

Efficient Heap Management for Declarative Data Parallel Programming on Multicores

Clemens Grelck^{1,2} and Sven-Bodo Scholz¹

¹ University of Hertfordshire
Department of Computer Science
Hatfield, United Kingdom
`c.grelck,sscholz@herts.ac.uk`

² University of Lübeck
Institute of Software Technology and Programming Languages
Lübeck, Germany
`grelck@isp.uni-luebeck.de`

Abstract. Declarative data parallel programming for shared memory multiprocessor systems implies paradigm-specific demands on the organisation of memory management. As a key feature of declarative programming implicit memory management is indispensable. Yet, the memory objects to be managed are very different from those that are predominant in general-purpose functional or object-oriented languages. Rather than complex structures of relatively small items interconnected by references, we are faced with large chunks of memory, usually arrays, which often account for 100s of MB each. Such sizes make relocation of data or failure to update arrays in-place prohibitively expensive.

To address these challenges of the data parallel setting, the functional array language SAC employs continuous garbage collection via reference counting combined with several aggressive optimisation techniques. However, we have observed that overall memory performance does not only rely on efficient reference counting techniques, but to a similar extent on the underlying memory allocation strategy. As in the general memory management setting we can identify specific demands of the declarative data parallel setting on heap organisation.

In this paper, we propose a heap manager design tailor-made to the needs of concurrent executions of declarative data parallel programs whose memory management is based on reference counting. We present runtime measurements that quantify the impact of the proposed design and relate it to the performance of several different general purpose heap managers that are available in the public domain.

1 Introduction

The ubiquity of multicore processors has made parallel processing a mainstream necessity rather than a niche business [1]. Declarative languages may benefit from this paradigm shift as their problem-oriented nature and the absence of side-effects facilitate (semi-)implicit parallelisation or at least help in explicit

parallelisation on a high level of abstraction. One approach is to exploit data parallelism on arrays as pursued by functional languages such as SISAL [2], NESL [3] or SAC [4, 5] and recently also adopted by HASKELL [6].

Declarative processing of large arrays of data has a specific challenge known as the aggregate update problem [7]. The (otherwise very desirable) absence of side-effects prevents incremental in-place updates of array element values as they are abundant in imperative array processing. A naive solution requires to copy the entire array which quickly becomes prohibitive with increasing array size. Efficient declarative array processing requires a mechanism that determines when it is safe to update an array in place and when not. The decision to reuse memory associated with an argument array to store a result array also depends on the characteristics of the operation itself. With the prevailing tracing garbage collectors [8] this is generally not feasible. The only way to achieve in-place updates of arrays in main-stream functional languages seems to be making arrays stateful, either by language semantics as in ML [9] or through a proper functional integration of states via monads in Haskell [10] or uniqueness typing in Clean [11]. However, these approaches also enforce a very non-declarative style of programming as far as arrays are concerned [12].

To mitigate the aggregate update problem without compromising a declarative style of programming, SISAL, NESL and SAC use reference counting [8] as a basis for memory management. At runtime each array is associated with a reference counter that keeps track of the number of active references to an array. Reference counting allows us to release unused memory as early as possible and to update arrays destructively in suitable operations provided that the reference counter indicates no further pending references. Strict evaluation generally tends to reduce the number of pending references; it seems to be necessary to make this memory management technique effective.

Reference counting does have its well known downsides, e.g. memory overhead, de-allocation cascades or the difficulty to identify reference cycles. However, in the particular setting of array processing they are less severe: individual chunks of memory are relatively large, data is not deeply structured, and cyclic references typically precluded by language semantics. Static analysis can be used effectively to reduce the overhead inflicted by reference counter updates, to identify opportunities for immediate memory and even data reuse and to make reuse and de-allocation decisions already at compile time. Surveys of such techniques can be found in [13, 14].

Unlike most forms of tracing garbage collection, reference counting is just half the story. It leads to a sequence of allocation and de-allocation requests that still need to be mapped to the linear address space of a process by some underlying mechanism. This is often considered a non-issue because low-level *memory allocators* have been available for decades for explicit heap management in machine-oriented languages (see [8] for a survey). Yet, many (SAC) applications spend a considerable proportion of their execution time in the memory allocator. As soon as runtime performance is a criterion for the suitability of a declarative language for a certain application domain, every (milli-)second counts, and im-

improvements in the interplay between the reference counting mechanism and the underlying heap manager can have a significant impact on overall performance.

