PRADA: Predictable Allocations by Deferred Actions*

Florian Haupenthal and Jörg Herter

Saarland University – Computer Science
Saarland, Germany
`{s9flhaup@stud, jherter@cs}.uni-saarland.de`

Abstract

Modern hard real-time systems still employ static memory management. However, dynamic storage allocation (DSA) can improve the flexibility and readability of programs as well as drastically shorten their development times. But allocators introduce unpredictability that makes deriving tight bounds on an application's worst-case execution time even more challenging. Especially their statically unpredictable influence on the cache, paired with zero knowledge about the cache set mapping of dynamically allocated objects leads to prohibitively large overestimations of execution times when dynamic memory allocation is employed. Recently, a cache-aware memory allocator, called CAMA, was proposed that gives strong guarantees about its cache influence and the cache set mapping of allocated objects. CAMA itself is rather complex due to its cache-aware implementations of split and merge operations.

This paper proposes PRADA, a lighter but less general dynamic memory allocator with equally strong guarantees about its influence on the cache. We compare the memory consumption of PRADA and CAMA for a small set of real-time applications as well as synthetical (de-) allocation sequences to investigate whether a simpler approach to cache awareness is still sufficient for the current generation of real-time applications.

1998 ACM Subject Classification B.3.3 Performance Analysis and Design Aids

Keywords and phrases Dynamic Memory Allocation, Worst-Case Execution-Time, Cache Predictability

Digital Object Identifier 10.4230/OASIcs.WCET.2013.77

1 Introduction

(Hard) real-time applications raise the requirements on dynamic memory allocators. Constant (de-) allocation times and a bounded, predictable cache behaviour become equally important as *good* response times and low memory consumption. Short, constant response times can be guaranteed by a large set of dynamic memory allocators, ranging from conventional buddy systems [9, 13] to specialized real-time allocators like Half-Fit [12] and TLSF [11]. However, none of these allocators provide guarantees about their effects on the cache that a cache analysis may exploit to provide a subsequent timing analysis with a tight approximation of the program's cache behaviour.

With CAMA [8, 6], the first cache-aware constant-time dynamic memory allocator was proposed. This allocator guarantees constant execution times as well as a bounded cache influence on just a statically known set of cache sets for allocations and deallocations.

* This work was supported by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

Furthermore, CAMA can be guided to which cache sets newly allocated objects shall be mapped. Common techniques to counteract memory fragmentation like splitting[1] and merging[2] are used. To implement these operations without introducing unpredictable cache effects, CAMA relies internally on an indirect management of free blocks. Internal free lists and split/merge operations work on so-called descriptors instead of the free blocks themselves. Strict memory placement policies exist for descriptors to ensure a statically predefined cache mapping of descriptors as well as the absence of accesses to unknown cache sets.

PRADA is a lighter implementation without the need for descriptors, providing the same predictability guarantees: constant execution times and a statically known effect on the cache. PRADA tackles the challenge of not introducing cache unpredictabilities by performing (partial) splits/merges only when no other cache set than the one already touched during the (de-) allocation procedure is accessed. To enable PRADA to choose when to perform (partial) split/merge operations, all these operations are initially deferred and executed when the prerequisites for the operation are met. However, the allocator does not provide any guarantee that these deferred actions will be executed at all. It only stores a fixed, but configurable amount of deferred actions. Surplus ones are simply dropped, i.e., never executed.

For general purpose dynamic memory allocators, deferred split and merge actions have been shown to be inferior to immediate splits and merges [14]. But does this still hold true when restricting the class of programs in which an allocator may be used to (hard) real-time applications? In this paper, we investigate on whether an implementation as complex as CAMA is actually necessary to fulfil the raised demands of hard real-time systems; or whether we can do with a simpler approach like PRADA.

