Allocating Memory in a Lock-Free Manner*

Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas

Department of Computer Science and Engineering,
Chalmers University of Technology, SE-412 96 Göteborg, Sweden
{andersg, ptrianta, tsigas}@cs.chalmers.se

Abstract. The potential of multiprocessor systems is often not fully realized by their system services. Certain synchronization methods, such as lock-based ones, may limit the parallelism. It is significant to see the impact of wait/lock-free synchronization design in key services for multiprocessor systems, such as the memory allocation service. Efficient, scalable memory allocators for multithreaded applications on multiprocessors is a significant goal of recent research projects.

We propose a lock-free memory allocator, to enhance the parallelism in the system. Its architecture is inspired by Hoard, a successful concurrent memory allocator, with a modular, scalable design that preserves scalability and helps avoiding false-sharing and heap blowup. Within our effort on designing appropriate lock-free algorithms to construct this system, we propose a new non-blocking data structure called flat-sets, supporting conventional "internal" operations as well as "inter-object" operations, for moving items between flat-sets.

We implemented the memory allocator in a set of multiprocessor systems (UMA Sun Enterprise 450 and ccNUMA Origin 3800) and studied its behaviour. The results show that the good properties of Hoard w.r.t. false-sharing and heap-blowup are preserved, while the scalability properties are enhanced even further with the help of 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 a memory allocator is 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

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

application. The memory allocator is not allowed to move or otherwise disturb memory blocks that are currently owned by the application.

A good allocator should aim at minimizing *fragmentation*, i.e. minimizing the amount of free memory that cannot be used (allocated) by the application. *Internal fragmentation* is free memory wasted when the application is given a larger memory block than it requested; and *external fragmentation* is free memory that has been split into too small, non-contiguous blocks to be useful to satisfy the requests from the application. Multi-threaded programs add some 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 outlined below, which have significant impact on application performance when the application is run on a multiprocessor [1]. 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 as fast as a good sequential one when executed on a single processor and its performance should scale with the load in the system.

The Hoard [2] 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, where each superblock handles 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.

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, as 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 [3,4,5,6]. The potential of this type of synchronization in the performance of system-services and data structures has also been pointed out earlier, in [7,4,8].

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 operations of common sets, as well as "inter-object" operations, for moving an item from one flat-set to another in a lock-free manner. The lock-free algorithms we introduce make use of standard synchronization primitives provided by multiprocessor systems, namely single-word *Compare-And-Swap*, or its equivalent *Load-Linked/Store-Conditional*.

We have implemented and evaluated the allocator proposed here on common multiprocessor platforms, namely an UMA Sun Enterprise 450 running Solaris 9 and a ccNUMA Origin 3800 running IRIX 6.5. We compare our allocator with the standard "libc" allocator of each platform and with Hoard (on the Sun system, where we had the original Hoard allocator availableusing standard benchmark applications to test the efficiency, scalability, cache behaviour and memory consumption behaviour. The results show that our system preserves the good properties of Hoard, while it offers a higher scalability potential, as justified by its lock-free nature.

In the next section we provide background information on lock- and wait-free synchronization (throughout the paper we use the terms non-blocking and lock-free interchangeably). Earlier and recent related work is discussed in section 7, after the presentation of our method and implementation, as some detail is needed to relate these contributions.

2 Background: Non-blocking Synchronization

Non-blocking implementations of shared data objects are an alternative to the traditional solution for maintaining the consistency of a shared data object (i.e. for ensuring *linearizability* [9]) by enforcing mutual exclusion. Non-blocking synchronization allows multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion [3,4,5,6,10]. Non-blocking 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. Non-blocking algorithms have been shown to have significant impact in applications [11,12], and there is also a library, NOBLE [13], containing many implementations of non-blocking data structures.

One of the most common synchronization primitives used in lock-free synchronization is the *Compare-And-Swap* instruction (also denoted CAS), which atomically executes the steps described in Fig. 1. CAS is available in e.g. SPARC processors. Another primitive which is equivalent with CAS in synchronization power 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*. Another useful primitive is *Fetch-And-Add*

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

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

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

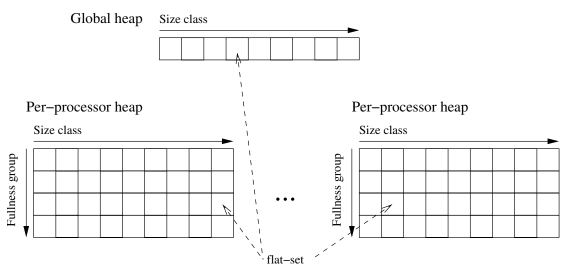
(also denoted *FAA*), described in Fig. 1. FAA can be simulated in software using CAS or LL/SC when it is not available in hardware.

