Blocking Universal Constructions

Panagiota Fatourou
Department of Computer Science
University of Crete
& FORTH ICS
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Department of Computer Science
University of Ioannina
nkallima@cs.uoi.gr

May 15, 2011

Abstract

In this paper, we present two new blocking universal constructions. The first one, called CC-BSim, is very efficient in systems that support coherent caches (CC), while the second one, called DSM-BSim, is more efficient in cache-less NUMA machines (where processors do not have caches). CC-BSim performs at most $O(h + t)$ remote memory references (RMR), where h is an upper bound on the number of operations that a process may help, and t is the size of the part of the object's state that it should be accessed in order to execute these h operations. DSM-BSim performs at most $O(hw)$ RMR, where w is the number of memory words accessed by the sequential version of an operation applied on a sequential data structure. DSM-BSim is better suited in a distributed shared memory (DSM) model. Algorithm CC-BSim uses just one `Swap` object in addition to read-write registers, thus exhibiting a performance advantage in machines that support `Swap` objects. On the other hand, DSM-BSim uses `CAS`, `Swap`, and read-write registers.

Our experiments show that CC-BSim and DSM-BSim outperform state-of-the-art synchronization techniques, like Sim (SPAA 2001), flat-combining (SPAA 2010), and others. We also present and experimentally analyze common shared data structures (like shared stacks and queues) based on CC-BSim and DSM-BSim. Our experiments show that these implementations have very good performance in practice.

1 Introduction

Nowadays multicore processors are used in any computing device. In such systems, processes communicate by (concurrently) accessing shared objects. Designing efficient shared objects from simpler that are provided by the hardware is therefore of crucial importance. However, this is not always an easy task. Using universal constructions, the design and implementation of shared objects is significantly simplified. A *universal construction* is a shared object that is able to simulate any shared object given the sequential implementation of the object. A universal construction supports just one operation called ApplyOp which takes as a parameter the sequential implementation of the operation that a process wants to apply to the simulated object (and its arguments). Universal constructions have been studied extensively in [1, 2, 6, 10, 11, 13, 14, 15]; however, most of them are mainly of theoretical interest. Recently, Hendler *et. al* presented in [13] a synchronization technique, called *flat combining*. This technique was used in [13] to get a blocking universal construction that achieves good performance in practice. Fatourou and Kallimanis [11] presented a wait-free universal construction that exhibits the nice performance properties of flat combining [13] although it ensures the strongest progress property of wait-freedom.

In this paper, we present two blocking universal constructions. The first, called CC-BSim, is suitable for cache coherent (CC) shared memory systems where accesses to shared objects are performed via cached copies of them. An access to a shared object is a remote memory reference (RMR) if the cached copy of this object is invalid. The great majority of modern parallel architectures follow the CC shared memory model. The second universal construction called DSM-BSim is better suited for the distributed shared memory (DSM) model. In systems that support this model, processors do not have access to local caches, so any access to a shared object that is allocated in a part of the shared memory residing on another processor is a remote memory reference (thus, it is expensive to serve it).

CC-BSim and DSM-BSim exploit the combining technique where one of the threads undertakes the task of applying the operations of many active processes (in addition to its own). This technique has been successfully applied in past in several cases [11, 13, 20, 21, 23]. The high level idea of CC-BSim and DSM-BSim is that a linked list is maintained containing information about the currently active operations. Each active process is assigned one of the nodes of the list (where the operation of the process has been announced). The active process (let it be q) that owns the first node of the list is the process that undertakes the responsibility of modifying the state of the object by applying on it its operation as well as the operations of processes that it helps; more specifically, q traverses the list and applies the operations announced in its nodes to the shared object. Newly activated processes add nodes to the tail of the list. An application of ApplyOp at CC-BSim performs $O(h + t)$ RMR, where h is an upper bound on the number of operations that a process may help, and t is the size of the part of the object's state that it should be accessed in order to execute these h operations. CC-BSim uses just one Swap object in addition to read-write registers and its space complexity is $O(n + s)$, where s is the size of the simulated object's state. CC-BSim combines and extends ideas from the CLH spin lock [8, 17] and flat combining [13]. Similarly, DSM-BSim combines and extends ideas from the MCS spin lock [18] and flat combining; it also employs a similar helping technique to that of CC-BSim. A process executing an ApplyOp on DSM-BSim performs $O(hw)$ RMR. In contrast to CC-BSim, DSM-BSim uses CAS registers in addition to one Swap object and its space complexity is $O(n + s)$.

Fatourou and Kallimanis have presented in [11] some efficient wait-free universal constructions. The first one, called Sim, has constant time complexity, but it is only of theoretical interest since it uses unrealistically large registers; a practical version of Sim, called P-Sim, is also presented in [11]. P-Sim performs $O(n+s)$ shared memory accesses, while its space complexity is $O(n^2+ns)$. P-Sim exhibits good performance in practice; it can been used to implement very efficiently shared objects of small size. In [11], efficient concurrent stack and queue implementations, based on P-Sim, are also presented and experimentally analyzed. However, P-Sim does not efficiently cope with shared objects whose state size is large, since it copies the entire object's state locally. In contrast, the blocking universal constructions presented in this paper efficiently handle even shared objects whose state's size is large or even unbounded since the unique process that applies the updates can do so directly on the shared data structure. Non-blocking universal constructions that cope with large objects are presented in [1, 3, 4, 6]. However, their performance have not been evaluated in practice.

Hendler *et. al* have presented in [13] a blocking universal construction, called flat-combining. In flat-combining there is a global lock that protects the shared data structure. Also, a list of the processes that wish to execute operations is maintained. The process that acquires the global lock traverses the list and applies the operations listed there to the shared data structure. A process that initiates a new operation is added at the head of the list using CAS. The lock owner cleanups the list periodically keeping in it only the processes that have recently initiated an operation. The number of RMR that an operation may perform in flat combining is unbounded (a really slow process that tries to insert itself to the list may be blocked by faster processes that perform continuous cleanups and additions in the list).

We experimentally compare CC-BSim and DSM-BSim with some of the state-of-the-art synchronization techniques, like P-Sim [11], flat-combining [13], CLH spin locks [8, 17], and a simple lock free technique. Our benchmarks (Figures $1-8$) show that CC-BSim significantly outperforms all other algorithms in many cases. More specifically, there are cases that CC-BSim is up to 2.5 times faster than the fastest of the other algorithms (Figure 7). DSM-BSim also outperforms all algorithms other than CC-BSim. DSM-BSim has the advantage over CC-BSim that it is efficient even in machines that support the DSM model and it can be efficiently employed in applications that run on machine architectures that are not known to them (i.e., by general scope applications that should run efficiently on a lot of different architectures). It is worth pointing out that P-Sim is still valuable in cases that stronger progress guarantees are needed, while flat-combining is useful in systems that Swap registers are not provided.