In Section 2, we describe PRADA, an alternative cache-aware constant time dynamic memory allocator that uses deferred actions in order to implement split and merge operations in a (cache- and time-) predictable manner. Section 3 studies the memory consumption of several dynamic memory allocators when presented (de-) allocation sequences representative for real-time applications. Related work is summarized in Section 4.

2 PRADA

PRADA is a dynamic memory allocator which manages free *memory blocks* in segregated free lists to allow for constant time allocation and deallocation routines. It defers actions which would introduce unpredictable behaviour when always immediately executed by remembering them in form of *requests*. These actions are executed during subsequent (de-) allocations to the cache set they need to access. This section briefly summarizes how PRADA works and achieves its predictability goals. A more detailed description of PRADA can be found in [3].

The memory managed by PRADA is divided into memory blocks. Every memory block consists of a size field storing its current size and the actual payload area itself. PRADA uses the payload area of currently free blocks to build-up its free lists. Therefore, the payload area of deallocated memory blocks contains three fields. Two list pointers linking to the previous block and the next block within the free list, respectively. The third field is a pointer to a potentially pending request.

PRADA and CAMA use their respective free lists in the same way. They use an adapted

[1] Splitting denotes the use of (split parts of) larger blocks in order to satisfy a request for a smaller block.
[2] Merging denotes the joining of two physically consecutive free blocks in order to have a larger block with higher probability to be useful in satisfying an allocation request.

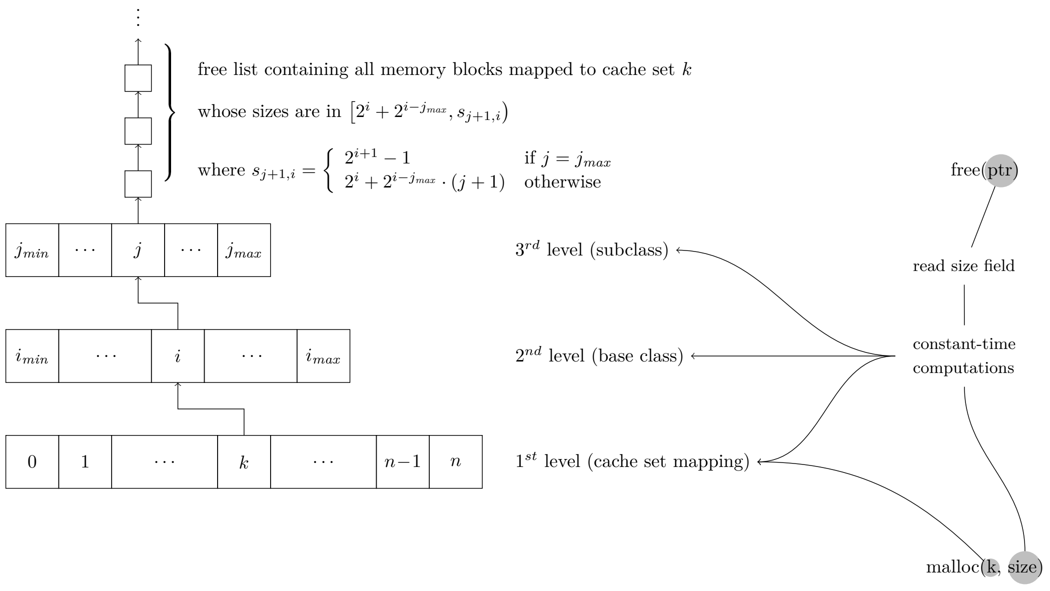
free list containing all memory blocks mapped to cache set k
whose sizes are in $[2^i + 2^{i-j_{max}}, s_{j+1,i})$
where $s_{j+1,i} = \begin{cases} 2^{i+1} - 1 & \text{if } j = j_{max} \\ 2^i + 2^{i-j_{max}} \cdot (j+1) & \text{otherwise} \end{cases}$
j_{min} ... j ... j_{max}
i_{min} ... i ... i_{max}
0 1 ... k ... $n-1$ n
3rd level (subclass)
2nd level (base class)
1st level (cache set mapping)
free(ptr)
read size field
constant-time computations
malloc(k, size)

