CAMA: A Predictable Cache-Aware Memory Allocator

Jörg Herter*, Peter Backes*, Florian Haupenthal*, and Jan Reineke†
*Department of Computer Science, Saarland University, Saarbrücken, Germany
E-Mail:{jherter, rtc}@cs.uni-saarland.de, haupenthal@gigasun.cs.uni-saarland.de
† Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
E-Mail: reineke@eecs.berkeley.edu

***Abstract*—General-purpose dynamic memory allocation algorithms strive for small memory fragmentation and good average-case response times. Hard real-time settings, in contrast, place different demands on dynamic memory allocators: worst-case response times are more important than average-case response times. Furthermore, predictable cache behavior is a prerequisite for timing analysis to derive tight bounds on a program's execution time.**

This paper proposes a novel algorithm that meets these demands. It guarantees constant response times, does not cause unpredictable cache pollution, and allocations are cache-set directed, i.e., allocated memory is guaranteed to be mapped to a given cache set. The latter two are necessary to enable a subsequent precise static cache analysis.

***Keywords*-Dynamic storage allocation; WCET analysis; predictability**

I. Introduction

General-purpose dynamic memory allocation algorithms are designed to provide good average-case response times while causing little fragmentation. While this behavior meets the demands of most applications, different requirements arise for real-time programs. In real-time applications, each operation must have a bounded execution time. To be able to give tight bounds on the worst-case execution times for real-time systems, memory allocators need to guarantee constant response times. Most general memory allocators fail to satisfy this criterion by having worst-case response times in $\mathcal{O}(n)$ or $\mathcal{O}(\log n)$, where n is the number of free blocks managed by the allocator.

However, constant response times are not enough. Due to the large gap between processor and memory speed, execution times strongly depend on cache performance. Today's dynamic memory allocators introduce unpredictable cache behavior. Therefore, static WCET analyses [1] fail to determine precise bounds on a program's execution time when dynamic memory allocation is used.

Current dynamic memory allocators introduce unpredictability in two ways. First, such allocators provide no guarantees about the cache set that an allocated block maps to. Static cache analyses rely on this information to classify memory accesses as cache hits or cache misses, which is necessary to determine precise WCET bounds. The second source of cache unpredictability stems from the execution of the allocator itself. Modern allocators manage free memory blocks in internal data structures on the heap. Finding a free block to satisfy allocation requests involves traversals of these structures to find suitable blocks. These traversals and hence their effects on the cache are unpredictable. As a consequence, information about cache contents is lost, whenever the allocator is run and its internal data structures are traversed. Furthermore, the unpredictability of the cache performance during the traversal itself leads to drastic overestimations of the execution times of memory (de)allocations.

In this paper, we present CAMA, a novel constant-time dynamic memory allocator that eliminates these sources of unpredictability. To enable cache-aware memory allocation, CAMA receives allocation requests with a target cache set as an additional argument. The cache influence of the allocation procedure itself is bounded and statically known.

Predictability and hence compatibility with WCET analyses are achieved as follows. Free blocks are managed in segregated free lists to allow for constant look-up times and hence constant response times. Our allocator uses cache-aware splitting and coalescing techniques to keep external fragmentation low. Internal fragmentation is reduced by using a multi-layered segregated-list approach similar to TLSF's approach [2].

In the design of CAMA, we explicitly avoided to introduce the unpredictability general purpose allocators inflict on the cache behavior. With these issues removed, existing techniques [3], [4] and tools [5] for WCET analysis should be able to cope with applications for which tight bounds on the WCETs could previously not be determined due to the unpredictability of dynamic allocation, once CAMA is used as the allocator.

A. Overview

The remainder of this paper is organized as follows. We elaborate on the problems that dynamic memory allocation imposes on the determination of WCET bounds in Section II and present related work in Section III. In Section IV, we

This work was partly 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). This work was also partly supported by the European Network of Excellence on Embedded System Design ARTIST-Design.

describe our cache-aware memory allocation algorithm and its implementation. For a set of real-time applications, we compare worst-case execution time bounds and memory consumption of CAMA to that of TLSF, and, in the case of memory consumption, to that of Doug Lea's allocator in Section V.

II. Hard Real-Time, Caches, Dynamic Memory Allocation, and Worst-Case Execution Times

Real-time applications are programs for which the correctness of operations depends not only upon their functional correctness, but also upon the time in which they are performed. In soft real-time systems, operations may sometimes miss their deadlines, upon which the system may respond with decreased quality of service. Consider, for example, a software for displaying video. If some operation does not finish in a timely manner, single frames may be dropped while displaying a video. Hence, the application does not fail completely, but the quality is decreased. In hard real-time systems, however, all operations must meet their deadlines. Within operations of a hard real-time system it is imperative that an event is reacted to within a strict deadline. Such strong timing constraints are required of applications for which reacting in time is safety-critical. Consider for example an airbag controller that has to correctly and timely detect a crash and fire the appropriate airbags. Failure to finish within a few milliseconds may be fatal.

Caches are used to bridge the increasing gap between processor speeds and memory access times. A cache is a small but fast memory that stores a subset of the contents of the main memory. Still, due to the *principle of locality*, the cache is able to serve most of the memory accesses. This enables caches to drastically improve the average latency of memory accesses. To reduce cache management and data transfer overhead, the main memory is logically partitioned into a set of *memory blocks*. The size of such a memory block is usually a power of two. Memory blocks are cached as a whole in *cache lines* of equal size. This way the block number is determined by the most significant bits of a memory address. When accessing a memory block, the system has to determine whether the memory block is currently present in the cache or if it needs to be fetched from main memory. To enable an efficient look-up, each memory block can be stored in a small number of cache lines only. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set, i.e., the number of cache lines it consists of, is called the *associativity* of the cache. At present, the associativity ranges typically from 1 to 32. The number of cache sets is usually also a power of two so that the set number can be determined by the least significant bits of the block number, the *index*. The remaining bits, known as the *tag*, are stored along with the data to decide whether and where a memory block is cached within a set. Since the number of memory blocks that map to a set is usually far greater than the associativity of the cache, a so-called *replacement policy* must decide which memory block to replace upon a cache miss.