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 [14]. 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 [15]. An appropriate garbage-collection mechanism, such as [16], can also solve the problem.

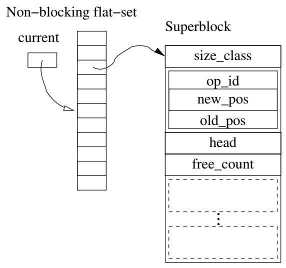
3 The New Lock-Free Memory Allocator: Architecture

The architecture of our lock-free memory allocator is inspired by Hoard[2], 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 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 bins based on *fullness*(cf. Fig. 2(a)). As the fullness of a particular superblock changes it is moved between the groups. A memory request (*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.



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



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

Fig. 2. The architecture of the memory allocator

4 Managing Superblocks: The Bounded Non-blocking Flat-Sets

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. [17,18,14], 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 *bounded non-blocking flat-set*, supporting conventional "internal" operations (*Get_Any* and *Insert* item) as well as "inter-object" operations, for moving an item from one flat-set to another.

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: (i) the 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; (ii) 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 the operations of the lock-free flat-set data structure and the lock-free algorithm, *move*, which is used to implement the "inter-object" operation for moving a reference to a superblock from one shared variable (pointer) to another satisfying the above requirements.

4.1 Operations on Bounded Non-blocking Flat-Sets

A bounded non-blocking flat-set provides the following operations: (i) *Get_Any*, which returns any item in the flat-set; and (ii) *Insert*, which inserts an item into the flat-set. An item can only reside 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 consists of an array of M shared locations set.set[i], each capable of holding a reference to a superblock, and a shared index variable set.current. The data structure and operations are shown in Fig. 3 and are briefly described below.

The index variable set.current is used as a *marker* to speed up flat-set operations. It contains a bit used as an *empty flag* for the flat-set and a index field that is used as the starting point for searches, both for items and for free slots. The *empty flag* is set by a *Get_Any* operation that discovers that the flat-set is empty, so that subsequent *Get_Any* operations know this; the *Insert* operation and successful *Get_Any* operations clear the flag. The *empty flag* is not always set when the flat-set is empty as superblocks can be moved away from the flat-set at any time, but it is always cleared when the flat-set is nonempty. The *Insert* operation scans the array set.set[i] forward from the position marked by set.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 in detail below). The *Get_Any* operation first reads set.current to check the *empty flag*. If the *empty flag* is set, *Get_Any* returns immediately, otherwise it starts to scan the array set.set[i] backwards from the position marked by set.current, until it finds a location that contains a superblock reference. If a *Get_Any* operation has scanned the whole set.set[i] array without finding a reference it will try to set the *empty flag* for the flat-set. This is done at line G13 using CAS and will succeed if and only if set.current has not been changed since it was read at line G2. This indicates that the flat-set is empty so *Get_Any* sets the empty flag and returns failure. If, on the other hand, set.current has changed between line G2 and G13, then either an *Insert* is in progress or has finished during the scan (line I6 and I9) or some other *Get_Any* has successfully found a superblock during this time (line G10), so *Get_Any* should redo the scan. To facilitate moving of superblocks between flat-sets via *Insert* *Get_Any* returns both a superblock reference and a reference to the shared location containing it.

4.2 How to Move a Shared Reference: Moving Items Between Flat-Sets

The algorithm supporting the operation *Move* moves a superblock reference sb from a shared location from to a shared location to. The target location (i.e. to) is known via the Insert operation. The algorithm requires the superblock to contain an auxiliary variable mv_info with the fields op_id, new_pos and old_pos and all superblock references to have a version field (cf. Fig 3).

