CAMA: Cache-Aware Memory Allocation for WCET Analysis

Jörg Herter
Department of Computer Science
Saarland University, Germany
Email: jherter@cs.uni-sb.de

Jan Reineke
Department of Computer Science
Saarland University, Germany
Email: reineke@cs.uni-sb.de

Reinhard Wilhelm
Department of Computer Science
Saarland University, Germany
Email: wilhelm@cs.uni-sb.de

***Abstract*—Current WCET analyses do not support dynamic memory allocation. This is mainly due to the unpredictability of the cache performance if standard memory allocators are used. We present a novel dynamic memory allocator that makes cache performance predictable and (de)allocates memory in constant time. It thereby enables WCET analysis in the presence of dynamic memory allocation.**

I. Introduction

At present, static worst-case execution time (WCET) analyses exist exclusively for programs with static memory allocation. However, supporting dynamic memory allocation as well would be desirable for a number of reasons:

- It often allows to save memory space, e.g. by immediately reusing the newly available space when converting one data structure into another.
- It is sometimes more natural to use, i.e. it gives a clearer program structure.

Why is dynamic memory allocation not supported by current WCET analyses? In order to give safe and reasonably precise estimations of the WCET, analyses have to derive bounds on the cache performance. They have to be able to statically classify most memory accesses producing cache hits during program execution as such. For standard malloc implementations this is impossible. Since these do not provide information about the addresses of allocated memory. In particular, a cache analysis does not know which cache sets allocated memory will map to. To obtain guarantees on the cache performance, an analysis would need to know which blocks of data compete within the cache, i.e. which blocks may evict each other from the cache. In addition, memory allocators cause cache pollution themselves. Maintaining and traversing their internal data structures, they influence the cache contents in an unpredictable way. Another somewhat less severe problem with standard malloc is that its execution time cannot be easily bounded.

This work is supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and the German-Israeli Foundation (GIF) in the "Encasa" project.

Our novel dynamic memory allocator alleviates these problems:

- It allocates and deallocates memory in constant time.
- It causes only a small, constant amount of *cache pollution*, completely predictable in the sense that one can statically determine which cache sets are affected.
- Allocation to cache sets can be controlled by an additional parameter.

We describe the main ideas of the new memory allocator and possible challenges in its implementation. Then we discuss how its properties can be exploited to obtain safe and precise WCET estimations.

A. Previous and Related Work

Dynamic memory allocators with bounded worst-case execution times have been investigated for many years. The binary buddy system is a long-known allocation algorithm whose WCET can be calculated. However, it suffers from a relatively high internal fragmentation of about 28% [9]. Ogaswara proposed the – to our knowledge – first constant time allocation algorithm ([7]; as cited by Wilson et al. [13]) besides simple segregated lists which produce very high fragmentation. However, its fragmentation is still high compared to other existing allocators. TLSF [5], a dynamic memory allocator for real-time systems, achieves constant run times while producing tolerable fragmentation. For real-life programs, fragmentation produced by TLSF is similar to that caused by Doug Lea's memory allocator [4], currently considered to be the best general allocator available [6]. Our allocator was greatly influenced by TLSF. In fact, it can be regarded as a cache-conscious modification of TLSF.

Chilimbi et al. proposed a cache-conscious memory allocator (ccmalloc) in order to improve program execution times [2]. Compared to malloc, Chilimbi's ccmalloc takes as an additional argument a pointer to an existing data structure/object that is likely to be accessed contemporaneously (or at least contemporary) with the element to be allocated. ccmalloc achieves its goal by trying to allocate the newly requested storage next to the one pointed to by its second argument. As a result, newly

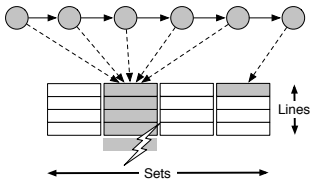

Fig. 1. Possible mapping from list elements to cache sets of a 4-way set-associative cache with 4 cache sets.

allocated storage is often located in the same cache set as the referenced one.

