RC23377 (W0404-063) April 9, 2004
Computer Science

IBM Research Report

Handling and Profiling the Increasingly Large and Complex Memory Allocation Patterns of the 64 Bit Era

Ulrich Finkler
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Handling and Profiling the Increasingly Large and Complex Memory Allocation Patterns of the 64 bit Era

Ulrich Finkler*

May 5, 2004

Abstract

The transition from 32 bit to 64 bit processors caused a sudden increase in memory use for a variety of workloads. Additionally, object oriented programs and advanced data structures predominantly use small memory blocks with complex allocation patterns. The increasing size and complexity of allocation patterns has exposed limitations in the scalability and performance of memory allocators. BFM, the allocator presented in this paper, was motivated by such limitations observed in the processing of VLSI designs.

BFM provides low memory overhead (best fit allocation) and competitive performance. Its worst case complexity, $O(\log(N))$ with small constants, holds not only amortized, but for each individual operation. N is the number of heap operations performed in the past. Thus it ensures the absence of pathological cases and is well suited for applications that wish to control response times.

Together with the allocator, a tracing infrastructure is presented that allows to plot memory use over execution time for every call chain. The overhead for tracing is very low such that tracing of multi-hour and multi-GB runs is feasible without moving to much larger machines.

In addition to the description of the data structures and algorithms of the allocator and tracer, experimental comparisons with the glibc allocator, Hoard and the IBM®AIX®[1] allocator are presented, along with experimental results of tracing.

*IBM T.J. Watson Research Center, Route 134, Yorktown Heights, New York 10548.

[1]IBM and AIX are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

1 Introduction

With the introduction of 64-bit processors a hard barrier for memory use by a single process fell, removing a critical bottleneck for a variety of applications, as for example VLSI layout processing. Now there are SMP machines with hundreds of GB of main memory and programs that use such amounts. Additionally, the transition to 64-bit increases memory waste in allocators relative to the actual memory use by the application since the size of pointers doubles while string, integer and floating point variables maintain their size.

Furthermore, object oriented programs and advanced linked data structures allocate memory in large numbers of small blocks in complicated patterns. The combination of larger core memories and more complex allocation patterns exposed bottlenecks in memory allocation.

Performance variations in memory allocation are mitigated by the fact that an application spends only a fraction of its time with allocation, but differences in memory usage are fully exposed. Main memory is a dominating cost factor for large machines. Memory waste limits not only the capacity of a single process but also has an impact on the usable number of concurrent large applications and threads.

The allocator presented in this paper was initially motivated by the observation of performance problems in VLSI layout processing applications on AIX. One of the design goals was to ensure the absence of pathological behavior. Later we found that out of a selection of allocators on AIX and Linux, each had cases with a problematic behavior.

The glibc allocator exhibited already in a 32-bit environment excessive runtimes with certain allocation patterns. The average time for a malloc/free pair varies by a factor 200 and more. Thus, an ap-

plication spending on one input about 1 % of its time in allocation may increase its runtime by a factor of 3 on a similar input. Such a case was encountered with a large VLSI processing application, turning an overnight execution into days. On the other hand the glibc allocator is fairly memory efficient.

AIX (4.x to 5.1) provides two flavors of allocator, the default allocator and a 'bucket' allocator. The default allocator showed no excessive execution time increases, but the handling of small blocks is consistently slow and memory usage not always optimal. The AIX 'bucket'-allocator treats small blocks more efficiently, but occasionally requires even more memory than the default allocator.

We also took the Hoard allocator [4] into consideration. The experiments show that Hoard is efficient across all experiments. But the Hoard allocator showed cases of significant memory waste similar to the AIX bucket allocator.

BFM, the allocator presented in this paper, combines provably good worst case performance and low memory usage without sacrificing performance in common cases. It provides (almost) best fit allocation and a worst case complexity of $O(\log(N))$ per operation, not only amortized. Its performance and memory usage for all investigated test patterns were competitive with the best results of the four tested alternatives.

Section 2 discusses the data structures of the BFM heap and section 3 covers experiments that compare BFM to the two flavors of the AIX allocator, the Hoard and the glibc allocator. In addition to the actual heap structure, a memory tracer for BFM is presented in section 4 that allows to plot memory usage over the time line for every call chain. The tracers execution time and memory overhead is sufficiently small that tracing of multi-hour and multi-GB runs is feasible without moving to much larger machines. This is particularly beneficial if the untraced application is already testing the limits of the largest available machines.

2 Data Structures

The BFM heap structure consists of two components. The first component handles larger blocks of memory and is based on a pair of binary trees (indexed fit [5]). It implements a best fit allocation with immediate coalescing.

The second component handles small blocks and is based on bucketing. It implements a best fit scheme with 'group-wise recombination', i.e. a block with minimal waste is used out of a limited selection of sizes but blocks are not coalesced until a consecutive group of blocks is free.

Neither of the two basic schemes is new, but the specific implementation provided by BFM reduces the performance disadvantage of best fit schemes for simple patterns and avoids the sometimes suboptimal memory usage of bucketing schemes.

The two different coalescing schemes fit the different properties of small and large blocks [5][14][15]. The allocator performance on small blocks has the largest impact (allocation time versus usage time) on the total execution time. Small blocks are used frequently and the time to initialize and use a small block is often in the order of magnitude or less than the cost of its allocation and deallocation. The efficient (constant time) group-wise coalescing depends on high usage rates and a limited number of different sizes to avoid unacceptable external memory waste. The for this case critical internal memory waste is low.

The tree-based immediate coalescing spends more time to reduce memory waste for less frequent allocations with a larger variation in block size but this is mitigated by the fact that the initialization and usage time of a large block is typically larger than the allocation cost.

