Technical Report no. 2004-04

Allocating memory in a lock-free manner *

Anders Gidenstam **Marina Papatriantafilou**
Philippas Tsigas

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2004

*This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC).

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2004-04
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2004

Abstract

The large potential of multiprocessor systems is frequently not fully realized by the way that they are supported by the system services. It is often the case that certain synchronization methods, such as lock-based ones, may limit the parallelism. But it is possible to share data without using locks, by employing wait/lock-free (a.k.a. non-blocking) synchronization, which offers guarantees for time-dependability and fault-tolerance. It is significant to see the impact of wait/lock-free synchronization design in key services and data structures for multiprocessor systems, such as the memory allocation service. Efficient, scalable memory allocators for multithreaded applications in multiprocessor systems is a significant goal of recent research projects.

In this paper we propose a lock-free memory allocator, to allow to take advantage of the parallelism in the system. Its architecture is inspired by Hoard, which is a successful system for concurrent memory allocators, based on a modular, scalable design that preserves scalability and help circumvent significant obstacles such as false sharing and heap blowup. Within our effort on designing appropriate lock-free algorithms for the synchronization in this system, we propose and give a non-blocking implementation of a new data structure called flat-sets, supporting conventional "intra-object" operations (to insert and find elements) as well as "inter-object" operations, for moving an element from one flat-set to another.

We have implemented the memory allocator in a set of multiprocessor systems (UMA Sun SunFire 880, ccNUMA SGI Origin 2000, ccNUMA SGI Origin 3800) and studied its behaviour. The results show that the good properties of circumventing false sharing and heap blowup provided by Hoard are preserved, while the scalability properties are enhanced even further with the help of the lock-free synchronization.

1 Introduction

Some form of dynamic memory management is used in most computer programs for multiprogrammed computers. It comes in a variety of flavors, from the traditional manual general purpose allocate/free type memory allocator to advanced automatic garbage collectors.

In this paper we focus on conventional general purpose memory allocators (such as the "libc" malloc) where the application can request (allocate) arbitrarily-sized blocks of memory and free them in any order. Essentially the memory allocator is an online algorithm that manages some pool of memory (heap), for example a contiguous range of addresses or a set of such ranges, keeping track of which parts of that memory are currently given to the application as well as which parts are unused and can be used to meet future allocation requests from the application. The memory allocator is not allowed to move or otherwise disturb pieces of memory that are currently owned by the application.

A good allocator should aim at minimizing *fragmentation*, i.e. the splitting of free memory into small, non-contiguous blocks that cannot be used (allocated) by the application. Besides, multithreaded programs add a number of extra complications to the memory allocator. Obviously some kind of synchronization has to be added to protect the heap during concurrent requests. There are also other involved issues, which have significant impact on application performance when the application is run on a multiprocessor [Ber02]. Summarizing the goals, a good concurrent memory allocator should (i) avoid *false sharing*, which is when different parts of the same cache-line end up being used by threads running on different processors; (ii) avoid *heap blowup*, which is an overconsumption of memory that may occur if the memory allocator fails to make memory deallocated by threads running on one processor available to threads running on other processors; (iii) ensure *efficiency* and *scalability*, i.e. the concurrent memory allocator should be at least as fast as a good sequential one when executed on a single processor and its performance should scale with the load in the system. In the next section these issues and goals are explained in more detail, together with an overview of concurrent allocator designs.

The Hoard[BMBW00] memory allocator introduces a design to meet the above goals. The allocation is done on the basis of per-processor heaps, which helps avoiding false sharing and reduces the synchronization overhead in many cases, thus improving both performance and scalability. Memory requests are mapped to the closest matching size from a fixed set of size-classes, which bounds internal fragmentation. The heaps consist of sets of superblocks, where each superblock handles blocks of one size class, which helps in coping with external fragmentation. The avoidance of heap blowup is achieved via the algorithm that returns blocks to the heaps where they were allocated from and via the algorithm that moves superblocks among

heaps.

Regarding efficiency and scalability, it is known that the use of locks in synchronization is a limiting factor, especially in multiprocessor systems, since it reduces parallelism. Constructions which guarantee that concurrent access to shared objects is free from locking are of particular interest, since they help to increase the amount of parallelism and to provide fault-tolerance. This type of synchronization is called lock-/wait-free, non-blocking or optimistic synchronization [Bar93, GC96, Her91, Rin99]. The potential of this type of synchronization in the performance of system-services and data structures has also been pointed out earlier, in [DG02, GC96, MP91].

The contribution of the present paper is a new memory allocator based on lock-free, fine-grained synchronization, to enhance parallelism, fault-tolerance and scalability. The architecture of our allocation system is inspired by Hoard, due to its well-justified design decisions, which we roughly outlined above. In the process of designing appropriate data structures and lock-free synchronization algorithms for our system, we introduced a new data structure, which we call *flat-set* which supports a subset of internal operations of common sets, as well as "inter-object" operations, for moving an element from one flat-set to another in a lock-free manner. The lock free synchronization algorithms we introduce make use of standard synchronization primitives provided by multiprocessor systems, namely `compare-and-swap`, or its equivalent `load-linked/store-conditional`, commonly denoted by `CAS` and `LL/SC`, respectively. In the next section we provide a description of the above synchronization primitives as well as some more background information on wait-/lock-free synchronization. We should mention here that throughout the paper we use the terms non-blocking and lock-free interchangeably.

We have also implemented and evaluated the allocator proposed here on common multiprocessor platforms, namely a UMA Sun SunFire 880 running Solaris 9, a ccNUMA SGI Origin 2000 running IRIX 6.5 and a ccNUMA SGI Origin 3800 also running IRIX 6.5. In all these systems we compare our allocator with the standard "libc" allocator of the respective platform using standard benchmark applications, which test the efficiency, scalability, cache behaviour and memory consumption behaviour. On the Sun platform, for which we had the original Hoard allocator available, we also compare with Hoard itself. The results show that our system preserves the good properties of Hoard, while it offers an even higher scalability potential, as expected by its lock-free nature.