Figure 1 Logical view on the partitioning of the memory in PRADA and CAMA and how to connect (de-) allocation requests to free lists.

version of TLSF's two-level approach of building size classes (which in turn correspond to its free lists). During an allocation, the first block of a suitable free list, i.e. size class, is used to satisfy the request. How does the two-level approach to set-up size classes of TLSF, PRADA, and CAMA work? The (logically) first level sorts blocks in exponentially growing size classes, i.e., for class i all associated memory blocks are of $size \in [2^i, 2^{i+1})$. A higher granularity for sorting blocks is achieved by a second level which is a linear subdivision of these classes. For an allocation of size s, two computations are needed. First, the base class needs to be determined, then the correct subclass. Both classes can be computed in two constant-time computations:

$$class = \lfloor \log_2(s) \rfloor \text{ and } subclass = \left\lfloor \frac{(s - 2^{class}) \cdot j_{max}}{2^{class}} \right\rfloor$$

where j_{max} is the number of linear subclasses. CAMA and PRADA add an additional level to setting up free lists by firstly sorting free blocks according to the cache set they start in. In contrast to non-cache-aware allocators, they use an additional argument which allows to guide their allocations to a certain cache set. This additional argument selects which free list structure is searched. Figure 1 illustrates how (de-) allocation requests are mapped to a suitable free list.

PRADA defers split and merge actions to avoid unpredictable effects on the cache during allocations and deallocations. Therefore, actions need to be remembered. Remembering actions has to be done in a way which preserves the predictable behaviour of the allocator. Therefore, we use an array of fixed size which contains requests for actions. These requests are used for the deferred performing of splits and merges. For each cache set, there is the same, fixed amount of entries reserved. Since this array is statically allocated, the impact on the cache state of accesses to this array is known. Due to this static setting, requests are

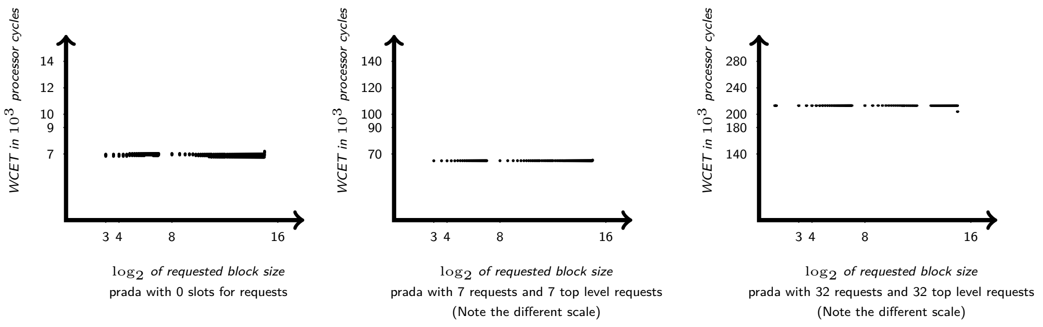

Figure 2 WCET bounds of PRADA0, PRADA7, and PRADA32.

never created or deleted, but get filled and cleared. These requests differ from the descriptors used by CAMA in two ways. The number of descriptors managed by CAMA depends on the number of managed memory blocks. For PRADA, the reserved space for requests is fixed. Furthermore, a request itself is smaller than a descriptor (8 bytes and 24 bytes, respectively).

PRADA executes one pending request, if there exists one for the current cache set during each allocation and deallocation. The following two paragraphs describe the procedures for allocations and deallocations and highlight when deferred requests are executed.