Cache analyses [6], [7] strive to derive tight bounds on the cache performance. To obtain such bounds, they have to classify memory accesses as cache hits or cache misses. A memory access constitutes a *cache hit* if it can be served by the cache, otherwise, in case of a *cache miss*, it has to be relayed to main memory. The better the classification of accesses as cache hits or cache misses, the tighter are the obtained bounds on execution times. To classify memory accesses, a cache analysis needs to know the mapping of program data to cache sets to determine which memory blocks compete for the cache lines of cache sets.

Using a standard, general purpose dynamic memory allocator, no knowledge about the mapping of allocated data structures to cache sets is statically available to a cache analysis. Assume, for example, a program would allocate six memory blocks to store objects of a linked list structure. Two possible mappings from those allocated memory blocks to cache sets are given in Figure 1 (a) and (b), respectively. For clarity, we assume a simple four-way cache-associative cache with four cache sets.

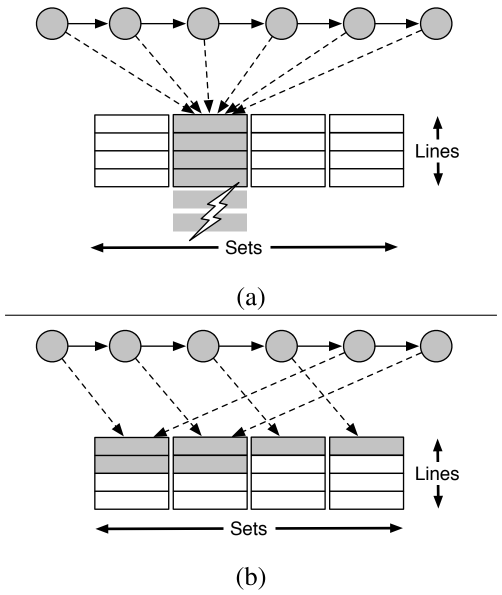

Figure 1. Two possible cache mappings of six dynamically allocated objects organized in a singly-linked list.

For the mapping depicted in Figure 1 (a) a further traversal of the list would yield only cache misses. However, given the mapping depicted in Figure 1 (b), in a subsequent traversal of the list, all six accesses to memory could be served by the cache. A conservative cache analysis with no knowledge about the addresses of allocated memory has no option other than to classify all accesses to allocated memory as cache misses, while during actual program runs all accesses may be cache hits.

In order to give precise WCET bounds, a WCET analysis has to implement such a cache analysis. Conservatively classifying each memory access as a cache miss is not an option, as turning off the cache easily causes a thirty-fold increase in execution time [8].

Dynamic memory allocators manage free and in-use blocks of memory. They need on the one hand to have fast response times to allocation and deallocation requests and on the other hand to minimize fragmentation, i.e., the amount of free memory not usable to satisfy allocation requests. Free memory blocks are managed by an allocator in some internal data structure. To satisfy an allocation request, a dynamic allocator chooses a free block according to some *placement choice*. I.e., the allocator has to decide which free block to return in order to minimize fragmentation. In general, this process contains a traversal of—at least parts of—the internal data structure. Each free block traversed will be loaded into the cache. If it is statically not derivable which blocks will be traversed, allocating memory influences the cache in an unpredictable manner (*unpredictable cache pollution*). Deallocating in-use blocks means to reinsert these blocks into the internally managed data structure holding free blocks. This also causes unpredictable cache pollution when a suitable place at which to insert the block has to be determined by a traversal of the data structure.

Hence, to enable precise WCET analyses, more demands on dynamic memory allocators arise. For such scenarios, dynamic allocators (1) need to have constant or tightly bounded allocation and deallocation times, (2) they must not cause unpredictable cache pollution, and (3) the cache sets newly allocated blocks start in must be statically derivable. While allocators meeting the demand for constant or at least reasonable tightly boundable execution times exist [2], [9], [10], cache awareness as implied by demands (2) and (3) has not yet been considered.

III. Related Work

Cache-conscious allocators have also been proposed by Chilimbi et al. [11]. However, their motivation was to improve program execution times, while we strive for predictability. Chilimbi's memory allocator takes as a second argument to the allocation request a pointer to an already existing object that is likely to be accessed contemporaneously with the one to be allocated. The allocator then tries to allocate memory for the new object next to the given one. As a result, newly allocated storage is likely to be mapped to the same memory block as the referenced storage, leading to a cache hit when accessing the second object.

Constant-time segregated-list allocators are well-known and understood. However, they tend to produce very high fragmentation. TLSF [2], a dynamic memory allocator for real-time systems, alleviates the fragmentation problem of segregated list allocators. TLSF first groups blocks whose sizes are within the same power-of-two interval, i.e., larger than 2^n but smaller than 2^{n+1}. Each power-of-two interval is then divided into a fixed number of k classes, s.t. the j^{th} such class contains all blocks with sizes s where

$$s \in \left(2^n + \frac{2^n}{k} \cdot j; 2^n + \frac{2^n}{k} \cdot (j+1)\right]$$

This two-layered approach to building size classes possesses less potential for internal fragmentation. For real-world programs, TLSF produces fragmentation similar to Doug Lea's memory allocator [12], currently considered to be amongst the best general allocators.