2.1 The Bucket Component

All blocks smaller than a given threshold – we chose 64*sizeof(void*) – are handled in the bucket component. Using a lookup table the size of a request is matched to a bucket such that memory waste is less than a certain threshold. For example requests of sizes smaller 6*sizeof(void*) are matched exactly and all blocks between 56 and 63 sizeof(void*) are mapped to 63*sizeof(void*). The choice of different tables allows to reduce the memory waste at the cost of an increasing number of buckets and thus a lower likelihood of reuse and vice versa.

Figure 1 shows the structures of the bucket components. The size in pointers is used to obtain from the first table an index into a second table, i.e. the the first table determines which fields in the second

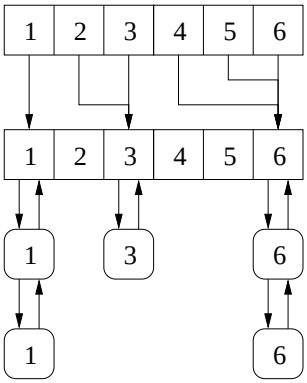

Figure 1: *Bucket data structures, numbers indicate block size in* sizeof(void*). Rounded rectangles *are page records.* Rectangles *are lookup tables with block sizes as indices.*

table are used. Each (used) entry in the second table holds a doubly linked list of *page records* with blocks of the same size.

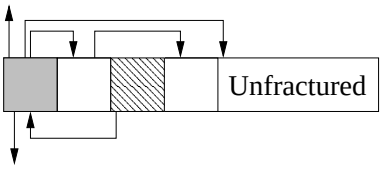

Figure 2: *Memory layout of a* page record. Gray*: Administrative information.* Striped*: Block in use, its prefix points to its page record.* White*: Blocks in singly linked free list.* Unfractured*: Not used yet.*

A *page record* (figure 2) is a consecutive block of memory that provides 1024*sizeof(void*) bytes for blocks of a fixed size. It also holds a small structure for administrative information which contains pointers for the doubly linked list of page records, a pointer to a singly linked free list and a pointer to a consecutive piece of unfractured memory.

A block that is in use reserves the first sizeof(void*) bytes as a prefix which holds a pointer to its page record. Large blocks handled in the tree component use a prefix with room for two pointers, the first containing the size of the block and the second being zero to allow differentiation between bucket and tree managed blocks. If we rely on 4-byte alignment it is possible to use the lowest bit of a one-pointer prefix to differentiate, but the savings are minor.

2.1.1 Operations

An allocation obtains a memory block as follows:
1. *Look up the bucket for the required size.*
2. *If the first page record has no free block, insert a new page record from the tree component at the beginning of the page record list.*
3. *Select the first page record.*
4. *If possible use a block from the free list, otherwise cut a block from the unfractured part.*
5. *Reduce the free block count. If the page record is now entirely used, move it to the end of the page record list.*

A free operation releases a memory block as follows:
1. *Use the prefix to find the page record. Insert the free block into the free list.*
2. *Increase the free count.*
3. *If the page is now entirely free, give it back to the tree component.*
4. *If the page record is not entirely free but its predecessor is completely used, move it to the front of the page record list.*

2.1.2 Analysis

Allocation and deallocation take constant time in the bucket component itself. An allocation may only take $O(\log(N))$ time if a page is exchanged with the tree component.

The simple reordering scheme (step 5 for allocation, step 4 for deallocation) has some interesting properties with regard to heap fragmentation. All completely used blocks reside as a compact sublist at the end of the list of page records of a bucket. Conceptually this sublist is a separate list, the *black list*. The rest of the page records which all have at least one free block form the *white list*.

If a block is returned to a black page, that page turns white and moves to the front of the white list. The expected number of free blocks in a page record increases with the time it spent in the white list without being in front. The distance of a white block from the front of the white list also grows with the time it spent in the white list. Thus, in the average the number of free blocks per page increases with its distance from the front of the white list. Since allocations are always satisfied from the front of the white list, the pages with the largest amounts of free

blocks are least likely to be used. This increases the probability to obtain entirely free pages and thus minimizes heap fragmentation.

Since allocation patterns and lifetimes of allocated memory blocks are not necessarily random one must ask the question how the scheme performs in practice. The experimental comparisons of the 'maximum resident set size' under AIX and Linux (section 3) for a variety of allocation patterns show that the memory footprint of BFM, together with that of the glibc allocator, is the smallest out of the five evaluated allocators. Furthermore, the bucket component realizes two mechanisms that increase locality compared to a pure 'coalesce by address' scheme as for example implemented by the tree component, *type separation* and *regrouping*.

Type separation occurs if different data types fall into different buckets. Often several structures, e.g. a graph and a binary tree, are constructed simultaneously, creating an *alternating* allocation pattern *ABABAB* or *ABCABC*. If there are no reusable blocks, an allocator like the tree component cuts pieces alternatingly for tree nodes and graph edges, mixing them in memory. If blocks are obtained from different buckets graph edges and tree nodes are in disjoint page records.

Regrouping occurs when free blocks belonging to the same page record are reused as a group, even if they do not form a consecutive memory block. Consider the construction of a binary tree and subsequent deletion of a part of this tree. Since there is typically no correlation between the address and the tree order, the deallocated blocks are more or less evenly distributed over the page records initially used for the tree nodes. The allocator reuses blocks one page record at a time, i.e. in groups that are within an area of 1024*sizeof(void*). Thus, a subsequent set of allocations from the same bucket has at least initially a higher locality (lower average distance between memory accesses) compared to reusing blocks in random or deallocation order. This helps in particular to reduce the number of TLB misses.

2.2 The Tree Component

The tree component of the allocator handles blocks larger than 64*sizeof(void*) and is based on a pair of binary trees. Each free block is present in both trees. The ptrtree holds free blocks ordered by their start address. The sizetree holds linked lists of free blocks of size S ordered by size (figure 3). This is an *indexed fit* [5], the efficiency results from the choice for the binary tree (see section 2.2.2).