The next section gives some background/overview information on concurrent memory management and non-blocking synchronization. Section 3 describes our memory allocator, including the lock-free implementation of the data structures designed for this purpose. Section 4 describes details on the implementation done for the experimental evaluation of the system,

including the platforms on which it was implemented and benchmark information. The results of our experimental evaluation are also presented in that section. We conclude with a section that provides a discussion on the achieved results and description of future work.

2 Background

This work builds on results in memory management and results in non-blocking synchronization. In this section we present related background information on concurrent memory allocators in the first subsection and on non-blocking synchronization in the second.

2.1 Concurrent Memory Allocators

As mentioned in the introduction, there is a set of extra complications introduced in the memory allocation issues from the perspective of multiprocessor systems and multithreaded applications.

The first of these is *false sharing* which is when different parts of the same cache-line end up being used by threads running on different processors. This will put a potentially large and completely unnecessary load on the cache-coherence mechanism. This can never be avoided completely since application threads may pass allocated memory between themselves but a memory allocator can *actively induce* false sharing by directly satisfying memory requests from different processors with memory from the same cache-line.

The second issue is *heap blowup* which is a potentially unbounded overconsumption of memory that might occur if the memory allocator fails to make memory freed by one processor available to others, e.g. as the result of a coarse policy for avoiding false sharing. A typical application that triggers this is an application that have producer and consumer threads, where the producer allocates memory and passes it to the consumer which in turn frees the memory. If the memory freed by the consumers is never made available to the producers then the resulting heap blowup can be unbounded.

The last issues are *scalability* and *speed*. For a memory allocator to be scalable, its performance has to scale well with the number of processors and the load in the system. In terms of speed, the concurrent memory allocator should be about as fast as a good sequential one in order to ensure good performance even when a multithreaded program is executed on a single processor.

Below follows a brief overview of some concurrent memory allocator designs based on the taxonomy presented in [Ber02]:

Single serial heap. A normal sequential memory allocator protected by a global lock. This type of memory allocator will scale poorly on multiprocessors, since only one thread can access the heap at a time, and it may, in

fact, in some instances also perform bad on a single processor since a thread holding the heap lock might easily be delayed inside the memory allocator code (for example by preemption or a page fault). It is also prone to induce false sharing, but does not suffer from heap blowup and should be fast on a single processor in most cases (but note the bottleneck problem caused by the lock as mentioned above). Examples: The standard memory allocators on Solaris, Irix and Windows 2000.

Concurrent single heap. A memory allocator that uses a single heap implemented using fine-grained synchronization so that several threads may operate on it concurrently. This kind of memory allocator avoids heap blowup and has the potential to be scalable. But it is prone to induce false sharing and is not easy to make fast due to the synchronization overhead.

Pure private heaps. The memory allocator maintains a separate heap for each processor, where threads running on that processor allocate memory. When a thread frees some memory it is added to the heap of the processor running that thread. This causes pure private heaps to suffer from potentially unbounded heap blowup. On the other hand this type of memory allocator is scalable, fast and less prone to induce false sharing (if designed correctly). Examples: STL allocator [SGI], Cilk [BL94].

Private heaps with ownership. These memory allocators are similar to pure private heaps but freed memory is always returned to the heap it was allocated from. This bounds the worst case heap blowup to $O(P)$, where P is the number of processors. Examples: MTmalloc (Solaris), Ptmalloc (glibc).

Private heaps with thresholds. Private heaps with thresholds use a global heap in addition to the per-processor heaps in order to avoid the $O(P)$ heap blowup that private heaps with ownership suffer from. This is done by transferring a part of the free memory in a per-processor heap to the global heap when the amount of free memory in the per-processor heap becomes too large. The memory in the global heap can be transferred to other per-processor heaps for reuse as needed. Examples: Hoard[BMBW00], LFmalloc[DG02](mostly) and the memory allocator presented in this paper.

2.2 Non-blocking Synchronization

The most commonly required consistency guarantee for shared data objects is *atomicity*, a.k.a. *linearizability* [HW90]. A shared object (its implementation) is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect in an atomic time instant which lies in its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification. The latter means that if we speak of e.g. read/write objects, the value returned by each read equals the value written by the most recent write according to the sequence of "shrunk" operations in the time axis.

Compared to the traditional solution for maintaining the consistency of a shared data object by enforcing mutual exclusion, non-blocking implementations of shared data objects are an alternative approach. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion [Bar93, GC96, Her91, Rin99]. Non-blocking (a.k.a. optimistic) synchronization can be lock-free or wait-free. *Lock-free* algorithms guarantee that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. However, there is a risk that the progress of other operations might cause one specific operation to take unbounded time to finish. In a *wait-free* algorithm, every operation is guaranteed to finish in a bounded number of its own steps, regardless of the actions of concurrent operations.

Lock-free algorithms typically involve fine-grained synchronization, with attempts to commit updates using certain synchronization primitives (like CAS, LL/SC; see below) or to verify non-interfered access to small amounts of shared data. If such an attempt fails, then that process/thread needs to *retry*. The above may happen due to preemption or due to interleaving by threads running in parallel. *Helping* is a method proposed to alleviate the problem: an operation that detects that it has preempted or has otherwise interleaved steps with another operation, helps the latter operation to progress before proceeding with its own steps, as to reduce the fail-retry overhead.

It has been shown that there exist *universal synchronization primitives*, that can implement, in a wait-free manner –hence, also in lock-free manner–, any object with a sequential specification using those primitives [Her91]. There exist also *universal constructions* that can implement any object using these universal synchronization primitives [Her91]. These constructions also use helping. However, being generic, they are are introduced for showing feasibility and not performance.

Some of the aforementioned universal primitives are common instructions available in most processors, e.g. the *Compare-And-Swap* instruction (also denoted *CAS*), which atomically executes the steps described in Fig. 1. If CAS cannot assign the new value to the location for which it is invoked, we say that it *fails*, otherwise, it *succeeds*. CAS is available in, e.g. SPARC processors.