A completely different approach to enable dynamic memory allocation for hard real-time systems was proposed in [13]. This paper presents algorithms to compute a static allocation for programs using dynamic memory allocation. Their algorithms strive to produce static allocations that lead to minimal derivable WCET bounds in a subsequent WCET analysis. Of course, transforming dynamic allocation to static allocation needs further assumptions: e.g. that all loop bounds and the block sizes that are requested are statically known. These assumptions are reasonable for hard real-time applications. However, good memory performance can only be achieved with this approach when exploitable regularities in the program's allocation behavior can be statically derived.

IV. Cache-Aware Allocation

In the design of CAMA, we aimed at constant response times for allocation and deallocation requests, the possibility to guide the allocator with respect to which cache set the returned memory addresses are mapped to, and the absence of unknown influences on the cache by (de)allocation operations. We added an additional parameter to allocation requests, so that the memory allocation function now has two parameters: the requested block size and the cache set that the block's memory address shall map to.

A. General Memory Management

CAMA is a multi-layered segregated fit allocator very similar to TLSF. That is, CAMA organizes free blocks in segregated lists, while a single segregated list contains all memory blocks within the same size class and whose memory addresses map to the same cache set.

Formally, we associate with each free memory block a tuple $(addr, size)$, where $addr$ is the starting address of the free block and $size$ is its size (in bytes). Let $\mathcal{B}$ denote the set of all such tuples associated with the free blocks currently managed by our allocator. Furthermore, let $S_{k,i,j}$ denote the set of tuples associated with all the free blocks whose start addresses map to cache set k and whose size is in the interval $I_{i,j}$ defined as

$$I_{i,j} = \left(2^i + \frac{2^i}{j_{max}+1} \cdot j; 2^i + \frac{2^i}{j_{max}+1} \cdot (j+1)\right].$$

I.e., all $S_{k,i,j}$ are disjoint and formally defined as

$$S_{k,i,j} = \{(addr, size) \in \mathcal{B} \mid \left\lfloor \frac{addr}{size_{cline}} \right\rfloor \equiv k \mod sets \wedge size \in I_{i,j} \}$$

where $size_{cline}$ denotes the size of a cache line and $sets$ the number of sets. Note that each power-of-two size class is divided into $j_{max} + 1$ linearly increasing classes.

If we enforce a minimum size for managed memory blocks and choose i_{min} appropriately, the set of all $S_{k,i,j}$ is a partition on $\mathcal{B}$, i.e.

$$\biguplus_{\substack{0 \leq k \leq sets \\ i_{min} \leq i \leq i_{max} \\ 0 \leq j \leq j_{max}}} S_{k,i,j} = \mathcal{B}.$$

We associate a segregated list with each set $S_{k,i,j}$ and put all memory blocks associated with tuples in $S_{k,i,j}$ into the segregated list associated with $S_{k,i,j}$. Satisfying an allocation request for a block of size $size$ whose memory address is mapped to cache set k' is reduced to determining the set $S_{k,i,j}$ with which such a block would be associated and then returning any block from the segregated list associated with this set. This set can be determined in constant time by computing its index triple (k, i, j) as

$$\begin{aligned} k &= k', \\ i &= \lfloor \log_2(size) \rfloor, \\ j &= \left\lfloor \frac{size - 2^i}{2^{i - j_{max}}} \right\rfloor. \end{aligned}$$

For deallocation, we have to determine the appropriate segregated list to hold the newly freed memory block. This list depends on the size of the block. We directly store a pointer to the appropriate free list at the block to save the computation of i, j and speed up deallocation.

B. Splitting and Merging Techniques

While an allocator as described above would already meet all stated real-time demands, it might cause enormous fragmentation for real programs. With the approach described so far, two small consecutive free blocks cannot be used to satisfy a request for a larger block. Nor can larger blocks be used to satisfy requests for smaller blocks. To alleviate this, we need to add cache-aware splitting and merging techniques.

Splitting denotes the ability to split larger blocks into smaller blocks at allocation time in order to satisfy requests for smaller blocks. To enable splitting, we proceed as follows.

For each cache set k we introduce a bit vector

$$v_k \in \{0, 1\}^{(i_{max} - i_{min})(j_{max} + 1)}$$

so that the n^{th} component of v_k is 1 if and only if

$$S_{k, n/j_{max}, j'} \neq \emptyset$$

where $j' \equiv n \mod j_{max}$.

Hence, if we want to be able to use larger blocks to satisfy requests for smaller blocks, we still compute the address triple (k, i, j). However, we do not directly access the segregated list associated with $S_{k,i,j}$ but scan v_k for the first bit set to 1 starting from the $(i \cdot (j_{max} + 1) + j)^{th}$ bit. Assume this bit to be the n'^{th} component of v_k. We can then take a block from the list containing the free blocks associated with the set

$$S_{k, n'/j_{max}, (n' \bmod j_{max})}$$

to satisfy the request. In the current state of the allocator, this list contains the smallest blocks large enough to satisfy this request.

We split blocks from this list if they have a minimum size large enough that after splitting a block

$$b \in S_{k, n'/j_{max}, (n' \bmod j_{max})}$$

into two blocks b_1 and b_2 it holds that (a) b_1 is suitable to satisfy the original request, i.e., a tuple $(addr, size)$ associated with b_1 would be an element of $S_{k,i,j}$ according to our partitioning. And (b) the size of b_2 is at least the minimum size for managed free blocks. Otherwise we do not split but just return a larger block than requested from the determined segregated free list to satisfy the allocation request.

After splitting we need to insert b_2 into the appropriate free list. However, as we in general do not statically know what block size was originally requested, we cannot derive b_2's size nor its address' mapping to cache sets. And even if this were possible, we would not want the additional cache pollution from accesses to arbitrary cache sets when. Consider organizing our free lists in the usual way, i.e., consisting of the free memory blocks themselves, with each free block containing pointers to its predecessor and successor within the free list. Adding b_2 to a free list would generate four memory accesses for adjusting these links and setting b_2 as the new first block in the list. However, no cache set mapping for any of these four accesses can be statically determined. Hence, a subsequent cache analysis would have to cope with four unknown cache accesses.

