Hierarchical PLABs, CLABs, TLABs in Hotspot

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Hannes Payer
University of Salzburg
hpayer@cs.uni-salzburg.at

Harald Röck
University of Salzburg
hroeck@cs.uni-salzburg.at

***Abstract*—Thread-local allocation buffers (TLABs) are widely used in memory allocators of garbage-collected systems to speed up the fast-path (thread-local allocation) and reduce global heap contention yet at the expense of increased memory fragmentation. Larger TLABs generally improve performance and scalability but only up to the point where more frequent garbage collection triggered by increased memory fragmentation begins dominating the overall memory management overhead. Smaller TLABs decrease memory fragmentation but increase the frequency of executing the slow-path (global allocation) and thus may reduce performance and scalability. In the Hotspot JVM a complex, TLAB-growing strategy implemented in several thousand lines of code determines the TLAB size based on heuristics. We introduce hierarchical allocation buffers (HABs) and present a three-level HAB implementation with processor- and core-local allocation buffers (PLABs, CLABs) in between the global heap and TLABs. PLABs and CLABs require low-overhead OS-provided information on which processor or core a thread executes. HABs may speed up the slow-path of TLABs in many cases and thus allow using smaller TLABs decreasing memory fragmentation and garbage collection frequency while providing the performance and scalability of otherwise larger TLABs. Our implementation works with or without the TLAB-growing strategy and requires two orders of magnitude less code. We evaluate our implementation in the Hotspot JVM and show improved performance for a memory-allocation-intensive benchmark.**

***Keywords*-memory management, garbage collection, virtual machines, scalability**

I. Introduction

Memory management in runtime systems like Java virtual machines (JVMs) may be a scalability bottleneck in applications with multiple threads accessing the global heap frequently. Thread-local allocation buffers (TLABs) reduce global heap contention by preallocating large pieces of memory from the global heap. The preallocated memory is stored thread-locally to handle allocation requests of a given thread. This approach does not only reduce contention on the global heap but also allows a fast-path for memory allocation that does not require any synchronization or atomic operations since the TLAB of a thread is not shared with any other threads. However, larger TLABs introduce additional memory fragmentation that depends linearly on the number of threads since large blocks of memory are committed to thread-local use only. High memory fragmentation may result in more frequent garbage collection which may decrease application throughput. To trade-off scalability and memory fragmentation in modern JVMs complex TLAB-growing strategies incorporate different factors like allocation rate, number of threads, heap size and feedback from the garbage collector to determine TLAB sizes for all threads. The implementation of such a strategy in the garbage collector of the Hotspot JVM [7] requires several thousand lines of code and thus significantly contributes to its complexity.

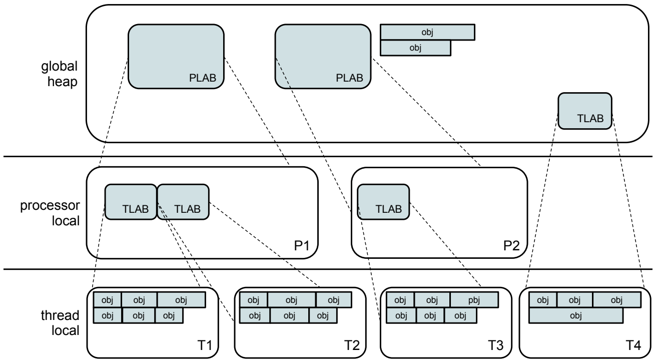


Figure 1. Hierarchical allocation buffers (HABs) with three levels: on the highest level is the global heap, in the middle are the PLABs or CLABs $P1$ and $P2$, and on the lowest level are the TLABs $T1 - T4$.

We introduce hierarchical allocation buffers (HABs), which consist of multiple levels of allocation buffers where an allocation buffer on a given level preallocates memory out of an allocation buffer on the next higher level. The traditional approach with TLABs is thus a two-level HAB system with the global heap on top and TLABs below. For recent multi-core architectures with several cores per CPU and several CPUs per machine we propose to use a three-level HAB system with one more level in between as depicted in Figure 1. In our implementation this level uses processor- or core-local allocation buffers (PLABs, CLABs) which require low-overhead OS-provided information on which processor or core a thread executes. PLABs and CLABs speed up the slow-path of TLABs in many cases and thus allow using smaller TLABs decreasing memory fragmentation and garbage collection frequency while providing the performance and scalability of otherwise larger TLABs. We show in experiments that a statically configured HAB system may provide similar performance as a TLAB-only system using a TLAB-growing strategy.

