Algorithmica (2010) 58: 304–338
DOI 10.1007/s00453-008-9268-x

NBMALLOC: Allocating Memory in a Lock-Free Manner

Anders Gidenstam · Marina Papatriantafilou · Philippas Tsigas

Received: 3 February 2006 / Accepted: 10 December 2008 / Published online: 22 January 2009

Abstract Efficient, scalable memory allocation for multithreaded applications on multiprocessors is a significant goal of recent research. In the distributed computing literature it has been emphasized that lock-based synchronization and concurrency-control may limit the parallelism in multiprocessor systems. Thus, system services that employ such methods can hinder reaching the full potential of these systems. A natural research question is the pertinence and the impact of lock-free concurrency control in key services for multiprocessors, such as in the memory allocation service, which is the theme of this work. We show the design and implementation of NBMALLOC, a lock-free memory allocator designed to enhance the parallelism in the system. The architecture of NBMALLOC is inspired by Hoard, a well-known concurrent memory allocator, with modular design that preserves scalability and helps avoiding false-sharing and heap-blowup. Within our effort to design appropriate lock-free algorithms for NBMALLOC, we propose and show a lock-free implementation of a new data structure, flat-set, supporting conventional "internal" set operations as well as "inter-object" operations, for moving items between flat-sets. The design of NBMALLOC also involved a series of other algorithmic problems, which are discussed in the paper. Further, we present the implementation of NBMALLOC and a

This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC). This paper is an extended version of [1].

A. Gidenstam (✉)
Algorithms and Complexity, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
e-mail: andersg@mpi-inf.mpg.de

M. Papatriantafilou · P. Tsigas
Computer Science and Engineering, Chalmers University of Technology, 412 96 Göteborg, Sweden

M. Papatriantafilou
e-mail: ptrianta@cs.chalmers.se

P. Tsigas
e-mail: tsigas@cs.chalmers.se

study of its behaviour in a set of multiprocessor systems. The results show that the good properties of Hoard w.r.t. false-sharing and heap-blowup are preserved.

Keywords Memory allocation · Lock-free synchronization · Non-blocking synchronization · Multithreaded applications

1 Introduction

In this paper we investigate the impact of lock-free synchronization in the memory-allocation system service on shared memory multiprocessor computers. We also propose a data structure useful for such a system, with new algorithms for lock-free and linearizable implementation of operations involving more than one instance of it; the latter is of independent interest in lock-free synchronization. *Lock-free algorithms* is an *efficient, fault-tolerant* alternative to using locks for synchronization, since they enable higher levels of parallelism and prevents slow operations from blocking the progress of other operations. A common consistency requirement from algorithms that access data objects concurrently is *linearizability*, which ensures that each operation on the data appears to take effect instantaneously during its actual duration and the effect of all operations are consistent with the object's sequential specification.

Dynamic memory management, used in most computer programs, comes in a variety of flavors, from the traditional manual general purpose allocate/free type memory allocators (e.g. as provided by the C runtime library) to advanced automatic garbage collectors. The focus here is on conventional, general-purpose memory allocators (such as the C runtime library "libc" malloc) where the application can request (allocate) arbitrarily-sized blocks of memory and free them in any order. A memory allocator is essentially an online algorithm that manages a pool of memory (heap), e.g. 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 and 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 memory blocks that are currently owned by the application.

We propose NBMALLOC, a new memory allocator based on lock-free, fine-grained synchronization, to enhance parallelism, fault-tolerance and *scalability*, the latter characterizing how well throughput (total and per thread) is increased/preserved as the number of threads (and/or processors) increase. The architecture of NBMALLOC is inspired by Hoard [2], due to its well-justified design decisions, which we outline below. Further, in the process of designing appropriate data structures and lock-free synchronization algorithms for NBMALLOC, we introduced a data structure, which we call *flat-set*. The flat-set is a container data structure with a subset operations on normal sets, as well as "inter-object" operations, for moving an item from one flat-set to another. The lock-free algorithms we introduce make use of standard hardware synchronization primitives provided by multiprocessor systems and provide linearizable implementation of the flat-set operations. Analytical proofs of lock-free algorithms are commonly a challenge; the correctness (linearizability) proofs of operations involving more than one instances of an object are an even bigger challenge

and as such, the linearizability proofs in this paper can be considered as a contribution of its own interest. The design of NBMALLOC also involved a set of other interesting algorithmic issues, which are discussed and analysed here, along with their solutions.

We implemented and studied NBMALLOC on a set of common multiprocessor platforms, namely an UMA Sun Fire 880 running Solaris 9, a ccNUMA Origin 2000 running IRIX 6.5 and a Intel Xeon PC running Linux 2.9.6. We studied NBMALLOC in connection with the standard "libc" allocator of each platform and with Hoard (on the systems where the Hoard allocator was available) using standard benchmark applications to test the efficiency, scalability, cache behaviour and memory consumption behaviour. The results show that NBMALLOC preserves the good properties of Hoard, while offering higher scalability potential (i.e. potential to increase/preserve throughput as the number of threads (and/or processors) increase), as justified by its lock-free nature.

Related Work

The Hoard [3] concurrent memory allocator is designed to meet the above goals. The allocation is done on the basis of per-processor heaps, which avoids false-sharing and reduces the synchronization overhead in many cases, improving both performance and scalability. Memory requests are mapped to the closest matching size in a fixed set of size-classes, which bounds internal fragmentation. The heaps are sets of superblocks, with each superblock handling blocks of one size class, which helps in coping with external fragmentation. To avoid heap-blowup, freed blocks are returned to the heap they were allocated from and empty superblocks may be reused in other heaps.

Concurrently with and independently from our work on NBMALLOC, Michael presented a lock-free allocator [4] which, like our contribution, is loosely based on the Hoard architecture and uses *Compare-And-Swap*. Despite both having started from the Hoard architecture, we have used two different approaches to achieve lock-freeness. Another allocator which tries to reduce the use of locks is LFMalloc [5]. To be able to relate these contributions with the one presented here, some more detail is needed and for this reason we describe this relation in Sect. 6.

Also of relevance to lock-free memory allocators are algorithms for lock-free memory management and reclamation. Such schemes need to be used in lock-free dynamic data-structures to provide safe reclamation of dynamically allocated shared memory blocks, i.e. to make sure that a deleted memory block is not reused until it is certain that it cannot be accessed by any concurrent or future operation. Some schemes focus on the safety of thread-local references to objects only, such as the efficient hazard pointer algorithm by Michael [6, 7] and the algorithm by Herlihy et al. [8], but these cannot be used with all data-structures. Other schemes based on reference counting can guarantee the safety of local as well as global references to objects. In this category we have the work of Valois et al. [9, 10], Detlefs et al. [11], Herlihy et al. [12] and Gidenstam et al. [13].

Recently, Streamflow, another memory allocator which reduces use of locks and contention on shared memory locations, but is not entirely lock-free appeared in [14].

Earlier related work in similar direction is the work on non-blocking operating systems by Massalin and Pu [15, 16] and Greenwald and Cheriton [17, 18]. The respective algorithms, however, made extensive use of the *2-Word-Compare-And-Swap* (*2CAS*) primitive, which can update two arbitrary memory locations in one atomic step, while this primitive is not available in current systems and is expensive to simulate in software.

Document Structure

Section 2 provides technical background on concurrent memory allocation and lock- and wait-free synchronization (throughout the paper we use the terms non-blocking and lock-free interchangeably). Section 3 describes NBMALLOC, including the lock-free flat-sets data structure designed for this purpose. In Sect. 4 we show that the flat-sets data structure implementation is linearizable and lock-free. Section 5 describes details on the implementation done for the experimental study of the system, including the platforms on which it was implemented and benchmark information. The results of our experimental study are also presented here. Sect. 6 relates NBMALLOC to other related work in detail. We conclude in Sect. 7 with a discussion of the achieved results and future work.

2 Background and Problem Description

2.1 Concurrent Memory Allocators

An important optimization goal of a good allocator is to minimize *fragmentation*, i.e. minimize the amount of free memory that cannot be used (allocated) by the application. Fragmentation is classified as either internal or external. *Internal fragmentation* is free memory wasted when the allocator gives the application a larger memory block than the application requested. *External fragmentation* is free memory that have been split into non-contiguous blocks too small to be used to satisfy the requests from the application.

Moreover, multi-threaded programs add more 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 issues which have significant impact on application performance when the application is run on a multiprocessor [2]:

(i) *False-sharing*, i.e. 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. False-sharing can never be avoided completely since application threads may pass allocated memory between themselves but a memory allocator should avoid to *actively induce* false-sharing by satisfying memory requests from different processors with memory from the same cache-line.

(ii) *Heap-blowup*, i.e. an overconsumption of memory that may occur if the memory allocator fails to make memory deallocated by threads running on one processor

Table 1 Properties of different memory allocators. False-sharing refers to actively induced false-sharing

	Type	False-sharing	Heap-blowup	Scalable	Lock-free
The standard memory allocators on Solaris, Irix and Windows 2000	Single heap	Yes	No	No	No
STL allocator [19]	Pure private heaps	No	Unbounded	Yes	No
Cilk [20]	Pure private heaps	No	Unbounded	Yes	No
MTmalloc (Solaris)	Private heaps w. ownership	No	Linear	Yes	No
Ptmalloc (glibc) [21]	Private heaps w. ownership	No	Linear	Yes	No
Hoard [3]	Private heaps w. thresholds	No	Constant	Yes	No
LFMalloc [5]	Private heaps w. thresholds	?	Constant	Yes	Almost
M. Michael's allocator [4]	Private heaps w. ownership	?	Constant	Yes	Yes
StreamFlow [14]	Private heaps w. thresholds	No	Constant	Yes	Mostly
NBMALLOC	Private heaps w. thresholds	No	Constant	Yes	Yes

available[1] to threads running on other processors. The worst case heap-blowup of a memory allocator is often classified w.r.t. the number of per-processor heaps (e.g. as constant, linear or unbounded). A typical source of heap blowup is an application that has producer and consumer threads, where the producers allocate memory and pass it to the consumers which in turn free the memory. If the memory blocks freed by the consumers are not made available to the producers an unbounded heap blowup could occur.

(iii) Ensure *efficiency* and *scalability*. For a memory allocator to be scalable, its performance has to scale well with the number of processors, threads 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.

Table 1 gives a brief overview of concurrent memory allocator designs and their respective properties based on the taxonomy presented in [2].

Separate handling of thread-local allocations In many applications most of the dynamic memory requests concern memory that will only be used by one thread. Therefore, it might be advantageous to distinguish between these thread-local allocations and allocations of memory that is to be shared between threads. In particular, the thread-local memory allocator might not need any synchronization. The crucial distinction between thread-local allocations and shared memory allocations could be made either explicitly by the programmer (using an extension of the traditional malloc interface) or automatically by compile time analysis, as e.g. in [22].