A cache-aware memory allocator should support subsequent cache analyses and provide some guarantees about which cache sets may be accessed during allocation requests. To achieve this behavior, we do not store free blocks directly in the segregated free lists, but some *descriptor blocks* as defined in Figure 4. Each allocated, i.e., in-use block then stores a pointer to its descriptor block instead of a pointer to its appropriate free list. The free lists contain the descriptor blocks of free memory blocks instead of the free blocks themselves. The appropriate free list for a descriptor block can be determined in constant time from the information stored in this block (namely, its size and starting address). Additionally, the information located at descriptor blocks is

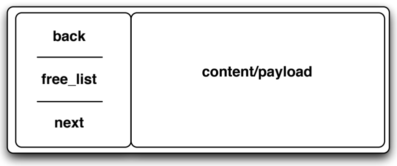

Figure 2. A *memory block* managed by CAMA. *back* is a pointer to the associated descriptor (for a large block); *next* points to the next block in the free list (for a small block that is free); *free_list* points to the free list of the associated size class (for a small black that is in use).

sufficient to enable cache-aware constant-time merging upon deallocation as discussed later.

A descriptor block contains the following information. A pointer to the free block for whose management it is used as well as the size of this block. This pointer $start$ and the size entry $size$ are used to compute the address triple (k, i, j) to identify the appropriate free list to insert the block into when it is deallocated.

$$\begin{aligned} k &= \left\lfloor \frac{start}{size_{cline}} \right\rfloor \bmod sets, \\ i &= \lfloor \log_2(size) \rfloor, \\ j &= \left\lfloor \frac{size - 2^i}{2^{i-j_{max}}} \right\rfloor. \end{aligned}$$

where $size_{cline}$ again denotes the size of a cache line.

One bit of the size entry is used as a *free bit* indicating whether the memory block referred to by this descriptor block is currently free or in-use. To enable merging, the descriptor block further stores pointers to the free blocks physically adjacent to the memory block associated with this descriptor. Finally, as descriptor blocks are organized in doubly-linked lists, pointers to its pre- and successors in the list are also stored at each block. CAMA guarantees to place such descriptor blocks exclusively at memory locations mapped to a predefined range of cache sets. Figures 2 depicts the layout memory blocks.

Figure 3 illustrates this memory management. The figure shows a fragment of memory with 3 free memory blocks managed by our algorithm. Each block has a corresponding descriptor block located in a part of memory that is mapped to the predefined range of cache sets (depicted as dark gray shaded areas).

Storing management information in descriptor blocks instead of the free blocks themselves enables us to do splitting—and merging—in a cache-aware manner. However, this comes at the price of increasing internal fragmentation. Working on descriptor blocks instead of working directly on the free blocks they manage, reduces inserting a block b_2 split from a larger block to creating a new descriptor block and inserting this descriptor block into the appropriate free list. This way, all memory accesses performed during insertion are to memory locations whose cache-set mappings

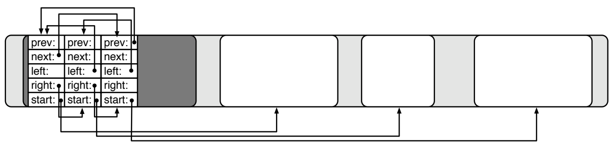

Figure 3. A fragment of memory managed by CAMA. The dark gray shaded memory area may be used for descriptor blocks, light gray areas are for the free or in-use memory blocks managed by the allocator only.

are statically known.

Merging denotes the ability to merge consecutive free blocks into a single large free block at deallocation time in order to later satisfy requests for larger blocks. As noted earlier, each in-use block stores a pointer to its descriptor block. Such a descriptor block contains pointers to managed blocks residing in memory locations adjacent to the memory location the block managed by this descriptor resides in. If a memory block is deallocated, we just have to check if its left-adjacent or right-adjacent neighboring blocks are currently free. To determine whether a memory block is currently in-use or free, we use an additional bit within the block's descriptor block. If one or both adjacent blocks are currently free, we merge these blocks into a single free block. This merging again reduces to updating entries in descriptor blocks requiring only predictable memory accesses. Upon merging two memory blocks, one descriptor block becomes free. As the memory blocks are merged into one block, one descriptor is used to manage this new block, while the descriptor of the second memory block is not needed anymore. For deallocating descriptor blocks, CAMA manages an additional free list for descriptors into which after merging unneeded descriptor blocks are inserted. The splitting operation uses this free list to allocate and thus reuse descriptor blocks. Only if the descriptor free list is empty, new memory for descriptor blocks is requested from the underlying operating system.

Descriptor blocks may pose a problem for very small blocks. If a program allocates many small memory blocks and the descriptor blocks for these are almost as large or even larger than the blocks themselves, memory waste would be overly high. And even worse, as descriptor blocks are restricted to map to a certain cache set range, many small blocks would force the allocator to request many almost unused memory pages just to store management information. Hence, descriptor blocks should only be used for memory blocks of a reasonable minimal size. This size threshold $size_{thresh}$ is in our current implementation set to

$$size_{thresh} = \frac{size_{cpage} - size_{ds}}{number_{ds}}$$

where $size_{cpage}$ denotes the size (in bytes) of a cache page[1],

[1]The size of a cache page equals the size of a cache line multiplied by the number of cache sets.

$size_{ds}$ the bytes usable for descriptor blocks per cache page according to the predefined mapping to cache sets, and $number_{ds}$ the number of descriptor blocks that fit into $size_{ds}$ bytes. This way, we ensure that the number of memory blocks managed via descriptors per cache page does not exceed the number of descriptors available on the same cache page. For blocks smaller than this size threshold, we deactivate splitting and merging completely. Hence, the only management overhead occurring from blocks smaller than the size threshold are the back links to the segregated lists the blocks reside in after deallocation.