II. Caches, Cache Analysis, and Dynamic Memory Allocation

Caches are used to bridge the increasing gap between processor speeds and memory access times. A cache is a small, fast memory that stores a subset of the main memory contents. It is located at or near the processor. Due to the *principle of locality*, most memory accesses can be serviced by the cache, although it is much smaller than main memory, thereby drastically improving the average latency of memory accesses. In order to give safe and reasonably precise estimations of the WCET, *cache analyses* [3] have to derive tight bounds on the cache performance. In modern processors, turning off the cache easily causes a thirty-fold increase in execution time [8]. Conservatively classifying each memory access as a cache miss is thus not an option. To classify memory accesses as hits, the cache analysis needs to know the mapping of program data to cache sets. Otherwise, it does not know which memory blocks compete for cache lines. See the example of a linked list in Figure 1. In the example, the elements of the dynamically-allocated linked list map to the cache sets very unevenly. Five of the six list elements map to one of the four cache sets. While traversing the list, one of the list elements is already being evicted, although the list is much smaller than the cache. If the LRU replacement policy would be employed, a subsequent second traversal of the list would result in only one cache hit. With a standard malloc, a cache analysis would not even be able to guarantee this single hit: it has no knowledge of the mapping to cache sets. Furthermore, all knowledge of the cache analysis about previous cache contents would be lost while traversing the list.

We propose to extend the malloc-routine by an additional parameter that constrains allocation to a specific cache set. malloc$(size, set)$ shall then return memory at an address that maps to the cache set set. Given this routine it is easily possible to allocate memory relative to a given address in the cache. A second routine malloc$(size, pointer, rel_distance)$ shall return memory at an address $rel\_distance$ cache sets away from $pointer$. Using the latter, one can construct a list, allocating consecutive list nodes to consecutive cache sets and thereby evenly distributing the list in the cache in a predictable manner. Another possibility would be to allocate all list nodes in the *same* cache set. This scheme minimizes cache damage to other structures as only one cache set is affected, independently of the size of the list.

This approach breaks with the philosophy of caches being transparent to the programmer. However, we may automatically generate the additional parameters. Only the two schemes – distributing the elements of a data structure evenly in the cache and allocating all elements of a structure to the same cache set – seem reasonable to preserve cache predictability. Once decided which scheme to employ, we merely need to associate calls to malloc with data structures. The latter can be obtained from a shape analysis [11].

III. Cache-Aware Memory Allocation

A memory allocator manages free and in-use blocks of memory. Allocators must satisfy two conflicting demands: they should have fast response times to (de)allocation requests and minimize fragmentation, i.e. the amount of free memory not usable to satisfy requests[1]. For a memory allocator used in real-time systems, the demand for constant response times arises. We also need our allocation algorithm to be able to allocate memory blocks mapped to a given cache set. A survey on existing dynamic memory allocators can be found in [13].

In general, allocators strive to minimize fragmentation by applying some *placement choice*, i.e. decide where to allocate new blocks in order to keep fragmentation low. Splitting techniques to satisfy requests for smaller blocks and coalescing techniques to serve larger blocks supplement the set of main techniques utilized by allocators. We follow Wilson et al. by viewing allocators as a *mechanism* that implements a placement *policy*, motivated by a *strategy* for minimizing fragmentation [13]. The overall strategy determines acceptable, implementable policies for placing blocks in memory. These policies are then implemented by a set of algorithms and data structures, the mechanism. We propose the following strategy: Separately manage regions of free blocks mapped to the same cache set. Within those memory regions select a suitable free block whose size may be slightly larger than the requested size if that allows for finding such a block in constant time. The policies "manage free blocks mapped to the same cache set in several disjoint size classes" and "always

[1]Traditionally, fragmentation is classed as external and internal fragmentation. Internal fragmentation is due only to the allocation algorithms itself. It arises when larger blocks are served than requested. The wasted memory then occurs internal within the block. External fragmentation is due to an unability to serve a large contiguous block, although enough small non-contiguous free blocks are available. External fragmentation is caused by properties of the allocation algorithm and the sequence of allocation/deallocation requests.

select the most/least recently freed block from the smallest size class large enough to satisfy the request" (LIFO/FIFO good-fit[2]) meet the proposed strategy. It is reasonable to believe that the ordering of the free lists has a significant impact on the overall fragmentation. Whether we settle for a LIFO or FIFO good-fit policy, resulting in a most recently freed and least recently freed ordering, respectively, will be determined by a series of experiments using real-life programs[3]. These policies can be implemented as follows. Logically, we may think of the available memory as a partition of n disjoint regions where n is the number of cache sets. Each region is mapped to a distinct cache set. We can further partition those regions into memory blocks of distinct size classes. That is, for each cache set, we obtain a set of size classes consisting of memory blocks whose sizes are within that size class. The free memory blocks of a single size class can be managed and organized in a simple linked list (free list). We can further store pointers to the heads of all such free lists in a consecutive memory area, for example an array. This way, we reduce the problem of finding a suitable free block satisfying an allocation request to computing the index within that array where the address of an appropriate free list is stored. When a suitable free list is found, we simply return the first element of that list. If a memory block is to be deallocated, we compute the index of an appropriate free list into which to reinsert this block. We may then either add the free block as first or last element of that list, resulting in either a most or a least recently freed ordering on the list.