Another primitive which is equivalent with CAS in synchronization power (in the sense described above) is the *Load-Linked/Store-Conditional* (also denoted *LL/SC*) pair of instructions, available in, e.g. MIPS processors. LL/SC is used as follows: (i) LL loads a word from memory. (ii) A short sequence of instructions may modify the value read. (iii) SC stores the new value into the memory word, unless the word has been modified by other process(es) after LL was invoked. In the latter case the SC *fails*, otherwise the SC *succeeds*.

```
CAS(mem : access integer; new, old : integer) return integer
begin
  tmp := *mem;
  if tmp == old then *mem := new;  /* CAS succeeded */
  return tmp;
end CAS;
```

Figure 1: The compare-and-swap (denoted CAS) synchronization primitive.

An issue that sometimes arises in connection with the use of CAS, is the so-called ABA problem. It can happen if a thread reads a value A from a shared variable, and then invokes a CAS operation to try to modify it. The CAS will (undesirably) succeed if between the read and the CAS other threads have changed the value of the shared variable from A to B and back to A. A common way to cope with the problem is to use version numbers of b bits as part of the shared variables [Val95]. Then, for the same problem to occur, it would be necessary to have a sequence of 2^b successful CAS operations between the read A and its corresponding CAS, with the last such operation storing value A to the shared variable. By choosing b appropriately, this is made extremely unlikely. An alternative method to cope with the ABA problem is to introduce special NULL values. This method is proposed and used in a lock-free queue implementation in [TZ01b]. Another method for the same problem is to use an appropriate garbage-collection mechanism, as introduced in [Mic02b].

There is a plethora of research articles that focus on wait-free and lock-free synchronization (for a few examples cf. [Bar93, Har01, Her91, Her93, HPT02, Mic02b, Moi97, Rin99, PT95, PT97, ST03, TZ01a, Val94, Val95]). Non-blocking algorithms have been shown to also have significant impact in applications [TZ01a, TZ02], and recently NOBLE, which is a library of shared data structures and includes blocking and non-blocking implementations, has been presented [ST02].

3 Lock-free memory allocation

3.1 Architecture

The architecture of our lock-free memory allocator is inspired by and closely follows that of Hoard[BMBW00], which is a well-known and practical concurrent memory allocator for multiprocessors.

The memory allocator provides allocatable memory of a fixed set of sizes, called *size-classes*. The size of memory requests from the application are rounded upwards to the closest size-class. In order to reduce false-sharing and contention, the memory allocator distributes the memory into *per-processor heaps*. The managed memory is handled internally in units

called *superblocks*. Each superblock contains allocatable blocks of one size-class. Initially all superblocks belong to the *global heap*. During an execution superblocks are moved to per-processor heaps as needed. When a superblock in a per-processor heap becomes empty (i.e. none of its blocks are allocated) it is moved back to the global heap. The superblocks in a per-processor heap are stored and handled separately, based on their size-class. Within each size-class the superblocks are kept sorted into bins based on *fullness*(cf. Fig. 3.1). As the fullness of a particular superblock changes it is moved between the groups.

A memory request ("malloc" call) will first search for a free block among the superblocks in the "almost full" fullness-group of the requested size-class in the appropriate per-processor heap for a one with available space. If no suitable superblock is found there, it will proceed to search in the lower fullness-groups, and, if that, too, is unsuccessful, it will request a new superblock from the global heap. Searching the almost full superblocks first limits external fragmentation.

When an allocated block is freed (by a "free" call) it is returned to the superblock it was allocated from and, if the new fullness requires so, the superblock is moved to another fullness-group.

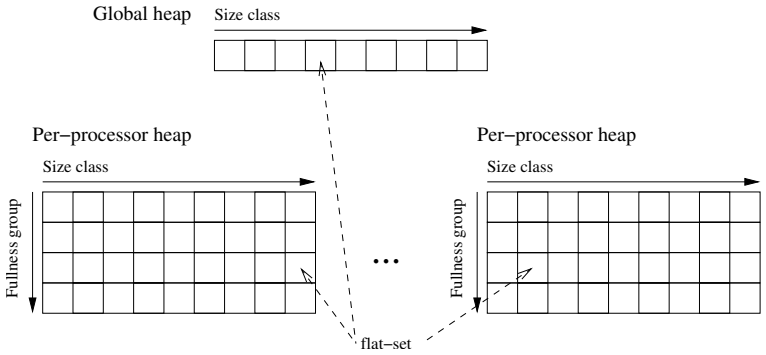


Figure 2: The organization of the global and per-processor heaps.

3.2 Managing superblocks

Since the number of superblocks in each fullness-group varies over time, a suitable collection-type data structure is needed to implement a group. Hoard, which uses mutual-exclusion on the level of per-processor heaps, uses linked-lists of superblocks for this purpose, but this issue becomes very different in a lock-free allocator. While there exist several lock-free linked-list implementations, e.g. [Har01, Mic02a, Val95], these cannot be applied here, because not only do we want the operations on a list to be lock-free, but we also need to be able to move a superblock from one list to another without making it inaccessible to other threads during the move.

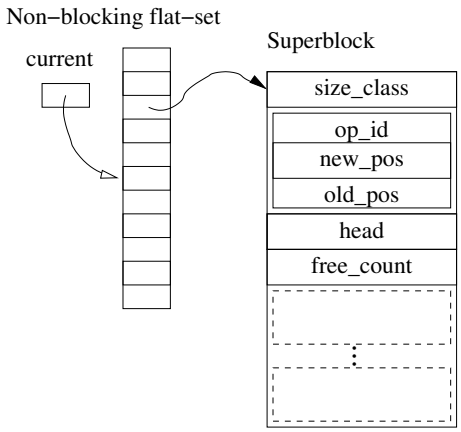


Figure 3: A non-blocking flat-set and a superblock.

To address this, we propose a new data structure we call a *bounded non-blocking flat-set*, supporting conventional "internal" operations (find_any and insert elements) as well as "inter-object" operations, for moving an element from one flat-set to another. A fundamental part of this structure is a lock-free algorithm for moving a superblock reference between two shared locations. We will present this algorithm first and then the algorithms supporting the "internal"-operations on these *flat-sets*.