C. Summary

In summary, the proposed algorithm works as follows. For the sake of readability, we omit the special treatment for small blocks.

Logically, we partition the free blocks managed by our allocator into disjoint sets of blocks that start in the same cache set and are within the same size class. Our allocator keeps a doubly-linked list for each such set. However, instead of organizing the free blocks directly in segregated lists, these linked lists instead store descriptor blocks for the free blocks. Those descriptor blocks are of fixed size and the allocator guarantees to place these blocks exclusively in memory locations mapped to a known range of cache sets. This range of cache sets can be increased to decrease potential fragmentation at the price of reducing the number of cache sets available for regular memory blocks. To enable the allocator to use larger blocks to satisfy requests for smaller blocks, we also store a bit string of length of the number of segregated lists. Each bit is associated with a segregated list and indicates whether this list is empty or not.

The allocator handles an allocation request for $size$ bytes *starting* at cache set k by computing which segregated list L contains the smallest blocks large enough to satisfy this request. The bit sequence is then read and the first bit set to 1 is searched within the substring starting at the bit associated with L and ending with the bit associated with the list containing the largest free blocks whose starting addresses are mapped to cache set k. If such a bit is found, the first entry from the associated segregated list is removed and the free block referred to by this entry is used to satisfy the request. Depending on its size, it is either directly returned or split into a block just large enough to satisfy the request, and a remainder. Splitting is done by creating a new descriptor block for the emerging new free block and updating the descriptor block of the block to be split. If no suitable blocks are available, new memory is requested from the underlying operating system. All these operations need constant time and touch only memory locations with a statically known mapping to cache sets.

The same holds for deallocation requests. Upon deallocation the allocator checks whether adjacent blocks in memory are currently unused. If so, adjacent free blocks are merged into a larger free block by adjusting the descriptor blocks of the involved memory blocks. Finally, the descriptor block of the now free block is added to its appropriate free list. Given the address and size information within the descriptor block, the appropriate free list can be easily computed.

```
struct descriptor {
  struct descriptor *left, *right, *prev, *next;
  void *start;
  ssize_t size /*includes an in-use bit*/;
};
```

Figure 4. Definition for descriptor block structures/objects.

```
Input: $size$: size of the block to be allocated
       $set$: cache set at which to start the block
Output: pointer to first byte of allocated memory
  $size \mathrel{+}= \text{sizeof}(*block)$;
  $size = \max\{size, \text{MIN\_BLOCK\_SIZE}\}$;
  $size = \lceil size/\text{ALIGNMENT} \rceil \cdot \text{ALIGNMENT}$;
  set $i$ and $j$ to the levels corresponding to $size$;
  set $i_2$ and $j_2$ to the levels of the first list $atab[set][i_2][j_2]$
  that has at least one free block, with either $i_2 > i$, or
  $i_2 = i$ and $j_2 \geq j$;
  if no such list exists then
    $block = add\_memory(set, i, j, size)$;
  else
    $block = split\_memory(set, i_2, j_2, size)$;
  end if
  return $block + \text{sizeof}(*block)$;
```

Figure 5. *camalloc*: allocate a memory block

Pseudo code for the main operations of our prototype implementation of CAMA is given in Figures 5 and 6. The structure definition for the descriptor blocks used in these algorithms is given in Figure 4. We use the sign bit of the size component of such a descriptor to indicate whether a memory block is free or in use. However, we address this information in the pseudo code like a normal component (named in_use).

The following program constants are used within the pseudo code. CS denotes the number of cache sets provided by the target hardware. The number of size classes per cache set is given by SL and TL, where SL equals to $i_{max} - i_{min}$, TL to j_{max}. In the current implementation, those constants are set to 11 and 4, respectively, with $i_{min} = 4$ and $i_{max} = 15$. The size of a descriptor block is denoted by DESC_BLOCK_SIZE. MIN_BLOCK_SIZE denotes the minimum block size for memory blocks managed by CAMA.

V. Evaluation

A. Memory Fragmentation

We used the standard approach to measure the memory fragmentation behavior of CAMA. That is, we extracted

```
Input: $ptr$: address of the memory block to be freed
  $block = ptr - \text{sizeof}(*block)$;
  if $block\text{→}header \geq \&atab[0][0][0]$
      $\wedge$ $block\text{→}header < \&atab[\text{CS}][\text{SL}][\text{TL}]$ then
      append $block$ to free list $block\text{→}header$;
      set respective bit;
      return
  end if
  $d = block\text{→}header$
  $d_l = d\text{→}left$
  $d_r = d\text{→}right$
  $d\text{→}in\_use = \text{false}$
  if $d_r \neq \text{NULL}$
      $\wedge$ $d_r\text{→}start = block + d\text{→}size$
      $\wedge$ $d_r\text{→}in\_use = \text{false}$ then
      set $i$ and $j$ to the levels corresponding to $d_r\text{→}size$;
      set $set$ to cache set of the byte pointed to by $d_r\text{→}start$;
      remove $d_r$ from $atab[set][i][j]$;
      clear bit if list is now empty;
      $d\text{→}size \mathrel{+}= d_r\text{→}size$;
      $d\text{→}right = d_r\text{→}right$;
      remove $d_r$ from descriptor list;
      put $d_r$ into descriptor free list;
      $d_r = d\text{→}right$;
  end if
  if $d_l$ $\wedge$ $d_l\text{→}in\_use = \text{false}$
      $\wedge$ $d_l\text{→}start + d_l\text{→}size = block$ then
      set $i$ and $j$ to the levels corresponding to $d_l\text{→}size$;
      set $set$ to the cache set of $d_l\text{→}start$;
      remove $d_l$ from $atab[set][i][j]$;
      clear bit if list is now empty;
      $d_l\text{→}size \mathrel{+}= d\text{→}size$;
      $d_l\text{→}right = d_r$;
      remove $d$ from descriptor list;
      put $d$ into descriptor free list;
      $d = d_l$;
      $d_l = d\text{→}left$;
  end if
  if $d_r = \text{NULL}$
      $\wedge$ program break is at $d\text{→}start + d\text{→}size$ then
      $d_l\text{→}right = \text{NULL}$;
      remove $d$ from descriptor list;
      move program break down to $d\text{→}start$;
      put $d$ into descriptor free list;
      return
  end if
  set $i$ and $j$ to the levels corresponding to $d$;
  set $set$ to the cache set of $d\text{→}start$;
  append $d$ to free list $atab[set][i][j]$;
  set respective bit;
```