Our three-level HAB implementation reflects the underlying processor architecture of a server machine with four Intel Xeon E7 processors where each processor comes with ten cores and two hardware threads per core. The allocation buffers of the middle level can be configured to be PLABs or CLABs. The TLABs on the lowest level allocate from the PLABs or CLABs associated with the processor or core on which the allocating thread is currently running on. We evaluate the performance of our three-level HAB implementation integrated into the Hotspot JVM and show performance improvements due to better cache utilization and less contention on the global heap.

We summarize the contributions of this paper: (1) the notion of hierarchical allocation buffers (HABs), (2) the three-level HAB implementation with processor- or core-local allocation buffers (PLABs, CLABs), and (3) an experimental evalulation of HABs in the Hotspot JVM.

In Section II, we present the design of HABs. In Section III, we discuss the implementation of HABs in the Hotspot JVM and the required operating system support. Related work is discussed in Section IV, our experiments are presented in Section V, and conclusions are in Section VI.

II. PLABs, CLABs, TLABs

TLABs are an architecture-independent concept for implementing allocation buffers. Each thread maintains its private allocation buffer for fast allocation of memory. However, allocation buffers may also be implemented in an architecture-dependent fashion. For example, allocation buffers can be assigned to processors, i.e., a thread running on a processor may use the allocation buffer of the processor for its allocation requests [3]. We study the use of processor-local allocation buffers (PLABs) as well as core-local allocation buffers (CLABs) situated in between TLABs and the global heap. A PLAB is assigned to a given processor which may comprise of multiple cores. Threads running on different cores but on the same processor share the same PLAB. A CLAB is assigned to a given core. Threads running on the same core share the same CLAB. Using PLABs may increase parallelism and cache utilization and thus reduce contention. On multicore machines using CLABs over PLABs may increase parallelism and cache utilization even further. Note that the size of PLABs and CLABs should be multiples of the TLAB size to avoid additional internal memory fragmentation.

Access to PLABs and CLABs is done in two steps. First, the processor or core on which a given thread currently runs is determined (selection). Then, the actual allocation in the selected PLAB or CLAB is performed atomically (allocation). Selection and allocation is done non-atomically for better performance and scalability. Thus a thread may migrate to another processor or core in between selection and allocation resulting in what we call a foreign allocation. Note that the probability of foreign allocations is low and we show in experiments that foreign allocations indeed rarely happen.

III. Implementation

In this section, we discuss the implementation of HABs in the Hotspot JVM and present simplified pseudo-code of the core algorithm. Garbage-collection-specific details were removed for simplicity. We modified the heap implementation of Hotspot's parallel garbage collector for the upcoming JDK7. The modifications are limited to the code path of allocating new and refilling existing TLABs. Additionally, we disallow direct inlined access to the global heap. In our benchmarks we did not observe any slow down when removing inlined access to the global heap.

Listing 1 shows the method for allocating a TLAB of a given size. If a thread's TLAB is full the thread invokes this method to allocate a new memory region for its TLAB. TLABs larger than PLAB size are directly allocated from the global heap. For smaller TLABs, we try to allocate them in a PLAB with preference to the PLAB of the current processor. We iterate over all PLABs starting at the PLAB of the current processor and try to allocate a new TLAB. As soon as a TLAB is successfully allocated it is returned to the caller. If a TLAB could not be allocated in any PLAB, we fall back to allocate memory from the global heap. If this also fails the method returns NULL, and eventually a GC cycle will be triggered.

Listing 2 shows the method for allocating a TLAB of a given size on a dedicated processor (or core), and if the PLAB is full how it is refilled. If an allocation request cannot be handled the method returns NULL. The implementation uses an optimistic non-blocking approach to avoid expensive locking. The access to a PLAB is synchronized using the PLAB's top variable. The top variable indicates where the free memory in the PLAB starts or whether the PLAB is currently being refilled. If a thread detects that the PLAB is currently being refilled by another thread it returns NULL indicating to the caller that no allocations are currently possible in this PLAB. The top variable is always modified using a compare-and-swap operation.