We propose a heap manager that is tailor-made for the needs of multithreaded declarative array processing and reference counting. Our design aims at outperforming existing allocators by exploiting three distinct advantages: Firstly, we adapt allocation/de-allocation strategies to the specific characteristics of array processing and reference counting. For example, we expect a large variety in the size of requested memory chunks from very small to very large, but only a relatively small number of different chunk sizes. Furthermore, reference counting (unlike manual memory management) guarantees that any allocated chunk of memory is released eventually. Consequently, overhead may arbitrarily be split between allocation and de-allocation operations.

Secondly, we use a rich (internal) interface between reference counting mechanism and allocator, that allows us to exchange extra information and let our allocator benefit from static code analysis. In contrast, the standard interfaces between applications and allocators are very lean (e.g. `malloc` and `free` in C or `new` and `delete` in C++) and restrict the flow of information from the application to the allocator.

Thirdly, we tightly integrate our allocator with the multithreaded runtime system [15]. As soon as threads run truly simultaneously on a multiprocessor system or multicore processor, access to heap-internal data structures requires synchronisation, which adversely affects performance and may even serialise program execution through the back door. While some general-purpose memory allocators do take multithreading into account [16–18], they need to deal with a wide range of multithreaded program organisations reasonably well. In contrast, the multithreaded runtime organisation of compiler-parallelised code typically is rather restricted. For example, the automatic parallelisation feature of the SAC compiler [15] results in a runtime program organisation where the number of threads is limited by the number of hardware execution units, certain threads are a-priori known to execute exclusively and memory allocated by one thread is known to be de-allocated by the same thread.

The contributions of this paper are

- to quantify the impact of heap management on compiler-parallelised declarative array processing code,
- to identify specific aspects of heap management that allow a tailor-made heap manager to exploit distinctive advantages over off-the-shelf implementations,
- to propose the design of a tailor-made heap manager in the context of SAC
- and to evaluate the benefit of using private heap management.

The remainder of the paper is organised as follows. In Section 2 we illustrate the problem using a microbenchmark. In Section 3 we outline the design of the SAC private heap manager and demonstrate its impact on overall runtime performance in Section 4. Finally, Section 5 outlines some related work before we draw conclusions in Section 6.

2 Problem illustration

We illustrate the impact of memory allocators on overall runtimes by means of the small SAC example program shown in Fig. 1. The program emulates a memory allocation and de-allocation pattern typical for many data parallel applications: a data parallel operation is repetitively applied to a vector **A** of length $[10000000/X]$, as generated by the library function `mkarray`. The data parallel operation within the `for`-loop is defined in terms of a `with`-construct, a SAC array comprehension. For each element of **A**, it recomputes its value by first allocating a vector of length **X** and subsequently summing these elements up.³ This creates a very common memory demand pattern: within a data parallel section each element computation requires the creation of a temporary array and, hence, some memory allocation and de-allocation. Furthermore, all allocations are of the same size which, again, is typical for many scientific applications such as those investigated in [20, 21, 19].

```
int main()
{
  A = mkarray( [10000000/X], 0);
  for (i=0; i<50; i+=1) {
    A = with {
      (. <= [idx] <= .) : A[idx] + sum( mkarray( [X], idx));
    }: modarray( A);
  }
  return( sum( A));
}
```

Fig. 1. Example SAC program

It should be noted here that we carefully restricted compiler optimisations in order to ensure that for this simple example allocations and de-allocations of intermediate vectors of length **X** effectively happen at runtime. Normally, the SAC compiler would fold the reduction operation (`sum`) and the build operation (`mkarray`) [22]. Even if that failed, memory management optimisations would pre-allocate memory for the temporary vector outside of the `with`-construct [14].

In order to quantify the impact of the memory allocator, we measure the runtimes of the program from Fig. 1 with varying values for **X**: 1000, 250, 100 and 25. The definition of the vector lengths of **A** and that of the intermediate vectors guarantee that, irrespective of the value of **X**, we always perform 500 million additions. This allows us to observe the impact of the allocator as the decrease of **X** corresponds to an increase in the number of memory allocations (and de-allocations). We observe the effect of 500.000, 2 million, 5 million and 20 million memory allocations and de-allocations, respectively. Fig. 2 shows program execution times of our example program on a 12-processor SUN Ultra Enterprise 4000 multiprocessor using the standard SOLARIS memory allocator.

³ For any details about the `with`-construct as well as about the purely functional semantics of this rather imperative looking code see [4, 5].

Additional experiments using the Gnu allocator coming with LINUX essentially led to the same observations.