[1] E.g. as the result of a coarse policy for avoiding false-sharing.

2.2 Non-blocking Synchronization

The most commonly required consistency guarantee for shared data objects is *linearizability* [23]. A shared object (its implementation) is *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 within its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification.

Compared to the traditional solution for maintaining the consistency of a shared data object (i.e. for ensuring linearizability) 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 [17, 24–26]. 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, 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 (e.g. *CAS*; 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, so 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 [25].

Some of the aforementioned universal primitives are available in several common 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 and Intel x86 processors. Another useful primitive is *Fetch-And-Add* (also denoted *FAA*), described in Fig. 1. *FAA* can be simulated in software using *CAS* when it is not available in hardware.

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 [27]. 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

Fig. 1 Compare-And-Swap (denoted *CAS*) and Fetch-And-Add (denoted *FAA*)

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

atomic CAS(mem : pointer to integer;
           new, old : integer) return boolean
begin
   tmp := *mem;
   if tmp == old then
      *mem := new; /* CAS succeeded */
   return tmp == old;
end CAS;

atomic FAA(mem : pointer to integer;
           increment : integer) return integer
begin
   tmp := *mem;
   *mem := tmp + increment;
   return tmp;
end FAA;
```

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 was proposed and used in a lock-free queue implementation in [28]. An appropriate memory-reclamation mechanism, such as [6], can also solve the problem.

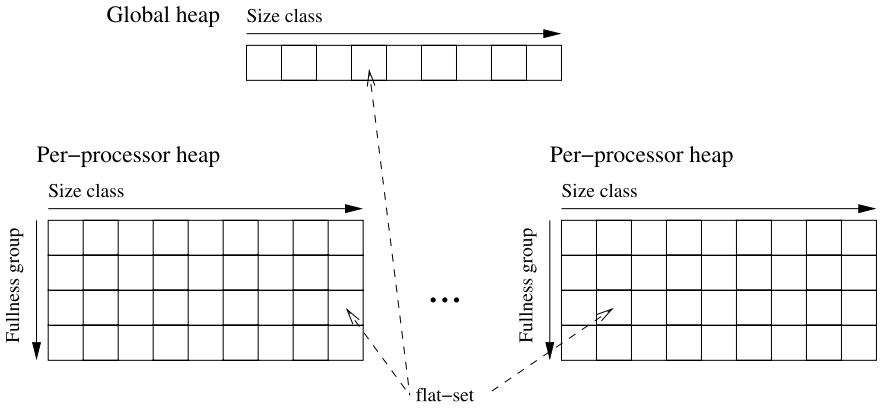
There is a plethora of research articles that focus on wait-free and lock-free synchronization (for a few examples cf. [6, 24, 25, 27, 29–35] and references therein).

3 NBMALLOC

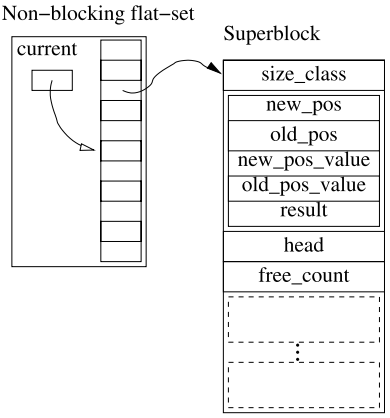
3.1 Architecture

The architecture of NBMALLOC is inspired by Hoard [3], 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. 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



(a) The organization of the global and per-processor heaps



(b) A non-blocking flat-set and a superblock

Fig. 2 The architecture of NBMALLOC

in a per-processor heap becomes almost empty (i.e. few 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 groups based on *fullness* (cf. Fig. 2(a)). As the fullness of a particular superblock changes, it is moved between the groups. A *malloc* call first searches for a superblock with a free block among the superblocks in the "almost full" fullness-group of the requested size-class in the appropriate per-processor heap. 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 reduces external fragmentation. When freed (by a call to *free*) an allocated block is returned to the superblock it was allocated from and, if the new fullness requires so, the superblock is moved to another fullness-group.

3.2 Managing Superblocks: The Flat-Set Data Structure

Since the number of superblocks in each fullness-group varies over time, a suitable collection-type data structure is needed to implement a fullness-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. [27, 29, 36], we cannot apply those here, because not only do we want the operations on the list to be lock-free, but we also need to be able to move a superblock from one set to another without making it inaccessible to other threads during the move. To address this, we propose a new data structure we call a *flat-set*, supporting conventional "internal" set operations (*Get_Any* and *Insert* item) as well as "inter-object" operations, for moving an item from one flat-set to another.

To have an entirely lock-free implementation including "inter"-flat-set operations it is crucial to be able to move superblocks from one set to another in a lock-free fashion. The requirement that makes this difficult is that the operations should be linearizable, and in particular we want the following properties to be satisfied:

1. a superblock should be reachable for other threads even while it is being moved between flat-sets, i.e. a non-atomic *first-remove-then-insert* sequence is not acceptable;
2. *Get_Any* and *Insert* should behave as if a superblock is referenced by exactly one shared location at any time.

Below we present the operations of the lock-free and linearizable flat-set data structure and the algorithm, *Move*, which implements the "inter-object" operation for moving a reference to a superblock from one shared variable (pointer) to another, while satisfying the above requirements.

3.3 Operations on Bounded Non-blocking Flat-Sets

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

1. *Get_Any*, which returns a reference to an item currently in the flat-set (which is unspecified); and
2. *Insert*, which inserts an item into a flat-set.

Note that an important property to satisfy is that an item only resides inside one flat-set at the time; when an item is inserted into a flat-set it is also removed from its old location.

The flat-set data structure implementation consists of an array of M shared locations set.set[i], each capable of holding a reference to an item, and a shared index variable set.current. The flat-set data structure and operations are shown in Algorithms 1 and 2 and are explained in the following paragraphs. The flat-set operations use two internal sub-operations *Move* and *SB_Deref* to move and read item references, respectively. These operations are further described in Sect. 3.4.

To speed up flat-set operations there is an index variable set.current that is used as the starting point for searches, both for items and for free slots.

Algorithm 1 The flat-set and superblock data structures in NBMALLOC

```
type flat-set_t is record
    size : constant integer
    current : flat-set_info
    set[size] : array of superblock_ref_t

type flat-set_info is atomic record
    /* Fits in one 32 bit machine word. */
    index : integer_16
    version : integer_16

type superblock_ref_t is atomic record
    /* Fits in one 32 bit machine word. */
    ptr : integer_16
    version : integer_16

/* superblock_ref_t utility functions. */
function pointer(ref : superblock_ref_t) return pointer to superblock_t
function version(ref : superblock_ref_t) return integer_16
function make_sb_ref(sb : pointer to superblock_t, op_id : integer_16)
    return superblock_ref_t

type superblock_t is record
    mv_info : move_info_t
    freelist_head : block_ref_t
    free_block_cnt : integer
    ...

type move_info_t is record
    new_pos : pointer to superblock_ref_t
    cur_pos : pointer to superblock_ref_t
    new_pos_value : superblock_ref_t
    cur_pos_value : superblock_ref_t
    result : enum {SB_MOVED_OK, SB_NOOP, NA} /* The initial value is always NA. */

type block_ref_t is atomic record
    /* Fits in one machine word. */
    offset : integer_16
    version : integer_16

function pointer(ref : block_ref_t) return pointer to block_t
function version(ref : block_ref_t) return integer_16
function make_ref(sb : pointer to block_t, version : integer_16) return block_ref_t

type block_t is record
    /* Block header. */
    owner : pointer to superblock_t
    next : pointer to block_t
```

3.4 How to Move a Shared Reference: Moving Items between Flat-Sets

The two operations *SB_Deref* and *Move* provide lock-free and linearizable reading and moving of superblock references. *SB_Deref* reads a superblock reference from a shared location and *Move* moves a superblock reference sb from a shared location

Algorithm 2 The flat-set operations *Get_Any* and *Insert*

```
function Get_Any(set : in out flat-set_t, sb : out superblock_ref_t,
      loc : out pointer to superblock_ref_t)
return status_t
        i, j : integer;   old_current : flat-set_info_t;
begin
G1   loop
G2      old_current := set.current;
G3      i := old_current.index;
G4      for j := 1 .. set.size do
G5         sb := SB_Deref(&set.set[i]);
G6         if pointer(sb) ≠ null then
G7            loc := &set.set[i];
G8            CAS(&set.current, old_current,(i,));
G9            return SUCCESS;
G10        if i == 0 then i := set.size - 1; else i--;
G11     if set.set not changed since prev. iter. then
G12        return EMPTY;
end Get_Any;

function Insert(set : in out flat-set_t, sb : in superblock_ref_t,
      loc : in out pointer to superblock_ref_t)
return status
        i, j : integer;   old_current : flat-set_info_t;
begin
I1   loop
I2      old_current := set.current;
I3      i := (old_current.index + 1) mod set.size;
I4      for j := 1 .. set.size do
I5         while SB_Deref(&set.set[i]) == null do
I6            if Move(sb, loc, &set.set[i]) == SB_MOVED_OK then
I7               CAS(&set.current, old_current, (i,));
I8               loc := &set.set[i];
I9               return SUCCESS;
               else  /* Move returned SB_NOOP */
I10              if *loc ≠ sb then
I11                 return MOVED_AWAY;
I12         i := (i + 1) mod set.size;
I13     if set.set not changed since prev. iter. then
I14        return FULL; /* The flat-set is full. */
end Insert;
```

from to another shared location to. The target location (i.e. to) is known e.g. via the Insert operation.

Move works by extending the helping mechanism, into constructing a "bridge" across two locations (which could belong to different objects). In particular, the algorithm requires the superblock to contain an auxiliary variable mv_info —of type move_info_t, which contains the fields new_pos, cur_pos new_pos_value, cur_pos_value and result. Further, all superblock references need to have a version field (cf. Algorithm 1). The move_info_t structure may also contain additional information in case of application needs. This is the case in NBMALLOC where two additional fields, cur_owner and new_owner, are used to keep track of which flat-set the superblock is currently located in.

Algorithm 3 The superblock *Move* operation

