world work would mask this effect. The benchmark was run on a dual-socket machine with two Intel Xeon quad-core processors, with each core run dual-way hyper-threaded, resulting in 16 hardware threads. Each experiment was run with 16 worker threads, and each worker thread was given a unique CPU affinity via the pthreads library, to give more consistent results between runs.

As expected for a complex and dynamic work scheduling algorithm, there is more variability than for the micro-benchmarks used elsewhere in the paper, and the results are not as clearly defined. Also the ratio of dominant accesses to non-dominant accesses is dynamic for these benchmarks; it is primarily determined by the ratio of worker threads to fibers. Clear trends are evident in the results however. With one slight exception the biased locks are faster than spinlocks, with Vasudevan-style locks and our function-pointer and token-passing mechanisms all having similar performance. As expected the biased locks return a good speed-up over spin-locks when non-dominant access is low (i.e. fibers greater than workers). However the biased locks return the best speed-up for the case when attempted non-dominant access is highest (i.e. workers greater than fibers) as the biased mechanisms give fairer queue access and better prevent the worker pool turning into a den of thieves, where workers spend their time attempting to steal scarce work from each others queues, instead of running fibers.

1. CONCLUSIONS

Building on previous lock reservation and biased and asymmetric locking work, we have presented a new, novel approach to biased synchronization which is roughly analogous to a message passing approach, along with two mechanisms for passing work to the owner thread and a taxonomy of different design choices which affect the performance of our algorithms in different ways. Our algorithms have a number of unique advantages compared to previous biased locking work. The ability to cache small amounts of shared data in an owner thread’s CPU’s registers is a potentially very useful advantage to our schemes.

Another unique advantage is the ability for non-dominant threads to make asynchronous critical section accesses. A potential application of asymmetric synchronization mechanisms such as these is in packet processing applications such as intrusion detection systems in which pipelining and flow-pinning are used [Intel 2006; Wun et al.]

\(^1\)An alternative to locking in the implementation of queues for work-stealing is to use lock-free data structures, such as in work on the Cilk language by Frigo et al.
In applications like these, different threads own different sections of larger data structures, such as TCP packet reassembly tables. While the great majority of accesses will be by a section’s owner, occasionally another thread might need access to update something in another’s section, an obvious lopsided access scenario in which biased locking would be useful. Our benchmarking of the fiber library shows that for a typical job scheduling algorithm, biased locking mechanisms return significant, measurable performance advantages.

Applications processing live market data feeds would also benefit from this approach. In a standard Options Price Reporting Authority (OPRA) feed decoder, where high rates of UDP packet ingestion is common (typically gigabits per second), the need for parallel processing is evident [OPRA 2011]. The dominant thread in this case performs workload distribution activities and notifications to tasks higher up the processing pipeline (i.e. the actual client application for the feed decoder). This leaves the non-dominant threads free to focus on actual message decoding, using biased locking to inform the dominant thread of decode success, thread readiness, and similar activities.

Our bias transfer mechanism requires almost zero overhead, as it is simply a special case of passing work, which could be very useful in situations where different threads periodically become dominant.

Our experiments show that overall, our biased synchronization mechanisms compare very well with biased locking algorithms at very high dominance, and out-perform Vasudevan et al’s biased locking at lower dominance levels. We have also shown allowing non-dominant threads to push work to the non-dominant thread in an asynchronous, non-blocking manner delivers further performance improvements, particularly as dominance levels fall. This is a particularly important result, since it makes these techniques potentially applicable to a wider range of applications with lower levels of critical section access dominance.

Finally we demonstrate that for applications with multiple critical sections, with each accessed by a dominant thread, we measure speed-ups of up to 19x over standard spinlocks in our micro-benchmarks for high levels of dominance and scales very well with decreasing dominance. We built on this result to measure performance for a typical highly-parallel PowerEN use-case, and showed substantial improvements for biased locks on the PowerEN architecture.

Our queue-based algorithms perform well alongside our other algorithms, however, we feel there is more potential in using queues, particularly for a full application where the threads perform significant amounts of work. We plan on experimenting with different forms of lockless queues in the future. In particular, a fast, lock-less multiple-writer single-reader queue could potentially offer an optimal solution when allowing asynchronous critical section accesses.

ACKNOWLEDGMENTS

This work was supported in part by a 2010 IBM Faculty Award to David Gregg, and in part by Science Foundation Ireland grant 10/CE/I1855 to Lero — The Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES


A:22 J. Cleary et al.


OPRA. 2011. Options price reporting authority website.


Received July 2011; revised April 2012; accepted August 2012