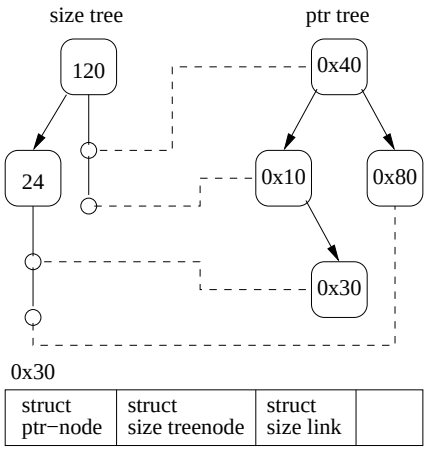

Figure 3: *The* sizetree *(doubly linked lists per size),* ptrtree *and memory layout of a free block creating implicit links (dashed lines).*

The trees are implicitly linked since the memory for the tree nodes and list links resides at fixed locations at the beginning of the free block, requiring 16 sizeof(void*). This space is available in the free block since smaller blocks are handled in the bucket component.

2.2.1 Operations

An allocation performs the following operations:
1. *Find smallest block in* sizetree *that satisfies request, obtain a new block via* sbrk *if necessary.*
2. *If rest is less than* 64 sizeof(void*)*, remove it from* ptrtree *and* sizetree*.*
3. *If rest larger than* 63 sizeof(void*)*, split it and update the* sizetree*. The* ptrtree *remains unchanged.*
4. *Create prefix that holds the size and return it.*

A deallocation performs the following operations:
1. *Use the prefix to determine the size of the block.*
2. *Find predecessor and successor in the* ptrtree*, merge if necessary.*
3. *Remove any unused nodes from* ptrtree *and* sizetree *and create appropriate new node in* sizetree*.*

The use of the trees is fairly straightforward, with the exception that coalescing takes advantage of the fact that merging a freshly deallocated block with a previously deallocated block is possible without

modification of the structure of the ptrtree, even if the key (start address) of the resulting block is different. The majority of the manipulations in the sizetree are simple operations in the singly linked lists.

It is possible to replace the ptrtree with Knuth's *boundary tags* [9] and thus to replace the logarithmic operation of the ptrtree with a constant operation. But due to the operations in the sizetree the worst case complexity is still $O(\log(N))$ and the impact on the elapsed time of actual programs is small since the time spent in allocation of a larger block compared to the time using the larger block is small. Boundary tags require additional space at the end and the beginning of each allocated block and working without boundary tags appears to help locality.

2.2.2 Choice of Balanced Tree

While the dual-tree mechanism is fairly straightforward, the choice and implementation of the trees is critical. Analysis and experiments show that a careful implementation of red-black trees provides significant advantages with respect to performance and robustness.

Red-black trees guarantee balance within a constant factor. Furthermore, red-black trees rotate only a constant number of times per operation in the worst case, at most once per insertion and at most twice per deletion. The remaining operations are comparisons between elements during the search and comparisons and modifications of the color bits after insertions and deletions. Furthermore, in many cases a constant number of color bit modifications is sufficient to fix the tree structure after an insertion or deletion. Thus, in many cases red-black trees require only a constant number of write operations per insertion or deletion, which are particularly expensive on large SMP machines.

Splay trees require amortized $O(\log(N))$ write operations for each insertion and deletion. Furthermore, they require splaying and thus write operations during searches to avoid multiple searches in a temporarily unbalanced tree. Both bottom-up splaying and top-down semi-splaying [1] performed significantly worse than the red-black tree in our tests. Some allocators use Cartesian trees [8]. In comparison to red-black trees Cartesian trees may become quite unbalanced and thus require rebuilding to maintain a logarithmic complexity. It is also not clear how well Cartesian trees approximate a best fit [5].

Last, but not least, red-black trees provide a logarithmic worst case *per operation*, not only amortized. Thus they are not only fast, but also well suited for environments that are sensitive to the latency of individual operations.

2.2.3 Tree Implementation

The red-black tree implementation itself is based on the description in [2] with a few modifications. We found a few opportunities to reduce the number of comparisons by code restructuring. The most significant change affects the use of the sentinel node.

The algorithm presented in [2] uses a shared sentinel node such that a value written to the sentinel is subsequently used in the tree fix up. This requires that a modification in one tree has to be completed before such an operation is started in another tree. This hampers thread safety and interoperation of linked trees. This is not hard to avoid and using the sentinel as a 'write only' memory field saves a branch instruction by not testing for the terminal node in several frequently executed places.

3 Experiments

This section provides experimental results for several allocators and comparisons to BFM. Under Linux both the glibc (2.3.2) allocator and the Hoard allocator (2.1.0) are compared against BFM. Under AIX 5.1 the default allocator and the bucket allocator (MALLOCTYPE=buckets) are compared against BFM. We also tried the latest version of the Hoard allocator (3.0.2), but frequent segmentation violations both on a SuSE 8.2 and a RedHat 9 system prevented its testing.

This section describes two types of experiments. The first set of experiments uses patterns with varying degree of complexity inspired by patterns seen in subroutines of applications. The second set of experiments uses an allocation intensive task in VLSI layout processing, a scanline based *netbuild*.

We also ran a few comparisons with standard programs using `LD_PRELOAD` under Linux. The differences in compiling a source module using a few larger template headers with g++ were minor (glibc 2.4 s,

Hoard 2.5 s, BFM 2.4 s). The same is true for running an 80 page postscript file through ghostscript (glibc 7.2 s, Hoard 7.5 s, BFM 7.3 s).

3.1 Patterns

A large application is typically composed out of smaller subroutines which show characteristic allocation patterns. Figures 5, 6, 7 and 8 show results for 28 patterns described in this section.

A routine that collects a number of elements in a queue and then processes and deallocates the elements one-by-one creates a *fifo:S* pattern with a single block size S (in units of sizeof(void*)). A similar routine using a stack creates a *lifo:S* pattern. Patterns 1 and 2 in the results are *fifo:13* and *fifo:111*, patterns 3 and 4 *lifo:13* and *lifo:111*.