```
function Move(sb : in superblock_ref_t,
      from : in pointer to superblock_ref_t,
      to : in pointer to superblock_ref_t)
return status
        new_op, cur_op : pointer to move_info_t;
        cur_from, cur_to : superblock_ref_t;
begin
     /* Step 1: Initiate move. */
M1   loop
M2      cur_op := HP_Deref(&sb.mv_info);
M3      cur_from := *from;
M4      if cur_from ≠ sb then
M5         HP_Release(cur_op); return SB_NOOP;
M6      if cur_op.new_pos == null then /* No current operation. */
M7         cur_to := SB_Deref(to);
M8         if pointer(cur_to) ≠ null then
M9            HP_Release(cur_op);
M10           return SB_NOOP;
M11        new_op := HP_New((to, from, cur_to, cur_from, NA));
M12        if CAS(&sb.mv_info, cur_op, new_op) then
M13           HP_Delete(cur_op);
M14           return Move_Help(sb, new_op);
           else
M15           HP_Delete(new_op); HP_Release(cur_op);
        else
M16        Move_Help(cur_op.old_pos_value, cur_op);
end Move;
```

A *Move* operation may *succeed* by returning SB_MOVED_OK or *fail* (abort) by returning SB_NOOP (if the block was moved away by another overlapping *Move* or the to location was occupied).

To ensure the lock-free property and linearizability, the move operation is divided into a number of atomic suboperations. The first step (1), to *register* the *Move* operation, is done in the *Move*(*sb, from, to*) function in Algorithm 3, while the other three steps, (2) to update location to, (3) to clear location from and (4) to update sb.mv_info, are done in the *Move_Help* function in Algorithm 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. The helping procedure is performed by *Move_Help* and is identical to steps 2–4 of the move operation as described below. Since all information required to finish an ongoing operation is stored in the descriptor mv_info, any thread that encounters the superblock can continue and finish any ongoing operation.

1. *Initiate Move*: First a *Move*(sb, from, to) collects the current state of from, and sb.mv_info (lines M2–3). If sb is no longer present in from, SB_NOOP is returned (lines M4–5). Line M6 checks whether there is an ongoing operation. If so, that operation is helped to completion via *Move_Help* (line M16). Otherwise the to location is read using *SB_Deref* (line M7). If the to location is occupied SB_NOOP is returned (line M10). Otherwise this *Move* will try to register itself in sb by updating sb.mv_info to point to new a descriptor containing all information needed to finish this operation using CAS (line M12). Note in particular that the value

Algorithm 4 The superblock *Move_Help* operation

```
function Move_Help(sb : in superblock_ref_t, op : pointer to move_info_t)
return status
        old, new, res : superblock_ref_t;
        new_op : pointer to move_info_t;
begin
    /* Step 2: Update "TO". */
H1  res := CAS(op.new_pos, op.new_pos_value,
                make_sb_ref(pointer(sb), version(op.new_pos_value) + 1));
H2  if res ≠ op.new_pos_value then /* I.e.CAS failed. */
H3      if res ≠ make_sb_ref(pointer(sb), version(op.new_pos_value) + 1) then
            /* To seems to be occupied. */
H4          if sb.mv_info == op then /* Finish this op. */
H5              CAS(&op.result, NA, SB_NOOP);
H6              new_op := HP_New((null, op.cur_pos, null, null, NA));
H7              if CAS(&sb.mv_info, op, new_op) then
H8                  HP_Release(new_op); HP_Delete(op);
                else
H9                  HP_Delete(new_op); HP_Release(op);
H10             return SB_NOOP;
H11         HP_Release(op); return op.result;
H12 CAS(&op.result, NA, SB_MOVED_OK);
    /* Step 3: Clear "FROM". */
H13 CAS(op.cur_pos, op.cur_pos_value, make_sb_ref(null, version(op.cur_pos_value) + 1));
    /* Step 4: Remove operation information.*/
H14 new_op := HP_New((null, op.new_pos, null, null, NA));
H15 if CAS(&sb.mv_info, op, new_op) then
H16     HP_Release(new_op); HP_Delete(op);
    else
H17     HP_Delete(new_op); HP_Release(op);
H18 return SB_MOVED_OK;
end Move_Help;
```

of the result field in the new descriptor is NA, indicating that the outcome of this operation is not yet decided. If the CAS succeeds this operation proceeds to Step 2 which is in *Move_Help*. Otherwise this operation retries from start as there might be another ongoing operation.

2. *Update to location*: The second step of a *Move* operation (and the first in the *Move_Help* operation) is to update the to location. This is done with the CAS at line H1. If the CAS succeeds the result field in the descriptor is set to SB_MOVED_OK (line H12) and the operation proceeds to Step 3. If the CAS fails, this could either mean that to has become occupied and this operation fails or that a(nother) helper has already performed the line H1 CAS for this operation. These two cases are distinguished below. First, if to contains the value this operation would have written there, the result field in the descriptor is set to SB_MOVED_OK (line H12) and the operation proceeds to Step 3. Otherwise, the value of sb.mv_info is checked to see if this operation has been completed by a(nother) helper (line H4). If it has been completed the return code stored in the descriptor is returned. Otherwise, it is certain that the operation will return SB_NOOP since the value of to at line H1 was neither the old value expected nor the new this operation would write, yet the operation is still not completed (which it had to be if sb had been successfully moved to to and then moved elsewhere by

Algorithm 5 The superblock *SB_Deref* operation

```
function SB_Deref(loc : in pointer to superblock_ref_t)
return superblock_ref_t
        sb : superblock_ref_t; op : pointer to move_info_t;
begin
      loop
SD1     sb := *loc;
SD2     if pointer(sb) == null then return sb;
SD3     op := HP_Deref(&sb.mv_info);
SD4     if op.new_pos == null and *loc == sb then
SD5       HP_Release(op); return sb;
SD6     if op.new_pos ≠ null then
          /* Help ongoing move operation to finish. */
SD7       Move_Help(sb, op);
        else
SD8       HP_Release(op);
end SB_Deref;
```

a subsequent *Move*.) So, SB_NOOP is written to the descriptor (line H5) and then the descriptor in sb is replaced by an idle one (line H7).

3. *Clear from location*: The from location is set to null using *CAS* (line H13). The *CAS* succeeds if and only if from still contains the expected superblock reference (including the right version). If it does not then someone else has already helped this move to complete this step.
4. *Operation finished*: The last step is to remove the descriptor from the superblock. The value of sb.mv_info is updated to reference a new idle descriptor using CAS (line H15). If the CAS fails a(nother) helper has already performed this step. The *Move* operation is now finished and returns SB_MOVED_OK (line H18).

In the presentation of the algorithm and in the pseudo-code in Algorithms 3 and 4 we use the atomic primitive *CAS* to update shared variables that fit in a single memory word, but other strong atomic synchronization primitives, such as *Load-Linked*/*Store-Conditional* could be used as well (cf. e.g. [37]).

The auxiliary mv_info variable in a superblock needs to be larger than the one or (for some platforms) two words of adjacent data the hardware *CAS* primitive can handle. To handle that we use an extra layer of indirection and a lock-free method, called *hazard pointers* [7] that can be implemented efficiently using the common single-word *CAS*, that provides safe handling of pointers to shared data blocks. The method provides four lock-free and linearizable operations: *HP_Deref* to read a shared reference to a data block and prevent that block from being reclaimed; *HP_Release* to release a previously dereferenced block for (potential) reclamation; *HP_New* to allocate a new data block; and *HP_Delete* to schedule a data block for reclamation once all claims to it via *HP_Deref* have been released. It is possible to safely use *CAS* to update a shared reference to a data block.

Dereferencing a superblock reference The operation *SB_Deref*, in Algorithm 5, is used to read a superblock reference variable (i.e. a value of type superblock_ref_t) in shared memory in a way that is linearizable with respect to concurrent *Move* and *SB_Deref* operations. *SB_Deref* achieves this by helping any ongoing *Move* operation concerning the referenced superblock to finish before it returns the reference.

Algorithm 6 The superblock operations *Get_block* and *Put_Block*

```
function Get_Block(sb : in pointer to superblock_t) return pointer to block_t
        nb : block_ref_t;
begin
GB1 loop
GB2     nb := sb.freelist_head;
GB3     if pointer(nb) ≠ null then
GB4         if CAS(&sb.freelist_head, nb, make_ref(nb.next, version(nb) + 1)) then
GB5             FAA(&sb.free_block_cnt, -1);
GB6             return pointer(nb);
        else
GB7         return null;
end Get_Block;

procedure Put_Block(sb : in pointer to superblock_t, bl : in pointer to block_t)
        oh : block_ref_t;
begin
PB1 loop
PB2     oh := sb.freelist_head;
PB3     bl.next := pointer(oh);
PB4     if CAS(&sb.freelist_head, oh, make_ref(bl, version(oh) + 1)) then
PB5         FAA(&sb.free_block_cnt, 1);
PB6         return;
end Put_Block;
```

3.5 Managing Blocks within a Superblock

The allocatable memory blocks within each superblock are kept in a lock-free IBM free-list [38]. The IBM free-list is essentially a lock-free stack implemented as a single-linked list where the push and pop operations are done by a *CAS* operation on the head-pointer. To avoid ABA-problems the head-pointer contains a version field. 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 Algorithm 6. The free blocks counter, sb.free_block_cnt, is used to estimate the fullness of a superblock.

3.6 Interacting with NBMALLOC

The user application interacts with the memory allocator via the two operations: *malloc*, to make a memory request, and *free*, to release previously allocated memory.

These two operations together with the main global data structures of the memory allocator are shown in Algorithm 7. The Global_Heap structure consists of an array of flat-sets, one for each size-class, which contains all empty or nearly empty superblocks in the system (cf. also Fig. 2(a)). A per-processor heap is a two dimensional array of flat-sets indexed by size-class and superblock fullness group.

A *malloc* call for a memory block of a certain size-class sc will first search the flat-sets for the required size-class in the appropriate per-processor heap for a superblock with a free block. The search begins in the flat-set for the "almost full" fullness-

Algorithm 7 The *malloc* and *free* operations