We used CC-BSim and DSM-BSim to implement efficient shared stack and queue implementations. Our stack implementation based on CC-BSim, called CC-BStack, outperforms state-of-the-art shared stack implementations (like the wait-free stack implementation, called SimStack, recently presented in [11], the linked stack implementation based on flat-combining [13], and the stack implementation based on CLH spin locks [8, 17]). DSM-BSim also outperforms all implementations other than CC-BSim. We also use CC-BSim and DSM-BSim to get two highly efficient shared queue implementations, called CC-BQUEUE and DSM-BQUEUE. More specifically, the two-locks queue implementation presented in [19] is enhanced by simply replacing the ordinary locks either with CC-BSim or with DSM-BSim. These implementations were experimentally compared to the wait-free queue implementation presented in [11] (called SimQueue), the two-locks implementation [19], and the queue implementation based on flat combining presented in [13]. The queue implementation based on CC-BSim performs up to 2.5 times faster than the queue implementation of [13] and outperforms SimQueue by a factor of up to 2.2. The good performance levels achieved by our queue implementation makes it suitable for applications that are heavily based on shared queues, like the hierarchical lock implementation that has been recently presented in [9]. In such applications, we expect that, by employing our queue implementation, we would observe a significant performance speed-up (however, implementing and experimentally analyzing this has been left for future work).

Many hardware manufactures have been influenced by the universality result [14], and they have equipped their machines with the strongest atomic primitives (like `CAS` and `LL/SC`). As shown in [11], machines that additionally support `Fetch&Add` instructions, can have important performance advantages, while it is additionally possible to ensure wait-freedom. Our experiments show that machines that support `Swap` objects have significant performance benefits as well. Fortunately, `Swap` instructions are already supported by a large variety of machine architectures (x86, sparc, etc.). Weak atomic primitives, like `Swap`, can be implemented much easier and more efficiently. More specifically, a `CAS` may fail to modify the register, thus a loop that repeatedly executes `CAS` operations is usually employed; in contrast, `Swap` always succeeds, thus causing less contention. Furthermore, the execution of a `CAS` includes the evaluation of a condition in the cpu microcode, so it is expected to be slower than executing a simple `Swap` instruction.

We note that CC-BSim, similarly to Sim [11] and flat-combining [13], cannot be efficiently applied in data structures such as search trees, where m lookups can be executed in parallel performing just a logarithmic number of shared memory accesses each. In such cases, it is expected that CC-BSim will not perform the same well, since like most previous universal constructions [6, 10, 11, 13, 14, 15], it performs each operation sequentially. More efficient implementations of such data structures can possibly be obtained by using many instances of CC-BSim in a way similar to the enhanced two-lock queue implementation (Section 5).

This paper is organized as follows. Section 2 presents the model. CC-BSim is presented in Section 3 and DSM-BSim in Section 4. Section 5 presents the experimental evaluation of CC-BSim and DSM-BSim.

2 Model

There are n *asynchronous processes* $p_1, \ldots, p_n$ running in the system. The processes communicate by accessing shared objects. Each *shared object* stores some information and offers to the processes the capability to access and modify the stored information using atomic *operations* that may be executed by the processes concurrently.

The most basic shared object is a read-write register which is provided by the hardware in all systems. A *read-write* register R, stores a value v from a set V and supports two atomic operations, `read`(R) and `write`(R, v). Operation `read`(R) returns the current value of register R, while `write`(R, v) stores value v to R and returns *ack*. A `CAS` register stores a value from a set V and supports two atomic operations, `read`(R) and `CAS`$(R, vold, vnew)$. Operation `read`(R) returns the value stored in register R. Operation `CAS`$(R, vold, vnew)$ stores v to R if the current value of R is equal to *vold* and returns `TRUE`, otherwise the contents of R remain unchanged and `CAS`$(R, vold, vnew)$ returns `FALSE`. A `Swap` register stores a value from a set V and supports two atomic operations, `read`(R) and `Swap`(R, v). Operation `read`(R) returns the value stored in register R. Operation `Swap`(R, v) returns the current value of R and stores v to R.

We consider shared memory systems where parts of the shared memory are associated to each processor, so every shared object is allocated (and resides) in the shared memory part associated with some processor. We consider two memory models. In cache-less NUMA machines (this model is also known as DSM), a process p performs a *remote memory reference* (RMR) if it accesses a shared object residing in the shared memory part of some processor other than that where p resides; all other memory accesses by p are called *local*. In *cache-coherent* (CC) machines, accesses in shared memory are performed on cached copies of the data items. In this case, a memory reference is called *remote* if the cached copy of the accessed data item is invalid. Then, a cache miss occurs and a valid copy of the data item should be locally fetched first before it can be accessed. It is worth pointing out however that once this occurs and as long as the data item is not updated by other processors, all future accesses to the data item by the processor are local. This is not the case in the DSM model, where every access to a data item that resides in a remote memory is *remote*. We remark that since an RMR is significantly more costly than a local memory reference, our goal is to design algorithms that perform as few RMR as possible.

A *universal* object simulates any other concurrent object. It supports an operation, called ApplyOp(*op-eration op*), which simulates the execution of operation *op* on the simulated object; ApplyOp returns the return value of operation *op*. An implementation of an object from basic shared objects (like read-write registers) provides, for each process, an implementation of ApplyOp using the basic shared objects.

A *configuration* C is a vector containing the states of the processes and the values of the shared objects at any point in time. At the *initial* configuration, each shared object has an initial value and each process is at an initial state. A process completes the execution of a computation step, each time it accesses a primitive shared object, i.e., a step may include the execution of code on local variables. An *execution* is a sequence of execution steps by processes.

An implementation is *blocking* if a process may have to wait for some event caused by other processes in order to complete the execution of its operation. In blocking implementations, the *time complexity* of an operation is the maximum number of remote memory references that a process performs to complete any instance of the operation in any execution. The *time complexity* of an implementation is the maximum between the time complexities of its operations. The *space complexity* of an implementation is determined by the number, the type, and the size of the primitive shared objects used by the implementation.

Let α be any execution of an implementation of a (high-level) object from base objects. *Linearizability* [16] ensures that for each operation *op* on the simulated object in α, there is some point within its execution interval, called *linearization point*, such that the response returned by *op* in α is the same as the response *op* would return if all operations in α were executing serially in the order determined by their linearization points; when this holds, we say that the response of *op* is *consistent*. An implementation is *linearizable* if all its executions are linearizable.