If top is a valid pointer, the thread attempts to allocate the new TLAB in the PLAB. The allocation in the PLAB advances the top pointer by the size of the new TLAB using a compare-and-swap retry cycle. If the new TLAB does not fit into the PLAB it returns NULL, and the protocol to refill the PLAB with a new memory region is started. The refill protocol starts by setting top to PLAB_REFILL_IN_PROGRESS. The succeeding thread allocates a new memory region from the global heap and reinitializes the PLAB with the new memory region. If the allocation fails at any step, for example, when trying to set top to PLAB_REFILL_IN_PROGRESS or when trying to allocate a new PLAB, the method returns NULL. As mentioned in the previous section we tolerate context switches in

```
HeapWord* allocate_new_tlab(size_t size) {
  if (size <= PLAB_SIZE) {
    int hw_id = get_hw_id();
    for (int i = 0; i < PLABS; i++) {
      HeapWord* tlab = allocate_on_processor(size, (hw_id + i) % PLABS);
      if (tlab != NULL) {
        return tlab;
      }
    }
  }
  return allocate_in_global(size);
}
```

Listing 1. TLAB allocation

```
HeapWord* allocate_on_processor(size_t size, int id) {
  HeapWord* top = plabs_[id].top();
  if (top == PLAB_REFILL_IN_PROGRESS) {
    return NULL;
  }
  HeapWord* tlab = plabs_[id].allocate(size);
  if (tlab == NULL) {
    HeapWord* result = cmpxchg(plabs_[id].top_addr(), top, PLAB_REFILL_IN_PROGRESS);
    if (result == NULL) {
      return NULL;
    }
    tlab = allocate_in_global(PLAB_SIZE);
    if (tlab == NULL) {
      plabs_[id].reset();
      return NULL;
    }
    plabs_[id].init(tlab, size);
  }
  return tlab;
}
```

Listing 2. TLAB allocation on a given processor (or core)

between reading the processor ID and performing the actual allocation, which may result in foreign allocations. Moreover, a foreign allocation may also be performed by a thread that encounters that the PLAB_REFILL_IN_PROGRESS flag has been set. Note that this is a completely lock-free implementation for avoiding lock-related complications with system invariants in the stop-the-world garbage collector.

A. *Operating System Support*

Our current implementation requires that the underlying operating system provides a mechanism for threads to look up the processor and core they are running on. In recent Linux kernels the `getcpu()` system call is optimized to provide a low-overhead mechanism for determining the CPU on which the invoking thread runs. However, it is not guaranteed that the thread is still executing on the CPU after the call returns. Therefore, allocation in PLABs and CLABs have to be synchronized and could even result in foreign allocations where a thread on processor A allocates memory assigned to processor B.

In order to reduce the additional overhead when accessing PLABs or CLABs or even allow unsynchronized access to CLABs we would require additional support of the OS. For instance, a notification when the thread is preempted would suffice to detect possible migrations and context switches. If a thread detects that it was preempted in between determining the current processor and the actual allocation in the PLAB it could restart the operation. In [2] the authors introduce multi-processor restartable critical sections (MB-RCS) for SPARC Solaris, which provide a mechanism for user-level threads to know on which processor they are executing and to safely manipulate CPU-specific data.

IV. Related Work

Several JVMs already provide specific support for different processor architectures. For example, the latest version of the Hotspot JVM already supports NUMA architectures where the heap is divided into dedicated parts for each NUMA node.

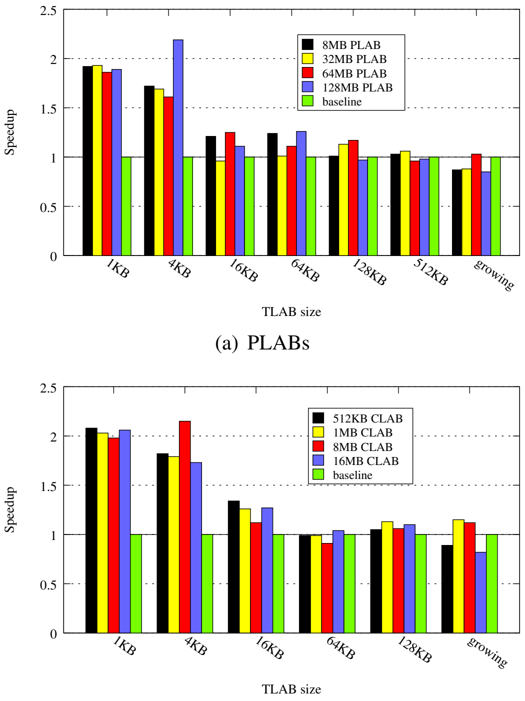
PLABs were previously discussed in [3]. The implementation is based on a special mechanism called multi-processor restartable critical section which allows to manipulate processor-local data consistently and guarantees that a thread always uses the PLAB of the processor it is running on. In our implementation we do not provide that guarantee. If there is a context switch between determining processor and PLAB operation and the thread is scheduled after that on a different processor we tolerate that. Moreover, just using PLABs eliminates the fast-path provided by TLABs. In our work we combine the benefits of both PLABs and CLABs with TLABs.