```
Global_Heap : global shared array [SIZE_CLASSES] of flat-set_t;
type per-processor_heap_t is array
     [SIZE_CLASSES] [MIN_FULLNESS .. MAX_FULLNESS] of flat-set_t;

function malloc(sc : in size_class) return pointer to block_t
        heap : pointer to per-processor_heap_t := select_heap(thread_id);
        sb : superblock_ref_t;  sbr : pointer to superblock_ref_t;
        bl : block_ref_t:= null;
begin
A1   for fg := MAX_FULLNESS - 1 .. MIN_FULLNESS do
A2      while Get_Any(heap[sc][fg], sb, sbr) == SUCCESS do
A3         bl := Get_Block(pointer(sb));
A4         if bl ≠ null then
A5            exit for loop;
           else
              /* Move the full superblock out of the way. */
A6            Insert(heap[sc][MAX_FULLNESS], sb, sbr);
A7   while bl == null loop
        /* Move a superblock from the global heap to the per-processor heap. */
A8      if Get_Any(Global_Heap[sc], sb, sbr) then
A9         if SB_MOVED_OK == Insert(heap[sc][MIN_FULLNESS], sb, sbr) then
A10           bl := Get_Block(pointer(sb));
        else
A11        return null; /* Out of memory. */
A12 if fullness(sb) ≠ fg then  /* Move the superblock to the right fullness group. */
A13    Insert(heap[sc][fullness(sb)], sb, sbr);
A14 return bl;
end malloc;