A move operation may *succeed* by returning SB_MOVED_OK or *fail* (abort) by returning SB_MOVED (if the block has been moved by another overlapping move) or SB_NOT_MOVED (if the to location is occupied). It will succeed if it is completed successfully by the thread that initiated it or by a helping thread. To ensure the lock-free property, the move operation is divided into a number of atomic suboperations. 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 identical to steps 2 - 4 of the move operation described below.

```
type superblock_ref {// fits in one machine word
      ptr : integer_16; version : integer_16;};
/* superblock_ref utility functions. */
function pointer(ref : superblock_ref)
return pointer to superblock
function version(ref : superblock_ref)
return integer_16
function make_sb_ref(sb : pointer to superblock,
         op_id : integer_16) return superblock_ref
type flat-set_info {// fits in one machine word
index : integer; empty : boolean; version : integer;};
function Get_Any(set : in out flat-set,
         sb : in out superblock_ref,
         loc : in out pointer to superblock_ref)
return status
         i, j : integer; old_current : flat-set_info;
begin
G1   loop
G2     old_current := set.current;
G3     if old_current.empty then
G4       return FAILURE;
G5     i := old_current.index;
G6     for j := 1 .. set.size do
G7       sb := set.set[i];
G8       if pointer(sb) /= null then
G9         loc := &set.set[i];
G10        set.current := (i, false);// Clear empty flag
G11        return SUCCESS;
G12      if i == 0 then i := set.size - 1 else i–;
G13    if CAS(&set.current, old_current,
G14      (old_current.index, true)) == old_current
G15    then
G16      return FAILURE;
function Insert(set : in out flat-set,
         sb : in superblock_ref,
         loc : in out pointer to superblock_ref)
return status
         i, j : integer;
begin
I1   loop
I2     i := (set.current.index + 1) mod set.size;
I3     for j := 1 .. set.size do
I4       while pointer(set.set[i]) == null do
I6         set.current := (i, false);
I7         case Move(sb, loc, &set.set[i]) is
I8         when SB_MOVED_OK:
I9           set.current := (i, false);
I10          loc := &set.set[i];
I11          return SB_MOVED_OK;
I12        when SB_MOVED:
I13          return SB_MOVED;
I14        when others:
I15        end case;
I16      i := (i + 1) mod set.size;
I17    if set.set not changed since prev. iter. then
I18      return FAILURE; /* The flat-set is full. */
function Get_Block(sb : in superblock_ref)
return block_ref
         nb, nh : block_ref;
begin
GB1 nb := sb.freelist_head;
GB2 while nb /= null do
GB3   nh := CAS(&sb.freelist_head,
GB4        nb, nb.next);
GB5   if nh == nb.next then
GB6     FAA(&sb.free_block_cnt, -1);
GB7     break;
GB8   nb := sb.freelist_head;
GB9 return nb;
```

```
structure flat-set {
      size : constant integer; current : flat-set_info;
      set[size] : array of superblock_ref;};
structure superblock {
      mv_info : move_info;
      freelist_head : pointer to block;
      free_block_cnt : integer;};
structure move_info {
      op_id : integer_16;
      new_pos : pointer to superblock_ref;
      old_pos : pointer to superblock_ref;};
type block_ref { // fits in one machine word
      offset : integer_16; version : integer_16;};
procedure Put_Block(sb : in superblock_ref,
                  bl : in block_ref)
        oh : block_ref;
begin
PB1 loop
PB2   bl.next := sb.freelist_head;
PB3   oh := CAS(&sb.freelist_head, bl.next, bl)
PB4   if oh == bl.next then break;
PB5 FAA(&sb.free_block_cnt, 1);
function Move(sb : in superblock_ref,
         from : in pointer to superblock_ref,
         to : in pointer to superblock_ref)
return status
      new_op, old_op : move_info;
      cur_from : superblock_ref;
begin
M1  /* Step 1: Initiate move. */
M2  loop
M3    old_op := Load_Linked(&sb.mv_info);
M4    cur_from := *from;
M5    if pointer(cur_from) /= pointer(sb) then
M6      return SB_MOVED;
M7    if old_op.from == null then // No cur. operation.
M8      new_op := (version(cur_from), to, from);
M9      if Store_Conditional(&sb.mv_info, new_op)
M10     then break;
M11   else
M12     Move_Help(make_sb_ref(pointer(sb), old_op.op_id),
M13         old_op.old_pos,
M14         old_op.new_pos);
M15 return Move_Help(cur_from, from, to);
function Move_Help(sb : in superblock_ref,
         from : in pointer to superblock_ref,
         to : in pointer to superblock_ref)
return status
      old, new, res : superblock_ref; mi : move_info;
begin
H1  /* Step 2: Update "TO". */
H2  old := *to;
H3  new := make_sb_ref(sb, version(old) + 1);
H4  res := CAS(to, make_sb_ref(null, version(old)), new)
H5  if pointer(res) /= pointer(sb) then
H6    /* To is occupied, abandon this operation. */
H7    mi := Load_Linked(&sb.mv_info);
H8    if mi == (version(sb), to, from) then
H9      mi := (0, from, null);
H10     Store_Conditional(&sb.mv_info, mi);
H11   return SB_NOT_MOVED;
H12 /* Step 3: Clear "FROM". */
H13 CAS(from, sb, make_sb_ref(null, version(sb) + 1));
H14 /* Step 4: Remove operation information.*/
H15 mi := Load_Linked(&sb.mv_info);
H16 if mi == (version(sb), to, from) then
H17   mi := (0, to, null);
H18   Store_Conditional(&sb.mv_info, mi);
H19 return SB_MOVED_OK;
```