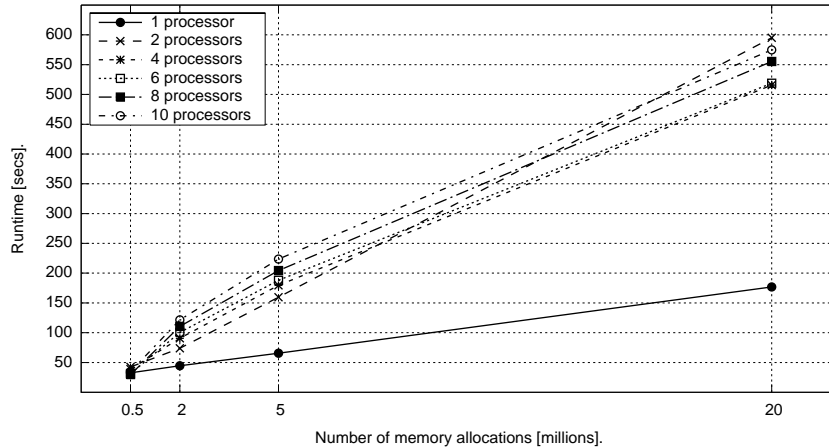


Fig. 2. Execution times of SAC program shown in Fig. 1

Focussing on single processor performance first, we can see that despite a fixed amount of overall computations, runtimes grow linearly with the number of memory allocations. Although this is not surprising in principle, the growth rate in fact is. From the measurements we can deduce that the pure computing time is roughly 30 seconds. Taking into account that even in the most allocation intensive case we have about 25 additions per allocation plus a certain loop overhead, the measured memory management overhead (roughly 140 seconds!) is extremely high. Obviously, the general-purpose allocators cannot benefit from the simple case of alternating allocations and de-allocations of always the same amount of memory. Our observations allow the conclusion that for these application scenarios the memory allocator is the key to overall runtime performance, whereas improvements in code generation and the compiler in general may easily be ineffective.

Looking at multiprocessor performance we observe that increasing parallelism yields substantial slowdowns, although our microbenchmark is almost embarrassingly parallel and the total workload is substantial. The reason for this behaviour lies in an inherently sequential design of the memory allocator, which seemingly has been adapted for multithreaded execution more or less naively by locking operations. This solution proves to be unsuitable for memory management intensive data parallel applications like our microbenchmark.

The bottom line of these observations is twofold: Firstly, a genuinely multi-threaded memory allocator design that reduces locking to a minimum is indispensable. Secondly, data parallel applications may spend considerable proportions of their overall runtime in memory management operations, making the memory allocator a prime target for optimisation.

3 Design of a SAC-specific heap manager

The memory organisation used by the SAC *private heap manager* (or SACPHM for short) is characterised by a hierarchy of multiple nested heaps, as illustrated in Fig. 3. At the top of the hierarchy is a single *global heap*, which controls the entire address space of the process. It may actually grow or shrink during program execution, as additional memory is requested from the operating system or unused memory is released to it.

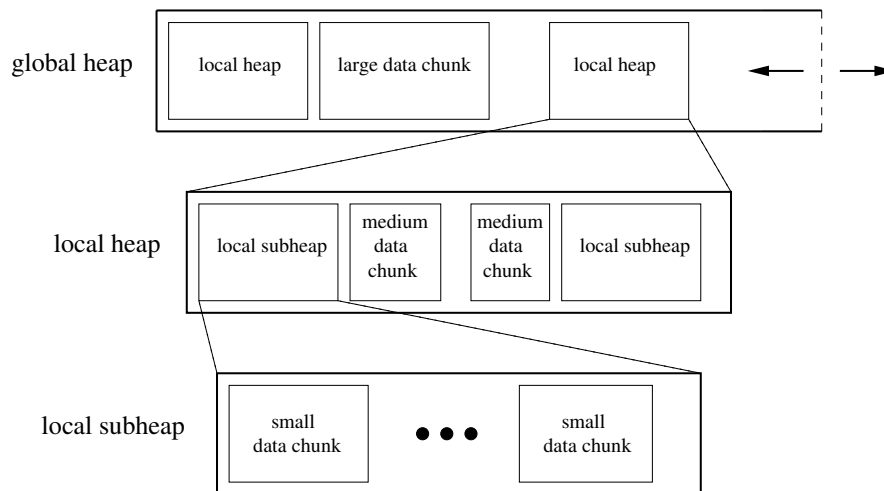


Fig. 3. Memory organisation using multiple nested heaps.

However, only very large chunks of memory are directly allocated from the global heap. Memory requests below some threshold size are satisfied from one of possibly several *local heaps*. A local heap is a contiguous piece of memory with a fixed size, which in turn is allocated from the global heap. Grouping together memory chunks of similar sizes tends to have a positive impact on memory fragmentation. Once the capacity of a local heap is exhausted, an additional local heap is allocated from the global heap.