In order to support inter-flat-set operations it is crucial to be able to move superblocks from one set to another in a lock-free fashion. The requirements that make this difficult are:

1. the superblock should be reachable for other threads even while it is being moved inside a per-processor heap; and
2. the number of shared references to the superblock should be the same after a move (or set of concurrent move operations) finish.

Below we present a lock-free algorithm that moves a reference to a superblock from one shared variable (pointer) to another and satisfies the above requirements.

How to move a shared reference

The algorithm is moving a superblock reference `sb` from a shared variable `FROM` to a shared variable `TO`. At this point we assume that the target location (i.e. `TO`) is known; later on we will show how such target locations are found. The algorithm requires the superblock to contain an auxiliary variable `mv_info`, which contains the three fields `op_id`, `old_pos` and `new_pos`, and that each reference variable contains a version field.

A move operation may *succeed* or *fail* (abort). It will succeed if it is

completed successfully by the thread that initiated it or by a helping-thread. It will fail if it detects that another, overlapping move of the same superblock has already removed the reference to the superblock from the `FROM` location or if it detects that a preceding or overlapping move of another superblock has occupied the `TO` location. To ensure the lock-free property, the move operation is divided into a number of atomic suboperations (cf. Fig. 4). A move operation that encounters an unfinished move of the same superblock will help the old operation to finish before it attempts to perform its own move. In the presentation here and in the pseudo-code in Fig. 5 we use the atomic primitive CAS but other atomic synchronization primitives, such as LL/SC could be used as well. The helping procedure is identical to steps 2 - 4 of the normal move operation described below.

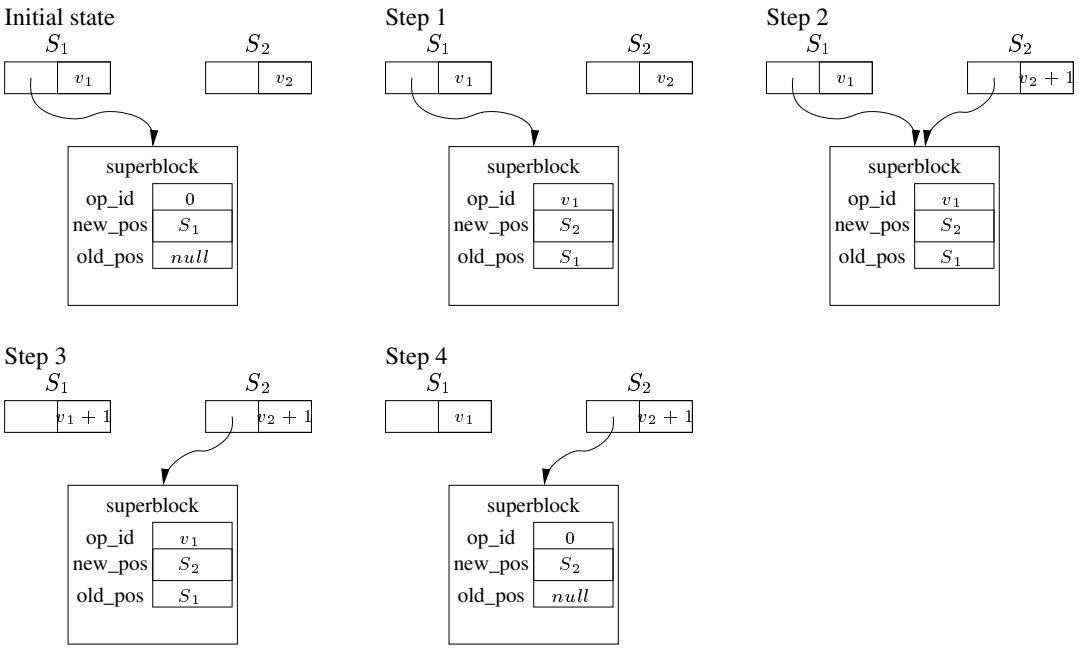


Figure 4: The steps of a lock-free move operation from location S_1 to location S_2.

1. The move is initiated by atomically *registering* the operation, by setting `mv_info.op_id = ver(FROM)`, `mv_info.old_pos = FROM` and `mv_info.new_pos = TO`. The assignment succeeds if and only if the current value of `mv_info.op_id` is zero. If the current value of `mv_info.op_id` is nonzero, there is an ongoing move that needs to be helped before this one can proceed. If the reference to the superblock disappears from `FROM` before this move has been registered, this move operation

```
M1  move(sb   : in     superblock_ref,
M2       from : access superblock_ref,
M3       to   : access superblock_ref)
M4      return status
M5    new_op, old_op, help_op : move_info;
M6  begin
M7    -- Step 1: Initiate move.
M8    loop
M9      new_op.op_id   := version(sb);
M10     new_op.old_pos := *from;
M11     new_op.new_pos := *to;
M12     old_op         := sb.mv_info;
M13     old_op.op_id   := 0;
M14     old_op.old_pos := null;
M15     help_op := CAS(sb.mv_info,
M16                    old_op,
M17                    new_op);
M18     if help_op == old_op then
M19       break;
M20     else
M21       move_help(help_op.op_id,
M22                 help_op.old_pos,
M23                 help_op.new_pos);
M24       if from /= sb then return SB_GONE;
M25     end if;
M26   end loop;
M27
M28   return move_help(sb,
M29                    from,
M30                    to);
M31 end move;
```

```
H1  move_help(sb   : in     superblock_ref,
H2            from : access superblock_ref,
H3            to   : access superblock_ref)
H4          return status
H5    old, new, res : superblock_ref;
H6  begin
H7    -- Step 2: Update "TO".
H8    old := *to;
H9    new := set_version(sb, inc(version(old)));
H10   if sb.mv_info.op_id == version(sb) then
H11     res := CAS(to,
H12                set_version(null, version(old)),
H13                new)
H14   else
H15     res := sb; -- The move has been helped
H16                -- to finish.
H17   end if
H18   if res /= old then
H19     if res == sb then
H20       -- We have been helped, continue.
H21     else
H22       -- To is occupied, abandon this operation.
H23       CAS(sb.mv_info,
H24           (version(sb), from, to),
H25           (0, null, null));
H26       return SB_NOT_MOVED;
H27     end if;
H28   end if;
H29   -- Step 3: Clear "FROM".
H30   CAS(from,
H31       sb,
H32       set_version(null, inc(version(sb))));
H33   -- Step 4: Remove operation information.
H34   CAS(sb.mv_info,
H35       (version(sb), from, to),
H36       (0, null, to));
H37   return SB_MOVED_OK;
H38 end move_help;
```

Figure 5: The lock-free move operation.

is abandoned and returns *fail*.