Fig. 3. The flat-set data structures and operations *Get_Any* and *Insert*, the superblock data structures and operations *Get_block* and *Put_Block* and the superblock *Move* operation.

1. A *Move*(sb, from, to) is initiated by atomically *registering* the operation. This is done by *Load-Linked/Store-Conditional* operations which sets sb.mv_info to (version(sb), to, from) iff the read value of sb.mv_info.from was null, which indicates that there are no ongoing move of this superblock. If the read value of sb.mv_info.op_id was nonzero, then 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 is abandoned and returns SB_MOVED.
2. If the current value of to is null then to is set to point to the superblock while simultaneously increasing its version. Otherwise this move is abandoned since the destination is occupied and the information about the move is removed from the superblock (as in step 4) and SB_NOT_MOVED is returned.
3. If from still contains the expected superblock reference (i.e. if no one else has helped this move) from is set to null while increasing its version.
4. If the move information is still in the superblock (i.e. if no one else has helped the move to complete) it is removed and the *move* operation returns SB_MOVED_OK.

In the presentation here and in the pseudo-code in Fig. 3 we use the atomic primitive CAS to update shared variables that fit in a single memory word, but other atomic synchronization primitives, such as LL/SC could be used as well. The auxiliary mv_info variable in a superblock might need to be larger than one word. To handle that we use the lock-free software implementation of Load-Linked/Store-Conditional for large words by Michael [19] which can be implemented efficiently from the common single-word CAS. Some hardware platforms provide a CAS primitive for words twice as wide as the standard word size, which may also be used for this.

The correctness proof of the algorithm is omitted due to space constraints; it can be found in [20].

5 Managing the Blocks Within a Superblock

The allocatable memory blocks within each superblock are kept in a lock-free IBM free-list [21]. The IBM free-list is essentially a lock-free stack implemented from 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 Fig. 3. The free blocks counter, sb.free_block_cnt, is used to estimate the fullness of a superblock.

6 Performance Evaluation

Systems. The performance of the new lock-free allocator has been measured on a two multiprocessor systems: (i) an UMA Sun Enterprise 450 with 4 400MHz

UltraSPARC II (4MB L2 cache) processors running Solaris 9; (ii) a ccNUMA SGI Origin 3800 with 128 (only 32 could be reserved) 500Mhz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

Benchmarks. We used three common benchmarks to evaluate our memory allocator: The **Larson** [2,1,22] benchmark simulates a multi-threaded server application which 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 (30 on the Origin 3800 due to job duration limits) second runs for each number threads.

The **Active-false** and **passive-false** [2,1] 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 read and write 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 which 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.

Implementation. [1] In our memory allocator we use the CAS primitive (implemented from the hardware synchronization instructions available on the respective system) for our lock-free operations. To avoid ABA problems we use the version number solution ([14], cf. section 2). We use 16-bit version numbers for the superblock references in the flat-sets, since for a bad event (i.e. that a CAS of a superblock reference succeeds when it should not) to happen not only must the 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 64KB to have 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, which would further reduce internal fragmentation at the cost of a larger fixed space overhead due to the preallocated flat-sets for each size-class. Blocks larger than 32KB 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

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

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).

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 (version 3.0.2). To the best of our knowledge, Hoard is not available for ccNUMA SGI IRIX platform.

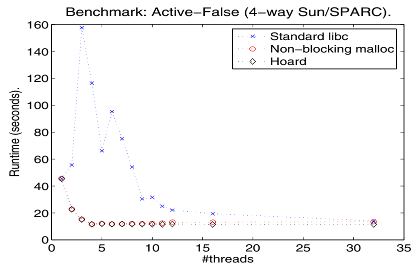
The benchmarks are intended to test scalability, fragmentation and false-sharing, which are the evaluation criteria of a good concurrent allocator, as explained in the introduction. When performing these experiments our main goal was not to optimize the performance of the lock-free allocator, but rather to examine the benefits of the lock-free design itself. There is plenty of room for optimization of the implementation.