In multithreaded execution each thread is associated with its individual local heap(s). This organisation addresses both scalability and false sharing [23] issues: Each thread may allocate and de-allocate arrays of up to a certain size without interfering with other threads. Small amounts of memory are guaranteed to be allocated from different parts of the address space if requested by different threads. Furthermore, housekeeping data structures for maintaining local heaps are kept separate by different threads. This allows us to keep them in processor-specific cache memories without invalidation by a cache coherence mechanism.

Three properties of the SAC multithreaded runtime system [15] are essential to make this design feasible. Firstly, the number of threads is limited by

the number of parallel processing units available (i.e. rather small), and thread creation/termination are limited to program startup/termination. Hence, it becomes feasible to a-priori associate each thread with some (non-negligible) local heap memory. Secondly, the data parallel approach encourages (though does not enforce) applications where the memory demands of the individual threads are rather similar. As a consequence, we may pre-allocate some heap memory for each thread at program startup when initialising the heap. Thirdly, the runtime system guarantees that any memory allocated by one thread is eventually released by the same thread. This restriction keeps thread-private local heaps in a coherent state throughout program execution. A general-purpose memory allocator cannot make such assumptions. Instead, it should work reasonably well both for large numbers of threads and heterogeneous allocation behaviour, including threads that mostly allocate memory while others mostly de-allocate memory following a producer/consumer pattern.

In our allocator design only accesses to the global heap may require synchronisation. The word *may* here is motivated by another restriction of the multithreaded runtime system: Program execution is organised as a sequence of alternating single-threaded and multithreaded supersteps [15]. Any memory management request to the global heap issued in a single-threaded superstep proceeds without synchronisation. Our experience is that very large arrays, allocated from the global heap, are predominantly allocated (and de-allocated) during single-threaded execution for subsequent multithreaded initialisation of elements. In practice, locking is reduced to the very rare case when the initially pre-allocated local heap of some thread is exhausted and needs to be extended during program execution.

The hierarchical memory organisation is repeated once again on the level of local heaps. Only medium-sized memory requests are directly satisfied by one of the local heaps. Allocations of memory chunks below a certain size are again grouped together in *local subheaps*, which in turn are allocated from local heaps. The distinction between heaps and subheaps is mainly motivated by different housekeeping mechanisms. In local heaps as well as in the global heap we allocate differently sized chunks from a contiguous address space, whereas subheaps use a fixed-size chunk allocation scheme. The latter allows us to quickly identify a suitable available memory chunk. Likewise, marking chunks as available or allocated inflicts very little time overhead. For larger chunks of memory, however, the resulting internal fragmentation is not tolerable. Therefore, we use a more expensive variable chunk size scheme above a certain threshold size. This scheme keeps track of chunk sizes and, in particular, splits larger parts of contiguous memory into pieces to accommodate allocation requests and coalesces adjacent free chunks of memory to conversely form larger chunks.

Both fixed and variable chunk size heap organisation schemes are well studied [8]. Nonetheless, we can customise our concrete implementation to specific aspects of data parallel array processing. In this context we often observe that applications only use a very restricted number of differently sized arrays (although the range of different array sizes may be very large). Therefore, we assume lo-

quality of time in similar way as cache memories do: If we de-allocate a memory chunk of some size, we consider it likely that we need to allocate a memory chunk of the same size very soon thereafter. Consequently, we employ a deferred coalescing scheme that only reconstructs larger chunks of memory if a concrete allocation request cannot be satisfied from the immediately available resources. Deferred coalescing moves overhead from de-allocations to allocations, which is not so desirable for general-purpose allocators because the number of allocations typically exceeds the number of de-allocations. However, with the allocator being a backend for the reference counting mechanism it is guaranteed that any allocated chunk of memory is released eventually. Hence, it doesn't matter whether we concentrate effort in allocation or in de-allocation operations.

		thread 0	thread 1	• • •	thread T - 1
local subheap	size range 0	arena 0 / 0	arena 0 / 1	• • •	arena 0 / T-1
	size range 1	arena 1 / 0	arena 1 / 1	• • •	arena 1 / T-1
	• • •	• • •	• • •	• • • • •	• • •
	size range K - 1	arena K-1 / 0	arena K-1 / 1	• • •	arena K-1 / T-1
local heap	size range K	arena K / 0	arena K / 1	• • •	arena K / T-1
	size range K + 1	arena K+1 / 0	arena K+1 / 1	• • •	arena K+1 / T-1
	• • •	• • •	• • •	• • • • •	• • •
	size range M - 1	arena M-1 / 0	arena M-1 / 1	• • •	arena M-1 / T-1
global heap	size range M	arena M / 0			
	size range M + 1	arena M+1 / 0			
	• • •	• • •			
	size range N - 1	arena N-1 / 0			

Fig. 4. Matrix of allocation arenas

The hierarchy of nested heaps requires a coarse-grained classification of memory requests into small, medium and large chunks. In order to accelerate searching for appropriate memory chunks, we effectively use a finer-grained classification and introduce an entire matrix of *allocation arenas*, as illustrated in Fig. 4. Allocation arenas represent the basic organisational entities for heap manage-

ment. Each request for allocation or de-allocation of memory first identifies the appropriate allocation arena, which provides

- a list of available, appropriately sized chunks of memory,
- an allocation strategy,
- a de-allocation strategy and
- a backup strategy to obtain more memory.

