Flat-Combining NUMA Locks

Dave Dice
Oracle Labs
dave.dice@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Nir Shavit
Oracle Labs
nir.shavit@oracle.com

ABSTRACT

Multicore machines are growing in size, and accordingly shifting from simple bus-based designs to NUMA and CC-NUMA architectures. With this shift, the need for scalable hierarchical locking algorithms is becoming crucial to performance. This paper presents a novel scalable hierarchical queue-lock algorithm based on the flat combining synchronization paradigm. At the core of the new algorithm is a scheme for building local queues of waiting threads in a highly efficient manner, and then merging them globally, all with little interconnect traffic and virtually no costly synchronization operations in the common case. In empirical testing on an Oracle SPARC Enterprise T5440 Server, a 256-way CC-NUMA machine, our new flat-combining hierarchical lock significantly outperforms all classic locking algorithms, and at high concurrency levels, provides up to a factor of two improvement over HCLH, the most efficient known hierarchical locking algorithm.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

hierarchical locks, queue locks, flat combining

1. INTRODUCTION

Queue locks [1, 2, 3, 4], and in particular the CLH [2, 3, 4] and MCS [3] locks, have long been the algorithms of choice for locking in many high performance systems. They are known to reduce the overall cache coherence traffic by forming queues of threads, each spinning on a separate memory location as they await their turn to access the critical section.

Current trends in multicore architecture design imply that in coming years, there will be an accelerated shift towards distributed nonuniform memory-access (NUMA) and cache-coherent NUMA (CC-NUMA) architectures. Such architectures, examples of which include Intel's 4 chip/32 way Nehalem-based systems and Oracle's 4 chip/256 way Niagara-based systems, consist of collections of computing cores with fast local memory (as found on a single multicore chip), communicating with each other via a slower (inter-chip) communication medium. Access by a core to the local memory, and in particular to a shared local cache, can be several times faster than access to the remote memory located on another chip [5].

Radović and Hagersten [5] were the first to show the benefits of designing locks that improve locality of reference on CC-NUMA architectures by developing *hierarchical locks*: general-purpose mutual-exclusion locks that encourage threads with high mutual memory locality to acquire the lock consecutively, thus reducing the overall level of cache misses when executing instructions in the critical section.

Radović and Hagersten introduced the hierarchical backoff lock (HBO): a *test-and-test-and-set* lock augmented with a new *backoff scheme* to reduce contention on the lock variable. Their hierarchical backoff mechanism allows the backoff delay to be tuned dynamically, so that when a thread notices that another thread from its own local cluster owns the lock, it can reduce its delay and increase its chances of acquiring the lock consecutively. However, because the locks are test-and-test-and-set locks, they incur invalidation traffic on every modification of the shared global lock variable, which is especially costly on NUMA machines. In their work [5], Radović and Hagersten did introduce a heuristic technique to throttle inter-chip coherence traffic. However, as we show in our evalution (Section 3), it does not necessarily translate to better scalability. Moreover, the dynamic adjustment of backoff delay time in the lock introduces significant starvation and fairness issues: it becomes likely that two or more threads from the same cluster will repeatedly acquire a lock while threads from other clusters starve. Radović and Hagersten also introduced a heuristic to improve fairness, but this requires fine tuning of the backoff parameters, which can change with the underlying application's characteristics.

Luchangco et al. [6] overcome these drawbacks by introducing a hierarchical version of the CLH queue-locking algorithm (HCLH). Their HCLH algorithm collects requests on each chip into a local CLH style queue, and then has the thread at the head of the queue integrate each chip's queue into a single global queue in a highly effective manner. This avoids the overhead of spinning on a shared location and eliminates fairness and starvation issues. The algorithm's drawback is that it forms the local queues of waiting threads by having each thread perform an atomic *register-to-memory-swap* (SWAP) operation[1] on the shared head of the local queue. These SWAPs to a shared location cause a bottleneck and introduce an overhead, implying that the thread merging the local queue into the global one must either wait for a long period (10s of microseconds) or globally merge an unacceptably short local queue. Furthermore, HCLH has complex condition checks along its critical execution path in order to determine if a thread must perform the operations of merging local CLH queues with the global queue. All of these drawbacks result in performance degradation, which takes away some of the benefits of the HCLH lock's locality of reference for operations in the critical section. Nevertheless, as we show empirically, the HCLH algorithm can improve on the original MCS and CLH locks, as well as the HBO lock, by a factor of 2 to 3.

This paper presents the *flat-combining MCS lock* (FC-MCS), a new hierarchical queue-lock design based on a combination of the flat-combining synchronization paradigm [7] and the MCS lock algorithm [3]. Flat combining is a novel mutual-exclusion based client-server style synchronization paradigm introduced to speed up operations on shared data structures. In this paper, we provide the first use of flat combining to add scalability to locks.

The key algorithmic breakthrough in our work is a new efficient way for threads to use a flat-combining methodology to build MCS-style local queues of waiting threads, each spinning on its own node, and then splice them seamlessly into a global MCS queue. In our new algorithm, threads spin on their local nodes while attempting to select one thread as a designated combiner. This combiner in turn constructs a local queue by collecting the requests of all spinning threads, and then splices this queue into the global one.

The use of the combiner approach at high concurrency levels allows our algorithm to overcome the main drawback of HCLH: threads are collected into the local queue quickly by allowing threads to post requests in parallel using only a simple write (without even a write-read memory barrier) to an unshared location, as opposed to using sequences of SWAP operations on a shared location to create the local queues in HCLH. This allows the combiner thread, the one putting together the local queue, to form relatively long local queues (empirically we found them to contain 90% of locally waiting threads), and to do so with little delay. Moreover, the common case critical path in the new algorithm is significantly shallower than HCLH, which, as is typical with locking algorithms, has a measurable effect on performance.

In a set of tests conducted on an Oracle SPARC Enterprise T5440 Server, a 256-way CC-NUMA multicore machine, we found that our new FC-MCS hierarchical locking algorithm significantly outperforms all prior locking algorithms. In particular FC-MCS can outperform the previous best hierarchical lock, the HCLH lock, by a factor of 2 at high concurrency levels.

The one drawback of our new FC locking algorithm is that unlike prior hierarchical locks, the use of the flat combining structure is not memory efficient: if multiple locks are being accessed, a thread will have to keep a node per lock for any lock it is repeatedly accessing, and these nodes will be recycled only after a thread has ceased to access a given lock. Our new FC-based locks are therefore unsuited for applications in which memory resources are limited.