3 An efficient blocking universal construction for the CC model

In this section we present CC-BSim, a blocking universal construction for the CC shared memory model which has $O(h + t)$ time complexity.

CC-BSim combines some ideas from CLH locks [8, 17] and flat-combining [13] to get a highly-efficient algorithm for CC systems. More specifically, a process p that wants to execute an operation appends a node at the end of a linked list. A pointer, called $Tail$, points to the last element of the list. Each node in the list other than the last one, contains information about some operation; the last node in the list is empty and it will be used by a process (let it be q) in the future to announce its operation. The new node that q will append in the list will play the role of the new empty node of the list. Thus, the last node in the list is always empty. Initially, the list contains an empty node which will be used by the first process that wants to perform an operation for its announcement. We say that a node *is assigned* to the process that has initiated the operation recorded in it. Once q appends its node, it spins on the *locked* field of its assigned node.

In contrast to the CLH algorithm where the list is implicit (i.e., no next pointers are maintained in each node of the list), the list maintained by CC-BSim is explicit. The process at the head of the list, has the lock. However, in contrast to what CLH does, this process does not give up the lock when it applies its operation. It rather continues accessing other elements of the list (by traversing the next pointers of the list nodes) and helps the processes spinning on these nodes by executing their operations and then setting the *locked* fields of these nodes to false to stop them from spinning.

We now give a more detailed description of CC-BSim (Algorithm 1). Pointer $Tail$ is a `Swap` object which initially points to the empty node that is initially inserted in the list. Whenever process p_i wants to execute some operation op, it executes a `Swap` operation to $Tail$ (line 6) in order to read the pointer to the empty node pointed to by $Tail$ and update $Tail$ to point to a new empty node pointed by p_i's local variable $next_i$. Once this has been performed, p_i has been assigned the node that was previously pointed by $Tail$, so it first announces its operation by recording op in the op field of the node (line 7) and then it sets the *next* field of the node to point to the new empty node (line 8). This results in the expansion of the shared linked list by one node which is empty and is pointed to by $Tail$. After that, p_i starts spinning on field *locked* of its assigned node until this field becomes `FALSE`. When p_i reads `FALSE` in *locked*, either its operation has been executed by some other process q or p_i is the first process in the list and therefore it has the lock. So, it should modify the simulated object's state by applying the operations currently recorded in the list on it. Thus, p_i applies its operation to the shared object and starts traversing the list to help any other pending operation.

We remark that the list could grow forever while p_i traverses it (a process may add a node at the end of the list more than once after its operation has been applied by p_i). In order to prevent p_i from traversing a continuously growing list, an upper bound h (line 14) on the number of operations that p_i can help is employed; once p_i helps h other operations, it stops helping and returns. Our experiments show that the choice of this parameter does not significantly impact the performance of the algorithm in practice. More specifically, setting h to a value equal to cn, where $c > 0$ is a small constant, is a good choice in terms of performance.

The total number of nodes used by the processes to maintain the list are $2n + 1$, two nodes for each process plus the empty node that is initially in the list. These nodes are recycled applying a similar technique than that of CLH. More specifically, each process uses the two nodes it has allocated for its first two operations, and then it uses the nodes it was assigned in its last two operations for future operations. We explain below why it is not enough to have each process p_i use just one list node instead of two. Assume first that every time that p_i wants to allocate a new node, it simply reuses the node it has last been assigned in the list. Let q be the process that currently has the lock. Suppose that q helps p_i by applying its operation but delays its execution before executing line 19. If p_i wants to immediately apply another operation, it may initialize its node again and exchange it with the empty node pointed to by $Tail$. Then, q continues by executing line 19, it sees that it's at the end of the list and it finishes its operation. Thus, q never sees the pending operation of p_i and it never helps p_i to finish its operation, so p_i spins forever. To avoid this bad scenario, every process maintains two nodes and uses them in a round robin manner.

Time and Space complexity. By the pseudocode (Algorithm 1), it follows that each process

Algorithm 1 Pseudocode for CC-BSim algorithm.

```
struct LockNode {
    LockNode *$next$;
    Operation $op$;
    RetVal $ret$;
    boolean $locked$;
    boolean $completed$;
};
shared LockNode *$Tail$ = $\langle null, \bot, \bot,$FALSE, FALSE$\rangle$;           // Initially points to an empty node
// Process $p_i$ owns a private copy of this variable
private LockNode *$node_i[0..1]$ = {$\langle null, \bot, \bot,$FALSE, FALSE$\rangle$, $\langle null, \bot, \bot,$FALSE, FALSE$\rangle$};
private int $toggle_i = 0$;                   // Takes values form set {0, 1}

RetVal applyOp(Operation op) {       // Pseudocode for process $p_i$
    LockNode *p, *cur, *next_node;
    int counter = 0;
1.  next_node = $node_i[toggle_i]$;          // a new recycled empty node is pointed by next_node
2.  $toggle_i$ = 1 - $toggle_i$;                  // $p_i$ toggles its toggle variable
3.  next_node $\rightarrow$ next = $\bot$;             // the new empty node is initialized (lines $3-5$)
4.  next_node $\rightarrow$ locked = TRUE;
5.  next_node $\rightarrow$ completed = FALSE;
6.  cur = Swap(Tail, next_node);         // $Tail$ points to a new empty node, the returned node is now owned by $p_i$
7.  cur $\rightarrow$ op = op;                       // $p_i$ announces its operation
8.  cur $\rightarrow$ next = next_node;            // next field points to the new empty node, also pointed by $Tail$
9.  $node_i[toggle_i]$ = cur;                // node pointed by cur will be re-used in a next operation
10. while (cur $\rightarrow$ locked == TRUE)      // process spins until its operation is applied or until unlocked
        nop;
11. if (cur $\rightarrow$ completed == TRUE)    // $p_i$'s operation has been applied by some other process
12.     return cur $\rightarrow$ ret;                // $p_i$ returns its returned value
13. p = cur;
14. while (p $\rightarrow$ next $\neq$ $null$ AND counter < $h$) {
15.     counter = counter + 1;            // counter is increased by one, since another operation is applied
16.     apply p$\rightarrow$op to object's state and store the return value to p $\rightarrow$ ret;
17.     p $\rightarrow$ completed = TRUE;         // announce to the spinning process that its operation is applied
18.     p $\rightarrow$ locked = FALSE;            // unlocks the spinning process
19.     p = p $\rightarrow$ next;                   // proceed to the next node
    }
20. p $\rightarrow$ locked = FALSE;                 // unlocks the next the owner of the next node (if any)
21. return cur $\rightarrow$ ret;                    // process returns
}
```