Alternating patterns are also common. Consider two data structures that are constructed concurrently. After evaluation their destructors deallocate them. If the block sizes are A and B, the allocation forms a sequence $ABABAB$ and the deallocation forms $AAABBB$, providing the content of each list in lifo or fifo order. We denote such patterns with *alt:7,17* (pattern 5), the numbers separated by commas indicating the different sizes involved. Patterns 6 and 7 are *alt:17,176* and *alt:93,192*.

A third common type are actually random patterns. For example a list is constructed, sorted and subsequently destructed. Typically the sort order does not correlate with the initial location of the blocks in memory and thus elements are deallocated in pseudo-random order with respect to the order of allocation. An analogous scenario is the filling and destruction of a balanced binary tree. Another pseudo-random type of pattern is created by scanline algorithms that involve several data structures as described in section 3.2. When and what sizes of elements are allocated and deallocated depends on the geometric input and varies greatly for example by the type of VLSI design level being processed. Patterns 20 to 24 are *r:1-63*, *r:64-2023*, *r:1024-2048*, *r:1000* and *r:700-1000*, respectively. The notation *a-b* indicates that multiple sizes between a*sizeof(void*) and b*sizeof(void*) bytes are used.

If more than two sizes are involved and different destructors operate in lifo or fifo or random order more complex patterns are created. We denote such patterns with 'multi:a-b'. These patterns require coalescing to maintain a small memory footprint. They typically allocate N elements in a certain grouping, then deallocate one half of the allocated elements in a certain pattern. Next they allocate $N/2$ new blocks with slightly different sizes before cleaning up. The patterns 8 to 19 are off the type m_i*:1-N/128* for even pattern indices i and m_{j-1}*:67-N/128* for odd pattern indices j, i.e. with and without small blocks. Pattern 25 (figure 4), 26, 27, 28 are of type *ma:1-250*, *mb:1-90*, *mb:1-10000* and *mb:1-20000*, respectively. All 28 patterns initialize the allocated memory and use a few hundred MB of memory except for *r:1-63*, which uses a few dozen MB.

```
ptrtype *field = malloc(N*sizeof(ptrtype));
for (i=0; i<N; i++) {
  field[i]=malloc((of+((1+i)>>8))%hs);
  memset(field[i],0xDF,(of+((1+i)>>8))%hs);
}
for (i=0; i<N; i+=2) free(field[N-2-i]);
for (i=0; i<N; i+=2) {
  field[i]=malloc((of+((1+i)>>7))%hs);
  memset(field[i],0xDF,(of+((1+i)>>7))%hs);
}
for (i=0; i<N; i+=4) free(field[i]);
for (i=0; i<N; i+=4) free(field[i+1]);
for (i=0; i<N; i+=4) free(field[i+2]);
for (i=0; i<N; i+=4) free(field[i+3]);
free(field);
```

Figure 4: *Generator for pattern 25.* N *controls the number of allocations in the patter,* of *is the smallest allocation size and* hs *the largest.*

3.1.1 Results

Figures 5, 6, 7 and 8 show comparisons of elapsed times and memory usage for BFM and the four memory allocators mentioned above. The Linux experiments were performed on a 2.8 GHz Pentium4®with 1 GB of dual channel DDR400 RAM. The AIX experiments were performed on an IBM pSeries®[2] model 630 with 4 processors and 16 GB RAM.

Figures 5, 6 show Linux results relative to the glibc allocator, i.e. Hoard/glibc and BFM/glibc. Figures 7 and 8 show AIX results relative to the AIX default allocator. The figures comparing performance mark the 4/3 and 3/4 ratio. Execution

[2]IBM and pSeries are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

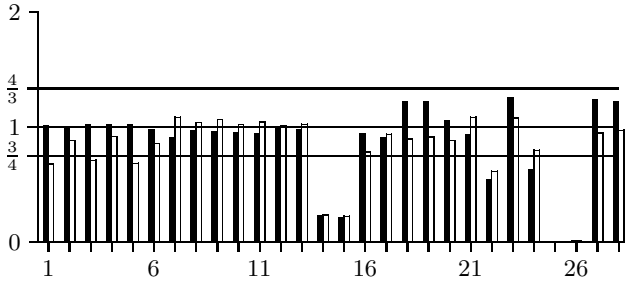

Figure 5: *Total elapsed times with Hoard (black bars) and BFM (white bars) for 28 patterns relative to the glibc-allocator.*

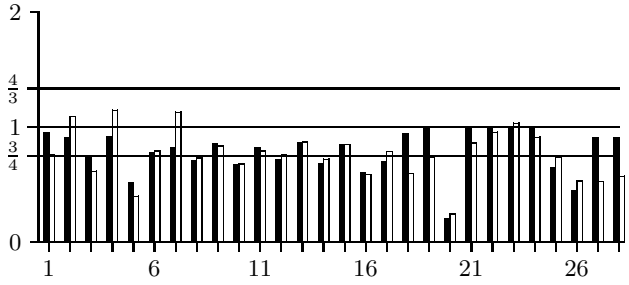

Figure 7: *Total elapsed times with the AIX bucket allocator (black bars) and BFM (white bars) relative to the AIX default allocator.*

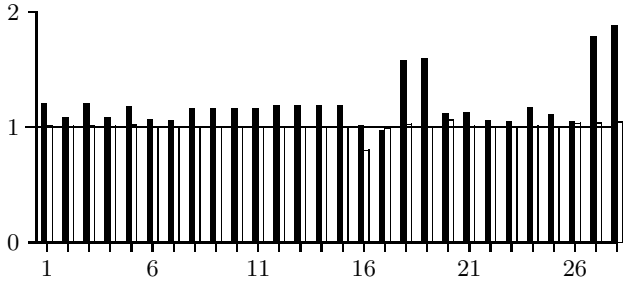

Figure 6: *Maximum VmSize from /proc of Hoard (black bars) and BFM (white bars) relative to the glibc allocator.*