We describe our algorithm in detail in Section 2. This includes our basic algorithm that combines flat combining and MCS lock algorithms, and a key optimization that enables good performance at low contention levels. Section 3 presents our experimental results which show that our FC-MCS lock can outperform the best of existing locks by up to a factor of 2. We conclude in Section 4.

[1]On some architectures the SWAP operation is emulated using repeated *compare-and-swap* operations in a loop.

2. THE NEW HIERARCHICAL LOCK ALGORITHM

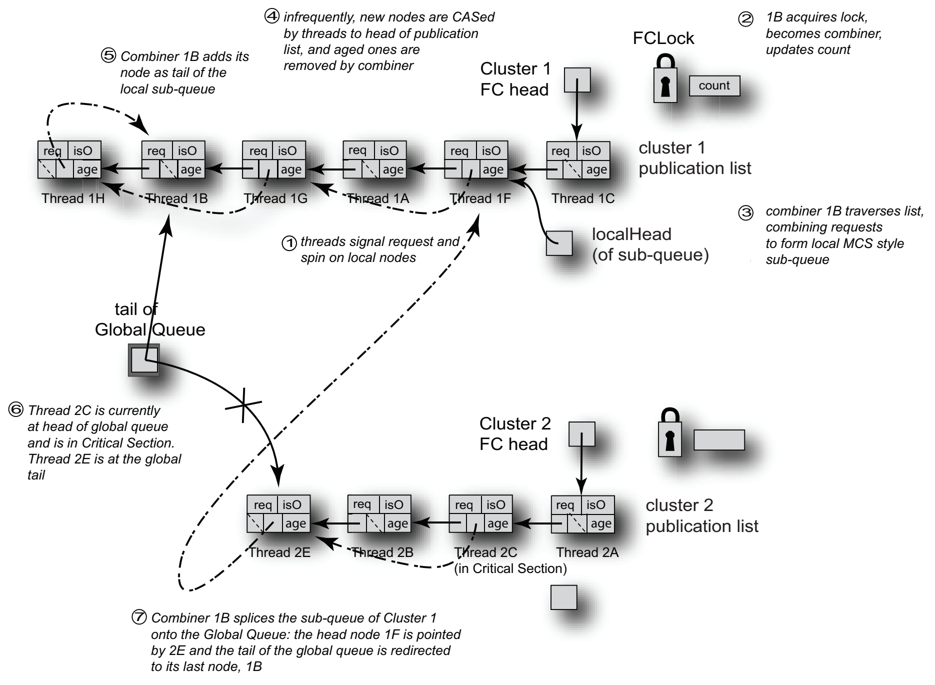
In this section, we describe our new flat-combining based hierarchical locking algorithm in detail. We assume that the system is organized into clusters of computing cores, each of which has a large cache that is shared among the cores local to that cluster, so that inter-cluster communication is significantly more expensive than intra-cluster communication. We use the term cluster to capture the collection of cores, and to make clear that they could be cores on a single multicore chip, or cores on a collection of multicore chips that have proximity to the same memory or caching structure; it all depends on the size of the NUMA machine at hand. We will also assume that each cluster has a unique *cluster id* known to all threads on the cluster.

Hendler et al. [7] showed how, given a sequential data structure, one can design a *flat combining* (henceforth FC) concurrent implementation of the structure that incurs very low synchronization overheads. The FC implementation uses mutual exclusion to repeatedly pick a unique "combiner" thread that will apply all other threads' operations to the structure. In our new FC-MCS algorithm we devise a simplified and streamlined variant of the FC algorithm and use it to effectively construct a local queue among threads in a given cluster. Combiners from various clusters will then repeatedly merge their local queues into a single global queue that will have virtually the same efficient handover structure as an MCS queue lock [3]. As we show, the use of flat combining will allow us to reduce overheads and introduce parallelism into the local queue creation process, and this parallelism, in turn, allows us to deliver improved performance.

2.1 Flat-Combining Local Queues

Our hierarchical FC-MCS lock consists of a collection of local flat-combining queues (FCQueues), one per cluster, and a single global queue (GlobalQueue). Figure 1 depicts an example of our lock construction for two clusters. We begin by describing the key elements of our algorithm, whose pseudocode appears in Figures 2 and 3. For the sake of clarity, some of the details presented here are omitted from the pseudocode.

As seen in Figure 1, each instance of a FCQueue consists of: a local FCLock, a *count* of the number of combining passes, a pointer to the *head* of a *publication list*, and pointers to the localHead and localTail of a sub-list. The publication list consists of thread specific nodes of a size proportional to the number of threads that are concurrently accessing the lock. Though one could implement the list in an array,



④ infrequently, new nodes are CASed by threads to head of publication list, and aged ones are removed by combiner
② 1B acquires lock, becomes combiner, updates count
⑤ Combiner 1B adds its node as tail of the local sub-queue
FCLock
count
Cluster 1 FC head
cluster 1 publication list
req
isO
age
Thread 1H
Thread 1B
Thread 1G
Thread 1A
Thread 1F
Thread 1C
③ combiner 1B traverses list, combining requests to form local MCS style sub-queue
① threads signal request and spin on local nodes
localHead (of sub-queue)
tail of Global Queue
⑥ Thread 2C is currently at head of global queue and is in Critical Section. Thread 2E is at the global tail
Cluster 2 FC head
cluster 2 publication list
Thread 2E
Thread 2B
Thread 2C (in Critical Section)
Thread 2A
⑦ Combiner 1B splices the sub-queue of Cluster 1 onto the Global Queue: the head node 1F is pointed by 2E and the tail of the global queue is redirected to its last node, 1B

Figure 1: A Hierarchical FC Queue Lock for two clusters. At the start of the depicted scenario, Thread 2C is in the critical section. Upon release it will update the isOwner (isO in the figure) field of node 2E, which is at the tail of the global queue. The scenario starts when Thread 1B becomes a combiner in Cluster 1, collects a local list consisting of nodes 1F, 1G, and 1H. It then adds its own node 1B as the last in the sublist. Finally, it splices this sublist onto the global queue by setting node 2E to point to its local list head node 1F, and setting the globalTail to 1B. The global queue at the end of the scenario is the one along the dotted pointers.

the dynamic publication list using thread specific pointers is necessary for a practical solution because the number of potential threads is unknown, and is typically much greater than the array size. Using an array one would have forced us to solve a renaming problem [8] among the threads accessing it. This would imply a CAS per location, which would give us little advantage over existing techniques. We therefore have a list rather than an array.