Figure 6. *cafree*: deallocate a memory block

(de)allocation traces from real programs and let the allocator process these sequences of allocation and deallocation requests. We refrained from generating synthetic traces for benchmarking as we agree with Wilson et al. that the regularities existing in real programs are not well enough understood to model them formally and perform probabilistic analyses that are directly applicable to real program behavior [14].

The field of application for a cache-aware constant time memory allocator is restricted to hard real-time systems. Hence, the set of programs used for benchmarking should consist mainly of such applications. However, in hard real-time applications, dynamic memory allocation is usually avoided due to the unpredictable timing behavior introduced by the allocator. Still, the MiBench benchmark suite [15], which consists of a set of commercially representative embedded programs, contains six test cases that make use of dynamic memory allocation. These six test cases execute the programs `Susan`, `Patricia`, and `Dijkstra`, each on a set of small and large input data, respectively.

`Susan` was developed for recognizing corners and edges in magnetic resonance images of the brain. The technique embodied by the program is also patented as a method for digitally processing images to determine the position of edges and/or corners therein for guidance of unmanned vehicles. The small input data run processes a black and white image of a rectangle while the large input data run processes a complex picture.

`Patricia`: A patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Patricia tries are often used to represent routing tables in network applications. These benchmarks use patricia tries to construct a routing table.

`Dijkstra` constructs a large graph (as an adjacency matrix) and then computes the shortest paths between pairs of nodes using repeated applications of Dijkstra's algorithm.

To compare CAMA to existing memory allocators, we used the same set of benchmark traces on Doug Lea's memory allocator[2] (DLMalloc) and TLSF[3].

Figure 8 shows the complete results of our measurements. Figure 7 compares the absolute memory consumption of the three allocators. We computed the degree of fragmentation as the percentage of (allocated) memory exceeding needed memory. I.e., if a program needs at most 100kb of contemporaneously allocated memory but 125kb are allocated, fragmentation would be 25%. The figures show that overall the memory consumption of all allocators is comparable. Only in the `Patricia large` test case, CAMA and TLSF have a—in absolute terms—significantly higher memory consumption.

For allocation requests CAMA needs an additional cache

[2]Version 2.8.4; from http://g.oswego.edu/dl/html/malloc.html
[3]Version 2.4.5; from http://rtportal.upv.es/rtmalloc/node/8

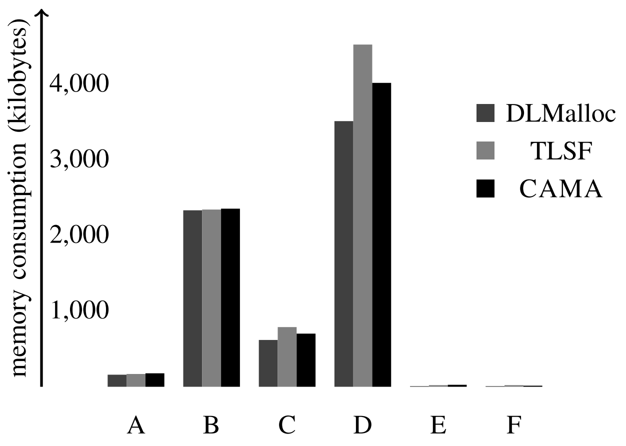

Figure 7. Absolute memory consumption for the following test cases taken from the MiBench test suite: `Susan small` (A), `Susan large` (B), `Patricia small` (C), `Patricia large` (D), `Dijkstra small` (E), and `Dijkstra large` (F).

Application	Memory Consumption & Fragmentation		
Susan small	Memory Need	155,979 bytes	
	DLMalloc	159,744 bytes	2.414%
	TLSF	169,216 bytes	8.486%
	CAMA	178,320 bytes	14.323%
Susan large	Memory Need	2,333,809 bytes	
	DLMalloc	2,334,720 bytes	0.039%
	TLSF	2,344,512 bytes	0.459%
	CAMA	2,356,156 bytes	0.958%
Patricia small	Memory Need	435,640 bytes	
	DLMalloc	610,304 bytes	40.094%
	TLSF	789,504 bytes	81.229%
	CAMA	703,128 bytes	61.401%
Patricia large	Memory Need	2,508,880 bytes	
	DLMalloc	3,514,368 bytes	40.077%
	TLSF	4,527,104 bytes	80.443%
	CAMA	4,020,312 bytes	60.243%
Dijkstra small	Memory Need	5,880 bytes	
	DLMalloc	8,192 bytes	39.320%
	TLSF	21,504 bytes	365%
	CAMA	26,168 bytes	445%
Dijkstra large	Memory Need	5,264 bytes	
	DLMalloc	8,192 bytes	55.623%
	TLSF	21,504 bytes	408%
	CAMA	14,324 bytes	272%

Figure 8. Memory consumption of DLMalloc, TLSF, and CAMA. *Memory need* denotes the minimal memory need of the program, i.e., the maximal number of bytes that are contemporaneously allocated at some program point. Fragmentation is computed as $\left(\frac{memory\ used}{memory\ need} - 1\right) \cdot 100\%$; CAMA was instantiated for 128 cache sets and a cache line size of 64 bytes.