2. Atomically set `TO = sb` (while increasing the version of `TO`) if and only if the current value of `TO` is `null`. If the value of `TO` is not `null` then this move is abandoned, step 4 is performed to remove the information about the move from the superblock and *fail* is returned.
3. Atomically set `FROM = null` (while increasing the version) if and only if `FROM == sb` (i.e. if no one else has helped the move to complete).
4. Atomically remove the move information from `sb` if and only if it is still there (i.e. if no one else has helped the move to complete). Return *success*.

The auxiliary `mv_info` variable in a superblock might need to be larger than one word. In that case one needs a multi-word CAS operation, which

can be implemented from the common single-word CAS (cf. [HT03, HFP02]).

Lemma 3.1 (Linearizability) *All executions of a set of (possibly concurrent) move operations*
$m_1, m_2, \ldots, m_n$ on the same superblock sb are linearizable.

Proof. (Sketch) The linearization point of each operation is the CAS of step 1: A new move operation m will successfully register when the operation information for m is successfully written into the superblock (see step 1 above). This can only happen when there is no other ongoing move concerning that superblock. Once the operation m has been registered (i.e. it has completed step 1 successfully) no other move operation m' concerning this superblock will get registered before m has finished (either by succeeding or failing) by itself or by having been helped by such an m'. Therefore, such an operation m' will be linearized after m. By recursively repeating the above argument, we get the lemma. □

Lemma 3.2 (Reachability) *Consider a superblock sb, a set of shared locations $s_1, s_2, \ldots, s_m$ and a set of move operations M_{sb} on sb, where, initially, exactly one shared location s_1 contains a reference to the superblock. Then at least one of the shared locations holds a reference to the superblock at all times during the execution of the move operations.*

Proof. (Sketch) The shared location `FROM` is not cleared before the shared location `TO` is successfully updated with a reference to the superblock being moved. Furthermore, the only way to remove the superblock reference from `TO` is to successfully use another move operation to move the superblock reference from `TO` to another shared location. □

Lemma 3.3 (No pointer multiplication) *Consider a superblock sb, a set of shared locations*
$s_1, s_2, \ldots, s_m$ and a set of move operations M_{sb} on sb, where, initially, exactly one shared location s_1 contains a reference to the superblock, then:
(i) After all moves of sb have finished there is exactly one shared reference to sb.
(ii) At any given time there are no more than two shared references to sb.

Proof. (Sketch) Since no new move can begin before all previous ones have finished (by themselves or through helping) there will only be one shared reference when a move begins and the move itself creates at most one more during its execution. □

Bounded non-blocking flat-sets

A bounded non-blocking flat-set provides the following operations:

1. `get_any` Returns any item in the flat-set.
2. `insert` Inserts an item into the flat-set.

The flat-set data structure consists of an array of M shared locations `SET[i]` (each capable of holding a reference to a superblock), a shared index variable `CURRENT` and two shared counters `OP_CNT1` and `OP_CNT2`. The operations are shown in Fig. 6 and are briefly described below.

The `insert` operation scans the array forward from `CURRENT` (while increasing `CURRENT`) until it finds an empty slot. It will then attempt to move the superblock reference to be inserted into this slot using the move operation described above. The `OP_CNT1` counter is incremented at the start of the operation and `OP_CNT2` at the end.

The `get_any` operation scans the array backwards starting from `CURRENT`, which points to the position after the one used by the last insert operation, until it finds a location that contains a superblock reference. The operation repeats the scan until either a reference is found or the operation counters have not changed during the scan.

Note that `get_any` returns both a superblock reference and a reference to the shared location containing it. To move a superblock reference from one flat-set to another we use that information in the call to `insert` on the destination flat-set.

3.3 Managing blocks

The allocatable memory blocks within a superblock are kept in an lock-free IBM free-list[Mic02a]. This is essentially a stack implemented from a single-linked-list where the push and pop operations are performed by doing CAS on the head-pointer. Each block has a header containing a pointer to the superblock it belongs to and a next pointer for the free-list. The two free-list operations `get_block` and `put_block` are shown in Fig. 7. The free blocks counter, `sb.free_block_cnt`, is used to estimate the fullness of a superblock.

4 Performance evaluation

Systems

There are two popular families among cache-coherent multiprocessor architectures – UMA (Uniform Memory Architecture) and NUMA (Non Uniform Memory Architecture). The difference between them is how the memory is accessed. In a UMA system all processors have the same latency to the

```
insert(set : in out flat_set,
       sb  : in     superblock_ref,
       sbr : in out (access superblock_ref))
      return status
  op_cnt1, op_cnt2, i, j : integer;
begin
  FAA(&set.op_cnt1, 1);
  loop
    op_cnt1 := set.op_cnt1;
    op_cnt2 := set.op_cnt2;
    for j := 0 .. set.size - 1 do
      i := set.current;
      FAA_MOD(&set.current, 1, set.size)
      if set.array[i] == null then break;
    end for;
    case move(s, sbr, &set.array[i]) is
      when SB_MOVED_OK:
        FAA(&set.op_cnt2, 1);
        sbr := &set.array[i];
        return SB_MOVED_OK;
      when SB_MOVED:
        return SB_MOVED;
      when others:
        -- Retry.
    end case;
  end loop;
end insert;
```

```
get_any(set : in out flat_set,
        sb  : in out superblock_ref,
        sbr : in out (access superblock_ref))
       return status
  op_cnt1, op_cnt2, i, j : integer;
begin
  loop
    op_cnt1 := set.op_cnt1;
    op_cnt2 := set.op_cnt2;
    i := set.current;
    for j := 0 .. set.size - 1 do
      sb := set.array[i];
      if sb /= null then
        sbr := &set.array[i];
        return SUCCESS;
      end if;
      if i == 0 then i := set.size else i--;
    end for;
    if op_cnt1 == set.op_cnt1 and
       op_cnt2 == set.op_cnt2
    then
      return FAILURE;
    end if;
  end loop;
end find_any;
```

Figure 6: The flat-set insert and find_any operations. FAA_MOD is the fetch_and_add operation modulo set.size.

memory. In a NUMA system, this is not the case, since access to memory on another node can be significantly slower.