p_i completes its operation either on line 12 or on line 21 of the pseudocode. In case that p_i completes *op* on line 12, it follows that p_i executes a constant number of RMR. Assume now that p_i completes *op* on line 21 of the pseudocode. By the pseudocode (line 14), p_i executes at most h iterations of the while loop (lines $15-19$). Lines $17-19$ contribute just a constant number of RMR, and line 15 is a local operation. Thus, p_i executes $O(w)$ RMR at each iteration of the loop (to apply each operation). However, different operations may access the same part of the state of the object, which once fetched in p_i's cache, accessing it does not cause any further RMR. So, the time complexity of CC-BSim is $O(h+t)$, t is the size of the part of the object's state that it should be accessed in order to execute these h operations. It is worth-pointing out that the amortized time complexity is $O(t)$; thus, the amortized time complexity of CC-BSim is as good as the time complexity of the sequential version. The space complexity of CC-BSim is $O(n+s)$, since each process has to allocate just two structs of type $LockNode$ at most.

4 An efficient blocking universal construction for the DSM model

CC-BSim performs an unbounded number of RMR in the DSM model. In this section, we present DSM-BSim, a blocking universal construction which performs $O(hw)$ RMR in the RMR model. DSM-BSim (Algorithm 2) is an extended version of the MCS spin locks [18] to support combining. Some of the techniques of CC-BSim are also employed. In contrast to CC-BSim, DSM-BSim uses `CAS` objects in addition to the `Swap` objects. Experiments show (Figures $1-8$) that CC-BSim achieves slightly better performance than DSM-BSim.

DSM-BSim maintains a list of operations to be executed in a manner similar to that of CC-BSim. In contrast to CC-BSim, the list is initially empty and its last node is a valid (not an empty) node. Each process q maintains two list nodes; q announces each operation it performs in one of these nodes, and then performs spinning on it. Notice that a process that appends its node to the list has to update the next field of the previous node to point to the new node. A process q helps all the operations recorded in list nodes up to the second last element in the list (see condition of the if statement of line 19). It does so to avoid arriving to a node where its next field has not yet been updated although this node is not the last node in the list any more. Thus, q will execute lines $22-25$ only if its node is the only node in the list. In this way, q performs a bounded number of remote memory accesses (whereas spinning on the last node of the list on line 25 would cause an unbounded number of memory accesses).

Time and Space complexity. DSM-BSim performs $O(hw)$ RMR, i.e. w RMR for each of the h operations that an operation helps. These RMR are performed during the execution of line 16. The only extra piece of code that may cause DSM-BSim to perform more RMR is that consisting of lines $22-25$. However, as explained above a process q executes these code lines only when the node it inserts is the single node of the list. So, if these lines are executed, q's local variable called p is equal to $mynode$, and therefore spinning on $p \rightarrow next$ on line 25 is local. The space complexity of DSM-BSim is the same as that of CC-BSim.

5 Performance evaluation

We evaluate CC-BSim and DSM-BSim in two different multiprocessor machine architectures. The first machine is a 32-core machine consisting of four AMD opteron 6134 processors (Magny Cours). Each processor of the Magny Cours machine consists of 2 dies and each die contains 4 processing cores. Communication among the cores of the same die is achieved with a fast L3 cache. Dies are communicating with Hyper-Transport links creating thus a complex topology that resembles a hypercube [7]. The second machine is an 128-way Sun machine consisting of 2 UltraSPARC-T2 processors (Niagara 2). Each processor in Niagara 2 machine consists of 8 processing cores, each of which is able to handle 8 threads. All experiments in the Magny Cours machine were performed using the gcc 4.3.4 compiler; all experiments in the Niagara 2 machine were performed using gcc 4.5.1. In order to avoid bottlenecks in memory allocation, the Hoard memory allocator [5] was used. The full code of CC-BSim and DSM-BSim is provided at http://www.cs.uoi.gr/~nkallima/bsim.

In order to evaluate CC-BSim's and DSM-BSim's performance, we compared them with state-of-the-art synchronization techniques simulating a simple `Fetch&Multiply` object and common

Algorithm 2 Pseudocode for DSM-BSim algorithm.

```
struct LockNode {
    LockNode *next;
    ArgVal arg;
    RetVal ret;
    Operation op;
    boolean locked;
    boolean completed;
};

shared LockNode *Tail = $null$;           // initially the list of active processes is empty

// process $p_i$ owns a private copy of these variables
private LockNode $MyNodes_i[0..1]$ = {⟨$null$, ⊥, ⊥, ⊥, FALSE, FALSE, ⟩, ⟨$null$, ⊥, ⊥, ⊥, FALSE, FALSE, ⟩};
private int $toggle_i$ = 0;                 // takes values form set {0, 1}

RetVal applyOp(Operation op) {             // pseudocode for process $p_i$
    LockNode *p, *mynode;
    int counter = 0;
1.  $toggle_i$ = 1 - $toggle_i$;             // $p_i$ toggles its toggle variable
2.  mynode = &$MyNodes_i$[toggle];         // a new empty node is pointed by mynode
3.  mynode→locked = TRUE;                  // $p_i$ initializes its node (lines $3-6$)
4.  mynode→completed = FALSE;
5.  mynode→next = $null$;
6.  mynode→op = op;                         // $p_i$ announces its operation $op$
7.  MyPred = swap(Tail, mynode);            // $Tail$ points to a new empty node, the returned node is now owned by $p_i$
8.  if (MyPred $\neq$ $null$) {              // in case that some node already exists in the list
9.      MyPred→next = mynode;               // set the previous node of the tail to point to my node
10.     while (mynode→locked == TRUE)       // $p_i$ spins until its operation is applied or until unlocked
            nop;
11.     if (mynode→completed == TRUE)       // $p_i$'s operation has been applied by some other process
12.         return mynode→ret;              // $p_i$ returns its returned value
    }
13. p = mynode;
14. while(TRUE) {                           // $p_i$ executes this loop at least once to apply its operation
15.     counter++;                          // counter is increased by one, since another operation is applied
16.     apply $op$ to object's state and store the return value to p→ret;
17.     p→completed = TRUE;                 // announce to the spinning process that its operation is applied
18.     p→locked = FALSE;                   // unlocks the spinning process
19.     if (p→next == $null$ or p→next→next == $null$ or counter $\geq h$)
20.         break;                          // $p_i$ has helped $h$ processes or less than two nodes are left in the list
21.     p = p→next;                         // proceed to ne next node
    }
22. if (p→next == $null$) {                 // $p_i$ is at the end of the list of active processes
23.     if (CAS(Tail, p, $null$) == TRUE)    // replace the tail of the node with $null$ (in case of success the list is empty)
24.         return mynode→ret;
25.     while (p→next == $null$)             // some process is adding a new node, it waits until it finishes its operation
            nop;                            // this loop is only executed on $p_i$'s node, it doesn't cost additional RMR
    }
26. p→next→locked = FALSE;                  // Unlocks the next the owner of the next node (if any)
27. p→next = $null$;
28. return mynode→ret;                      // Process returns
}
```