PRADA allocates memory blocks aligned to cache sets. Hence, all allocations (including the added space for the block header) are rounded up to the next multiple of a cache line. With the provided cache set mapping and the computed *class* and *subclass*, the free list containing the smallest blocks suitable to satisfy the allocation request is fully determined. If there exists a suitable block, i.e., the free list is not empty, there may also exist a pending, now obsolete merge request for this block. This request needs to be cleared first. Then, one pending request for the current cache set can be executed. If the selected block is exactly of the requested size, the block is simply returned. Otherwise, i.e. the found block is larger than requested, its size is set to the requested size and a split request is created. If no suitable block is found, new memory is requested from the operating system to create a suitable block, possibly requiring a deferred split operation. Even in case that no block was found, a requested action for the current cache set can be executed.

At deallocation, PRADA first checks whether there is still a pending split request for the deallocated block. If there is one, this split request is dropped and the blocks original size is restored. Then, one pending request for the current cache set is executed if one exists. Finally, the current block is inserted into the appropriate free list and marked as available for merges, i.e., a merge request is created.

Figure 2 depicts the WCET bounds for allocations of different sizes and mapped to different cache sets derived by aiT[4] for our prototype implementation of PRADA. For comparison, Figure 3 shows the respective WCET bounds derived for TLSF and CAMA. Deallocations take only a single pointer as an argument. For PRADA, the WCET bound for deallocations still depends on the number of requests. WCET bounds of 2,437 cycles, 59,947 cycles, and 182,783 cycles where derived for 0, 7, and 32 requests, respectively. For TLSF and CAMA, the bounds are 6,018 cycles and 98,156 cycles, respectively.

3 Evaluation

For the evaluation of PRADA and the comparison with CAMA, we used the relevant programs from the MiBench benchmark suite [2], i.e., those using dynamic memory allocation. The

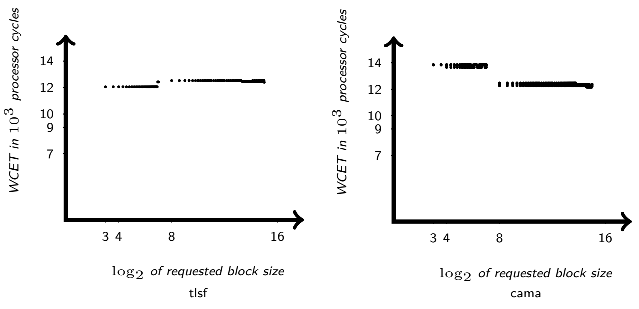

Figure 3 WCET bounds of `TLSF` and `CAMA`.

MiBench suite itself consists of a set of embedded programs, considered to be representative for commercial applications. However, most embedded systems avoid dynamic memory management. Therefore, we have only six relevant test cases from this suite. These six test cases run 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 software is, however, also used as image recognition in 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. 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. `Patricia` uses 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 get a better impression on their respective memory performances, we compare the total memory consumption of `CAMA` and `PRADA` against several other allocators:

- `TLSF`: a constant time, but cache-unaware *real-time* allocator.
- `aobf` (address ordered best fit), `aoff` (first fit), and `aowf` (worst fit): simple sequential fits with different allocation policies (best, first, and worst fit) that are able to allocate blocks according to a predefined cache set mapping. However, no useful WCET for allocation and deallocation requests can be given.
- DLMalloc: Doug Lea's allocator [10] which is considered to be the best general purpose dynamic memory allocator. However, this allocator is neither cache-aware nor does it provide useful WCET bounds for allocation requests.

We also measure the maximum amount of memory live, i.e. allocated, contemporaneously for the different benchmarks. Comparing Doug Lea's allocator and `TLSF` gives a good impression of the (isolated) costs in terms of memory consumption for constant response times. I.e., the spatial costs for switching from a *best fit* strategy to a *good fit* strategy in order to achieve constant allocation times. Our sequential fit allocators isolate the spatial costs of enforcing a certain, statically fixed cache set mapping on allocations. The difference between the maximum amount of live memory and `DLMalloc`'s memory consumption illustrates the spatial costs inherent to dynamic memory allocation; even without further demands for constant response times and cache guarantees.