The performance of the new lock-free allocator has been measured on a three multiprocessor systems, both on UMA and NUMA memory architectures. The three systems are

- An UMA Sun SunFire 880 with 4 900MHz UltraSPARC III+ (8MB L2 cache) processors running Solaris 9.
- A ccNUMA SGI Origin 2000 with 8 195Mhz MIPS R10000 (4MB L2 cache) processors running IRIX 6.5.
- A ccNUMA SGI Origin 3800 with 128[1] 500Mhz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

The experiments were run concurrently with other workloads, as we did not have exclusive access to the systems. However, on the SGI Origin 3800 system the the batch-system ensures that the requested number of CPUs (up to 32) is available to execute the threads.

The atomic synchronization primitives `compare_and_swap` and `fetch_and_add` were implemented in machine language from the hardware synchronization instructions available on the respective system.

[1]It is only possible to reserve up to 32 CPUs on this machine.

```
get_block(sb : in superblock_ref)
         return block_ref
   nb, nh : block_ref;
begin
  nb := sb.freelist_head;
  while nb /= null do
    nh = CAS(&sb.freelist_head,
             nb, nb.next);
    if nh == nb.next then
      FAA(&sb.free_block_cnt, -1);
      break;
    end if;
    nb := sb.freelist_head;
  end loop;
  return nb;
end get_block;
```

```
put_block(sb : in superblock_ref,
          bl : in block_ref)
  oh : block_ref;
begin
  loop
    bl.next = sb.freelist_head;
    oh = CAS(&sb.freelist_head,
             bl.next, bl)
    if oh == bl.next then break;
  end loop;
  FAA(&sb.free_block_cnt, 1);
end put_block;
```

Figure 7: The superblock free-list operations.

Benchmarks

- **Larson** [BMBW00, Ber02, LK98] Each thread allocates and deallocates objects and transfers some objects to other threads to be deallocated there. The intention is to simulate the behaviour of a server application. It tests throughput, heap blowup, false-sharing and scalability.
- **Active-false** and **passive-false** [BMBW00, Ber02] Each thread repeatedly allocates an object of a certain size, writes to that object a large number of times and then deallocates the object again. The difference between the tests is that *passive-false* first provides each thread with an object allocated by the main thread. These tests should induce cache trashing for allocators that induce false sharing.

Implementation

In our memory allocator we used the CAS primitive to implement the lock-free operations. To avoid ABA problems we used the version number solution ([Val95], cf. section 2.2). In this implementation we have used 16-bit version numbers. Further, we used superblocks of 64KB and size-classes that are powers of 2, starting from 8 bytes. Blocks larger than 32KB are allocated directly from the operating system instead of being handled in superblocks. Our implementation uses four fullness-groups and uses a fullness-change-threshold of $\frac{1}{4}$, i.e. a superblock is not moved to a new group until its fullness is more than $\frac{1}{4}$ outside its current group. This overlap prevents superblocks from rapidly oscillating between fullness-groups. Further, the flat-sets used in the global and per-processor heaps have a maximum size set to 4093 superblocks, which allows each per-processor heap to contain up to 255MB memory per size-class.

Results

In the evaluation we compare our allocator with the standard "libc" allocator of the respective platform using the above standard benchmark applications. On the Sun platform, for which we had the original Hoard allocator available, we also compare with Hoard itself.

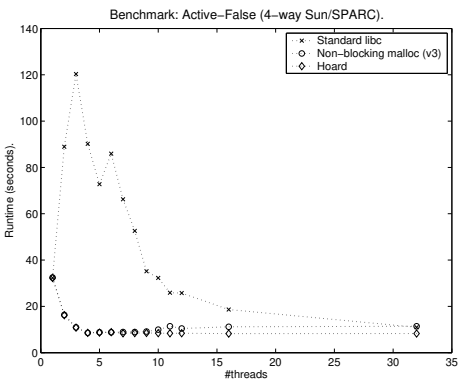
Another allocator which tries to reduce the use of locks is the LFMalloc [DG02]. It comes with an interesting approach for almost lock-free synchronization, called Multi-Processor Restartable Critical Sections (MP-RCS), a mechanism for restarting interrupted critical sections, which provide lock-free transactions on processor local data (i.e between threads running on the same processor). The implementation requires the possibility of efficient management of CPU-data and close interactions with the operating system's scheduler. To the best of our knowledge, this possibility is not directly available on all systems. LFMalloc is also based on the Hoard design, with the difference in that it limits each per-processor heap to at most one superblock of each size-class; when this block is full, further memory requests are redirected to the global heap where blocking synchronization is used and false-sharing is likely to occur. However, provided that it may become available for experimentation, it can be worthwhile to do a comparative study with that approach in the future.

The benchmarks are intended to test scalability, fragmentation and false-sharing. When performing these experiments we did not have as a goal to optimize the performance of the lock-free allocator, but rather to examine the benefits that can be brought by the lock-free design itself. There is certainly plenty of room for optimization of the implementation.