Moreover, in our case, using the list has an important secondary advantage: the same nodes used in the flat combining publication list will be used in the sub-list of the MCS-style queue locking algorithm. This means that threads will spin on the same nodes in both the combining and lock-awaiting phases.

Each thread t accessing the local FCQueue in a cluster executes the following sequence of steps, which allows the construction of a local list of threads, each spinning on a given node, waiting for its predecessor to notify it that the critical section is free. The following then is the local FC-Queue algorithm for a given thread t:

1. Thread t first indicates its lock acquisition request in the requestReady field of its thread specific publication node (no need to use a load-store memory barrier). If t's thread specific publication node is marked as active, t continues to step 2, otherwise it continues to step 5.
2. t checks if the local FCLock is taken. If so (another thread is an active combiner), t spins on the requestReady field waiting for a response to the lock acquisition request (one can add a yield at this point to allow other threads on the same core to run). Once in a while, while spinning, t checks to see if the lock is still taken and that its publication node is active. If the node is inactive, t proceeds to step 5. Once the response is available (in the form of requestReady getting reset to false), t proceeds to spin on its isOwner field, waiting for it to be set by its predecessor thread in the queue.
3. If the local FCLock is not taken, t attempts to acquire it and become a combiner. On failure it returns to spinning in step 2.
4. Otherwise, t holds the FCLock and is the combiner. Thereafter, t
   - increments the combining pass *count* by one.
   - traverses the publication list (our algorithm guarantees that this is done in a wait-free manner) from the publication list head, combining all "ready" (indicated via the requestReady flag) acquisition requests into an ordered logical queue (pointed to by the localHead and localTail from Figure 2), setting the *age* of each of these nodes to the current *count*, and resetting the requestReady fields after notifying each thread who its successor in the list is, that is, the node it must notify upon leaving the global critical section.

```
myFCNode.isOwner = false;
myFCNode.canBeGlobalTail = false;
myFCNode.requestReady = true;

FCNode localTail = NULL;
FCNode localHead = NULL;

// lock acquire code
while (true) {
  if (myFCNode is inActive) {
    InsertFC(myFCNode);
  }
  if (FCQueue.FCLock not Acquired) {
    if (CAS(FCQueue.FCLock, Free, Acquired)) {
      if (myFCNode.requestReady) {
        // become the flat combiner
        FCQueue.count++;
        for MaxCombiningIterations do {
          for each FCNode in FCQueue do {
            if (FCNode.requestReady == true
               && FCNode != myFCNode) {
              // add FCNode to the local wait queue
              if (localHead == NULL) {
                localHead = FCNode;
                localTail = FCNode;
              } else {
                localTail.next = FCNode;
                localTail = FCNode;
              }
              FCNode.age = FCQueue.count;
              FCNode.requestReady = false;
            } else {
              if (FCQueue.count - FCNode.age >
                  threshold) {
                remove FCNode from FCQueue;
              }
            }
          }
        }
        // add combiner's FCNode to the local
        // wait queue
        localTail.next = myFCNode;
        localTail = myFCNode;
        myFCNode.canBeGlobalTail = true;
        myFCNode.requestReady = false;
        // splice the local wait queue into
        // the global wait queue
        prevTail= SWAP(globalTail, localTail));
        if (prevTail != NULL) {
          prevTail.next = localHead;
        } else {
          localHead.isOwner = true;
        }
      }
      // release the FCQueue.FCLock
      FCQueue.FCLock = Free;
    }
  }
  if (myFCNode.requestReady == false) {
    break;
  }
}
// wait to become the lock owner
while (myFCNode.FCLock.isOwner == false);
```

Figure 2: Acquiring an FC Hierarchical Lock.

- At the end of the traversal, t enqueues its node at the localTail of the local queue, and sets its node's canBeGlobalTail flag. As we describe later, this flag is used by the lock releaser to determine if it needs to read the globalTail during the lock release operation. (In Figure 1, Thread 1B becomes a combiner in cluster 1, collects a local list consisting of nodes 1F, 1G, and 1H. It then adds its own node 1B as the last in the sublist.)
- If the *count* is such that a cleanup needs to be performed, t, during its traversal of the publication list, starting from the second item (as we explain below, we always leave the item pointed to by the *head* in the list), removes from the publication list all nodes whose *age* is much smaller than the current *count*. This is done by removing the node and marking it as inactive.
- Thread t releases the FCLock.

```
if (myFCNode.canBeGlobalTail == true) {
  while (true) {
    if (globalTail == myFCNode) {
      if (CAS(globalTail, myFCNode, NULL) == true) {
        // cleanup CAS succeeded
        break;
      }
    } else {
      // lock handoff
      if (myFCNode.next != NULL) {
        myFCNode.next.isOwner = true;
        break;
      }
    }
  }
} else {
  // lock handoff
  myFCNode.next.isOwner = true;
}
```

Figure 3: Releasing an FC Hierarchical Lock.

5. If thread t has no thread specific publication node, it allocates one, marked as active. If it already has one marked as inactive, it marks it as active. Thread t then executes a store-load memory barrier, and proceeds to insert its node into the head of the FC publication list by repeatedly attempting to perform a successful CAS on it. If and when t succeeds, it proceeds to step 1.

The combiner pass creates the local queue. As seen in cluster 1 in Figure 1, this local queue does not necessarily include all the nodes in the publication list. For example, node 1A is in the publication list but not in the local queue.

Notice that nodes are added to the publication list using a CAS only at the head of the list, and so a simple wait-free traversal by the combiner is trivial to implement [9]. Thus, the removal will not require any synchronization as long as it is not performed on the node pointed to from the head: the continuation of the list past this first node is only ever changed by the thread holding the global lock. Note that the first item is not an anchor or dummy node, we are simply not removing it. Once a new node is inserted, if it is unused it will be removed, and even if no new nodes are added, leaving it in the list will not affect performance.

The common case for a thread is that its node is active in the publication list and some other thread is the combiner, so it completes in step 2 after having only performed a store and a sequence of loads ending with a single cache miss. The end result is that there is a lot of parallelism, and little synchronization overhead, in the process of collecting nodes into the local queue, which in turn translates into reduced access time and longer local queues, giving our algorithm its advantage over the HCLH algorithm.

2.2 The Global Queue

Our global queue works in MCS style as opposed to the CLH style of HCLH. By this we mean that a thread spins on a field of its own node, notifies its successor upon leaving the critical section, and re-uses its own node in the next access attempt. This is in contrast with HCLH and CLH, where threads spin on the nodes of their predecessors, update their own node, and using their predecessor's released node in the next attempt. The choice of an MCS style lock is necessary and serves several roles in our algorithm: in the FC publication list, the lock access list, and the global queue. In order to remain in the publication list, a thread must use its own node repeatedly in all accesses, a property that holds for queuing in MCS style but not CLH style.