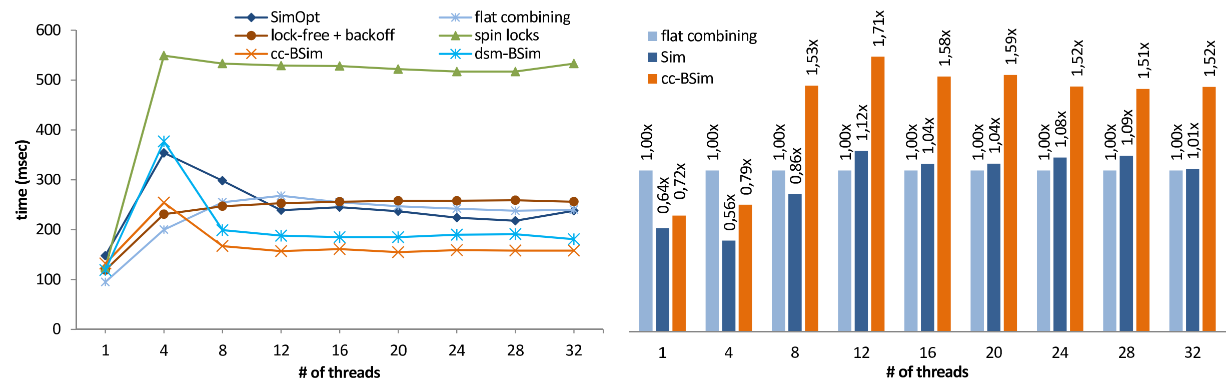

Figure 1: Absolute performance of the CC-BSim and DSM-BSim in Magny Cours machine (left) and relative performance of CC-BSim and Sim against flat-combining (right).

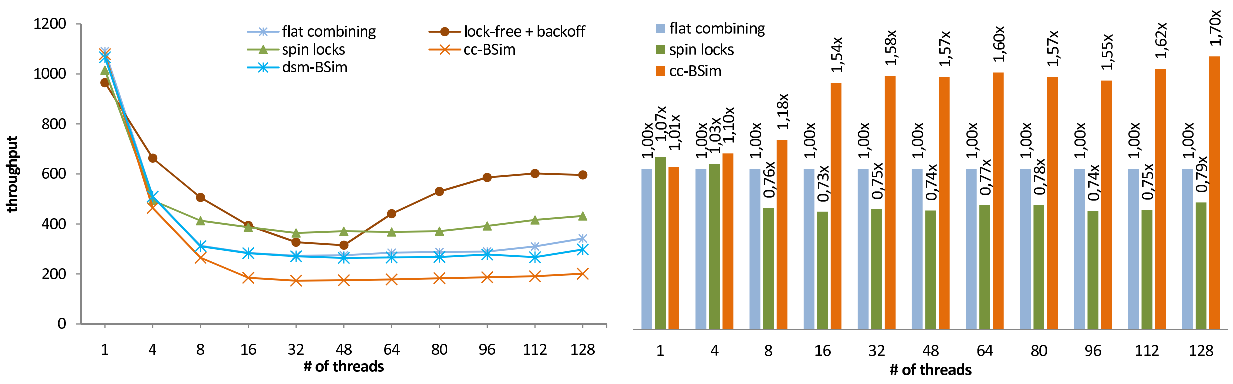

Figure 2: Absolute performance of CC-BSim and DSM-BSim in Niagara 2 machine (left) and relative performance of CC-BSim and CLH spin locks against flat-combining (right).

concurrent data structures like shared stacks and queues. Our first experiment is a synthetic benchmark (Figures 1, 2), where a simple `Fetch&Multiply` object is simulated using several state-of-the-art synchronization techniques. This object is simple enough to exhibit any overheads that a synchronization technique may induce while simulating a very small shared object. CC-BSim and DSM-BSim are compared with Sim a wait-free universal construction presented in [11], flat-combining [12, 13], the CLH spin-locks [8, 17], and a simple lock free implementation. Since the Niagara 2 machine does not support `Fetch&Add` instructions which are necessary for the good performance of Sim, no experiment was performed in the Niagara 2 machine for the Sim algorithm. The lock free simulation of the `Fetch&Multiply` object was implemented using a `CAS` object. Whenever a thread wants to apply a `Fetch&Multiply` instruction, it repeatedly executes `CAS` instructions until it succeeds. All algorithms were carefully optimized and for those that use backoff schemes, we performed a large number of experiments in order to choose the best backoff parameters. We used the flat-combining implementation that was provided by its inventors [12, 13] and we choose its parameters very carefully in order to achieve the best performance.

In the first experiment (Figures 1, 2), we measure the time that is needed for each of the synchronization techniques to perform 10^6 `Fetch&Multiply` instructions . The experiment was executed for many different values of n. Each thread executed $10^6/n$ `Fetch&Multiply` instructions. In order to make the experiment more realistic, some small random work was added between the

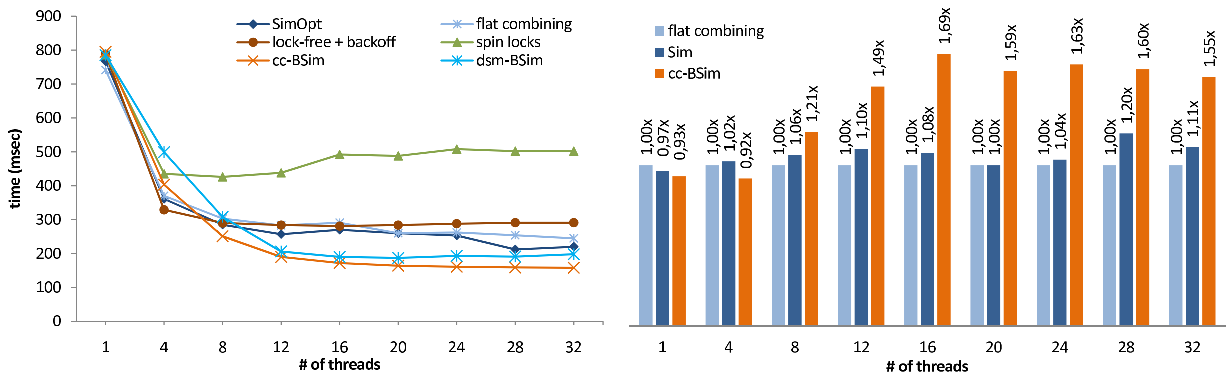

Figure 3: Absolute performance of the CC-BSim and DSM-BSim in Magny Cours machine (left) and relative performance of CC-BSim and Sim against flat-combining (right). Random work between consecutive **Fetch&Multiply** instructions is 512 at most.