procedure free(bl : in pointer to block_t)
        sbp : pointer to superblock_t := bl.owner;
        heap : pointer to per-processor_heap_t := sbp.owner;
        newfg, oldfg : fullness_group := fullness(sbp);
        status : pointer to move_info_t;  sb : superblock_ref_t;  sbr : pointer to superblock_ref_t;
begin
F1   Put_Block(sbp, bl);
F2   newfg := fullness(sbp);
F3   if newfg ≠ oldfg then
F4      status := HP_Deref(&sbp.mv_info);
F5      sbr := status.cur_pos;
F6      sb := *sbr;
F7      HP_Release(status);
F8      if pointer(sb) == sbp then
F9         if newfg == empty or almost empty then
F10           Insert(Global_Heap[sc], sb, sbr);
           else
F11           Insert(heap[sc][newfg], sb, sbr);
end free;
```

group. If no suitable superblock is found there, the search proceeds to search in the lower fullness-groups (lines A1 to A6 in Algorithm 7). Searching in the almost full superblocks set first is a strategy to reduce external fragmentation, since it allows less full superblocks in the per-processor heap to get fewer allocation requests and gradually become empty enough to be moved to the global heap. If no superblock

with a free block is found in the per-processor heap, *malloc* will attempt to get a new superblock from the Global_Heap (line A8) and move it to the per-processor heap (line A9). If such a superblock is found then *malloc* tries to allocate a block from it. If no such superblock is found the system is out of memory for this size-class and *malloc* will return null.

The operation *free* is used by the application to return a no longer needed block of memory to the memory allocator. The operation uses the owner field of the block header to find the superblock it belongs to. When the block is returned to the superblock (line F1), the superblock might need to be moved to a different fullness-group or, if it has become almost empty, to the global heap. To be able to move the superblock, its current location is needed. The location is read from the mv_info field in the superblock (line F4) and the superblock is then moved at line F10 or F11. The test at line F8 makes sure that it is the right superblock that is going to be moved—the superblock in question could have been removed from sbr between line F4 and F6 by some concurrent operation.

4 Correctness of the Non-blocking Flat-Set Algorithm

The main and central algorithmic construction in NBMALLOC is the flat-sets used to manage the superblocks. At the heart of the flat-set operations lie the two suboperations *Move* and *SB_Deref* which allow superblock references to be moved between shared locations in a lock-free and linearizable manner.

The SB_Deref *and* Move *operations* Let l_i be a set of locations and b_j a set of items/superblocks for i:s and j:s from any two sets. In the following we will use l_a and l_b to refer to two arbitrary (distinct) locations and x to refer to an arbitrary item/superblock. The operations *SB_Deref* and *Move* then have the following sequential semantics:

- a superblock x resides at (is referenced by) exactly one location at any time.
- *SB_Deref*(l_a) returns a reference r to the superblock presently at location l_a or null if l_a is empty.
- *Move*(r, l_a, l_b) has the following outcomes:
  SB_MOVED_OK when l_a contains a reference r to x and l_b contains null. The superblock reference r is moved from the location l_a to the location l_b.
  SB_NOOP when l_a does not contain a reference r to x or l_b does not contain null.

As a *Move* operation may be completed by a helper it is not obvious that the actual outcome of the operation is returned to the caller. The following lemma proves that the actual result of the *Move* is returned.

Lemma 1 (Move return code) *A* Move *operation always returns the actual outcome of the operation to its caller.*

Proof It is easy to see that the actual outcome is reported when the operation result is SB_NOOP due to the superblock no longer being present in from (line M5), as there are no helping in this case. The same holds for SB_NOOP reported at line M10.

Once an operation has been registered (line M12), helping may occur. There are two cases to consider:

(i) Assume the operation was successful (i.e. SB_MOVED_OK should have been returned) but SB_NOOP (or NA) is returned to the caller. Since the operation was successful the CAS at line H1 succeeded for one helper, denoted *Move_Help*$_H$. In the initiator's call to *Move_Help* the CAS at H1 did not succeed (else SB_MOVED_OK would have been returned). Now, if the initiator saw the expected new value of to at line H1 (tested at H3) SB_MOVED_OK would be returned, so to must hold some other value. Next, the initiator would check if sb.mv_info still contains the current operation (line H4). This cannot be the case, since according to our assumption the helper has updated to and the only way to remove the reference to x from to is with another subsequent *Move* operation. Consequently, the initiator will return the return-code stored in the operation descriptor (line H11) and since the operation has been completed (line H15) by the helper, the stored return code is SB_MOVED_OK.

(ii) The operation resulted in SB_NOOP but SB_MOVED_OK (or NA) is returned to the caller. Since the result is SB_NOOP the CAS at line H1 does not succeed for the initiator or any helper. Consider the first helper to reach line H5. It will reach this line since the operation is still unfinished and the CAS will succeed since the initial value of op.result is NA. The initiator will either take the same path (if it sees that the operation is still unfinished at line H4) and return SB_NOOP at H10 or find the operation finished and return op.result, which at this point cannot contain anything else than SB_NOOP as some helper must have passed line H5 before completing the operation at line H10. □

Proposition 1 *The linearization point of the operation SB_Deref is the memory read at line SD*1 *iff null is returned and the HP_Deref at line SD*3 *otherwise.*

Proposition 2 *The linearization point of the operation* Move$_1(r, l_a, l_b)$ *is*

- *the first CAS at* $H1_1$ *executed by the initiator or any helper iff the result is SB_MOVED_OK*;
- *the memory read at* $M3_1$ *iff the result is SB_NOOP and* $*\text{from} \neq r$ *at line* $M3_1$;
- *the SB_Deref at* $M7_1$ *iff the result is SB_NOOP and* SB_Deref$(l_b) \neq$ *null at line* $M7_1$; *and*
- *the first CAS at* $H1_1$ *executed by the initiator or any helper otherwise.*

To facilitate the presentation of the linearizability proofs we first outline the key events for an operation *Move*(r, l_a, l_b) to return a particular result here. These lists of events follow directly from the pseudo code.

SB_MOVED_OK

M2: *HP_Deref*(&sb.mv_info) gives cur_op with cur_op.new_pos equal to null;
M3: The value read from the from location equals r;
M7: The result of *SB_Deref(to)* is a null reference;
M12: The CAS succeeds, i.e. the old value is cur_op;

H1: The CAS succeeds, i.e. to still hold the value read at M7;
H12: The result code is set to SB_MOVED_OK;
H13: from is set to null;
H15: The operation descriptor is removed from sb.mv_info.

SB_NOOP
M3: The value read from the *from location is different from r;
or
M2: *HP_Deref*(&sb.mv_info) gives cur_op with cur_op.new_pos equal to null;
M3: The value read from the from location equals r;
M7: The result of *SB_Deref(to)* is not a null reference;
or
M2: *HP_Deref*(&sb.mv_info) gives cur_op with cur_op.new_pos equal to null;
M3: The value read from the from location equals r;
M7: The result of *SB_Deref(to)* is a null reference;
M12: The CAS succeeds, i.e. the old value is cur_op;
H1: The CAS fails, i.e. to does not hold the value read at M7;
H5: The result code is set to SB_NOOP;
H7: The operation descriptor is removed from sb.mv_info.

Lemma 2 (Linearizability I) *The operation SB_Deref is linearizable with respect to other SB_Deref and Move operations with linearization points according to Proposition 1.*

Proof According to the abstract semantics *SB_Deref* returns null or a reference to a superblock. First, consider the case when *SB_Deref* returns null. This is done at line SD2 if the current value of loc as read at line SD1 contains a null reference. In this case line SD1 is the linearization point of *SB_Deref*.

Now consider the case when *SB_Deref* returns a reference to a superblock x. This is done at line SD5 if and only if the following conditions hold: (i) loc was referencing x at line SD1; (ii) the move info read at line SD3 shows that x is not involved in any ongoing operation; and (iii) loc is still referencing x at line SD4 (with the same version number). Condition (i) and (iii) together implies that loc was referencing x at line SD3 when the move info was read. Therefore, the linearization point in this case is at line SD3.

Now consider a sequence of *SB_Deref* operations on two locations, l_a (initially occupied by a superblock x) and l_b (initially empty), concurrent with a successful *Move*(r, l_a, l_b). There are two nontrivial forbidden result sequences: (i) *SB_Deref*$(l_a) = \text{null}$; *SB_Deref*$(l_b) = \text{null}$; and (ii) *SB_Deref*$(l_b) = b$; *SB_Deref*$(l_a) = b$.

(i) Assume towards a contradiction that the sequence *SB_Deref*$(l_a) = \text{null}$; *SB_Deref*$(l_b) = \text{null}$ occurred. Then line SD1 of *SB_Deref*(l_a) was executed after line H13 of *Move* and line SD1 of *SB_Deref*(l_b) before line H1 of *Move*. This is a contradiction.

(ii) Assume towards a contradiction that the sequence *SB_Deref*$(l_b) = b$; *SB_Deref*$(l_a) = b$ occurred. then line SD3 of *SB_Deref*(l_b) was executed after line H1 of *Move* and line SD3 of *SB_Deref*(l_a) before line M12 of *Move*. This is also a contradiction. □

Lemma 3 (Linearizability II) *The operation* Move *is linearizable with respect to other* Move *and* SB_Deref *operations with linearization points according to Proposition 2.*

Proof Consider the operation $\mathit{Move}_1(r, l_a, l_b)$ for the superblock x. There are three main cases for its interaction with other concurrent *Move* operations:

1. Chained. There is a $\mathit{Move}_2(r', l_b, l')$ moving another superblock from l_b.
2. Same source. There is a $\mathit{Move}_2(r, l_a, l')$.
3. Same destination. There is a $\mathit{Move}_2(r', l', l_b)$.

Case 1. Chained. Assume there is a $\mathit{Move}_2(r', l_b, l')$. We will examine the potential linearizations below:

$\mathit{Move}_1 = \text{OK} \rightarrow \mathit{Move}_2 = \text{OK}$: This case is impossible. The successful Move_1 and Move_2 imply the following two sequences of events occurred as outlined above: M2_1, M3_1, M7_1, M12_1, H1_1, H12_1, H13_1 and H15_1 and M2_2, M3_2, M7_2, M12_2, H1_2, H12_2, H13_2 and H15_2, where H1_1 and H1_2 are the linearization points of Move_1 and Move_2. The key point here is that the *SB_Deref* at M7_1 has to return null, which implies that it occurred after H13_2 or helped Move_2 to completion. So we have $\mathit{Move}_2 \rightarrow \mathit{Move}_1$ which contradicts the assumed order.

$\mathit{Move}_1 = \text{OK} \rightarrow \mathit{Move}_2 = \text{NOOP}$: This case is impossible. There is no way Move_1 can succeed when l_b persistently contains r'.