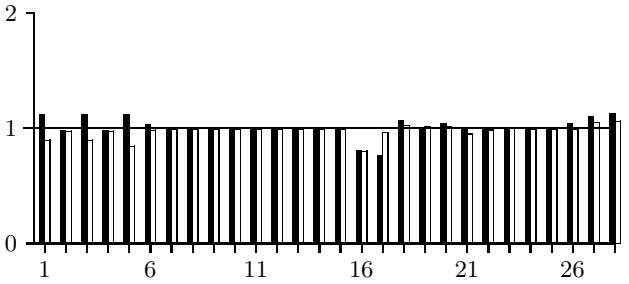

Figure 8: *Maximum size (getrusage) of AIX bucket allocator (black bars) and BFM (white bars) relative to the AIX default allocator.*

time variations of this order of magnitude in the patterns are in general mitigated by non-allocation computations. Effects of this size are for example caused by the type of lock that is used, to ensure a fair comparison BFM was set up with a spin lock analogous to Hoards.

The AIX bucket allocator, Hoard and BFM work efficiently for all patterns. The glibc allocator is slow on a few patterns, two of which show pathological behavior. Figure 9 shows how for pattern 25 the execution time grows much faster than linear with increasing pattern size. The execution of this pattern with 1.5 million elements (less than 1 GB memory) takes with the glibc allocator a few hundred times longer than with BFM or Hoard and the ratio is growing rapidly with pattern size.

The AIX default allocator is slow for patterns involving small blocks. While none of the cases is as bad as the execution time with the glibc allocator for pattern 25 and 26, BFM or the bucket allocator provide significant performance improvements for allocation intensive applications.

The glibc, AIX and BFM allocators waste little memory for all patterns, but Hoard sometimes does not coalesce blocks successfully. In the most extreme cases the total memory footprint is almost doubled.

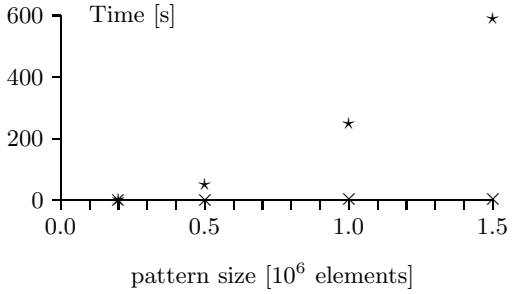

Figure 9: *Pattern 25 for different sizes for glibc ($\star$) and BFM ($\times$).*

3.2 Netbuilding

The input to a VLSI netbuild consists typically out of several large sets of rectangles, one set per VLSI design layer. In recent microprocessor designs millions of rectangles are common per layer in larger cells. The set of layers forms a 'stack' such that a layer 'B' connects electrically to the layer 'A' directly below and the layer 'C' directly above where two rectangles overlap. Within a layer 'A' electrical connectivity is established where two rectangles in 'A' overlap. Netbuilding establishes electrically con-

nected sets of rectangles, the nets. Figure 10 shows a flow diagram of the scanline algorithm.

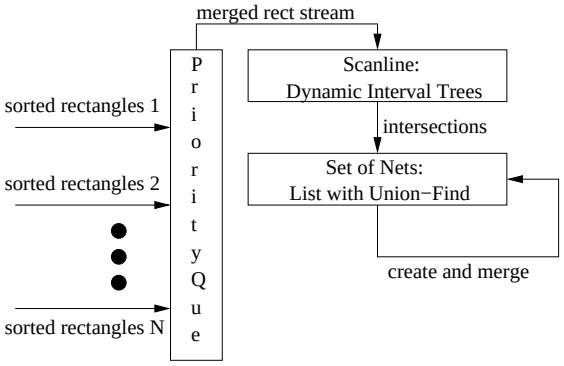

Figure 10: *Flow diagram for netbuilding.*

The code generates a number of lists with random rectangles simulating multiple layers of a VLSI design. Each of those lists is sorted. A priority queue merges the N streams into a single stream of geometry that is processed by a scanline. Rectangles intersecting the scanline are stored in N dynamic interval trees used to find rectangle intersections. Each processed rectangle may start a new net or merge existing nets. The set data structure representing the nets uses *union by rank with path compression* [10].

The code performs roughly 10 new/delete pairs per processed rectangle distributed over roughly the same number of data types over a size range of 16 to 64 bytes (64 bit). A significant improvement in locality and thus performance is achieved by copying geometry into the interval trees rather than referencing the original input. The scanline intersects roughly $O(\sqrt{N})$ elements at any given time and performs the majority of accesses. Netbuilding forms a pattern of the mixed type *multi-random:2-8*.

Figure 11 compares memory usage and execution time for the five allocators. The two Linux allocators and BFM are close in this case, which correlates to the performance result of pattern 20, pseudo random operations on small blocks.

The AIX default allocator uses significantly more memory and time than BFM. The AIX bucket allocator improves the execution time, but uses even more memory. While the performance correlates with pattern 20, memory waste is primarily caused by internal fragmentation due to the small block sizes. Note the increase of memory waste of 64-bit

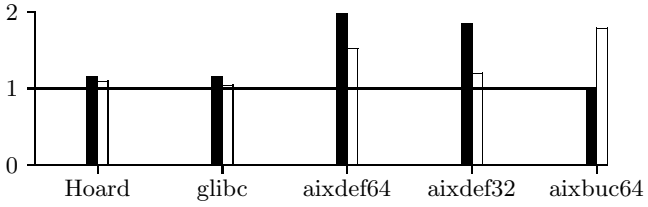

Figure 11: *Netbuilding (BFM='1': Intel 32-bit: 134 sec, 602 MB; AIX 64-bit: 191 sec, 989 MB, AIX 32-bit: 169 sec, 600 MB). Black bars show elapsed time, white bars show memory.*

compared to 32-bit code due to the larger pointer size but equal integer size.