Our splicing approach into the global queue will create an MCS global queue, spanning all the clusters' local queues, in which each thread knows only of its immediate successor, and yet all nodes are ordered in a global way that enhances chances that nodes from a given cluster follow one another. As in the HCLH algorithm, this is key to our algorithm's ability to exploit data locality to achieve better performance. If one can form large collections of requests from the same cluster, the algorithm can minimizes the lock handoff intervals.

To create the global queue, each combiner splices the node pointed by the localTail of its local queue into the node at the global queue's (globalTail), logically moving an entire collection of nodes into the global queue. After splicing the sub-queue into the global queue, the combiner spins on its node's isOwner flag.

In the original MCS lock algorithm, a lock releaser checks the globalTail to determine if its node is the last in the MCS queue, and if so, CASes globalTail to null. We note that this lookup of globalTail may lead to a cache miss and a bus transaction if globalTail was last modified by a remote cluster thread. Furthermore, we also observe that in our algorithm, since the local queue combiner is at the localTail, no other node in that local queue can be the last node in the global MCS queue. Hence, none of the intermediate nodes in the sub-queue needs to make the above mentioned check of globalTail. We use the canBeGlobalTail field to indicate to a thread if its node can possibly be the last node in the global MCS queue. The combiner sets its node's canBeGlobalTail field to true, and then checks the globalTail in its lock release operation. All other threads do not need to do the check.

If the lock releaser's node has a successor, the releaser sets the successor's isOwner field to true, thus handing over the lock ownership.

2.3 Adjustments for Low Concurrency

The scheme we describe above works exceptionally well at high concurrency levels. However, at low concurrency levels, combining of requests into sufficiently large sub-queues becomes impossible, leaving the unwanted overhead of attempting to combine. Our solution is a simple one: at low concurrency levels let threads skip the attempt to combine and access the global queue directly. The elegant part in the FC-MCS algorithmic design is that the batches of nodes are added by combiners seamlessly, still maintaining the properties of an MCS queue with respect to all others, which allows individual threads to access the global queue and add themselves to it in a straightforward manner. In the combined algorithm, threads can apply a simple local test to determine of they should combine or attempt to access the queue directly. Our choice was to have threads count the size of the sub-queue when they were last combiners, and if the size is low several times, they switch to direct access with high probability. If in some access the combined queue is large, they switch back.

In summary, our new FC hierarchical locking algorithm creates an MCS style global queue which, at high concurrency levels, builds long sublists of adjacent local nodes, delivering good locality of reference for threads accessing the critical section. It does so in a highly efficient way by parallelizing the process of creating the local sub-queues on each cluster using flat combining.

2.4 Comparison to HCLH

At a high level, the design of FC-MCS is very similar to that of HCLH: Build local wait queues. Designate a "master" for each local queue, which splices the local queue in the global queue. Finally, let each thread spin on its node (or its node's predecessor) to determine if the thread has acquired the lock.

However, we observe that there are two key differences between FC-MCS and HCLH:

- The methods of building the local queues are drastically different in HCLH and FC-MCS. The HCLH local queue is pretty much a CLH queue, where each thread atomically (using the SWAP instruction) adds itself in the queue. This leads to contention on the tail of the CLH queue, which, as we will see in Section 3, becomes a performance bottleneck. The FC-MCS local queue, however, is built by the combiner by simply perusing through the flat combining list. As a result, there is almost no synchronization required to build local queues; the exception being the atomic operation needed to determine the combiner thread (essentially a test-and-test-and-set lock acquisition). All the non-combiner threads simply post their requests (using a store instruction, without any store-load barriers) in their respective flat combining request nodes. This design methodology eliminates the contention bottleneck we observe in HCLH and all the other queue based locks, thus enabling greater parallelism in the local queue building process.

- In HCLH, after a thread enqueues its request in the local CLH queue, it spin waits on its node's predecessor to determine if it has become the lock owner. Because each thread needs to know if it is the local queue master the spin waiting logic is quite complicated, leading to several checks that appear in the critical path of the algorithm. This clearly adds to the latency of the lock handoff operation. On the other hand, the spin waiting of threads in FC-MCS is just like that of the MCS lock. The thread spins on its node's isOwner flag. This enables a much more streamlined and efficient lock handoff for FC-MCS.

Both the key differences, especially the first one, contribute to the drastically better performance (as we shall see in Section 3) of FC-MCS over HCLH.

3. EMPIRICAL EVALUATION

This section presents an empirical comparison of our new FC-MCS algorithm with the most efficient known locking algorithms, the CLH [2, 4] and MCS [3] queue locks, the HBO hierarchical backoff lock [5] and the HCLH hierarchical queue-lock [6]. Our collected data clearly shows that our algorithm outperforms all prior algorithms.

3.1 Methodology

We implemented all algorithms in C++, and compiled them with the Oracle Sun Studio 12.1 C++ compiler at optimization level xO5. The experiments were conducted on an Oracle T5440 series machine, which consists of 4 Niagara II chips, each chip containing 8 cores, and each core containing 8 hardware thread contexts, for a total of 256 hardware thread contexts running at a 1.4 GHz clock frequency. Each chip has an on board 4MB L2 cache, and each core has a shared 8KB L1 data cache.

To emphasize the impact of locality on these algorithms, we ensured that the threads in all workloads were evenly distributed among the 4 chips. For instance, for a test involving 32 threads, 8 threads were bound to each chip. This thread binding also helps ensure that the thread distribution is maintained throughout the runs. In all our tests, the main thread fires off a group of threads, each of which executes a critical region 100,000 times. Each iteration consists of a critical and a non-critical region, both of which can be configured differently to capture the behaviors of a diverse set of workloads.

We implemented our own versions of the mcs lock, clh lock, hbo lock, and naturally new-fc-mcs lock, and borrowed the highly tuned original hclh lock code developed by its authors [6].

3.2 Common Case: Scalability Under Standard Conditions

We first report the scalability results of a workload where each thread executes a counter increment (essentially a single read and a single write) in its critical region. Such short critical regions routinely occur in realistic workloads. The non-critical region consists of an idle spin loop for a randomly selected interval between 0 and 4 microseconds. This reflects a fair amount of work done between consecutive critical region executions, as is the case with most realistic applications.