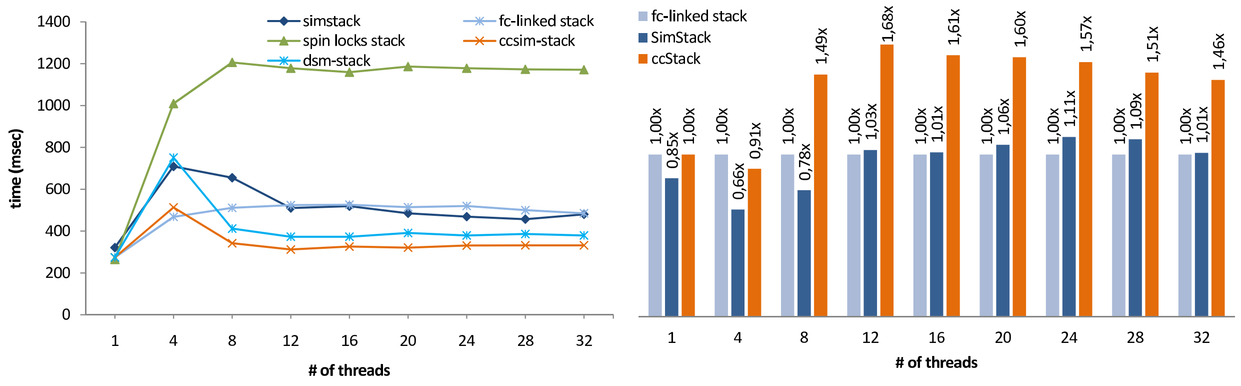

Figure 4: Absolute performance of the stack implementation based on CC-BSim in Magny Cours machine (left) and the relative performance of CC-BSim's stack implementation and SimStack against flat-combining (right).

execution of two consecutive **Fetch&Multiply** instructions to avoid unrealistically low numbers of cache misses and long runs. A similar strategy has been followed in [11, 19]. This random work consists of dummy loop iterations, which are at most 64. The horizontal axis of Figures $1-8$ represents the number of processes n. In Figure 1 (left) and Figure 2 (left), the vertical axis displays the time that each synchronization technique needs to complete the experiment. In Figure 1 (right) the vertical axis shows the relative performance of CC-BSim and DSM-BSim compared to that of flat-combining, i.e., it illustrates how much faster/slower CC-BSim and Sim are compared to flat-combining (where the performance of flat-combining is always considered equal to 1.0). Similarly, Figure 2 (right) shows the relative performance of CC-BSim and CLH spin locks compared to flat-combining in the Niagara 2 machine.

In the experiments performed in the Magny Cours machine (Figure 1), CC-BSim outperforms all other synchronization techniques. More specifically, CC-BSim is up to 1.71 times faster than flat-combining and up to 3.3 times faster than CLH locks. Furthermore, CC-BSim outperforms Sim by a factor up to 1.52. The lock free implementation of the **Fetch&Multiply** object is slightly slower than Sim and flat-combining. DSM-BSim performs also very well and its performance is close to that of CC-BSim, albeit it is better suited in machines following the DSM model.

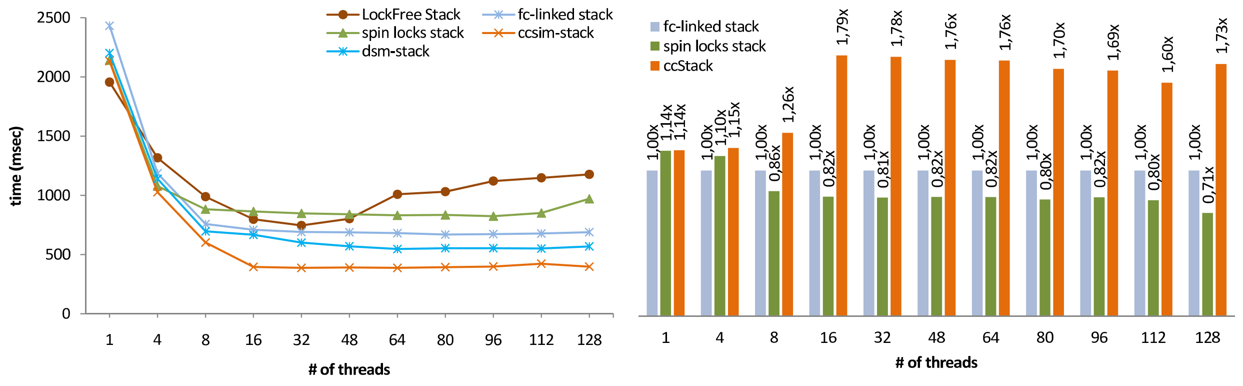

Figure 5: Absolute performance of the stack implementation based on CC-BSim in Niagara 2 machine (left) and the relative performance of CC-BSim's stack implementation and SimStack against flat-combining (right).

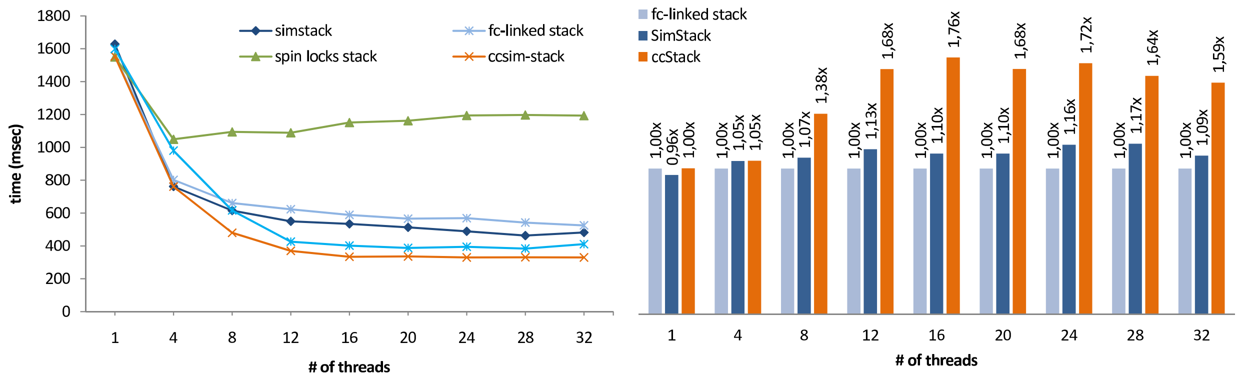

Figure 6: Absolute performance of the stack implementation based on CC-BSim in Magny Cours machine (left) and the relative performance of CC-BSim's stack implementation and SimStack against flat-combining (right). Random work is 512 at most.