Arena identification is a binary decision problem whose depth is logarithmic in the number of allocation arenas. However, in practice the exact amount of memory needed to represent some array is often known statically to the program and, hence, also the appropriate allocation arena can be determined at compile time. It is the restricted interface of (e.g. `malloc` and `free`) that prevents general-purpose memory allocators from taking advantage of static chunk size knowledge. In contrast, our integrated solution employs a much richer interface between reference counting mechanism and backend heap manager that already selects the allocation arena at compile time whenever possible.

Likewise, the tight integration of reference counting and heap management permits specific optimisations. For example, at runtime any SAC array is represented by a (typically large) data vector and a (always small) descriptor that accommodates the reference counter and dynamic shape information. This design enables the seamless flow of arrays between program parts written in SAC and program parts written in other languages using the SAC foreign language interface. The separation of SAC-specific structural and administrative information from actual data, unfortunately, also requires two allocations and two de-allocations per array. In most cases allocation and de-allocation of data vector and descriptor are made in conjunction, but this is not guaranteed and upon de-allocation it is undecidable whether a data vector has been allocated within the realm of SAC or outside. However, our private heap manager makes exactly this decidable. This optimisation alone accounts for about 10% execution time improvement through a range of applications.

4 Evaluation

We have repeated the initial experiment described in Section 2 with the SAC private heap manager and three off-the-shelf multithreaded memory allocators:

- `MTMALLOC` [16] is a replacement for the standard memory allocator, which is provided by SUN itself starting with SOLARIS-7.
- `PTMALLOC` [17] adapts the serial allocator `DLMALLOC` [24] for use with multithreaded applications. It also employs multiple heaps to reduce contention, but there is no static mapping of heaps to threads. Upon each memory request, threads search for an unlocked heap, lock the heap, and then apply the serial allocation/de-allocation techniques adopted from `DLMALLOC`.
- `HOARD` [18] seems to be the most recent development in multithreaded memory managers. It maps a possibly large number of threads to a generally much smaller number of separate heaps by means of hashing techniques.

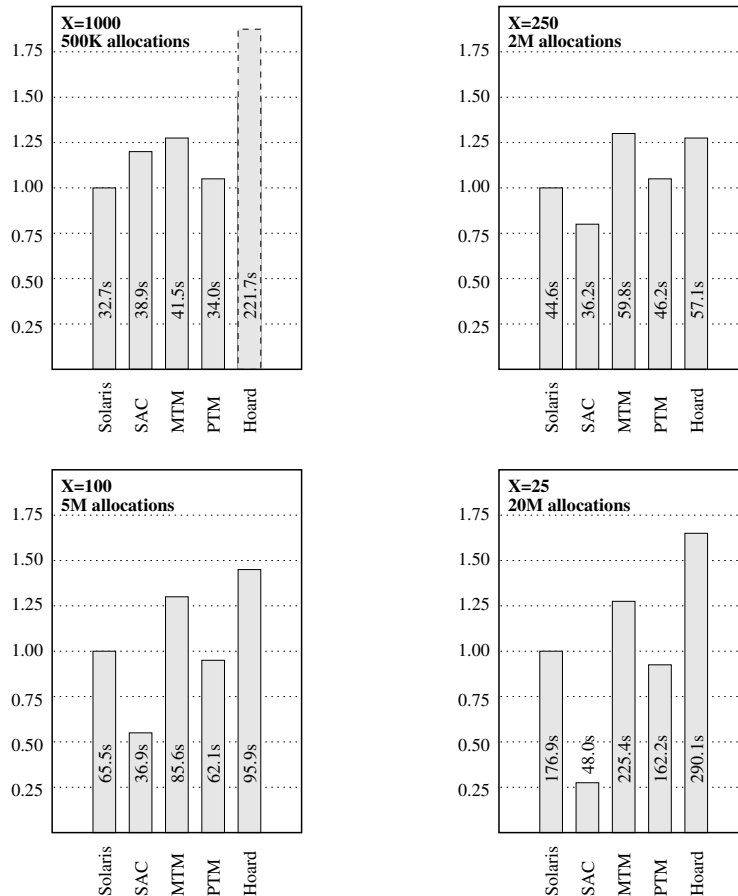


Fig. 5. Single processor performance of multithreaded allocators in comparison with the serial SOLARIS allocator as base line using the SAC microbenchmark of Fig. 1