Figure 4 depicts the performance of all the locking algorithms. Clearly, beyond 32 threads, the new-fc-mcs lock scales significantly better than all the others, improving on the mcs and clh locks by a factor of 4, and on hclh by a factor of 2 at high thread counts. Note that contrary to earlier results for the hclh lock [6], reported on a Sun E25K distributed memory machine, on our 256 way multicore architecture hclh significantly outperforms clh. We conjecture that this is due to threads sharing an L2 cache in our architecture, and therefore having a big win from locality of reference. As expected, at low thread counts, all the algorithms perform poorly because of the lack of pipelining in performing various synchronization operations. The non-critical region is large, and so throughput can only improve as concurrency increases since threads overlap executing the critical region with the lock acquisition operations and with execution of the non-critical region. As can be seen, below 32 threads, the new-fc-mcs lock performs similarly to the

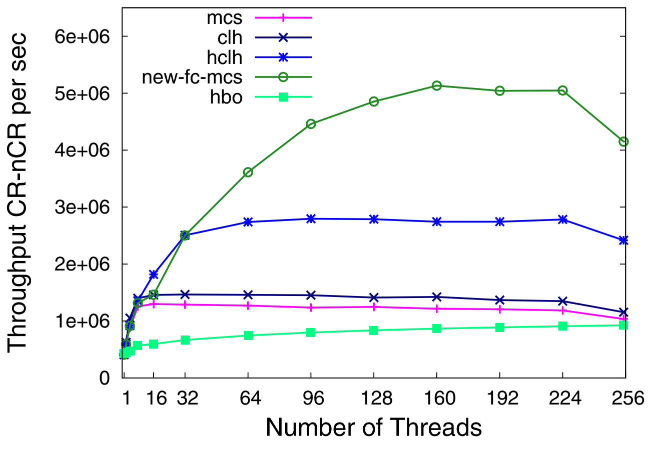


Figure 4: Average throughput in terms of number of critical and non-critical regions executed per second. In each critical region a thread increments a globally shared counter, while in the non-critical region it spins for a randomly selected interval of 0 to 4 microseconds.

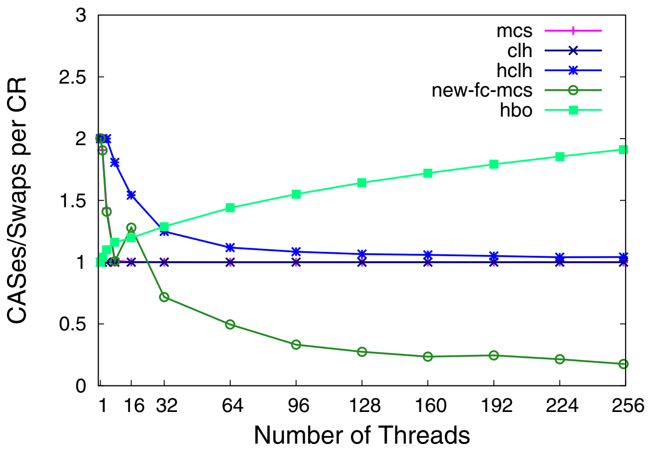


Figure 5: Average number of atomic CAS/SWAP instructions per critical region.

mcs lock: the low level of combining brings the adaptation mechanism into play, with threads skipping the combining attempts and directly accessing the global MCS lock.

Figures 5 and 6 help explain the superior performance of new-fc-mcs at high thread counts. The atomic read-modify-write operations (CAS and SWAP instructions) in all the algorithms are executed on shared locations and are thus indicative of bottlenecks. The average number of atomic read-modify-write instructions per critical region is drastically lower in new-fc-mcs compared to all other algorithms (Figure 5). In the mcs, clh, and hbo locks, all operations are applied sequentially on the same global location. The hbo algorithm suffers from increased CAS failures to an extent that overshadows the wins from improved cache locality. In the hclh lock, there is parallelism among local queues, but the building of the local CLH queues requires atomic oper-

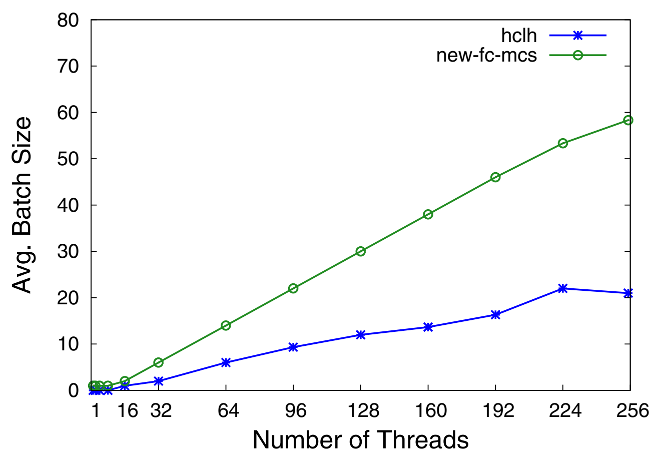


Figure 6: The average number of lock acquisition requests batched together by the combiner/master of each cluster. At low levels, we turned off the option to skip batching to expose the "natural" batching level. Since there is no batching of requests in mcs, clh, and hbo, we do not represent them in this graph.

ations on the local queue's Tail pointer, which introduces a sequential bottleneck in the buildup of the local CLH queue. This bottleneck, as witnessed in Figure 6, results in smaller "batches" of local CLH queue nodes in the global CLH queue (the alternative of improving the batch size by having the local CLH queue master wait for longer intervals to get a decent sized batch, does not help noticeably improve latency either).

In new-fc-mcs, there is no bottleneck in posting requests, as threads apply simple write operations in parallel, which the combiner then picks up. The result, as witnessed in Figure 6, is a high level of batching where close to 90% of requests are batched, which explains the factor-of-two better performance of new-fc-mcs over hclh. At lower concurrency levels (see Figure 6), obtaining large batches is impossible. However, our low contention adaptation mechanism helps the threads "bypass" the flat combining phase and directly enqueue their requests into the global queue. As a result, we observe that new-fc-mcs tracks the performance of mcs at low threading levels, and thus continues to be competitive.

3.3 Stress Test: Scalability with Empty Critical and Non-Critical Regions

In order to stress-test all the locking algorithms, we ran an experiment with empty critical and non-critical regions. All threads essentially acquire and release the lock for a 100,000 times. Although the experiment is not a realistic workload, it helps us avoid the issue of non-critical work masking any lock operation latencies, thus enabling us to do a "bare bones" comparison between the different algorithms.