The results from the two false-sharing benchmarks, shown in Fig. 4, show that our memory allocator, 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. Our allocator shows consistent behaviour as the number of processors and memory architecture changes.

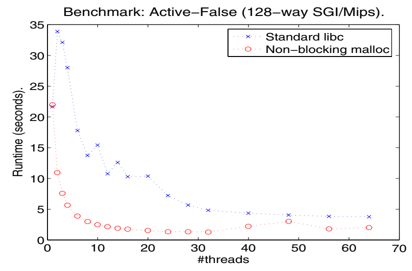
The throughput results from the Larson benchmark, shown in Fig. 4, show that our lock-free memory allocator has good scalability, not only in the case of full concurrency (where Hoard also shows extremely good scalability), but also when the number of threads increases beyond the number of processors. In that region, Hoard's performance quickly drops from its peak at full concurrency (cf. Fig. 4(e)). We can actually observe more clearly the scalability properties of the lock-free allocator in the performance diagrams on the SGI Origin platform (Fig. 4(f)). There is a linear-style of throughput increase when the number of processors increases (when studying the diagrams recall we have up to 32 processors available on the Origin 3800). Furthermore, when the load on each processor increases beyond 1, the throughput of the lock-free allocator stays high. In terms of absolute throughput, Hoard is superior to our lock-free allocator, at least on the Sun 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 is further enhanced by lock-free synchronization.

The results with respect to memory consumption, Fig. 4(g,h), show that for the Larson benchmark the memory usage (and thus fragmentation) of the non-blocking allocator stays at a similar level to Hoard and that the use of per-processor heaps with thresholds, while having a larger overhead than the "libc" allocator, still have almost as good scalability with respect to memory utilization as a single heap allocator.

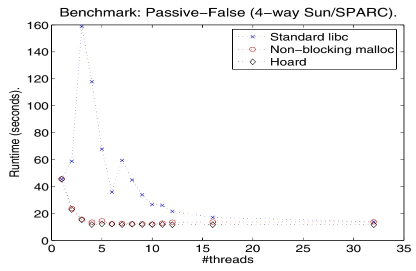
Moreover, that our lock-free allocator shows a very similar behaviour in throughput on both the UMA and the ccNUMA systems is an indication that



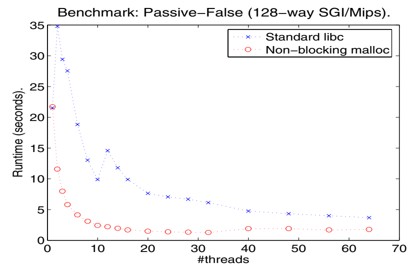
(a) Active-False: Sun SPARC 4 CPUs



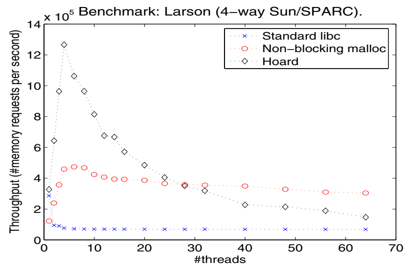
(b) Active-False: SGI MIPS 32(/128) CPUs



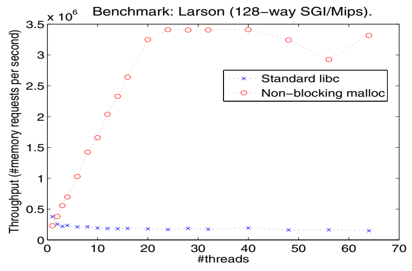
(c) Passive-False: Sun SPARC 4 CPUs



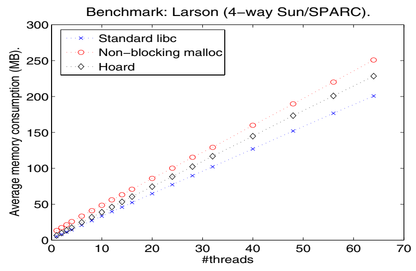
(d) Passive-False: SGI MIPS 32(/128) CPUs



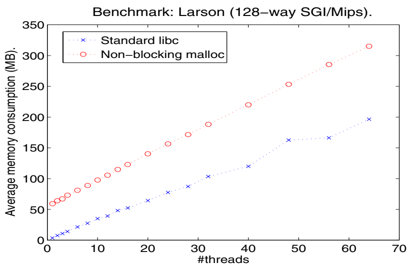
(e) Throughput: Sun SPARC 4 CPUs



(f) Throughput: SGI MIPS 32(/128) CPUs



(g) Memory consumption: Sun SPARC 4 CPUs



(h) Memory consumption: SGI MIPS 32(/128) CPUs

Fig. 4. Results from the benchmarks

there are few contention hot-spots, as these tend to cause much larger performance penalties on NUMA than on UMA architectures.