$\mathit{Move}_1 = \text{NOOP} \rightarrow \mathit{Move}_2 = \text{OK}$: The successful Move_2 implies the following events occured as outlined above: M2_2, M3_2, M7_2, M12_2, H1_2, H12_2, H13_2 and H15_2, where H1_2 is the linearization point of Move_2. There are two potential linearization points for Move_1: (i) M3_1 if $l_a \neq r$ at that point, which is rather uninteresting since the outcome of Move_1 doesn't depend on Move_2 in this case; (ii) M7_1 if $\mathit{SB\_Deref}(l_b)$ returns r' there, which implies that M7_1 occured before M12_2 (since *SB_Deref* would help Move_2 otherwise). The linearization point H1_1 for Move_1 would require that l_b was empty at M7_1 and imply that there is an operation Move' moving the block referenced by r' into l_b between Move_1 and Move_2. This is covered by case 3 below.

$\mathit{Move}_1 = \text{NOOP} \rightarrow \mathit{Move}_2 = \text{NOOP}$: In this case Move_1 is linearized at M3_1 or M7_1. In case Move_2 is linearized at H1_2 then $\text{M7}_1 \rightarrow \text{M12}_2$ or else the opposite linearization order would have occured due to helping.

$\mathit{Move}_2 = \text{OK} \rightarrow \mathit{Move}_1 = \text{OK}$: The successful Move_1 and Move_2 imply the following two sequences of events occured as outlined above: M2_1, M3_1, M7_1, M12_1, H1_1, H12_1, H13_1 and H15_1 and M2_2, M3_2, M7_2, M12_2, H1_2, H12_2, H13_2 and H15_2, where H1_1 and H1_2 are the linearization points of Move_1 and Move_2. The key point here is that the *SB_Deref* at M7_1 has to return null, which implies that it occured after H13_2 or helped Move_2 to completion. So $\mathit{Move}_2 \rightarrow \mathit{Move}_1$.

$\mathit{Move}_2 = \text{OK} \rightarrow \mathit{Move}_1 = \text{NOOP}$: The successful Move_2 implies the following events occured as outlined above: M2_2, M3_2, M7_2, M12_2, H1_2, H12_2, H13_2 and H15_2, where H1_2 is the linearization point of Move_2. The linearization point for Move_1 is M3_1, as the linearization point M7_1 would have implied $\text{M7}_1 \rightarrow \text{M12}_2$

and $H1_1$ would have implied that an other *Move* with destination l_b occured between $Move_2$ and $Move_1$.

$Move_2 = \text{NOOP} \rightarrow Move_1 = \text{OK}$: This case is impossible. There is no way $Move_1$ can succeed when l_b persistently contains r'.

$Move_2 = \text{NOOP} \rightarrow Move_1 = \text{NOOP}$: In this case $Move_1$ is linearized at $M3_1$ or $M7_1$ where in the $M7_1$ case $Move_1$ might help $Move_2$ to complete.

Case 2. Same source. Assume there is a $Move_2(r, l_a, l')$. We will examine the potential linearizations below (the cases where $Move_2$ is linearized first are symmetric):

$Move_1 = \text{OK} \rightarrow Move_2 = \text{OK}$: This is an impossible outcome. To have any possibility of succeeding an operation has to be registered by a successful CAS at line M12. When the old value passed to CAS was read at M2, location l_A must still contain r at line M4 and there must not be an ongoing operation on this superblock (line M6). Clearly, one operation will be register before the other and once that has happened the other have no hope of registering since when the first operation is finished (by itself or via helping) l_a isn't equal to r anymore.

$Move_1 = \text{OK} \rightarrow Move_2 = \text{NOOP}$: The successful $Move_1$ implies the following events occured as outlined above: $M2_1$, $M3_1$, $M7_1$, $M12_1$, $H1_1$, $H12_1$, $H13_1$ and $H15_1$, where $H1_1$ is the linearization point of $Move_1$. There are two potential linearization points for $Move_2$: (i) $M3_2$ if $l_a \neq r$ at that point, which implies that $H13_1$ has already occured; (ii) $M7_2$ if $M2_2$ occured before $M12_1$ and $M3_2$ before $H13_1$, which doesn't force $M7_2$ to happen after $H1_1$ but that linearization order is also fine since the outcome of $Move_2$ then didn't depend on $Move_1$. The linearization point $H1_2$ is impossible here since it implies $Move_2$ was successfully registered in x before $Move_1$ and therefore $Move_2 \rightarrow Move_1$.

$Move_1 = \text{NOOP} \rightarrow Move_2 = \text{OK}$: The successful $Move_2$ implies the following events occured as outlined above: $M2_2$, $M3_2$, $M7_2$, $M12_2$, $H1_2$, $H12_2$, $H13_2$ and $H15_2$, where $H1_2$ is the linearization point of $Move_2$. There are two possible linearization points for $Move_1$: (i) $M7_1$ if $M2_1$ occured before $M12_2$ and $M3_1$ before $H13_2$, which actually does not imply $Move_1 \rightarrow Move_2$ but is indistinguishable from the reverse case as seen above; (ii) $H1_1$ which implies the CAS at $M12_1$ was successful and occured before $M12_2$, hence $Move_1 \rightarrow Move_2$.

$Move_1 = \text{NOOP} \rightarrow Move_2 = \text{NOOP}$: Since $Move_1$ and $Move_2$ are directly next to each other in the linearized history and neither of them have any external effect their order doesn't matter. Hence it is fine linearize them according to the proposed linearization points.

Case 3. Same destination. Assume there is a $Move_2(r', l', l_b)$. We will examine the potential linearizations below:

$Move_1 = \text{OK} \rightarrow Move_2 = \text{OK}$: This is an impossible outcome. To return OK both *Move*s must succeed to change the null reference in l_b to a reference to the respective superblock using CAS at $H1_1$ and $H1_2$, respectively. That is clearly impossible.

$Move_1 = \text{OK} \rightarrow Move_2 = \text{NOOP}$: The successful $Move_1$ implies the following events occured as outlined above: $M2_1$, $M3_1$, $M7_1$, $M12_1$, $H1_1$, $H12_1$, $H13_1$ and $H15_1$, where $H1_1$ is the linearization point of $Move_1$. There are two interesting

possible linearization points for *Move*$_2$: (i) M7_2 which implies M7_2 occured after H1_1 and hence *Move*$_1$ $\rightarrow$ *Move*$_2$; (ii) H1_2 which implies M7_2 occured before H1_1 but (as *Move*$_2$ returns NOOP) H1_1 occured before H1_2, hence *Move*$_1$ $\rightarrow$ *Move*$_2$. The linearization point M3_2 is an uninteresting case as it fails without interacting with *Move*$_1$.

Move$_1$ = NOOP $\rightarrow$ *Move*$_2$ = OK: The successful *Move*$_2$ implies the following events occured as outlined above: M2_2, M3_2, M7_2, M12_2, H1_2, H12_2, H13_2 and H15_2, where H1_2 is the linearization point of *Move*$_2$. The linearization point for *Move*$_1$ is M3_1, which is an uninteresting case as it fails without interacting with *Move*$_2$. The linearization points M7_1 or H1_1 cannot occur as they either imply that H1_2 occured before M7_1 or H1_1 which contradict *Move*$_1$ $\rightarrow$ *Move*$_2$, or that another *Move*' moved another superblock away from l_b between *Move*$_1$ and *Move*$_2$ which implies that the *Move*s can be linearized according to case 2 above.

Move$_1$ = NOOP $\rightarrow$ *Move*$_2$ = NOOP: If one or both *Move*s are linearized at M3 the case is clear. Likewise for the other linearization points. □

Lemma 4 (Lock-free I) *The operation* Move *is lock-free.*

Proof There is only one loop in *Move* and none in *Move_Help*. For a *Move* invocation to remain in the loop it either has to find an already ongoing operation concerning the same superblock (line M6) or have the CAS at line M12 fail. In the first case this operation itself will help the other ongoing operation to complete (line M16), thus ensuring some operation makes progress. In the second case another *Move* operation successfully registered itself and has thus made progress. □

Lemma 5 (Lock-free II) *The operation* SB_Deref *is lock-free.*

Proof There is only one loop in *SB_Deref* and none in *Move_Help*. For a *SB_Deref* to remain in the loop it needs to read a non-null superblock reference from loc (line SD1) and find that the referenced superblock is involved in an ongoing operation (line SD4). If there is an ongoing operation *SB_Deref* will finish it via *Move_Help*. So, in the next iteration loc may contain: (i) null, (e.g. if the move operation moved the superblock from loc); or (ii) a reference to a superblock with no ongoing operation; or (iii) a reference to a superblock with an ongoing operation. In the first two cases *SB_Deref* will terminate. In (iii) *SB_Deref* would do another iteration, but in that case, clearly, some other concurrent operation made progress (at least) via the help provided by *Move_Help*, so it is acceptable for *SB_Deref* not to make progress. □

The flat-set Insert *and* Get_Any *operations* Flat-sets are containers with the following operations: *Get_Any*(s, r, loc) returns SUCCESS and a reference r to a superblock x and r's location loc in the flat-set s or EMPTY if the flat-set s is empty.

Insert(s, r, loc), which takes a superblock reference r and its current location loc as arguments, returns:

- SUCCESS and updates loc to reference the new location of r if the superblock is inserted into s (and removed from its previous location);

- MOVED_AWAY if the superblock is no longer present in *loc*; and
- FULL if the flat-set s is full.

Proposition 3 *The linearization point of the operation* Get_Any(s, r, loc) *is*

- *the* SB_Deref *at line G*7 *iff SUCCESS is returned*; *and*
- *at line G*1 *iff EMPTY is returned*.

Proposition 4 *The linearization point of the operation* Insert(s, r, loc) *is*

- *the* Move *at line I*6 *iff it returns SB_MOVED_OK*;
- *the memory read at line I*10 *iff* $^*\mathsf{loc} \neq \mathsf{sb}$; *and*
- *at line I*1 *iff FULL is returned*.

To shorten the linearizability proofs we first outline the key events for an operation Get_Any(set, r, loc) and Insert(set, r, loc) to return a particular result here. These lists of events follow directly from the pseudo code.
Get_Any(set, r, loc):

SUCCESS **G6**: SB_Deref returns a non-null reference;
G11: set.current is updated with (i,) if it still is equal to old_current.
EMPTY G6–12: No non-null reference is found in set.set;
G6–12: No non-null reference is found in set.set;
G11: The second iteration saw exactly the same values in set.set as the first.

Insert(set, r, loc):

SUCCESS I5: SB_Deref(&set.set[i]) is null;
I6: Move$(r, loc,$ set.set[i]$)$ = SB_MOVED_OK;
I7: set.current is updated with $(i,)$ if it still is equal to old_current.
MOVED_AWAY I5: SB_Deref(&set.set[i]) is null;
I6: Move$(r, loc,$ set.set[i]$)$ = SB_NOOP;
I10: $^*loc \neq r$.
FULL I1–12: set.set[i] $\neq$ null for all i;
I1–12: set.set[i] $\neq$ null for all i;
I13: The second iteration saw exactly the same values in set.set as the first.

Lemma 6 (Linearizability III) *The operations* Get_Any *and* Insert *are linearizable with respect to other* Get_Any *and* Insert *operations with linearization points according to Proposition* 3.

Proof There are three kinds of interesting interactions: (i) operations on or returning the same superblock, (ii) operations on an empty flat-set, and (iii) operations on a full flat-set.

(i) Consider the superblock x referenced by the reference r located at index i_1 in the flat set s_1. Note that any Get_Any returning r is linearized at a SB_Deref (line G5) and any Insert of x returning SUCCESS is linearized at a Move (line I6). Lemmas 2 and 3 show that these operations are linearizable w.r.t. each other.