set parameter telling the allocator the cache set the returned memory address shall be mapped to. In real life, this parameter is either set by a timing analysis while analyzing the program or by the programmer himself. Obviously, choosing unsuitable cache set arguments can almost arbitrarily increase fragmentation. Using just random numbers may not produce results applicable to real programs. Computing the optimal cache set arguments may on the other hand lead to too optimistic results. We therefore set the cache set arguments according to very simple heuristics as the average programmer would probably do. Two heuristics were sufficient for all test cases. The first assumes that memory is never deallocated and just put consecutively in memory. It then simulates this behavior and sets cache set arguments to the cache set that the start addresses of allocated blocks are mapped to in its simulation. This heuristics was used in the `Susan` test cases. The second returns cache set a n-times, then n-times cache set $(a + 1)$ and so on. The assumption here is that n successively allocated memory blocks fit into one cache line. We used this with the `Dijkstra` and `Patricia` tests.

In summary, this benchmark result is very encouraging. There is a test case where fragmentation seems overly high (`Dijkstra`), but this program allocates only a small amount of memory and hence CAMA's (and also TLSF's) address table has a large impact on the memory consumption. CAMA's address table to look up the appropriate segregated listsneeds roughly 6,000 bytes. This is already more memory than is needed by the `Dijkstra` test case. Hence, we believe the most meaningful results are obtained from the `Patricia` and `Susan large` test cases. In those cases the memory need of the programs is large enough to render the influence of the address tables on fragmentation negligible.

When compared to standard memory allocators, constant response times come at the cost of higher fragmentation. In contrast to general allocators, constant time allocators cannot apply some *best fit* strategy to find the best free block to satisfy an allocation request, but have to do with a *first fit* strategy that in constant time finds some block suitable to satisfy the request. That this price is not necessarily overly high was already shown by Masmano et al. for their constant time allocator TLSF [2]. Our benchmark results indicate that adding cache awareness does not significantly increase the costs already paid for constant-time behavior any further.

Considering the benchmark results more closely, it seems surprising that CAMA outperforms TLSF in some test cases. Intuitively, adding cache awareness can only increase fragmentation, not reduce it. The better memory performance of CAMA is due to the different treatment of small blocks compared to TLSF. CAMA stores no information for later splitting or merging for small blocks, resulting in less management overhead. `Patricia` and `Dijkstra`, however, request many small blocks that can never be split or merged anyway. Hence, CAMA's strategy works better for these programs. For the `Susan` test cases as well as `Dijkstra small`, we get the intuitively expected results. Here, cache awareness increases fragmentation slightly. These test cases allocate just a small number of blocks. Hence, the larger amount of segregated lists CAMA needs to manage gives TLSF an advantage. The fragmentation created by both allocators is roughly the same and CAMA's larger address table for the segregated lists causes its higher memory consumption.

B. Execution Times

In hard real-time settings, the most meaningful performance measure for execution time is the *provable worst-case execution time*, that is, a statically provable upper bound on a program's worst-case execution time.

Figure 9 shows the provable WCET for requests for differently sized free blocks posed to the two real-time allocators. In the case of CAMA, we requested each size mapped once to each cache set. All time bounds where determined using aiT, an industrial worst-case execution time analyzer (http://www.absint.de/ait/). As execution times are hardware dependent, we had to determine these values with respect to a fixed target hardware. To obtain meaningful results, we chose an existing target platform: the PowerPC MPC603e which is widely used in embedded systems. This hardware possesses separate data and instruction caches with 128 cache sets each and a cache line size of 32 bytes.

The figure shows two values for TLSF per requested block size. The black value is the one obtained from analyzing TLSF in the version as provided for download. The light gray value is obtained from a slightly modified version of TLSF in which we replaced the internal computation of logarithms by a faster, more predictable implementation and bypassed the check whether the allocator was already initialized, as CAMA provides an explicit initialization method and performs no such check. Hence, CAMA's WCET for allocation requests can be bounded by 9,935 cycles, TLSF's by 13,026 and 16,260 cycles, respectively.

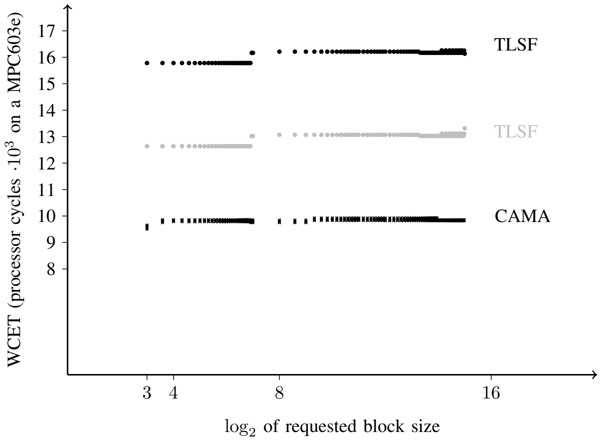

Figure 9. Provable WCET bounds for different allocation requests.

Surprisingly, the provable WCET of CAMA is still significantly smaller than that of TLSF, although one would expect TLSF to have a slightly smaller WCET bound, as both allocators perform basically the same operations in the worst case with CAMA having to deal with descriptor blocks and a three-dimensional addressing scheme and TLSF with a two-dimensional one. The reason for this is entirely implementation dependent. When additional memory needs to be requested from the underlying operating system, TLSF calls its free()-method to insert new memory blocks into appropriate free lists. While this prevents code duplication, the overhead of the method call adds significantly to TLSF's WCET.

```
struct task_descr* lowPriority = low;
struct task_descr* highPriority = high;
for(i = 0; i < HUGE_LIST_SIZE; i++) {
   for(j = 0; j < SMALL_LIST_SIZE; j++) { // high prioritized tasks waiting?
      high = high->next;
      ...
   }
   high = highPriority;
   low = low->next; // next lower prioritized task waiting?
   ...
}
low = lowPriority;
```

Figure 10. The main loop body of a simplified task scheduler.