Fig. 5 shows the single processor performance achieved by SACP_{HM} and that of the three other multithreaded memory allocators in comparison with the serial SOLARIS allocator used in Section 2. Regardless of the concrete problem size, MTMALLOC incurs a runtime overhead of about 25% compared with the serial allocator. For HOARD the respective overhead grows with increasing memory management frequency from about 25% to more than 60%. Surprisingly, performance is much worse for problem size $X=1000$, where single processor execution time exceeds that of any other allocator by almost an order of magnitude. Having a closer look at the implementation of HOARD reveals that memory requests exceeding a certain threshold size are directly mapped to virtual memory by using the `mmap` and `munmap` system routines. Obviously, their frequent application incurs prohibitive overhead.

In contrast, PTMALLOC performs similar to the serial allocators, slightly outperforming them with increased memory management frequency. For SACP_{HM}

it can be observed that starting out with a performance loss of about 20% relative to standard allocators for problem size $X=1000$, this slowdown turns into a significant speedup with increasing frequency of memory allocations and deallocations. With overall execution time being clearly dominated by dynamic memory management overhead for $X=25$, the SAC-specific memory allocator makes the overall program run 3.7 times faster than with the SOLARIS allocator.

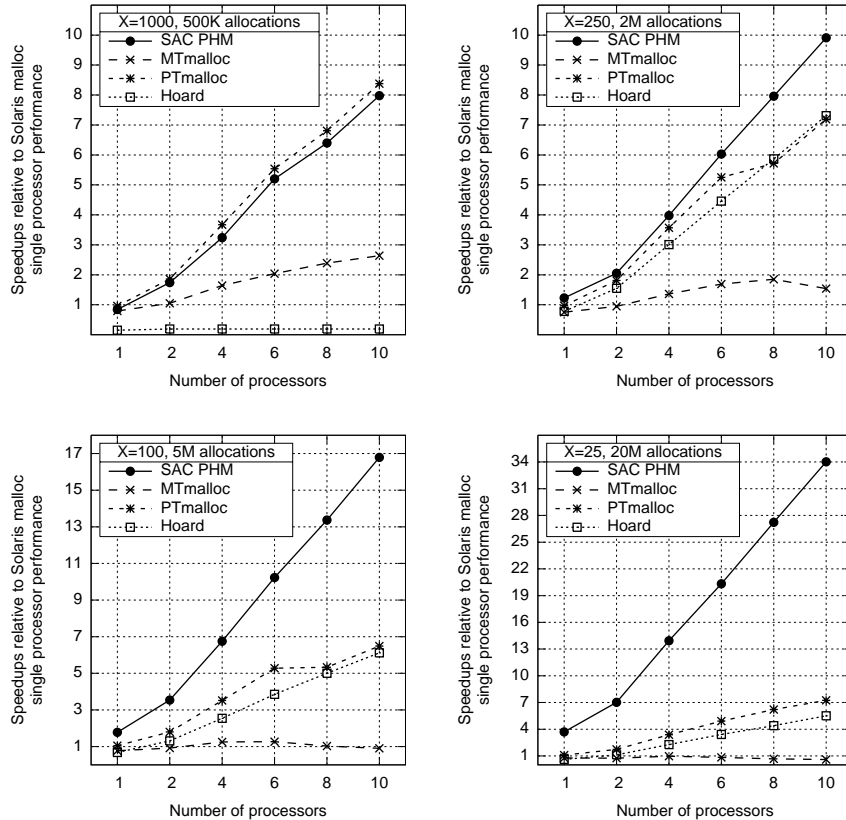


Fig. 6. Multi processor performance of multithreaded allocators in comparison with the serial SOLARIS allocator as base line using the SAC microbenchmark of Fig. 1

Fig. 6 shows the multiprocessor performance of the multithreaded allocators relative to the base line set by the serial SOLARIS allocator. This “true” parallel performance takes the different sequential performance levels into account, hence the different starting points of the curves for a single processor. First of all, we observe that MTMALLOC scales rather poorly for all problem sizes investigated. This observation is rather surprising for an allocator that is particularly designed for exactly this scenario. However, it coincides with much more thorough investigations made by the developers of HOARD [18]. In contrast, PTMALLOC scales

fairly well; it is not clear why hardly any speedup can be observed when switching from 6 to 8 processors for some problem sizes, but additional measurements have confirmed these figures. High scalability can be observed for HOARD for all problem sizes that are not mapped directly to the virtual memory manager (i.e. $X=1000$). SACPHM turns out to scale as well as HOARD, but provides this scalability on top of a substantially higher single processor performance.