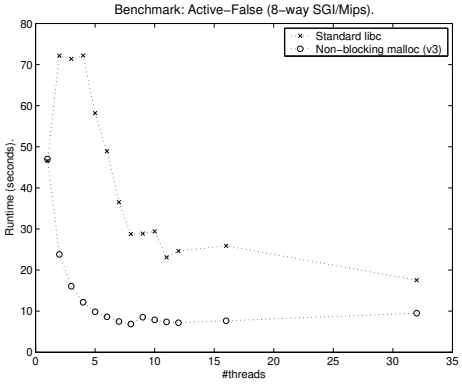
The results from the two false-sharing benchmarks, shown in Fig. 9 and Fig. 8, show that our memory allocator retains the nice properties of Hoard with respect to this issue. It also shows consistent behaviour as the number of processors and memory architecture changes. In the benchmarks the threads repeatedly allocate a 1-byte object, write to that object 1000 times and then deallocate that object. We measure the time it takes to complete a total of 1000000 such cycles.

The Larson benchmark used object-sizes between 5 and 500 bytes and run for 60 seconds (30 seconds on the Origin 3800).

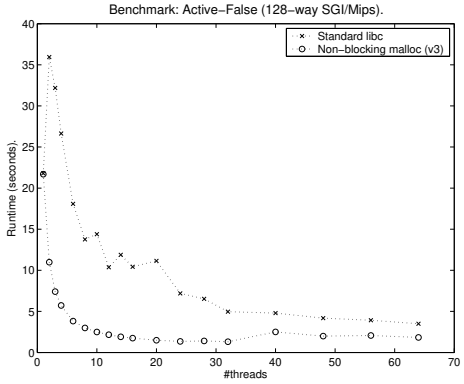
The throughput results from the Larson benchmark, shown in Fig. 10, show that our lock-free memory allocator has good scalability, not only in the case of full concurrency (where Hoard also shows extremely good scalability), but also when the number of threads increases beyond the number of processors. In that region, Hoard's performance quickly drops from its peak at full concurrency. We can actually observe more clearly the scalability properties of the lock-free allocator in the performance diagrams on the SGI Origin platforms. There is a linear-style of throughput increase when the number of processors increases (when studying the diagrams recall we have



(a) Sun SPARC 4 CPUs



(b) SGI MIPS 8 CPUs

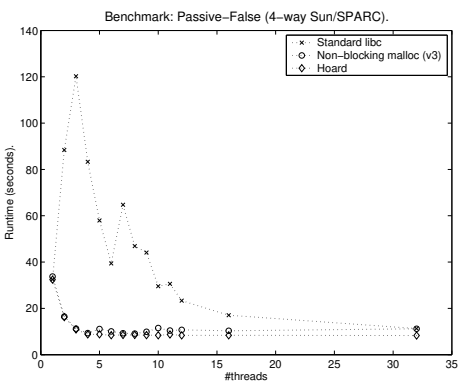


(c) SGI MIPS 32(/128) CPUs

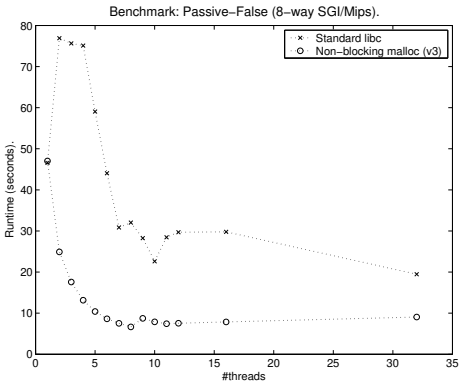
Figure 8: The Active-False benchmark.

8 and 32 processors available in the Origin 2000 and Origin 3800 platforms, respectively). Furthermore, when the load on each processor increases, the throughput of the lock-free allocator does not drop, but instead it may also increase, serving more requests. In terms of absolute throughput, Hoard is still superior to our lock-free allocator, at least on the SunFire platform where we had the possibility to compare them. This is not surprising, considering that it is very well designed and has been around enough time to be well tuned. An interesting conclusion is that the scalability of Hoard's architecture further enhanced by lock-free synchronization.

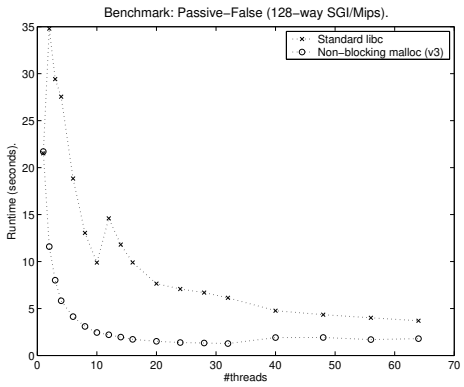
The results with respect to memory consumption, Fig. 11, show that for the Larson benchmark the memory usage (and thus fragmentation) of the



(a) Sun SPARC 4 CPUs



(b) SGI MIPS 8 CPUs



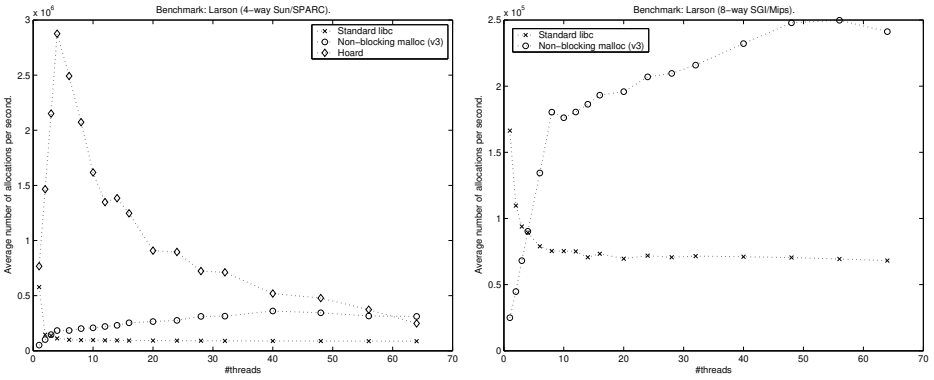
(c) SGI MIPS 32(/128) CPUs

Figure 9: The Passive-False benchmark.