Thread- and processor-local data is not only relevant in allocators of JVMs but also in explicit memory management systems. Multi-processor restartable critical sections are used in [2] to implement a memory allocator that holds allocator-specific metadata processor-locally to take advantage of the cache and to reduce contention. McRT-Malloc [4] is a non-blocking memory management algorithm, which avoids atomic operations on typical code paths by accessing thread-local data only. Hoard [1] is a memory allocator that combines, in more recent versions, thread-local with non-thread-local allocation buffers.

V. EXPERIMENTS

For our experimental evaluation we used a server machine with four Intel Xeon E7 processors where each processor comes with ten cores and two hardware threads per core, 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We ran the SPECjvm2008 memory-allocation-intensive javac benchmark [5] with 80 threads on the Hotspot JVM in server mode [6]. Each benchmark run was configured to last six minutes with two minutes of warm-up time in the beginning. We repeated each experiment seven times and measured the performed operations per minute. The default generational garbage collector of the JVM is configured with a maximum heap size of 30GB and a new generation size of 10GB. Parallel garbage collection is performed in the old and new generation.

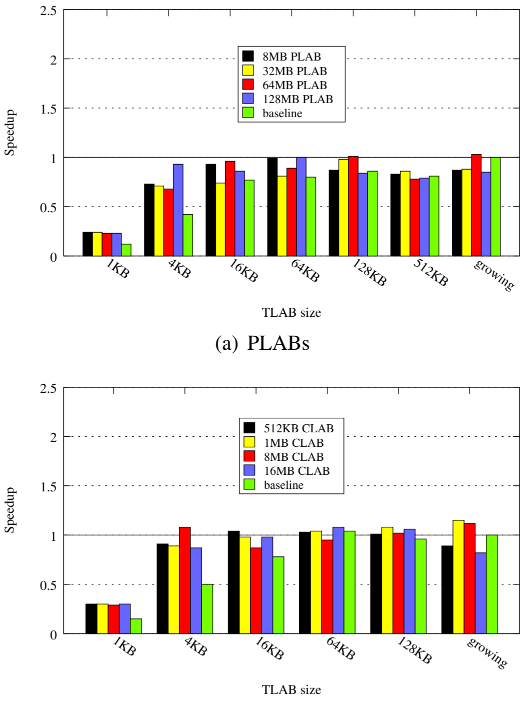
For the data in Figure 2 we removed the maximum and minimum from the seven runs and calculated the average of the remaining five runs. On the x-axis are TLAB sizes of increasing but fixed size, except where it says “growing” indicating that the TLAB-growing strategy of the unmodified Hotspot JVM is used. Note that with the TLAB-growing strategy the TLAB size settles at around 2MB. On the y-axis the speedup over the corresponding TLAB-only configurations of the unmodified Hotspot JVM (baselines) is depicted. In Figure 2(a) the results using HABs with PLABs and in Figure 2(b) the results using HABs with CLABs are depicted. For smaller TLABs a higher speedup can be achieved since the slow-path is triggered more often. However, performance also improves with the TLAB-growing strategy. In the presented results CLABs perform on average slightly better then PLABs.



(a) PLABs

(b) CLABs

Figure 2. Speedup of using HABs with different TLAB and PLAB/CLAB configurations over the corresponding TLAB-only configurations of the unmodified Hotspot JVM.



(a) PLABs

(b) CLABs

Figure 3. The data of Figure 2 but using the unmodified Hotspot JVM with the TLAB-growing strategy as common baseline.