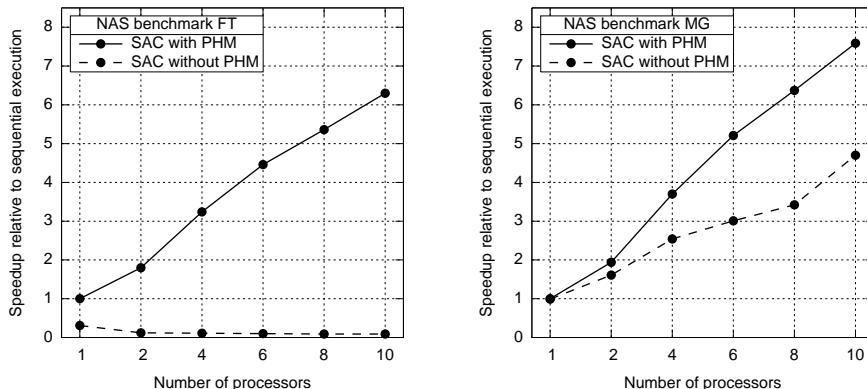


Fig. 7. Performance impact of SACPHM on NAS benchmarks FT (left) and MG (right)

In addition to our microbenchmark we have also investigated the impact of heap management on two non-trivial benchmarks from the NAS benchmark suite [25]: SAC implementations of 3-dimensional multigrid relaxation (benchmark MG [20]) and 3-dimensional fast-Fourier transforms (benchmark FT [21]). Fig. 7 quantifies the effect of SACPHM on runtime performance (size class A).

The two benchmarks show very different allocation/de-allocation patterns. FT uses fairly large arrays of complex numbers ($256 \times 256 \times 128$) that are exclusively allocated and de-allocated in single-threaded mode. We use the Danielson-Lanczos algorithm for 1d-FFTs on vectors of length 128 and 256, respectively, and explicitly create 2 (4) vectors of length 64, 4 (8) vectors of length 32, etc. All these allocations and the corresponding de-allocations happen during multi-threaded execution. Unsurprisingly, Fig. 7 shows SACPHM to be indispensable. However, our declarative implementation of FT is so much dominated by memory management overhead that the good single-processor performance of SACPHM on small chunks of data that are repeatedly allocated and de-allocated turns out to be crucial as well.

The benchmark MG implements a multigrid method that starts with arrays of size $256 \times 256 \times 256$ and performs alternating convolution and mapping steps. In each mapping step the array size shrinks by a factor of two in each dimension until the minimum size of $4 \times 4 \times 4$ is reached; further mapping steps let the array size grow again. Like in the FT benchmark, we are faced with a substantial number of allocations (and de-allocations) over a wide range of chunk sizes. However, in contrast to FT none of them occur during data-parallel operations.

As a consequence, the serial allocator performs reasonably well. Nevertheless, it takes SACPHM to achieve good overall speedups through a reduction of absolute overhead inflicted by dynamic memory management.

5 Related work

In the previous section have already acknowledged and evaluated several general-purpose memory allocators that are specifically designed for multithreaded program execution, namely SUN's MTMALLOC [16], PTMALLOC [17] and HOARD [18].

Only few declarative languages besides SAC explicitly focus on arrays. We mention SISAL [2] and NESL [3], which both use reference counting. While the developers of SISAL spent considerable effort into efficient reference counting [13], they left the underlying heap management issues to the C system library [26].

In NESL the VCode interpreter takes responsibility for memory management [27, 28]. Its design differs from our solution in various aspects. Firstly, by making the reference counter a part of the heap administration data structures NESL fully integrates reference counting with its own heap management. In contrast, we explicitly allocate reference counters (as part of a more general array descriptor) on the heap. This approach allows us to employ (third party) heap managers that are fully unaware of our reference counting scheme for experimental comparisons like the one in Section 4 as well as for backup reasons. Secondly, the NESL solution organises the heap differently. While NESL does use multiple free lists for different chunk sizes for the same purposes as we do, it nevertheless allocates all chunk sizes from the same contiguous address space. In contrast, our allocation arenas (inspired from general-purpose multithreaded allocator designs) actually keep differently sized chunks in different areas of the address space. This design has a positive impact on fragmentation and solves the false sharing problem. Thirdly, we haven't found any information concerning multithreaded heap management in the context of NESL, and the solution described in [27] does not support concurrently executing threads.