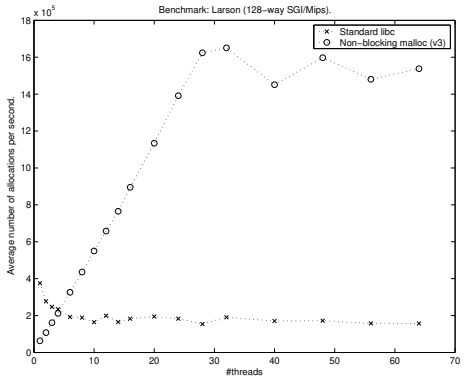
non-blocking allocator stays at the same level as for Hoard and that the use of per-processor heaps with threshold does not cause a much worse memory utilization than a sequential allocator.

5 Discussion and future work

The lock-free memory allocation method proposed in this paper confirms our expectation that fine-grain, lock-free synchronization is useful for enhancing the scalability under increasing load in the system. To the best of our knowledge, this is also the first lock-free allocator in the literature. We expect that this contribution may have an interesting impact in the domain of memory



(a) Sun SPARC 4 CPUs (b) SGI MIPS 8 CPUs

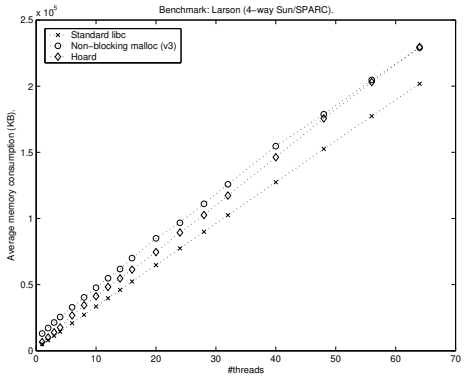


(c) SGI MIPS 32(/128) CPUs

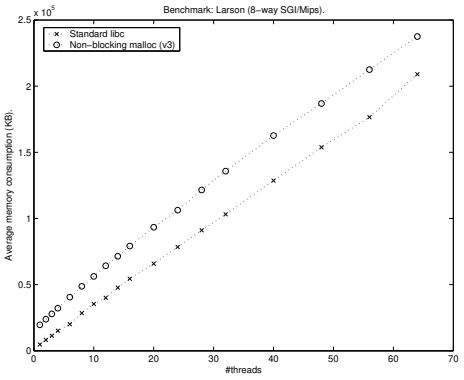
Figure 10: The Larson benchmark: Throughput.

allocators. Moreover, with further research in this direction, we expect that the results can be even more useful for multithreaded applications.

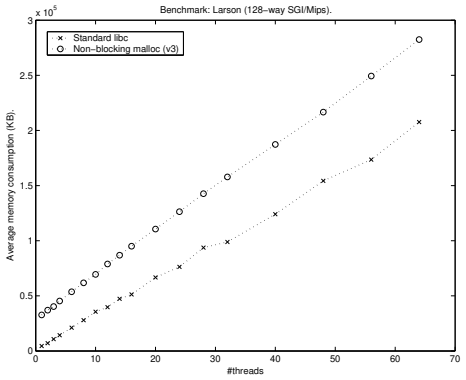
A couple of research issues whose investigation can help in that direction includes: (i) Self-tuning lock-free searching in lock-free flat sets for available space, to balance the corresponding trade-off between search time and memory overhead in the allocator: as a "malloc" operation progresses in its search, new "holes" may be created "behind" it, by concurrent "free" operations. It would be good to have a load-adaptive method to decide the right time-point (from the trade-off point of view) to stop searching and get a new empty superblock. (ii) Generalization of the method for "inter-object" operations, which is illustrated here in the move operation between sets. A general methodology in this direction would enable combinations of known



(a) Sun SPARC 4 CPUs



(b) SGI MIPS 8 CPUs



(c) SGI MIPS 32(/128) CPUs

Figure 11: The Larson benchmark: Average memory consumption.

lock-free implementation of data structures (e.g. other list-structures) into larger, "interconnected" data structures, to be integrated in systems such as the one studied here.

Acknowledgement

We would like to thank Håkan Sundell for informative and interesting discussions on non-blocking methods.

References

[Bar93] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, Velen, Germany, June 30–July 2, 1993. SIGACT and SIGARCH. Extended abstract.

[Ber02] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, August 1 2002.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing,. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, NM, USA, November 1994.

[BMBW00] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000.

[DG02] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 163–174, Berlin, June 2002. ACM Press.

[GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[Har01] Timothy L Harris. A pragmatic implementation of non-blocking linked lists. In *Distributed Computing, 15th International Conference*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, October 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[HPT02] Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[HT03] Phuong Hoai Ha and Philippas Tsigas. Reactive multi-word synchronization for multiprocessors. In *Proceedings of the 12th IEEE/ACM International Conference on Parall el Architectures and Compilation*

Techniques (PACT '03), pages 184–193. IEEE Computer Society Press, September 2003.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[LK98] Paul Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 176–185, Vancouver, October 1998. ACM Press.

[Mic02a] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel ALgorithms and Architectures (SPAA-02)*, pages 73–82, New York, August 10–13 2002. ACM Press.

[Mic02b] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM Press, 2002.

[Moi97] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, 1991.

[PT95] M. Papatriantafilou and P. Tsigas. Wait-free consensus in "in-phase" multiprocessor systems. In *Symposium on Parallel and Distributed Processing (SPDP '95)*, pages 312–319, Los Alamitos, Ca., USA, October 1995. IEEE Computer Society Press.

[PT97] M. Papatriantafilou and P. Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, September 1997.

[Rin99] Martin C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[SGI] SGI. The standard template library for C++. http://www.sgi.com/tech/stl/Allocators.html.

[ST02] Håkan Sundell and Philippas Tsigas. NOBLE: A non-blocking interprocess communication library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[ST03] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 03)*. IEEE press, 2003.

[TZ01a] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proceedings of the ACM SIGMETRICS 2001/Performance 2001*, pages 320–321. ACM press, June 2001.

[TZ01b] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143. ACM Press, 2001.

[TZ02] Philippas Tsigas and Yi Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67. ACM press, July 2002.

[Val94] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, October 1994.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 214–222, New York, August 1995. ACM.