(ii) Consider an empty flat-set s and two operations *Get_Any*(s, r', l') returning EMPTY and *Insert*(s, r, l) returning SUCCESS. Assume towards a contradiction that *Insert* $\rightarrow$ *Get_Any*. As mentioned above a successful *Insert* implies the following events I5, I6, and I7 where the linearization point is that of the *Move* at line I6. For *Get_Any* to return EMPTY it must complete two successive scans of the flat-set location array without finding a superblock in any location and without any location changing value between the two scans (lines G6–G12, G6–G12). The linearization point is between the two scans. According to our assumption, the *Move* at I6 is linearized before the start of the second scan in *Get_Any*. However, then Lemmas 2 and 3 guarantee that a *SB_Deref* (i.e. that of the location x was moved to) in the second scan returns a reference to x, hence *Get_Any* will not return EMPTY.

(iii) Consider a flat-set s with one empty location and two operations *Insert*$_1(s, r, l)$ returning SUCCESS and *Insert*$_2(s, r', l')$ returning FULL. Assume towards a contradiction that *Insert*$_2 \rightarrow$ *Insert*$_1$. The successful *Insert*$_1$ implies the following events: I5$_1$, I6$_1$, and I7$_1$ where the linearization point is that of the *Move* at line I6$_1$. For *Insert*$_2$ to return FULL it must complete two successive scans of the flat-set location array without finding any empty location and without any location changing value between the two scans (lines I1$_2$–I12$_2$, I1$_2$–I12$_2$). The linearization point is between the two scans. According to our assumption the linearization point in the *Move* at I6$_1$ must occur after the start of the second scan in *Insert*$_2$. However, for *Insert*$_2$ to return FULL the second scan must see exactly the same values in the location array as the first scan and all locations must be occupied. Hence, *Insert*$_1$ also cannot succeed. If the second scan in *Insert*$_2$ sees the value written by *Insert*$_1$ that value differs from what was read from that location during the first scan. So *Insert*$_2$ will scan the flat-set location array again, thereby moving its linearization point after that of *Insert*$_1$. □

Lemma 7 (Lock-free III) *The operation Get_Any is lock-free.*

Proof The operation *Get_Any* contains one unbounded loop. To remain in this loop the state of the flat-set location array must not remain unchanged for two iterations in a row (line G11). That the flat-set array changed implies some other operation made progress. □

Lemma 8 (Lock-free IV) *The operation Insert is lock-free.*

Proof The operation *Insert* contains two nested loops. First, note that the outer loop is exited if the state of the flat-set location array remains unchanged for two iterations (line I13). Further, as the inner loop tries to move the superblock to each empty location found in the array, two iterations of the outer loop without any change in the flat-set array guarantees that the flat-set was full at the start of the second iteration. If the flat-set array changed that implies some other operation made progress.

For *Insert* to remain in the inner loop the current location set.set[i] must be null at the start of each iteration (line I4) and *Move* at I6 must return SB_NOOP due to the destination set.set[i] being different from null. Recall that *Move* could also return SB_NOOP if the superblock is no longer present at loc but in that case *Insert* returns

at line I11. So, for *Insert* to remain in the inner loop set.set[i] has to change from null to non-null (and then back again) which can only be the result of other operations making progress. □

5 Experimental Study

5.1 Systems

There are two major families of cache-coherent multiprocessor architectures—UMA (Uniform Memory Architecture) and NUMA (Non-Uniform Memory Architecture). In a UMA system all processors have the same latency to the memory. In a NUMA system, this is not the case, since access to memory on another node can be significantly slower.

NBMALLOC has been studied on a three multiprocessor systems, both on UMA and NUMA memory architectures. The three systems are

(i) an UMA Sun Sun-Fire 880 with 6 900 MHz UltraSPARC III+ (4 MB L2 cache) processors running Solaris 9;
(ii) a ccNUMA SGI Origin 2000 with 30 250 MHz MIPS R10000 (4 MB L2 cache) processors running IRIX 6.5;
(iii) an UMA PC with 2 2.80 GHz Intel Xeon (512 KB L2 cache) processors running Linux 2.6.9-22 SMP.

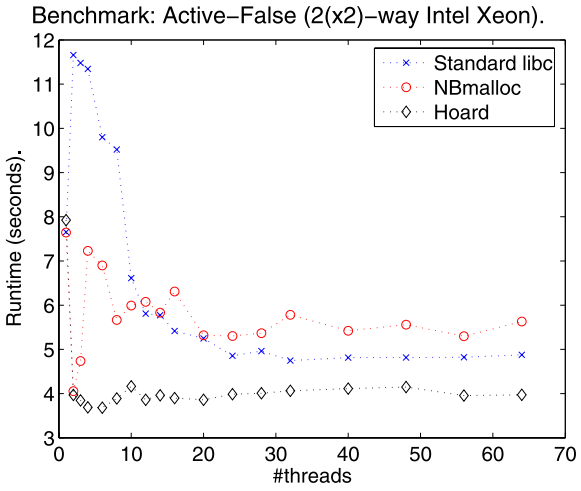
5.2 Benchmarks

We used the following common benchmarks to evaluate NBMALLOC:

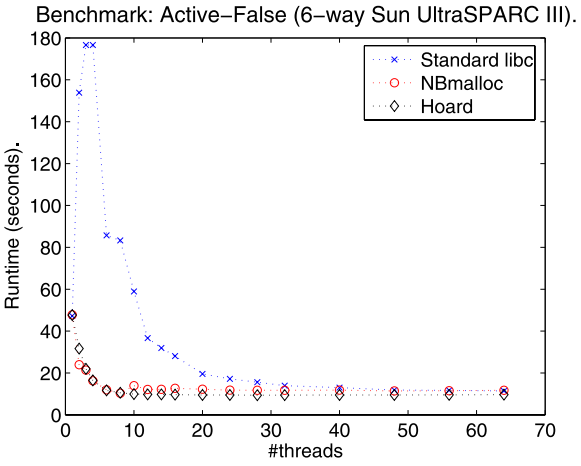
The *Larson* [2, 3, 39] benchmark simulates a multi-threaded server application that makes heavy use of dynamic memory. Each thread allocates and deallocates objects of random sizes (between 5 to 500 bytes) and also transfers some of the objects to other threads to be deallocated there. The benchmark result is throughput in terms of the number of allocations and deallocations per second, which reflects the allocator's behaviour with respect to false-sharing and scalability, and the resulting memory footprint of the process which should reflect any tendencies for heap-blowup. We measured the throughput during 60 second runs for each set of threads.

The *Active-false* and *passive-false* [2, 3] benchmarks measure how the allocator handles active (i.e. directly caused by the allocator) respective passive (i.e. caused by application behaviour) false-sharing. In the benchmarks each thread repeatedly allocates an object of a certain size (1 byte) and subsequently reads and writes to that object a large number of times (1000) before deallocating it again. If the allocator does not take care to avoid false-sharing, several threads might get objects located in the same cache-line and this will slow down the reads and writes to the objects considerably. In the *passive-false* benchmark all initial objects are allocated by one thread and then transfered to the others to introduce the risk of passive false-sharing when those objects are later freed for reuse by the threads. The benchmark result is the total wall-clock time for performing a fixed number (10^6) of allocate-read/write-deallocate cycles among all threads.

Fig. 3 The results from the active false-sharing benchmark



(a) Active-False: PC with 2 Intel Xeon CPUs



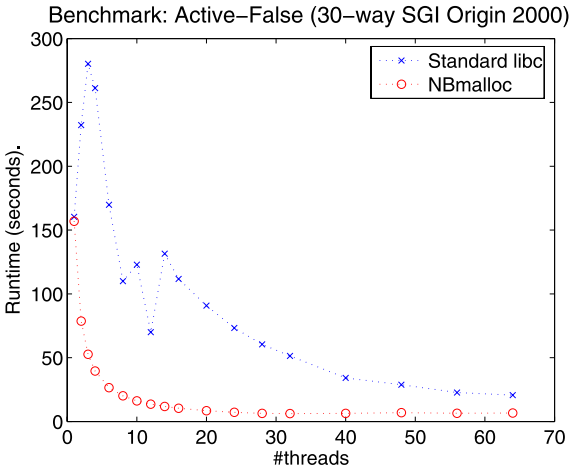
(b) Active-False: Sun with 6 UltraSPARC III+ CPUs

5.3 Implementation

In our implementation of NBMALLOC[2] we use the *CAS* primitive (implemented from the hardware synchronization instructions available on the respective system) for the lock-free operations. To avoid ABA problems we use the version number solution ([27], cf. Sect. 2.2). We use 16-bit version numbers for the superblock references in the flat-sets. The reason why this is safe, is that for a bad event (i.e. that a *CAS* of a superblock reference succeeds when it should not) to happen, not only must the

[2]Our implementation is available at http://www.cs.chalmers.se/~dcs/nbmalloc.html.

Fig. 3 (*Continued*)



(c) Active-False: SGI Origin 2000 with 30 MIPS 10k CPUs

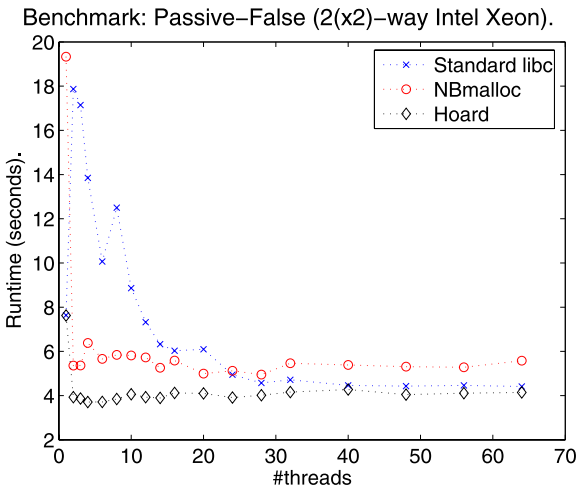
version numbers be equal, but also that same superblock must have been moved back to the same location in the flat-set, which contains thousands of locations. We use superblocks of 64 KB and this leaves enough space for version numbers in superblock pointers. We also use size-classes that are powers of two, starting from 8 bytes. This is not a decision forced by the algorithm; a more tightly spaced set of size-classes can also be used; this would impose some extra fixed space overhead due to the preallocated flat-sets for each size-class, but it would also further reduce internal fragmentation. Blocks larger than 32 KB are allocated directly from the operating system instead of being handled in superblocks. Our implementation uses four fullness-groups and 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 prevents superblocks from rapidly oscillating between fullness-groups. Further, we set the maximum size for the flat-sets used in the global heap and for those in per-processor heaps to 4093 superblocks each (these values can be adjusted separately).

5.4 Results

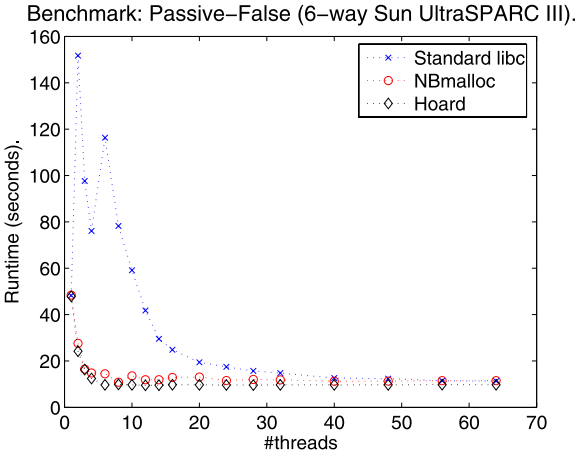
In the evaluation we study NBMALLOC in connection with the standard "libc" allocator of the respective platform using the above standard benchmark applications. On the Sun platform, for which we had the original Hoard allocator available, we also study in connection with Hoard (version 3.7.0 on the PC and 3.4.0 on the other platforms). To the best of our knowledge, Hoard is not available for ccNUMA SGI IRIX platforms. Note that on the PC/Linux platform, the default "libc" malloc is in fact Ptmalloc, a lock-based concurrent memory allocator with private per-processor heaps by Gloger [21].

The benchmarks are intended to test scalability, fragmentation and false-sharing behaviour, which are the evaluation criteria of a good concurrent allocator, as explained in the introduction. When performing these experiments our main goal was

Fig. 4 The results from the passive false-sharing benchmark



(a) Passive-False: PC with 2 Intel Xeon CPUs

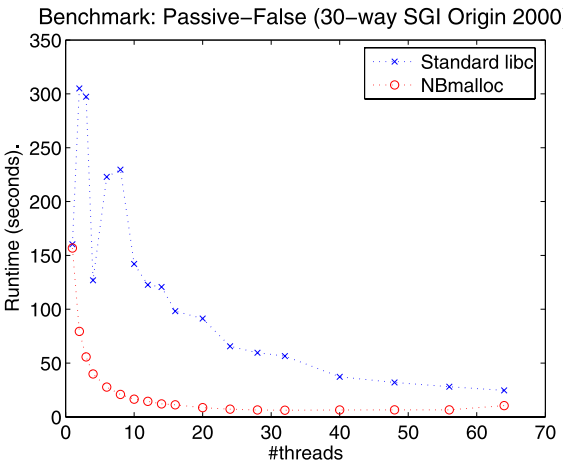


(b) Passive-False: Sun with 6 UltraSPARC III+ CPUs

not to optimize the performance of the lock-free allocator, but rather to examine the benefits of the lock-free design itself.

The results from the two false-sharing benchmarks, shown in Figs. 3 and 4, respectively, show that NBMALLOC and Hoard induce very little false-sharing. The standard "libc" allocator, on the other hand, suffers significantly from false-sharing as shown by its longer and irregular runtimes. For "libc" false-sharing causes the largest slowdown when there are few but fully concurrent threads, as they are the most likely to get objects in the same cache-line and also access them concurrently. When the number of threads gets larger, objects are more likely to be in different cache-lines and also, due to time-sharing, not all threads execute at the same time. An important

Fig. 4 (*Continued*)



(c) Passive-False: SGI Origin 2000 with 30 MIPS 10k CPUs

observation throughout the experiments is that NBMALLOC shows consistent behaviour as the number of processors and memory architectures change.

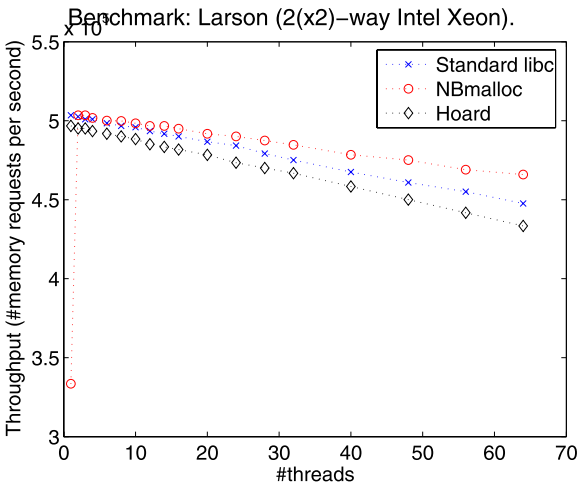
The throughput results from the Larson benchmark, shown in Fig. 5, show that NBMALLOC has good scalability, not only in the case of full concurrency (where Hoard also shows excellent scalability), but also when the number of threads increases beyond the number of processors. In that region, Hoard's performance quickly drops from its peak at full concurrency on the Sun (cf. Fig. 5(b)) and slowly on the PC (cf. Fig. 5(a)).

We can actually observe more clearly the scalability properties of the lock-free allocator in the performance diagrams on the SGI Origin 2000 platform (Fig. 5(c)). We can observe a linear-style of throughput increase when the number of processors increases (recall that we have 30 processors available in the Origin 2000). Furthermore, when the load on each processor increases beyond one thread, the throughput of the lock-free allocator stays high, as is desirable for scalability. In terms of absolute throughput, Hoard is superior to NBMALLOC on the Sun platform where we had the possibility to compare them. On the PC/Linux platform the situation is reversed and except for the sequential case NBMALLOC has higher throughput than both Hoard and Ptmalloc/glibc.