6 Conclusion

We have outlined an important aspect of the memory management subsystem of the functional array language SAC: the integration between the reference counting mechanism that decides when to allocate and de-allocate heap memory and the underlying heap manager that maps concurrent allocation and de-allocation requests of multiple threads to the linear address space of a process. Empirical data shows the significance of an integrated approach to achieve good runtime performance in declarative array processing on multiprocessor and multicore systems.

As soon as runtime performance is an issue (and in parallel processing it usually is), declarative programming languages often find themselves in a defensive position. In machine-oriented programming languages one typically blames the

application programmer (rather than the C compiler, for instance) for unsatisfactory performance. Frequently, additional effort and expert knowledge manage to improve performance, albeit often at the expense of readability and portability. Declarative programming languages raise the level of abstraction in programming from a machine-oriented to a problem-oriented view. Yet, they need to meet the programmer's performance expectations. The more performance matters, the more difficult this is to achieve.

Automatic memory management plays a crucial role here because it is a key feature of declarative languages and it must directly compete with manual dynamic memory management or even static memory layouts used by machine-oriented approaches. From the user's perspective it is indistinguishable whether unsatisfactory performance is caused by inefficiencies in compilation/parallelisation schemes or by false assumptions of an off-the-shelf memory allocator. The bottom line is that it takes a fully integrated approach to be successful: code generation needs to be well integrated with reference counting to directly reuse memory as often as possible, and reference counting needs to be well integrated with an underlying heap manager to reduce the overhead inflicted by remaining allocations and de-allocations. Furthermore, the heap manager needs to be integrated with the multithreaded runtime system to avoid costly synchronisation when concurrent threads access the implicitly shared heap.

References

1. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal* **30** (2005)
2. Cann, D.: Retire Fortran? A Debate Rekindled. *CACM* **35** (1992)
3. Blelloch, G.E.: Programming Parallel Algorithms. *CACM* **39** (1996)
4. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *J. Functional Programming* **13** (2003) 1005–1059
5. Grellck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. *Intern. Journal of Parallel Programming* **34** (2006) 383–427
6. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: *Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, Nice, France, ACM Press (2007)
7. Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: *12th ACM Symposium on Principles of Programming Languages (POPL'85)*, New Orleans, USA, ACM Press (1985) 300–313
8. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review. In: *International Workshop on Memory Management (IWMM'95)*, Kinross, UK. LNCS 986, Springer-Verlag (1995) 1–116
9. Milner, R., Tofte, M., Harper, R.: *The Definition of Standard ML*. MIT Press, Cambridge, USA (1990)
10. Peyton Jones, S., Launchbury, J.: State in Haskell. *Lisp and Symbolic Computation* **8** (1995) 293–341
11. Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, M.: Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. University of Nijmegen, The Netherlands (1993)

12. Serrarens, P.: Implementing the Conjugate Gradient Algorithm in a Functional Language. In: 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany. LNCS 1268, Springer-Verlag, (1997) 125–140
13. Cann, D., Evripidou, P.: Advanced Array Optimizations for High Performance Functional Languages. *IEEE Trans. on Parallel and Distributed Systems* **6** (1995)
14. Grelck, C., Trojahnner, K.: Implicit Memory Management for SAC. In: 16th International Workshop on Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany (2004) 335–348
15. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. *J. Functional Programming* **15** (2005) 353–401
16. Sun Microsystems Inc.: A Comparison of Memory Allocators in Multiprocessors. Solaris Developer Connection, Sun Microsystems Inc., Mountain View, USA (2000)
17. Gloger, W.: Dynamic Memory Allocator Implementations in Linux System Libraries. 4th International Linux Kongress (LK'97), Würzburg, Germany (1997)
18. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. In: 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, USA. ACM Press (2000) 117–128
19. Shafarenko, A., et.al.: Implementing a numerical solution of the KPI equation using Single Assignment C: lessons and experiences. In: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland. LNCS 4015, Springer-Verlag (2006)
20. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA, IEEE Computer Society Press (2002)
21. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. LNCS 2763, Springer-Verlag (2003) 230–235
22. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In: Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK. LNCS 1467, Springer-Verlag (1998) 72–92
23. Torellas, J., Lam, M., Hennessy, J.: False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers* **43** (1994) 651–663
24. Lea, D.: A Memory Allocator. *Unix/Mail* **6/96** (1996)
25. van der Wijngart, R.: NAS Parallel Benchmarks Version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, Moffet Field, USA (2002)
26. Cann, D.C.: Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Lab, Livermore, USA (1989)
27. Blleloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., Zagha, M.: Implementation of a Portable Nested Data-Parallel Language. Technical Report CMU-CS-93-112, Carnegie Mellon University, Pittsburgh, USA (1993)
28. Blleloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., Zagha, M.: Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing* **21** (1994) 4–14