We can think of the array storing all addresses of free lists as a three-dimensional construction as depicted in Figure 2. The first layer encodes to which cache set the memory block shall map. In the second layer, neighboring partitions of memory blocks constitute size classes a power of two apart. That is, at index i on the second layer, a third layer containing all free lists for blocks of sizes in $[2^i, 2^{i+1} - 1]$ mapped to the same cache set is referenced. Hence, the second layer constitutes a simple segregated list for size classes of powers of two. To diminish the fragmentation

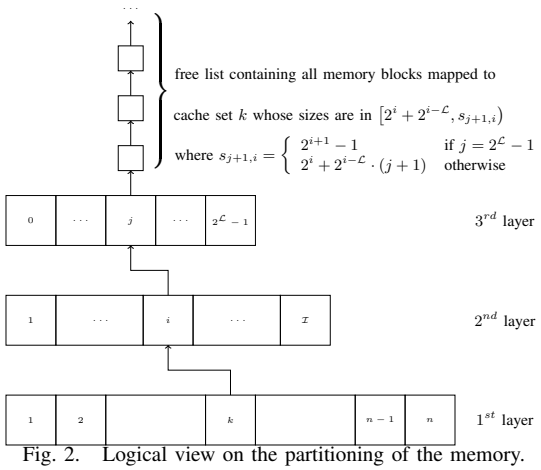

Fig. 2. Logical view on the partitioning of the memory.

that such a single segregated list would cause, we add a third layer in which size classes increase linearly. If all layers are organized in a single array by flattening their hierarchical structure, the corresponding index of the desired free list within that array can be computed in constant time by a mapping function $\mathcal{M} : \mathbb{N}^3 \mapsto \mathbb{N}$.

A request to deallocate a memory block will determine an appropriate free list for the given block and append it either to the head or the tail of this list, depending on our ordering strategy. While response times will be exceptionally good for our allocator, fragmentation might be a problem. We reduce fragmentation by a preanalysis of the program code in order to determine a safe approximation of the sequence of requests, both for allocation and deallocation, presented to the allocator during program execution. This information can be used to model the second and third layer of the allocator such that fragmentation is minimized. Hence, we first analyse the memory allocation behavior of the program, adjust the allocator accordingly, and then estimate the WCET. We will further investigate how well allocation behavior of real-life programs can be statically analyzed. This may result in a mechanism automatically selecting the best allocator for a program during compile time.

There are still some questions not answered in detail, most of them implementation specific. How can requests for blocks larger than one cache line be efficiently handled, what is a good initial partitioning into size classes, and how much of a problem is fragmentation in real-life programs? We are currently evaluating the following approach regarding requests for blocks larger than a single cache line. Requests for large blocks are in general very rare [13]. We may therefore allocate blocks destined to hold a large atomic object in a non-cached area of memory without significally increasing execution times. Large records (structs) can usually be split into smaller records that each fit into a cache line.

[2]Given a linked list of free memory blocks, a first fit algorithm would select the first block, starting from the head of the list, that is large enough to satisfy the allocation request. A best fit would select a smallest free block of the list large enough to satisfy the request, a worst fit would select a largest free block (given that this block is large enough to satisfy the request). A good fit selects the best (i.e. smallest) block chosen from some subset of fitting blocks. Hence, good fits in general are a tradeoff between best and first fit. They avoid an exhaustive search of the whole free list but might not select an optimal block.

[3]Similar experiments conducted by Weinstock [12] and more recently by Wilson et al. [14] showed that first fit with an address-ordered list produces significantly less fragmentation than LIFO-ordered first fit. Although a FIFO ordering for first fit has not been considered as thoroughly, there exist results suggesting that FIFO produces less fragmentation than LIFO; maybe as little as address-ordered first fit [14]. We believe that similar results will be obtained for our FIFO and LIFO good-fit policy.

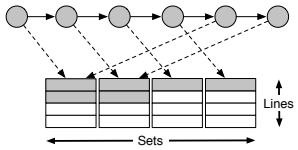

Fig. 3. Shape of a linked-list structure obtained from a shape analysis and its mapping to cache sets of a 4-way set-associative cache with 4 cache sets.

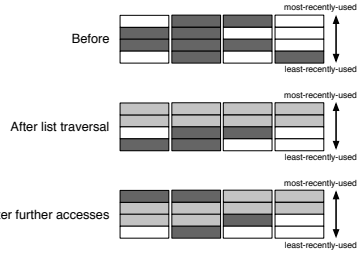

Fig. 4. Effect of traversing the linked list of Fig. 3 on static cache knowledge. Boxes shaded in dark gray indicate information about cache contents other than about list elements. Boxes shaded in light gray indicate information about list elements in the cache.