7 Other Related Work

Recently Michael presented a lock-free allocator [23] which, like our contribution, is loosely based on the Hoard architecture. Our work and Michael's have been done concurrently and completely independently, an early version of our work is in the technical report [20]. Despite both having started from the Hoard architecture, we have used two different approaches to achieve lock-freeness. 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 [23], in the experiments carried out on a 16-way POWER3 platform. By further studying the allocators, it is relevant to note that: Our allocator and Hoard keep all partially filled superblocks in their respective per-processor heap while the allocator in [23] does not and this may increase the potential for inducing false-sharing. Our allocator and Hoard also keep the partially filled superblocks sorted by fullness and not doing so, like the allocator in [23] 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 becomes empty and thus eligible for reuse. The allocator in [23], 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. As this is work that has been carried out concurrently and independently with our contribution, we do not have any measurements of the impact of the above differences, however this is interesting to do as part of future work, towards further optimization of these allocators.

Another allocator which reduces the use of locks is LFMalloc [7]. 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. However, a comparative study with that approach can be worthwhile, when it becomes available for experimentation.

Earlier related work is the work on non-blocking operating systems by Massalin and Pu [8,24] and Greenwald and Cheriton [4,25]. They, however, made extensive use of the *2-Word-Compare-And-Swap* primitive in their algorithms.

This primitive can update two arbitrary memory locations in one atomic step but is not available in current systems and expensive to do in software.

8 Discussion

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 the system. To the best of our knowledge, this, together with the allocator which was independently presented in [23] 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.

Acknowledgements

We would like to thank Håkan Sundell for interesting discussions on non-blocking methods and Maged Michael for his helpful comments on an earlier version of this paper.

References

1. Berger, E.D.: Memory Management for High-Performance Applications. PhD thesis, The University of Texas at Austin, Department of Computer Sciences (2002)
2. 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. (2000) 117–128
3. 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 (1993) 261–270 Extended abstract.
4. Greenwald, M., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: Operating Systems Design and Implementation. (1996) 123–136
5. Herlihy, M.: Wait-free synchronization. ACM Transaction on Programming and Systems **11** (1991) 124–149
6. Rinard, M.C.: Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. ACM Transactions on Computer Systems **17** (1999) 337–371
7. Dice, D., Garthwaite, A.: Mostly lock-free malloc. In: ISMM'02 Proc. of the 3rd Int. Symp. on Memory Management. ACM SIGPLAN Notices, ACM Press (2002) 163–174
8. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91 (1991)
9. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (1990) 463–492

10. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. Journal of Parallel and Distributed Computing **62** (2002) 766–791
11. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: Proc. of the ACM SIGMETRICS 2001/Performance 2001, ACM press (2001) 320–321
12. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In: Proc. of the 3rd ACM Workshop on Software and Performance (WOSP'02), ACM press (2002) 55–67
13. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. Lecture Notes in Computer Science, Springer Verlag (2002)
14. 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), ACM (1995) 214–222
15. 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, ACM Press (2001) 134–143
16. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proc. of the 21st annual symp. on Principles of distributed computing, ACM Press (2002) 21–30
17. Harris, T.L.: A pragmatic implementation of non-blocking linked lists. In: Proc. of the 15th Int. Conf. on Distributed Computing, Springer-Verlag (2001) 300–314
18. 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), ACM Press (2002) 73–82
19. Michael, M.M.: Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In: Proc. of the 18th Int. Conf. on Distributed Computing (DISC). (2004)
20. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Allocating memory in a lock-free manner. Technical Report 2004-04, Computing Science, Chalmers University of technology (2004)
21. IBM: IBM System/370 Extended Architecture, Principles of Operation. (1983) Publication No. SA22-7085.
22. 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, ACM Press (1998) 176–185
23. Michael, M.: Scalable lock-free dynamic memory allocation. In: Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation. ACM SIGPLAN Notices, ACM Press (2004)
24. Massalin, H.: Synthesis: An Efficient Implementation of Fundamental Operating System Services. PhD thesis, Columbia University (1992)
25. Greenwald, M.B.: Non-blocking synchronization and system design. PhD thesis, Stanford University (1999)