Figures 7, 8, and 9 depict the performance of all the locks in this workload. The performance trend is somewhat different than before: hbo outperforms all the other locks. We were initially surprised by this finding. However, on second thought this behavior makes sense: because remote threads have a bigger backoff range, the lock, upon being taken by a thread from a cluster, tends to remain in the cluster for

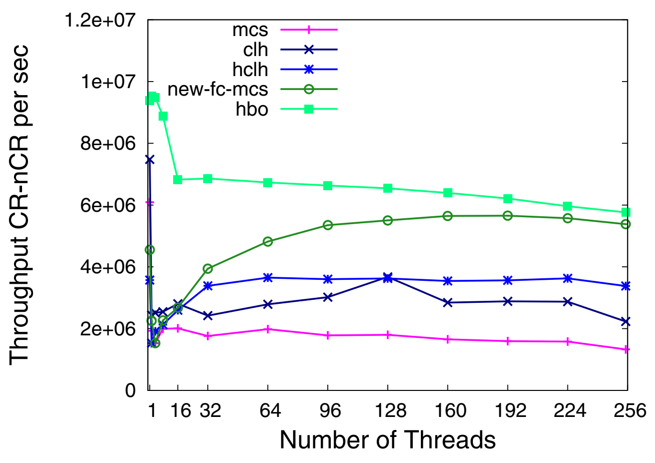


Figure 7: Average throughput of critical and non-critical region executions. Each thread's critical region and non-critical region are empty.

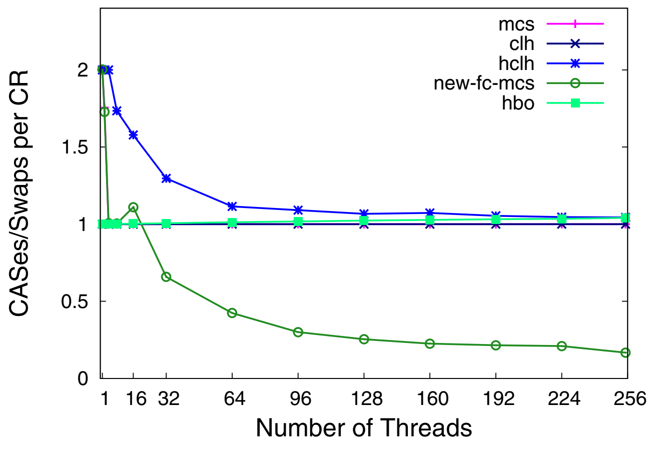


Figure 8: Average number of atomic CAS/SWAP instructions. Each thread's critical region and non-critical region are empty.

long intervals. In our experiments, we observed that the threads in a cluster are typically successful in locally acquiring and releasing the lock for about 10,000 times before a remote thread is able to acquire the lock. More importantly, because the non-critical region of this benchmark is empty, a lock releaser tends to quickly return to re-acquire the lock. As a result, the number of local lock acquisition requests, and their corresponding success rates (because of the backoff bias towards local threads), remain high at all times. This same behavior of hbo did not manifest itself in our earlier benchmark because of the non-zero duration of the non-critical region. This non-zero duration spaces apart the re-acquisition attempts of threads, and as a result, the number of local acquisition requests remains relatively low, and hence the chance of remote threads acquiring the lock increases.

For the remaining algorithms, as concurrency increases, mcs scales worst, followed by clh, followed by hclh, and fi-

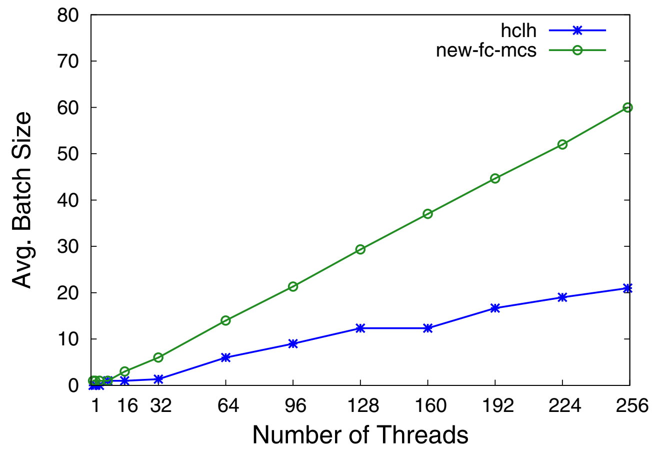


Figure 9: Average number of lock acquisition requests batched together by the combiner/master of each cluster. Since there is no batching of requests in mcs, clh, and hbo, we do not represent them in this graph.

nally new-fc-mcs, which performs best. We attribute this to the same reasons as in the prior sub-section – lower numbers of atomic operations, and quicker batching of lock acquisition requests. It is interesting that even at this very high load, new-fc-mcs continues to deliver very large batch sizes (Figure 9) because of lower number of atomics (Figure 8), while the batch size of hclh drops because of relatively higher number of atomic operations.

We would like to mention an implementation detail of our algorithm that appears to be relevant here. During our experiments, we noticed that the effectiveness of the flat combining operation is sensitive to the arrival rate of the threads. If the thread arrival rate is low, the flat combiner must iterate more times through the flat combining queue in order to batch together sufficiently many requests. However, we must be careful to prevent the combiner from iterating through the flat combining queue too many times in order to avoid an increase in the latency of the locking operation. This involves a delicate dance to dynamically adapt the combiner's iteration limit (if the limit is N, the combiner iterates through the flat combining queue N times) to the underlying workload. To that end, we have implemented a simple heuristic that appears to do very well for the workloads we tested: after a flat combining operation, if the combiner determines that the ratio of the number of requests (batched together by the combiner) to the size of the flat combining queue is below a particular threshold (50% in our tests), the combiner increments the iteration limit. This increment is subject to a maximum ceiling (128 in our tests). Similarly, if the ratio goes beyond a particular threshold (90% in our tests), the iteration limit is decremented by the combiner if it is greater than 1.

3.4 Impact of Our Low Concurrency Optimization

As discussed at the end of Section 2, the basic FC-MCS lock suffers from significant overheads at low thread counts, which we avoid by switching to applying operations directly to the global queue at low thread counts. Figure 10 shows the impact of not using our optimization, that is, always executing the combining operation, even in the absence of contention, on the same workload discussed in Section 3.2. Without the optimization (unopt-fc-mcs), the overhead of trying to combine dominates performance because there are not enough threads to combine. With the optimization, at low thread counts new-fc-mcs mimics the mcs lock behavior and continues tracking the performance of an unopt-fc-mcs at higher thread counts. Our optimization appears to be surprisingly effective for a wide range of workloads.