IV. WCET Analyses

How does a cache analysis exploit the properties of our new memory allocator? The main idea is as follows. Suppose, we have information about the shape of the dynamically-allocated data structures including the relative distances of objects in the cache. Such information can be obtained from a shape analysis. Consider, for example, a linked list as shown in Figure 3. If all six objects organized in that list are mapped to cache sets in such a way that neighboring elements are mapped to sets of relative distance 1, then traversing the list affects at most $\lceil \frac{6}{n} \rceil$ cache lines per cache set, where n denotes the number of cache sets. The number of cache lines per cache set affected by a traversal of a data structure can be used to (a) bound the information loss caused by that traversal and (b) infer hits for a second traversal. Figure 4 depicts this for our list example. In the example, we assume least-recently-used (LRU) replacement. Cache lines are sorted from most- to least-recently-used. Upon a cache miss, the least-recently-used element is evicted. Dark-gray-shaded boxes represent knowledge of a *must*-cache analysis [3]. For instance, a dark-gray-shaded box in the third line of a set indicates that the analysis "knows" that a certain memory block is in line 1, 2 or 3. Thus, dark-gray-shaded boxes represent upper bounds on positions in the LRU-stack. Traversing the list evicts at most 2 lines from each cache set. The analysis can thus safely infer that the two most-recently-used elements of each cache set are still contained in the cache (row 2), after list traversal. The upper bound on the position in the LRU-stack is increased by two. If the list is traversed again later in the program, it is also possible to safely predict cache hits for this traversal.

V. Summary and Conclusions

Our work is aimed at developing a static program analysis for determining WCET bounds for programs performing dynamic memory allocation. To enable such an analysis, we propose to replace the used memory allocator of the program by a predictable, cache-aware allocator and use this allocator to *guide* memory allocation with respect to the cache set mapping. Constant execution times are achieved by relying on segregated lists which is a common practice with real-time allocators [1], [5]. We combine a shape analysis with a WCET analysis to obtain WCET bounds for the analyzed programs. The shape analysis is necessary to compute heap shapes that contain information about the data structures arising during program execution. This information relates individual parts of the data structures to cache sets, allowing for a cache hit/miss classification of accesses to components of data structures as well as bounding the loss of information about contents of cache sets when loading structures into the cache.

References

[1] D. F. Bacon, P. Cheng, and V.T. Rajan, "A Real-Time Garbage Collector with Low Overhead and Consistent Utilization," *SPNOTICES: ACM SIGPLAN Notices*, 2003.

[2] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Making Pointer-Based Data Structures Cache Conscious," *Computer*, vol 33(12):67–75, 2000.

[3] C. Ferdinand and R. Wilhelm "Efficient and Precise Cache Behavior Prediction for Real-Time Systems," *Real-Time Systems*, 17(2-3):131–181, 1999.

[4] D. Lea, "A Memory Allocator," *Unix/Mail, 6/96*, 1996.

[5] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A New Dynamic Memory Allocator for Real-Time Systems," *IEEE Computer Society, ECRTS*, 2004.

[6] M. Masmano, I. Ripoll, A. Crespo, J. Real, and A. J. Wellings, "Implementation of a Constant-Time Dynamic Storage Allocator," *Software: Practice and Experience*, 2008.

[7] T. Ogasawara, "An Algorithm with Constant Execution Time for Dynamic Storage Allocation," *2nd Int. Workshop on Real-Time Computing Systems and Applications*, 1995.

[8] M. Langenbach, S. Thesing, and R. Heckmann "Pipeline Modeling for Timing Analysis," *Proceedings of the Static Analyses Symposium (SAS)*, volume 2477, 2002.

[9] J. L. Peterson, T.A. Norman, "Buddy Systems," *Communications of the ACM*, 20(6):421-431, 1977.

[10] M. Rezaei, K. M. Kavi, "Intelligent Memory Manager: Reducing Cache Pollution Due to Memory Management Functions," *Journal of Systems Architecture*, 2006.

[11] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric Shape Analysis via 3-valued Logic," *ACM Transactions on Programming Languages and Systems, Vol. 24, No. 3*, Pages 217–298, May 2002.

[12] C. B. Weinstock, "Dynamic Storage Allocation Techniques," PhD thesis, 1976.

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

[14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Memory Allocation Policies Reconsidered," technical report, 1995.