Similarly to the experiments performed on Magny Cours machine, CC-BSim outperforms all of other algorithms (Figures 2) in the Niagara 2 machine. More specifically, CC-BSim outperforms flat-combining by a factor of up to 1.70 and CLH spin locks by a factor of up to 1.57. Notice that CLH spin locks are more competitive in the Niagara 2 machine, since the Niagara 2 machine consists of a few cpus but each of them handles a lot of threads (so communication among these threads is very fast). The lock free implementation of the `Fetch&Multiply` object is much slower than Sim and flat-combining in most of the cases. It is worth-pointing out that implementing the appropriate backoff scheme has a significant impact on performance of the lock free implementation in the Niagara 2 machine. DSM-BSim exposes almost the same performance to flat-combining in the Niagara 2 machine. As shown in Figures 1, 2, all algorithms perform faster in case $n = 1$ in the Magny Cours machine than in cases where $n > 1$, while in the Niagara 2 machine they perform faster when $n > 1$. This is probably due to the fact that the Magny Cours machine consists of few but very fast processing cores, while the Niagara 2 machine is able to handle a lot of threads but its processing cores are much slower.

At the experiments presented in [11], the random work was at most 512. In this paper, a lower random work was chosen since greater values led to very low contention in the experiments executed in the Niagara 2 machine. Thus, the experiments presented in [11] are not directly comparable to

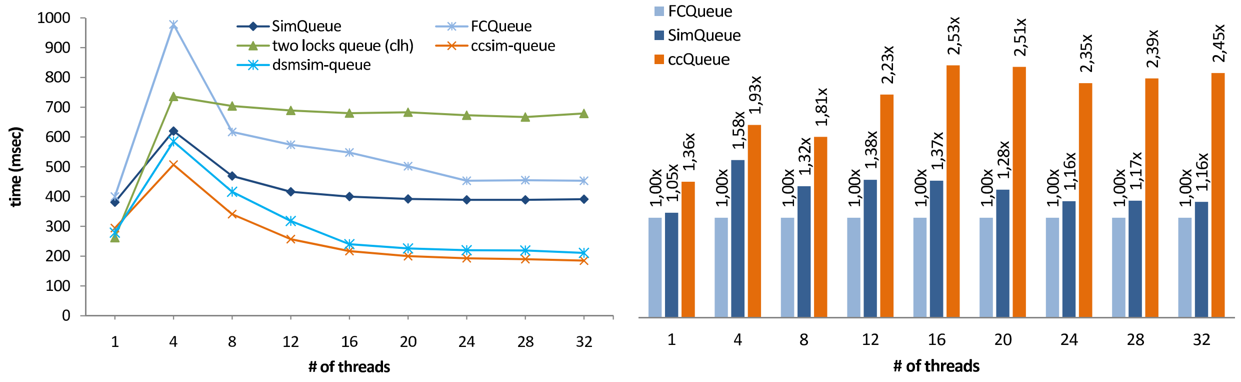

Figure 7: Absolute performance of the queue implementation based on CC-BSim in Magny Cours machine (left) and the relative performance of CC-BSim's queue implementation and SimStack against flat-combining (right).

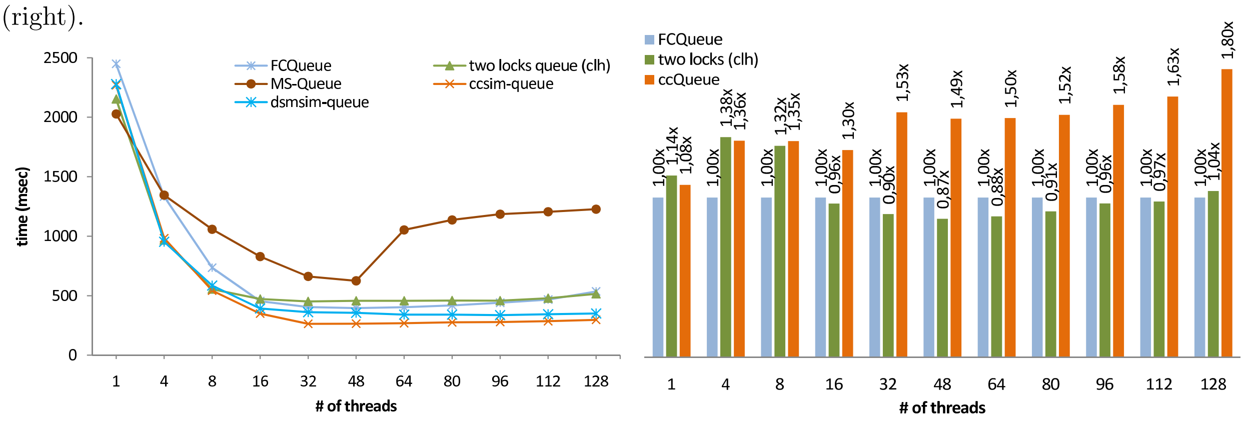

Figure 8: Absolute performance of the queue implementation based on CC-BSim in Niagara 2 machine (left) and the relative performance of CC-BSim's queue implementation and SimStack against flat-combining (right).

that presented in Figure 1. However, directly comparable experiments to that of [11] for the Magny Cours machine are presented in Figure 3. As expected, performance advantages of CC-BSim and DSM-BSim are similar to that presented on Figure 1.

In order to investigate the performance advantages of CC-BSim and DSM-BSim, we implemented a shared stack based on them and we compared the performance of these implementations with the state-of-the-art shared stack implementations. Specifically, the shared stack implementations based on CC-BSim and on DSM-BSim were evaluated against the wait-free stack implementation called SimStack presented by Fatourou and Kallimanis in [11], the lock free stack implementation presented by Treiber in [22][1], a stack implementation based on CLH spin lock [8, 17], and a linked stack implementation based on flat-combining [12, 13]. In order to evaluate these shared stack implementations, 10^6 pairs of PUSH, POP operations were executed with each thread executing $10^6/n$ pairs. Random work (at most 64) was added between the execution of two consecutive stack operations.