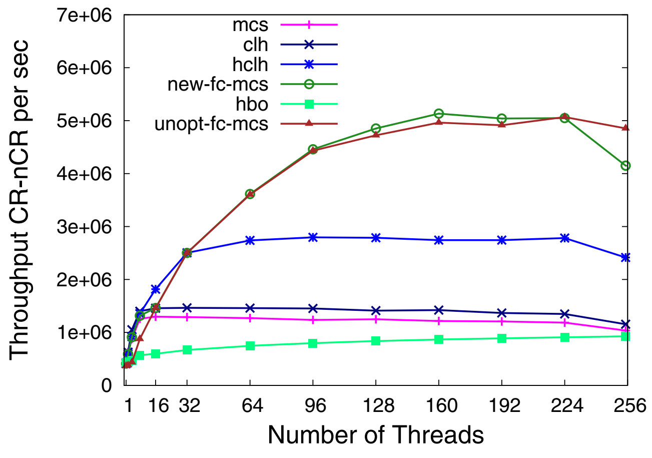


Figure 10: The throughput graph from Figure 4 with the curve for the unopt-fc-mcs variant of our algorithm.

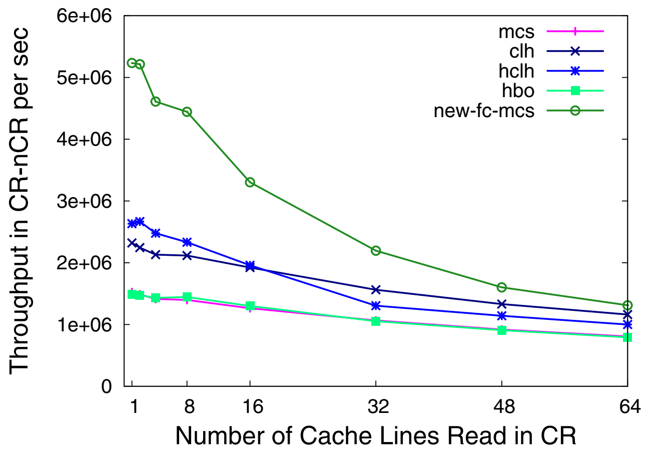


Figure 11: The effect of growing the footprint of the read-only critical region. The number of threads per test in this experiment was fixed at 248.

3.5 The Critical Region Footprint Effect

Our last set of experiments tests the effect of increasing the footprint of the critical region on the performance of the locks. We present results for two slightly different experiments. In both experiments, the footprint of the critical region increases from 1 cache-line to 64 cache-lines. In the

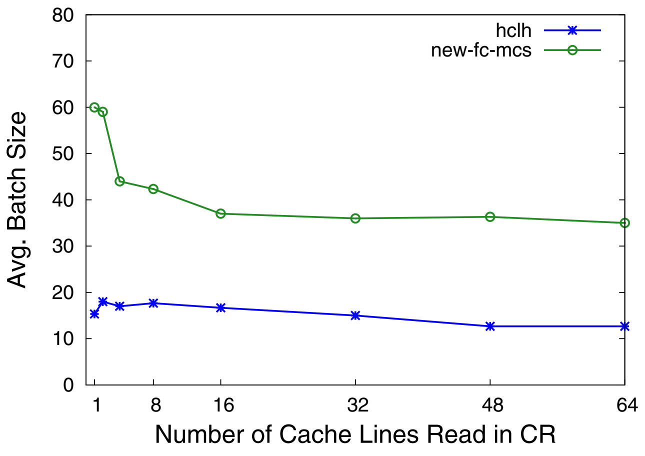


Figure 12: Average number of lock acquisition requests batched together by the combiner/master of each cluster for the growing read-only critical region. We represent data for only new-fc-mcs and hclh.

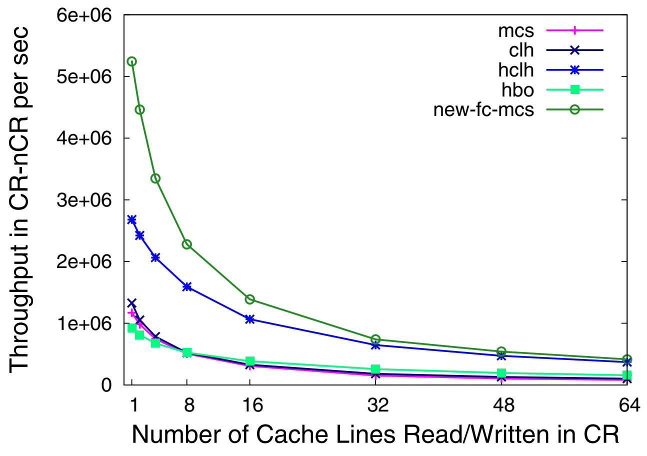


Figure 13: Effect of growing the footprint of the read-write critical region. The number of threads per test in this experiment was fixed at 248.

first experiment, the critical region only reads the shared variables (an integer counter on each cache line), whereas in the second experiment, the critical region both reads and writes the shared variables (incrementing an integer counter on each cache line). Each data point in the graphs represents a test run consisting of 248 threads, each repeatedly executing the critical region for 100,000 times. The non-critical region consists of an idle loop in which the thread spins for a randomly chosen interval between 0 and 4 microseconds.

Read-only Critical Regions.

Figure 11 depicts the effect of the growing footprint of the read-only critical region on the performance of all the locks. The new-fc-mcs lock continues to significantly outperform all other locks, starting with twice the performance of the other locks for small critical regions (1-4 cache lines). Notice however, that this performance gap dwindles as the critical region footprint (and consequently length) increases. We believe this is an expected result: Because the critical region is read-only, we expect all the cache-lines touched by it to be resident in host thread's L1 cache. Additionally, the locality benefits of new-fc-mcs and hclh, which remain more or less constant because the critical region is read-only, decline in a relative sense because the critical region length (and hence execution time) increases significantly (eventually its size becomes the dominating performance factor). Nevertheless, we observe that critical region lengths are typically small in real world applications, and new-fc-mcs is very effective in delivering much better throughput in that range – it outperforms all alternatives by a factor of 1.5 to 2 in the 1 to 16 read-only cache line range.

Figure 12 shows the average batching sizes of the new-fc-mcs and hclh locks for the tests reported in Figure 11. As in our previous experiments, new-fc-mcs is much more effective than hclh in batching lock acquisition requests. Notice the drop in batching size for new-fc-mcs. This is expected because as the length of the critical region increases, the interval between two lock acquisition attempts for each thread increases, and hence fewer threads are batched together by the flat combiner. The batching size remains more or less constant for larger critical regions. We believe the heuristic we applied – to dynamically re-adjust the combiner's combining iterations based on the batching size – helps new-fc-mcs maintain the constant batching size. The batching size of hclh remains more or less constant.