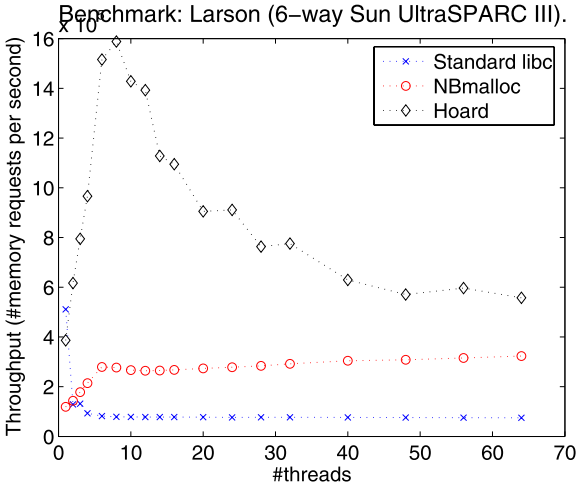
The results with respect to memory consumption, in Fig. 6, show that for the Larson benchmark the memory usage (and thus fragmentation) of NBMALLOC stays at a similar or better level than Hoard (cf. Fig. 6(a) and (b)). Despite having a larger initial overhead than single heap allocators such as the "libc" allocator the use of per-processor heaps with thresholds scales almost as well with respect to memory utilization.

To summarise, an interesting conclusion is that the scalability of Hoard's architecture is further enhanced by lock-free synchronization. Moreover, note that NBMALLOC shows a very similar behaviour in throughput on both the UMA and the ccNUMA systems. This is another positive implication of lock-free synchroniza-

Fig. 5 The Larson benchmark: Throughput



(a) Throughput: PC with 2 Intel Xeon CPUs



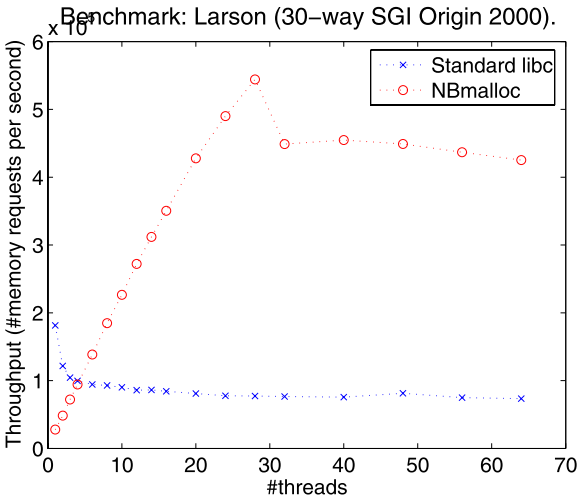
(b) Throughput: Sun with 6 UltraSPARC III+ CPUs

tion, as it means much fewer (to none) contention hot-spots. In connection with the architecture context, this is even better, as contention hot-spots tend to cause much larger performance penalties on NUMA than on UMA architectures.

6 Other Related Work

As mentioned in the introduction, concurrently with and independently from our work on NBMALLOC, Michael presented a lock-free allocator [4] that, like our contribution, is loosely based on the Hoard architecture. Despite both having started

Fig. 5 (*Continued*)

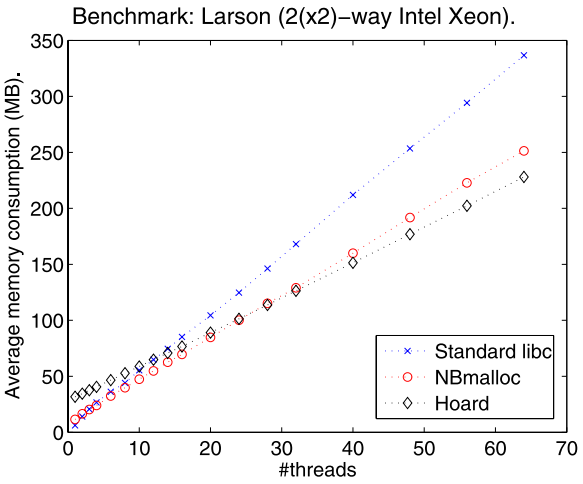


(c) Throughput: SGI Origin 2000 with 30 MIPS 10k CPUs

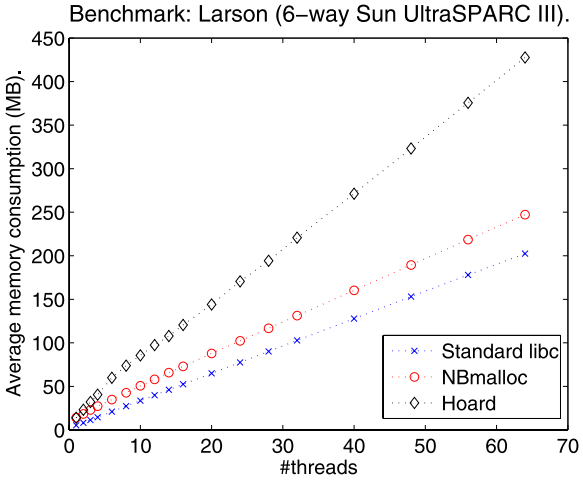
from the Hoard architecture, we have used two different approaches to achieve lock-freedom. In Michael's allocator each per-processor heap contains one active (i.e. used by memory requests) and at most one inactive partially filled superblock per size-class, plus an unlimited number of full superblocks. All other partially filled superblocks are stored globally in per-size-class FIFO queues. It is an elegant algorithmic construction, and from the scalability and throughput performance point of view it performs excellently, as is shown in [4], in the experiments carried out on a 16-way POWER3 platform. By further studying the allocators, it is relevant to note that: NBMALLOC and Hoard keep all partially filled superblocks in their respective per-processor heap while the allocator in [4] does not and this may increase the potential for inducing false-sharing. NBMALLOC and Hoard also keep the partially filled superblocks sorted by fullness and not doing so, like the allocator in [4] does, may imply some increased risk of external fragmentation since the fullness order is used to direct allocation requests to the more full superblocks which makes it more likely that less full ones become empty and thus eligible for reuse. The allocator in [4], unlike ours, uses the *first-remove-then-insert* approach to move superblocks around, which in a concurrent environment could affect the fault-tolerance of the allocator and cause unnecessary allocation of superblocks since a superblock is invisible to other threads while it is being moved.

Another allocator that reduces the use of locks is LFMalloc [5]. It uses a method for almost lock-free synchronization, whose implementation requires the ability to efficiently manage CPU-data and closely interact 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.

Fig. 6 The Larson benchmark: Average memory consumption



(a) Memory consumption: PC with 2 Intel Xeon CPUs



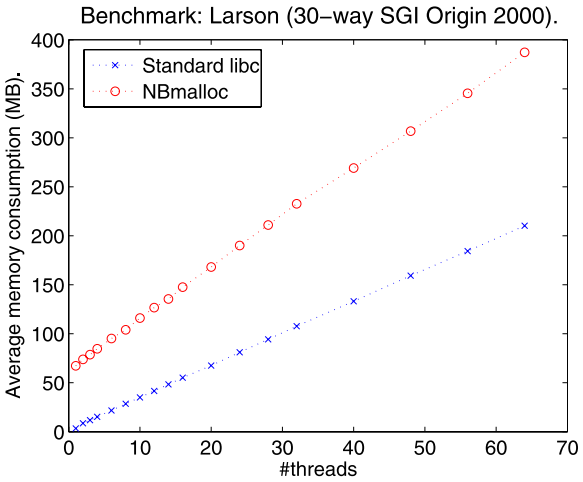
(b) Memory consumption: Sun with 6 UltraSPARC III+ CPUs

Besides the contribution corresponding to the functionality of memory allocation, in this work we also show a data structure implementation with lock-free and linearizable operations involving more than one of its instances ("inter-object" operations). This contribution may enable a new methodology in lock-free system components construction (cf. next section).

7 Discussion and Future Work

The lock-free memory allocator proposed in this paper confirms our expectation that fine-grain, lock-free synchronization is useful for scalability under increasing load in

Fig. 6 (*Continued*)



(c) Memory consumption: SGI Origin 2000 with 30 MIPS 10k CPUs

the system. To the best of our knowledge, this, together with the allocator that was independently presented in [4] are also the first lock-free general allocators (based on single-word CAS) in the literature. We expect that this contribution will have an interesting impact in the domain of memory allocators and service systems for multiprocessors. NBMALLOC has been used for memory allocation for dynamic lock-free data structure implementations within the C++ STL library research efforts by Dechev et al. [40].

It will be useful to study a generalization of the method for "inter-object" operations, which is illustrated here in the move operation. A general methodology in this direction would enable combinations of known lock-free data structures (e.g. list-structures) into larger, interconnected ones, to be integrated in systems such as the one studied here.

Acknowledgements We would like to thank the anonymous reviewers for their detailed and helpful comments, Håkan Sundell for interesting discussions on non-blocking methods and Maged Michael for helpful comments on an earlier version of this paper.

References

1. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Allocating memory in a lock-free manner. In: Proc. of the 13th Annual European Symp. on Algorithms (ESA'05). LNCS, vol. 3669, pp. 329–342. Springer, Berlin (2005)
2. Berger, E.D.: Memory management for high-performance applications. Ph.D. Thesis, The University of Texas at Austin, Department of Computer Sciences (2002)

3. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A scalable memory allocator for multithreaded applications. In: ASPLOS-IX: 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 117–128 (2000)
4. Michael, M.: Scalable lock-free dynamic memory allocation. In: Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation. ACM SIGPLAN Notices. ACM, New York (2004)
5. Dice, D., Garthwaite, A.: Mostly lock-free malloc. In: ISMM'02 Proc. of the 3rd Int. Symp. on Memory Management. ACM SIGPLAN Notices, pp. 163–174. ACM, New York (2002)
6. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, pp. 21–30. ACM, New York (2002)
7. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**, 491–504 (2004)
8. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In: Proceedings of the 16th International Symposium on Distributed Computing (DISC'02). LNCS, vol. 2508, pp. 339–353. Springer, Berlin (2002)
9. Valois, J.D.: Lock-free data structures. Ph.D. Thesis, Rensselaer Polytechnic Institute, Department of Computer Science, Troy, New York (1995)
10. Michael, M.M., Scott, M.L.: Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department (1995)
11. Detlefs, D.L., Martin, P.A., Moir, M., Guy L. Steele, J.: Lock-free reference counting. In: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 190–199. ACM, New York (2001)
12. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Nonblocking memory management support for dynamic-sized data structures. ACM Trans. Comput. Syst. **23**, 146–196 (2005)
13. Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P.: Practical and efficient lock-free garbage collection based on reference counting. In: Proc. of the 8th International Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN), pp. 202–207. IEEE Comput. Soc., Los Alamitos (2005)
14. Schneider, S., Antonopoulos, C., Nikolopoulos, D.: Scalabel locality-conscious multithreaded memory allocation. In: Proceedings of the 2006 International Symposium on Memory Management (ISMM'06), pp. 84–94. ACM, New York (2006)
15. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91 (1991)
16. Massalin, H.: Synthesis: an efficient implementation of fundamental operating system services. Ph.D. Thesis, Columbia University (1992)
17. Greenwald, M., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: Operating Systems Design and Implementation, pp. 123–136 (1996)
18. Greenwald, M.B.: Non-blocking synchronization and system design. Ph.D. Thesis, Stanford University (1999)
19. SGI: The standard template library for C++ (2003). http://www.sgi.com/tech/stl/Allocators.html
20. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. Assoc. Comput. Mach. **46**, 720–748 (1999)
21. Gloger, W.: Wolfram Gloger's malloc homepage (2003). http://www.malloc.de/en/
22. Steensgaard, B.: Thread-specific heaps for multi-threaded programs. In: ISMM 2000 Proc. of the Second Int. Symp. on Memory Management. ACM SIGPLAN Notices, vol. 36(1). ACM, New York (2000)
23. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**, 463–492 (1990)
24. Barnes, G.: A method for implementing lock-free shared data structures. In: Proc. of the 5th Annual ACM Symp. on Parallel Algorithms and Architectures, SIGACT and SIGARCH, pp. 261–270 (1993). Extended abstract
25. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Syst. **11**, 124–149 (1991)
26. Rinard, M.C.: Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. ACM Trans. Comput. Syst. **17**, 337–371 (1999)
27. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC '95), pp. 214–222. ACM, New York (1995)
28. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proc. of the 13th Annual ACM Symp. on Parallel Algorithms and Architectures, pp. 134–143. ACM, New York (2001)

29. Harris, T.L.: A pragmatic implementation of non-blocking linked lists. In: Proc. of the 15th Int. Conf. on Distributed Computing, pp. 300–314. Springer, Berlin (2001)
30. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. J. Parallel Distrib. Comput. **62**, 766–791 (2002)
31. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing, pp. 219–228 (1997)
32. Papatriantafilou, M., Tsigas, P.: Wait-free consensus in "in-phase" multiprocessor systems. In: Symp. on Parallel and Distributed Processing (SPDP '95), pp. 312–319. IEEE Comput. Soc., Los Alamitos (1995)
33. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Process. Lett. **7**, 321–328 (1997)
34. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. In: Proc. of the 17th IEEE/ACM Int. Parallel and Distributed Processing Symp. (IPDPS 03). IEEE Press, New York (2003)
35. Valois, J.D.: Implementing lock-free queues. In: Proc. of the Seventh Int. Conf. on Parallel and Distributed Computing Systems, pp. 64–69 (1994)
36. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA-02), pp. 73–82. ACM, New York (2002)
37. Jayanti, P.: A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In: Proceedings of the 12th International Symposium on Distributed Computing (DISC '98). Lecture Notes in Computer Science, vol. 1499, pp. 216–230. Springer, Berlin (1998)
38. IBM: IBM System/370 Extended Architecture, Principles of Operation (1983). Publication No. SA22-7085
39. Larson, P.P.Å., Krishnan, M.: Memory allocation for long-running server applications. In: ISMM'98 Proc. of the 1st Int. Symp. on Memory Management. ACM SIGPLAN Notices, pp. 176–185. ACM, New York (1998)
40. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Lock-free dynamically resizable arrays. In: Proc. of the 10th Int. Conf. on Principles of Distributed Systems OPODIS'06. LNCS, vol. 4305, pp. 142–156. Springer, Berlin (2006)