3.3 Multiple Threads

Various schemes to improve scalability for multiple threads can be applied to BFM, for example using *multiple heaps with ownership* (glibc allocator) or a scheme like Hoards multiple specialized heaps that fall back onto a single main heap.

In order to investigate how BFM performs in a parallel scheme, we adopted the approach of Hoard: Multiple bucket components that fall back to a single tree component. Note that the coalescing within a page record is not affected by using multiple bucket components, only bucket components that are not used by any thread may contain records whose reuse is missed. A form of *emptiness threshold* [4] is naturally provided by the ratio of block size to page size, since entirely empty pages return to the tree component.

The netbuilding application described in section 3.2 stresses the memory allocator as well as the memory subsystem of the hardware and exposes any limitations to scalability at a relatively small number of threads. In a multi threaded environment N netbuild jobs corresponding to cells may be parallelized over M processors (typically $N >> M$).

In large multi-threaded applications there are typically many more independent tasks than processors. Besides of potential dependencies between tasks there are two reasons not start all N tasks as threads at once. On one hand starting many more threads than processors increases the peak memory usage without the benefit of additional speedup. On the other hand starting all threads at once reduces load balancing since all threads obtain the same share of compute power regardless of their size.

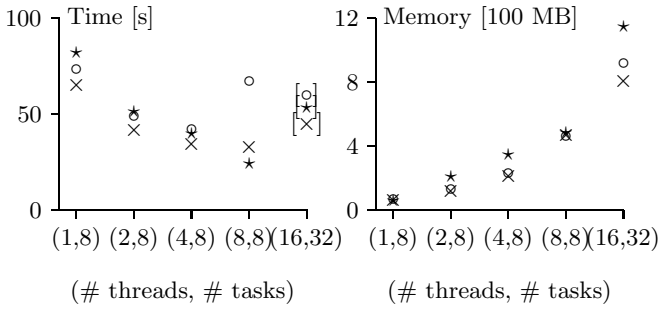

Figure 12: *Multi-threaded netbuild tasks (glibc: ⋆, BFM: ×, Hoard: ∘). [] indicate division by 4.*

To capture the character of multi-threaded applications with a large number of tasks we used a set of identical netbuilding tasks, each processing 10^6 shapes distributed over seven layers. The number of tasks is chosen as a multiple of the number of threads such that all threads remain busy during the entire run. Condition variables were used to coordinate scheduling to a fixed number of threads.

Figure 12 shows elapsed time and peak memory usage for a set of combinations (N, M) of N threads and M netbuilding tasks. The experiments were performed on a 4-way IBM xSeries®[3] model 330. BFM using Hoard's multi heap scheme scales similar to Hoard, carrying over an advantage in memory usage. For the execution time the same holds for *glibc*. The glibc allocator with its entirely separate heaps uses more and more memory compared to Hoard and BFM, demonstrating the impact of *blow up* [4].

There are two combinations, $(8, 8)$ and $(16, 32)$, that deserve additional attention. Case (16,32) tests the behavior for a number of threads larger than the number of (virtual) CPUs. Memory usage increases proportional to the number of concurrent tasks, speedup is reduced due to lower locality and higher thread scheduling overhead.

For the case $(8, 8)$ the number of tasks matches exactly the number of processors presented by the operating system (four hyper-threading CPUs appear as eight). All threads are scheduled at the start of the application and thus heap sharing provides no memory benefit. The glibc allocator appears to gain a performance benefit from its better locality since there is no exchange between the heaps. The untypically large execution time of Hoard for the case $(8, 8)$ (also present for $(4, 4)$) appears to be rather an artefact of the implementation than a problem of the scheme, since BFM using a very similar scheme does not experience it. Establishing affinity between threads and heaps is very sensitive to the hash function used.

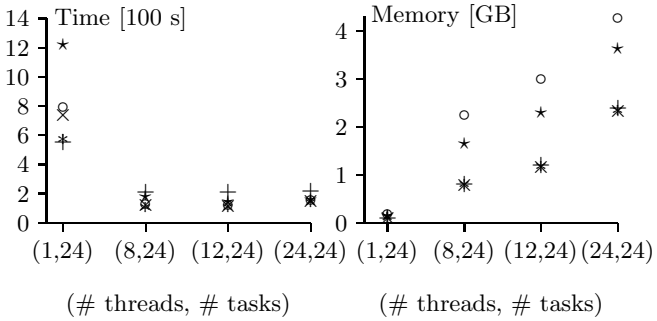

Figure 13: *Multi-threaded netbuild tasks (AIX multiheap: ⋆, BFM multiheap: ×, BFM and thread specific cache: +, BFM multiheap and thread cache: ∗, AIX bucket multiheap: ∘). Platform: 24-way IBM S80, 64-bit code.*

Figure 13 shows the analogous experiment on a 24-way IBM S80. BFM using multiple bucket components (symbol ×) uses less memory than the AIX allocators across all test cases. The AIX bucket allocator with multiple heaps (symbol ∘) delivers similar performance to BFM, but requires much more memory. The AIX default allocator using multiple heaps (symbol ⋆) is slower than the bucket flavor but uses slightly less memory.

With an increasing number of threads and thus CPUs, the performance difference between the multi-heap approaches shrinks together with the speedup as the execution time becomes dominated by memory accesses and processor bus use.

Figure 13 indirectly demonstrates the importance of memory usage. An application with higher computational cost per processed data volume than net building scales to much higher numbers of processors and the impact of memory allocation on total runtime is much lower. For this form of parallelism by data partitioning non-sequential memory usage increases proportional to the number of threads, such that at some point the amount of core memory dic-

[3]IBM and xSeries are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

tates how many threads can be executed concurrently without paging.

3.4 Thread Specific Caching

Multi-heap solutions as discussed in the last section have an Achilles heel, in the worst case all deallocations may hit the same heap. If only complexity is considered, the slowdown is at most a factor of two [4]. Unfortunately, the penalty is in practice much larger.