ISBN: 978-1-61208-184-7

	PLAB				
TLAB	4MB	8MB	16MB	32MB	64MB
No	0.32%	0.33%	0.51%	0.94%	1.93%
1KB	0.11%	0.06%	0.09%	0.05%	0.15%
4KB	0.11%	0.06%	0.06%	0.06%	0.19%
8KB	0.13%	0.10%	0.08%	0.10%	0.29%
16KB	0.10%	0.08%	0.11%	0.08%	0.12%
32KB	0.11%	0.08%	0.08%	0.11%	0.30%
64KB	0.22%	0.15%	0.16%	0.29%	0.45%
128KB	0.52%	0.47%	0.63%	1.31%	2.60%
growing	0.55%	0.47%	0.64%	1.10%	2.23%

Table I
PERCENTAGE OF FOREIGN ALLOCATIONS WITH DIFFERENT PLAB AND TLAB CONFIGURATIONS.

	CLAB			
TLAB	512KB	1MB	8MB	16MB
1KB	0.00%	0.01%	0.05%	0.07%
4KB	0.01%	0.01%	0.06%	0.15%
16KB	0.01%	0.02%	0.07%	0.13%
64KB	0.03%	0.03%	0.10%	0.17%
128KB	0.06%	0.05%	0.15%	0.26%
growing	-	-	1.42%	2.81%

Table II
PERCENTAGE OF FOREIGN ALLOCATIONS WITH DIFFERENT CLAB AND TLAB CONFIGURATIONS.

Figure 3 is based on the same data as presented in Figure 2 but the y-axis depicts the speedup over the TLAB-only configuration of the unmodified Hotspot JVM using the TLAB-growing strategy, which is the default setting of the JVM. The results confirm that for smaller TLAB sizes than with the TLAB-growing strategy (around 2MB) similar or even better performance can be achieved. Figure 3(a) shows the results using HABs with PLABs and Figure 3(b) shows the results using HABs with CLABs. In particular, the results show that HABs with a small TLAB size provide similar performance as the original TLAB-growing strategy of the Hotspot JVM. In this case, a statically configured HAB implementation may thus replace a significantly more complex implementation of a TLAB-growing strategy.

In Table I and Table II, the amount of foreign allocations using different PLAB and CLAB configurations with different TLAB sizes are presented. The amount of foreign allocations increases with increasing PLAB or CLAB size. Overall, however, the amount of foreign allocations is low, which shows that allowing allocations in PLABs or CLABs that do not match the current processor or core the thread is running on can be tolerated here. For the 512KB and 1MB CLAB sizes the TLAB-growing strategy immediately determines to use TLABs larger then the given CLAB size, so CLABs are not used at all. Evaluating different synchronization strategies and allocations policies remains future work.

VI. CONCLUSION

We introduced hierarchical allocation buffers (HABs) for improving performance and scalability of memory management on state-of-the-art multiprocessor and multicore server machines. We implemented and evaluated three-level HABs in the Hotspot JVM and showed performance improvements in a memory-allocation-intensive benchmark due to better cache utilization and less contention on the global heap. The results show that taking the underlying hardware architecture of general purpose machines into account even more than before may require significantly less complex code than architecture-oblivious solutions without a loss in performance. The concept of HABs is just a first step. In the future we plan to consider a cooperative thread scheduling mechanism for executing threads that share a significant amount of data on the same processor to further benefit from caching. Moreover, we plan to integrate the HABs architecture into the TLAB-growing strategy of the Hotspot JVM for tuning not only TLAB sizes but also PLAB or CLAB sizes automatically. Considering other HABs configurations may also be beneficial, e.g., a four-level system with TLABs, CLABs, PLABs, and the global heap.

Acknowledgements

This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23).

REFERENCES

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proc. ASPLOS*, pages 117–128. ACM, 2000.

[2] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proc. ISMM*, pages 163–174. ACM, 2002.

[3] A. Garthwaite, D. Dice, and D. White. Supporting per-processor local-allocation buffers using lightweight user-level preemption notification. In *Proc. VEE*, pages 24–34, New York, NY, USA, 2005. ACM.

[4] R. Hudson, B. Saha, A. Adl-Tabatabai, and B. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *Proc. ISMM*, pages 74–83. ACM, 2006.

[5] SPEC. SPECjvm2008 benchmarks, 2008. http://www.spec.org/jvm2008.

[6] Sun Microsystems. Description of HotSpot GCs: Memory Management in the Java HotSpot Virtual Machine White Paper. http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf, 2006.

[7] Sun Microsystems. The Java Hotspot Virtual Machine White Paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html, 2011.