The provable WCET for deallocation procedures on this hardware are 6,891 cycles for CAMA and 5,703 cycles for TLSF.

In summary, besides implementation issues, both CAMA and TLSF have about the same provable WCETs for allocations and deallocations. Hence, neither memory consumption nor provable WCET of a program is expected to be increased by using a cache-aware memory allocator compared to using a non-cache-aware allocator.

But what can actually be gained by adding cache awareness? To thoroughly answer this, a novel cache analysis would be needed which takes into account CAMA's cache set argument as well as its additional cache guarantees. Unfortunately, such an analysis does not yet exist. However, to still roughly demonstrate the impact of cache awareness on WCET bounds, we analyzed a simplified task scheduler for which we could manually annotate at least the statically available information about the cache-set mapping of dynamically allocated objects. The scheduler manages two singly-linked lists composed of task descriptors with maximally 4 and 16 entries, respectively. The smaller list is used for high priority tasks, the other for all other tasks. Assume CAMA is used to ensure that all high-priority objects map to cache set 0 and other objects are equally spread over sets 1 to 127. Analyzing the WCET of the program fragment given in Figure 10 using aiT then yields an upper bound on the WCET of 6,505 cycles on a PowerPC MPC603e. With statically unknown cache-set mappings, however, only a WCET of 10,915 cycles can be guaranteed on the same hardware.

Note that the analyzed code does not contain any invocations of the memory allocator, so the gain in WCET guarantees is completely attributed to the analysis being able to exclude that objects of the lower prioritized list evict higher priority objects from cache, which leads to the safe prediction of cache hits when traversing the higher priority list again.

VI. Conclusions

For hard real-time applications tight bounds on the execution times of memory (de)allocation requests are not sufficient. On modern embedded hardware, cache performance has a large influence on the overall system performance. Hence, dynamic memory allocators suitable for use in hard real-time systems have to provide predictable cache behavior. Otherwise, tight bounds on the cache performance of an application and hence its execution times cannot be statically determined.

CAMA provides a predictable cache behavior in the sense that it (a) can be guided with respect to which cache set allocated memory is mapped to and it (b) guarantees to access only a statically known subset of the available cache sets a constant number of times while processing allocation and deallocation requests. Furthermore, CAMA's execution times can be tightly bound.

Our benchmark results indicate that cache awareness does not necessarily increase the fragmentation already paid for constant response times any further and when it does, fragmentation is still not overly high.

TLSF is already successfully used within soft real-time systems. This indicates that the price paid for real-time behavior in fragmentation is not prohibitively high. Hard real-time systems still rely exclusively on static memory allocation as the cache influence of dynamic memory allocators introduces too much uncertainty about cache performance. CAMA provides guarantees about its cache behavior for a similar price paid in fragmentation as for real-time guarantees alone. It therefore seems reasonable to believe that CAMA can enable the use of dynamic memory allocation within hard real-time systems.

Acknowledgment

The authors thank Andreas Abel for his help in performing the aiT analyses.

References

[1] R. Wilhelm *et al.*, "The worst-case execution time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008. [Online]. Available: http://dx.doi.org/10.1145/1347375.1347389

[2] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–86. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2004.35

[3] S. Thesing *et al.*, "An abstract interpretation-based timing validation of hard real-time avionics," in *Proceedings of the International Performance and Dependability Symposium (IPDS)*. IEEE Computer Society Press, June 2003, pp. 625–632. [Online]. Available: http://dx.doi.org/10.1109/DSN.2003.1209972

[4] C. Ferdinand *et al.*, "Reliable and precise WCET determination for a real-life processor," in *Proceedings of EMSOFT 2001*. London, UK: Springer-Verlag, 2001, pp. 469–485. [Online]. Available: http://portal.acm.org/citation.cfm?id=646787.703893

[5] C. Ferdinand, "Worst case execution time prediction by static program analysis," *Parallel and Distributed Processing Symposium, International*, vol. 3, p. 125a, 2004. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2004.1303088

[6] C. Ferdinand and R. Wilhelm, "Fast and efficient cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17(2/3), pp. 131–181, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1008186323068

[7] J. Reineke, "Caches in WCET analysis," Ph.D. dissertation, Universität des Saarlandes, November 2008. [Online]. Available: http://rw4.cs.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalysis.pdf

[8] M. Langenbach, S. Thesing, and R. Heckmann, "Pipeline modeling for timing analysis," *Proceedings of SAS*, vol. 2477, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=647171.716098

[9] J. L. Peterson and T. A. Norman, "Buddy systems," *Communications of the ACM*, pp. 20(6):421–431, 1977. [Online]. Available: http://dx.doi.org/10.1145/359605.359626

[10] T. Ogasawara, "An algorithm with constant execution time for dynamic memory allocation," *2nd Int. Workshop on Real-Time Computing Systems and Applications*, 1995. [Online]. Available: http://dx.doi.org/10.1109/RTCSA.1995.528746

[11] T. M. Chilimbi, M. Hill, and J. R. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, pp. 67–75, 2000. [Online]. Available: http://dx.doi.org/10.1109/2.889095

[12] D. Lea, "A memory allocator," *Unix/Mail 6/96*, 1996.

[13] J. Herter and J. Reineke, "Making dynamic memory allocation static to support WCET analyses," in *Proceedings of 9th International Workshop on WCET Analysis*, June 2009. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2284/pdf/Herter.2284.pdf

[14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of the International Workshop on Memory Management*. Springer-Verlag, 1995, pp. 1–116. [Online]. Available: http://portal.acm.org/citation.cfm?id=645647.664690

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15