Using the AIX bucket allocator, experiment $(1, 24)$ (figure 13) takes 790 seconds on one CPU. Only a fraction of this time is spend in allocation. Experiment $(24, 24)$ takes 1201 seconds (over 28000 CPU seconds), the impact is much worse than just serialization of a fraction of the execution time. Microbenchmarks show that a lock transfer takes roughly 1000 clock cycles.

Thread specific caching addresses this worst case behavior for the most critical case of smaller blocks. A thread specific cache contains a stack of depth d, here $d = 8$, for each size range of the bucket component. Memory usage per thread is less than 3568 sizeof(void*) bytes for $d = 8$. Since the cache is thread specific, it is not necessary to protect it by a lock. When the stack is empty or full, $d/2$ blocks are obtained from or returned to the heap with a single lock acquisition, respectively.

Figure 13 shows that thread specific caching on a single heap (symbol +) drops the execution time of our worst case experiment from 1201 to 221 seconds. Additionally, the single-thread execution time drops since the cost of at least one quarter of the uncontended lock acquisitions is avoided.

Combining BFM's multi heap scheme with thread specific caching (symbol $*$) combines the benefit of using multiple bucket components (fastest for high thread numbers) with thread caching (fastest for low thread numbers) and significantly improves the 'single lock' worst case.

4 Memory Tracing

There are excellent tools that find memory leaks and errors, e.g. *Valgrind*. But those tools cause serious increases in execution time and memory use and are not able to locate causes of high memory use that are not a leak. A typical example are obsolete global variables. They are cleaned up properly by their destructors at the end of the program, but could have been released much earlier.

There are rather efficient tools that determine the total memory usage of a program run and even allow plotting usage over time. But only knowing at which point in time the usage peak occurred makes it hard to correlate memory usage with the actual allocation. Furthermore, the larger the program the harder becomes the task of finding memory inefficiencies.

In this section, we present a highly efficient memory profiler which allows to plot for each call chain the amount of memory it owned over the time line based on a trace file generated during a single run. The trace overhead is sufficiently small that tracing is feasible even for programs that run untraced for multiple hours and use multiple GB of memory. Furthermore the trace files are reasonably small. While the BFM profiler finds leaks and can check for false deallocations, it does not monitor load and store instructions as for example *Valgrind*.

Sampling as used for profiling of execution time is not feasible for this form of memory profiling. The allocation of a block has to be attributed to a call chain. Later the deallocation of the block, which typically occurs in a different call chain, has to be deducted from the allocating call chain. Thus it is necessary to capture each allocation event and to maintain a record for every memory block that is in use by the program.

There are three key components in order to accomplish this task: *Event generation*, *online event accounting* and *trace compression*. Each of these components described in the following subsections contributes to reducing the amount of work and trace data.

4.1 Event Generation

An allocation event consists of its call chain (as a sequence of return addresses) and a time stamp in addition to address and size of the allocated block. A deallocation event consists of the block address and a time stamp.

Creating an event for every allocation and deallocation takes too much time. Thus, event generation uses *page record allocation and deallocation* to generate events for small blocks, large blocks generate

events directly.

Two properties make the use of *page records* to collapse events a feasible approximation. The bucketing provides a certain amount of type separation, i.e. the page allocation events for two types that fall into different buckets are separated and thus pages are attributed correctly to the call chains. But obviously it is a common case that multiple call chains allocate from the same bucket B.

Let N_c be the number of *page record events* on B assigned to a call chain C. Let A_c be the number of block accesses (malloc/free) performed by C on B and let A be the total number of accesses performed on B. N is the total number of page record events performed for B. For large numbers of allocations it holds

$$\frac{N_c}{N} \approx \frac{A_c}{A}.$$

The formula is an approximation since the allocation requests may have small variations in waste and it is assumed that the allocation pattern does not correlate with the number of blocks in a page record. This is in practice the case, the approximation is analogous to time profiling by sampling the stack at fixed time intervals.

4.2 Online Accounting

Even with the reduction of events described in the last section, trace file sizes and the time to write them would be excessive if the resulting events were recorded directly. The next stage reduces the event stream to a stream of *amount changes* associated with call chains. One recorded *change* may cover multiple events. For this purpose two dictionaries are used.

The first dictionary uses call chains as keys and stores the amount currently owned for a call chain. Each allocation event for a call chain increases or decreases the amount owned.

The second dictionary provides the mechanism to find the record of the allocating call chain based on the address of a deallocated block. It uses the address of an allocated block as a key and holds a reference to the record of the allocating call chain as well as the block size for each block currently used. Upon deallocation the call chain record is updated and the block record deleted. Since each event carries a time stamp each update of a call chain record provides one data point for the plot of the affected call chain.

Note that using the page records helps also to limit the memory overhead due to tracing. For every used block the tracer has to hold a block record. The block record requires 6*sizeof(void*). But the smallest allocation event to record is of size 64*sizeof(void*) and most events are actually the size of a page record, 1024*sizeof(void*). Thus the application allocates at least 10 times as much memory as the tracer requires to keep track of the blocks. The remaining memory overhead of the tracer is required for the call chain records.

4.3 Trace Recording

Despite event reduction and online accounting the number of records with time, call chain index and size still is still too large to generate trace files for computations that run for multiple hours. But analysis of traces shows that especially for large program runs there is potential for lossy compression.

After the first allocation of a call chain C, which establishes the initial record, changes are recorded only if a call chain holds more memory than the size of a *page record* and if the change is larger than a given threshold, e.g. 5 % of the last recorded value. This accuracy is completely sufficient for applications that use multiple Megabytes and it eliminates the small fluctuations that otherwise bloat the trace files.