Also notice that clh begins to outperform hclh at around 32 cache line sized critical regions. We believe that since the locality benefits of hclh become increasingly insignificant (given the increasing size of the read-only critical region), the expensive lock handoff operation in hclh (consisting of multiple conditional checks) starts to become an important performance concern. The lock handoff operation of clh is much more streamlined and efficient, which helps it outperform hclh for larger critical regions.

Because of the contention on the central test-and-test-and-set (TATAS) lock, hbo is unable to leverage any significant locality benefits, and ends up performing similar to mcs. To its credit, hbo significantly improves scalability over a simple TATAS lock (not shown here), enabling performance comparable to that of mcs.

Read-Write Critical Regions.

Figure 13 depicts the effect on throughput of the growing footprints of critical regions that modify the cache lines they access. As in the read-only experiment, new-fc-mcs outperforms all other locks for all critical region sizes. However, the pattern of the graphs is significantly different than that of the read-only experiment (Figure 11).

First, all the non-hierarchical locks incur huge performance costs compared to new-fc-mcs and hclh because of the coherence traffic generated by the continuously modified cache lines in the critical region. The new-fc-mcs and hclh locks significantly reduce the coherence traffic by effectively batching together lock acquisition requests coming from the same cluster. The hbo lock continues to suffer from contention on its central TATAS lock, and hence performs comparably to the non-hierarchical locks.

Second, although new-fc-mcs outperforms hclh by a factor of 2 with 1 cache line, this performance gap narrows rather

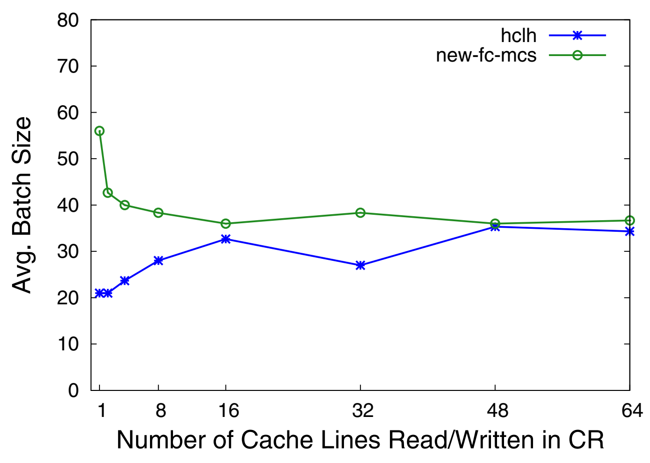


Figure 14: Average number of lock acquisition requests batched together by the combiner/master of each cluster for the growing read-write critical region. We represent data for only new-fc-mcs and hclh.

quickly as the number of lines is increased (with a difference of just about 25% at 16 cache lines, which dwindles down to about 10% at 64 cache lines). The explanation for this behavior appears in Figure 14 – the average batching size of new-fc-mcs decreases (much like in the read-only experiment), but more interestingly, the average batching size of hclh increases.

Reduction in the batching size of new-fc-mcs can be attributed to the increase in critical region length. However, the batching size increase for hclh appears to be counterintuitive. The explanation of this rather contradictory behavior of hclh lies in its implementation details: The hclh code is the original highly tuned implementation of hclh [6], which contains an interesting heuristic to help the master thread determine the amount of time it has to wait for more threads to join the local CLH queue. If the master determines that there was no other thread in its cluster that acquired the lock between its current and previous lock acquisition attempts, it halves the waiting time, assuming that the lock is lightly contended. Otherwise, if the master determines that no batching happened, it doubles the waiting time limit, assuming that it can do better batching the next time around. There are of course lower and upper bounds on the waiting time limits. What happens in the tests depicted in Figure 13 is that, since threads arrive at the lock (to acquire it) more slowly, the waiting time for the master effectively increases to a point where batching becomes more effective. This increase in batching size leads to better data locality, which helps "improve" the performance of hclh compared to that of new-fc-mcs. Note that since this is a read-write intensive critical region, data locality plays a critical role in a lock's performance. Thus, the benefits of better locality (achieved by better batching) outweigh the cost of waiting for longer durations in hclh.

Based on our finding, a natural question to ask is: Can we increase the waiting time for smaller critical regions to get better batching for hclh so as to get better locality and performance? Unfortunately that seems not be the case. Our testing indicates that hclh has to maintain a delicate balance between the amount of waiting and the resulting batching sizes to guarantee the best possible performance. If the waiting time is too long, the master ends up worthlessly waiting for new threads that never show up in the duration, thus compromising some of the benefits of better locality. The hclh authors have developed a highly tuned implementation that takes all these factors into account.

In any case, we observe that new-fc-mcs performs substantially (a factor of 1.5 to 2 times) better than hclh for what we consider to be critical regions with reasonably large read-write memory footprints (1 to 4 cache lines).

4. CONCLUSION

The growing size of multicore machines is likely to shift the design space in the NUMA and CC-NUMA direction, requiring the development of appropriate concurrent algorithms and synchronization mechanisms. This paper tackles the most basic of the multicore synchronization algorithms, the lock, presenting a new NUMA directed algorithm. The new lock is based on a combination of the flat combining synchronization paradigm and the MCS lock that adapts and scales significantly better than any of the prior locks on a wide range of workloads. We hope this paper will help to revive the line of research on locks and synchronization structures directed at NUMA architectures.

5. REFERENCES

[1] Anderson, T.: The performance implications of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel and Distributed Systems **1**(1) (1990) 6–16

[2] Craig, T.: Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Dept of Computer Science (1993)

[3] Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Computer Systems **9**(1) (1991) 21–65

[4] Magnussen, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: Proc. 8th International Symposium on Parallel Processing (IPPS). (1994) 165–171

[5] Radović, Z., Hagersten, E.: Hierarchical Backoff Locks for Nonuniform Communication Architectures. In: HPCA-9, Anaheim, California, USA (2003) 241–252

[6] Victor Luchangco and Dan Nussbaum and Nir Shavit: A Hierarchical CLH Queue Lock. In: Proceedings of the 12th International Euro-Par Conference. (2006) 801–810

[7] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat Combining and the Synchronization-Parallelism Tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures. (2010) 355–364

[8] Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. J. ACM **37**(3) (1990) 524–548

[9] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2007)