[1]In the experiments performed in the Magny Cours machine, the lock free implementation of a stack presented in [22] performed poorly; for this reason, we decided not to include results for it in the diagrams (Figure 4) for the Magny Cours machine.

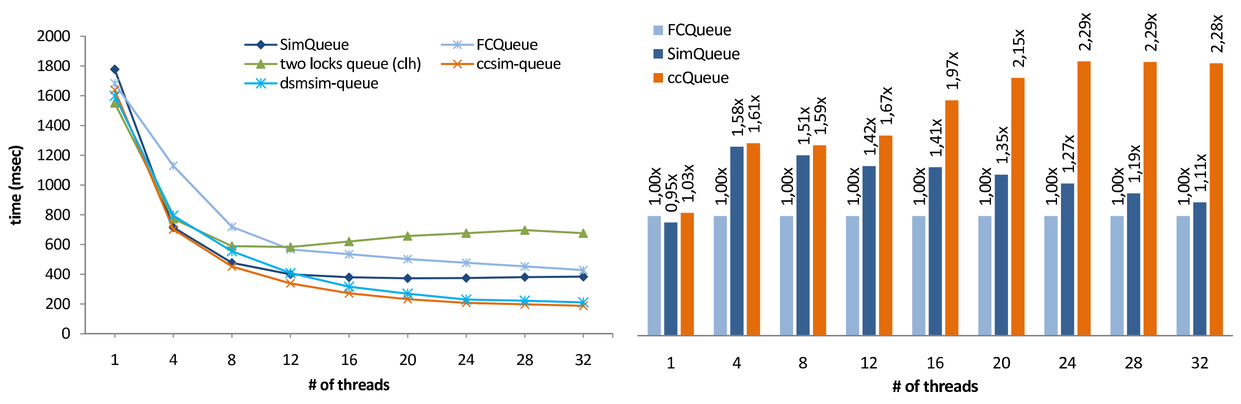

Figure 9: Absolute performance of the queue implementation based on CC-BSim in Magny Cours machine (left) and the relative performance of CC-BSim's queue implementation and SimStack against flat-combining (right). Random work is 512 at most.

As it is shown in Figure 4, the shared stack based on CC-BSim, called CC-BStack, performs up to 1.68 times faster that the linked stack based on flat-combining in the Magny Cours machine, and up to 1.59 times faster than SimStack. The stack implementation based on CLH spin locks did not perform well in the Magny Cours machine. The stack implementation based on DSM-BSim algorithm performs worse than CC-BSim but it is better than all other algorithms. Experiments with random work at most 512 for the Magny Cours machine are illustrated in Figure 6. In the Niagara 2 machine the shared stack based on CC-BSim performs 1.79 times faster that the linked stack based on flat-combining (Figure 5). The stack implementation based on DSM-BSim algorithm performs worse than CC-BSim but it is again better than all other algorithms. CLH spin locks perform worse than flat-combining and their performance is up to 2.4 times slower than CC-BSim.

In our final experiments, we implement and experimentally analyze a shared queue based on CC-BSim and DSM-BSim. More specifically, the two locks queue implementation presented in [19] is enhanced by replacing the ordinary locks either with CC-BSim or with DSM-BSim. Our queue implementation based on CC-BSim or DSM-BSim was compared with the wait-free queue implementation presented by Fatourou and Kallimanis in [11], the lock free queue implementation, and the two locks implementation presented in [19][2], and the queue implementation based on flat-combining [12, 13]. The queue experiment is similar to that for stacks.

As illustrated in Figure 7, SimQueue exhibits better performance than any algorithm other than CC-BQUEUE and DSM-BQUEUE in the Magny Cours machine, as it was expected [11]. It is worth pointing that, in this machine, CC-BQUEUE performs up to 2.53 times faster than the queue implementation based on flat-combining (Figure 7). Furthermore, CC-BQUEUE performs 2.1 times better than SimQueue. On the other hand, DSM-BQUEUE algorithm performs a little worse than CC-BQUEUE but its better than all other algorithms. Experiments with random work at most 512 for the Magny Cours machine are illustrated in Figure 9.

As illustrated in Figure 8, flat-combining performs better than all algorithms other than CC-BQUEUE and DSM-BQUEUE in the Niagara 2 machine (recall that SimQueue has not been imple-

[2]In the experiments performed in the Magny Cours machine, the lock free implementation of a queue presented in [19] performed poorly; for this reason, we decided not to include results for it in the diagrams (Figure 7) of the Magny Cours machine.

mented in this machine). It is worth pointing that CC-BQUEUE performs up to 1.8 times faster than the queue implementation based on flat-combining. It is also noticeable that the performance gap between the queue implementation based on flat-combining and the two locks queue is smaller in the Niagara 2 machine. This is due to the fact that the CLH spin locks perform very well in this machine and the parallel use of two different locks (one for enqueues and one for dequeues) gives a performance boost in the two locks algorithm. Again, DSM-BQUEUE performs slightly worse than CC-BQUEUE but its better than all other algorithms in the Niagara 2 machine.

Acknowledgments. We would like to thank Dimitris Nikolopoulos for arranging the provision of access to some of the multi-core machines of the Department of Computer Science at Virginia Tech where we ran our experiments. Many thanks also to Michael Scott for providing to us access to the Rochester's Niagara 2 machine. Thanks to Nir Shavit and Danny Hendler for providing the code of flat combining and of the concurrent data structures based on them.

Nikolaos Kallimanis is supported by a PhD scholarship from the Empirikion Foundation, Athens, Greece. The work of Panagiota Fatourou was supported by the European Commission under the 6th and 7th Framework Programs through the ENCORE (FP7-ICT-248647), Hi-PEAC2 (FP7-ICT-217068), and TransForm (FP7-MCITN-238639) projects.

References

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[2] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[3] James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, dec 1999.

[4] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[6] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.

[7] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Blade computing with the amd opteron processor (magny-cours). *Hot chips 21*, August 2009.

[8] T. S. Craig. Building fifo and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[9] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures (to appear)*, 2011.

[10] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue adaptive universal constractions. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 127–141, 2009.

[11] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures (to appear)*, 2011.

[12] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. The code for flat combining. http://github.com/mit-carbon/flat-combining.

[13] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 355–364, 2010.

[14] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, jan 1991.

[15] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, nov 1993.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, 1990.

[17] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, 1994.

[18] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[19] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[20] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *EuroSys*, pages 305–315, 2006.

[21] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[22] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[23] Pen-Chung Yew, Nian-Feng Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388 –395, April 1987.