4.4 Implementation

The implementation of the trace generation with lossy compression and the event generation is fairly straightforward. But the online accounting poses a few problems. On one hand the accounting generates a fair amount of allocations itself. On the other hand there are millions of dictionary lookups with rather long keys, the call chains. Some of the trace experiments were performed on a VLSI application with several hundreds of thousands of lines of code, call chain lengths of tens of steps are common. Also the number of call chains is rather unpredictable and can be large. The aforementioned VLSI application generated events in over 5000 call chains on optimized code with a significant amount of inlining.

In order to minimize the disturbance of the traced

process, only the *event generation* takes place there. The *online accounting* and *trace generation*, which allocates memory itself to maintain dynamic structures, is performed in a child process which is forked at the first trace event. This separates the allocations of the tracing and the traced code and takes advantage of available additional processors.

Efficient handling of the call chain records is a challenge. The keys are long and degenerate since there are large common subchains. Hashing has to evaluate the entire key and a comparison-based tree looks in the average $O(\log(N))$ times at a significant number of call chain steps to determine a difference. A tree structure called PATRICIA [6] turned out to be an excellent solution.

A PATRICIA is a binary tree based on comparison of individual bits. But rather than comparing a bit of a fixed position on each level of the tree, it records in each node the index of the differentiating bit. In the average case the number of bits evaluated in a search, insertion or deletion is independent of the key length and logarithmic in the number of elements in the PATRICIA.

4.5 Experiments

We traced several runs of a large VLSI application (runtime 10 hours on an IBM pSeries 690, memory peak 18 GB, single thread). Using a separate processor for the tracer, the elapsed time of the original program (with default AIX allocator) was a few minutes longer than for the traced version (BFM + tracer). Using BFM saved more time than the event generation required. The tracer used less than 2 GB additional memory. The zipped trace file was roughly 250 MB in size.

Figure 14 shows the two main contributors in the netbuild test case. Random input is created at the beginning and remains present until the end of the program. After the lists are sorted the nets are formed, the slope of the linear increase reflects the nearly constant throughput of the scanline. The nets are destructed first after the search.

Tracing the netbuild test case was performed on a single CPU Power4 machine. It increased the elapsed time from 173 seconds to 192 seconds. The memory usage of the traced process was unchanged (slightly below 1 GB) and the tracer process required 2 % of the memory of the traced process. There were 20 call chains, four with significant activity. The (uncompressed) trace file was 16 kB in size, less than 5 kB compressed. This test case performs a little less than 100 million allocation and deallocation events.

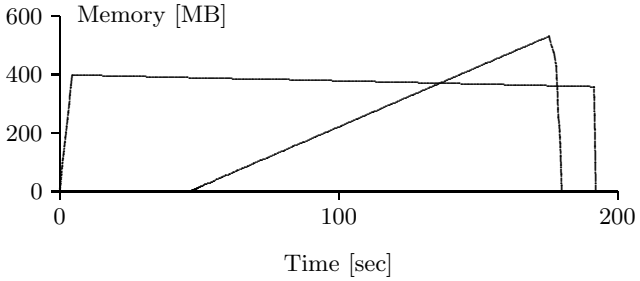

Figure 14: *The two main contributors to the netbuild, creation of random input and forming of nets during the search.*

5 Conclusion

Memory allocators with provably good worst case behavior and best fit allocation carry the stigma of being slow in practice. As the experiments and analysis presented in this paper show, the BFM heap structure combines competitive performance with very low memory waste. The worst case complexity of $O(\log(N))$ with small constants holds for each individual operation. The heap structure also performs well in parallelization schemes and allows efficient memory profiling.

Confirming the theoretical arguments, BFM's memory usage and performance was throughout the experiments competitive with the best results out of four widely accepted allocators without showing a bottleneck. This is particularly important for memory usage, since the transition from 32-bit to 64-bit address spaces tends to increase the impact of memory waste.

Affinity between heaps and threads in multi heap schemes as well as approaches like thread specific caching is becoming increasingly important as the gap between processor clock and memory latency widens. An open topic for future research is the question how a better integration between thread libraries and allocators may further improve scalability.

References

[1] Sleator, Tarjan. *Self Adjusting Binary Trees.*, Journal of the ACM, Vol. 32, No 3, 1985, pp. 652-686.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*, MIT Press.

[3] R. Sedgewick. *Algorithms*, 2nd Edition Addison-Wesley 1988.

[4] E.D. Berger, K.S. McKinley, R.D. Blumofe, P.R. Wilson. *Hoard: A Scalable Memory Allocator for Multithreaded Applications*. ASPLOS-IX, Cambridge, MA, 2000.

[5] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles. *Dynamic Storage Allocation: A Survey and Critical Review*, Proc. 1995 International Workshop on Memory Management, Springer Verlag LNCS.

[6] D. R. Morrison. *PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric*, Jrnl. of the ACM, 15(4) pp514-534, Oct 1968.

[7] I. Puaut. *Real Time Performance of Dynamic Memory Allocation Algorithms*, 14th Euromicro Conference on Real-time Systems, Vienna, Austria, June 2002.

[8] C.J. Stephenson. *Fast Fits: New Methods for Dynamic Storage Allocations*, Nineth symposium on operating system principles, October 1983.

[9] D.E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley, 1973.

[10] A. Aho, J. Hopcraft, and J. Ullmann. *Data structures and and algorithms*, Addison Wesley, Reading, Mass., 1983.

[11] *Valgrind, a GPL'd system for debugging and profiling x86-Linux programs.* http://valgrind.kde.org/

[12] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, Dafna Sheinwald. *Thread-local heaps for Java*, Proceedings of ISMM 2002, pages 76-87.

[13] Dave Dice, Alex Garthwaite. *Mostly Lock-free Malloc*, Proceedings of ISMM 2002, pages 163-174.

[14] David Detlefs, Al Dosser, Benjamin Zorn. *Memory allocation costs in large C and C++ programs*, Technical report CU-CS-665-93, University of Colorado at Boulder, 1993.

[15] Brad Calder, Dirk Grunwald, Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical report CU-CS-698-94, University of Colorado at Boulder, 1994.