For an unbiased comparison, we want to compare just the (de-) allocation routines without the actual program computations for the different allocators. Therefore, we record a trace of allocations and deallocations during one run of the benchmark application. In a subsequent step, we use this trace to synthesize a one-path program which only consists of these allocations and deallocations. This program is then compiled once for each allocator.

Since the used subset of the MiBench benchmark suite may not be as representative as the suite as a whole, we added three additional, synthetic benchmarks. These benchmarks are artificial traces which describe different types of typical behaviours of hard real-time systems. Due to their artificial character, these traces only cover some basic characteristics of real programs. The three characteristics we used for our evaluation are suggested by [14], which also points out the weaknesses of generating synthetic, randomized (de-) allocation sequences. Namely that real programs simply do not behave randomly, but exhibit regularities that a dynamic memory allocator may exploit. Hence, results from randomly generated (de-) allocation sequences generally tend to be overly pessimistic. One typical behavioural pattern is having all allocations in a set-up phase. After this phase, the application *works* on these allocated objects without allocating more objects. This behaviour is covered in the trace called *ramp*. Another typical behaviour pattern consists of round-wise allocations. This pattern is widely found in reactive systems. These programs often run in a loop and everything which gets allocated during one iteration gets deallocated in the same iteration. This behaviour is modelled in the trace called *peak*. The third behavioural pattern that we consider is a combination of the two patterns discussed so far. I.e., there is a base of allocated objects on which the program works, but there are also additional allocations and subsequent deallocations per iteration as in the *peak* pattern. This pattern is implemented in the trace called *plateau*.

Life spans and requested block sizes are randomly selected according to an exponential distribution with rate parameter $\lambda = 0.25$. However, we shifted this distribution such that we have 1 as the smallest possible life span. The random values were furthermore multiplied by 4 to obtain reasonable, aligned block sizes. The programs *ramp* and *peak* run until 10,000 allocations are performed. The *plateau* in the third program consists of 1,500 allocations on top of which 10,000 allocations and deallocations are performed. The additional cache set arguments of `PRADA` and `CAMA` are selected according to the same heuristics used in [6]. Those heuristics are intentionally very simple, with the intention that any programmer would use a heuristics at least as good. We use two simple heuristics A and B depending on the benchmark application. Heuristics A assumes that memory is never deallocated and just put consecutively in memory. It then simulates this behaviour and sets cache set arguments to the cache set that the start addresses of allocated blocks are mapped to in its simulation. Heuristics B simply returns cache set a n-times, then n-times cache set $(a + 1)$ and so on. This heuristics assumes that n successively allocated memory blocks fit into one cache line. We used heuristics A for the `Susan` test cases as well as for all synthetical benchmarks. For `Dijkstra` and `Patricia` test cases, heuristics B was used.

The number of deferred actions in `PRADA` can be configured. We used two configurations for our benchmarks. One with splitting and merging completely disabled, i.e. allowing for 0 actions to be stored, and one with space for 32 actions, denoted `PRADA0` and `PRADA32`. `PRADA` and `CAMA` are configured for an architecture with 128 cache sets, with $i_{min} = 0$, $i_{max} = 18$, $j_{min} = 0$, and $j_{max} = 3$. `TLSF` used 24 base classes with 32 subclasses, each. Figure 4 shows the memory consumption for all allocators on the MiBench programs.

We observe a measurable impact of disabling splitting and merging on the MiBench test cases. On these real-life benchmarks, forcing the allocator to adhere to a given cache set mapping for allocated blocks (`aobf`, `aoff`, and `aowf`) also causes a measurable increase in memory consumption. While this is to be expected, this increase is surprisingly lower than the increased memory consumption due to constant response times, i.e., when compared to `TLSF`. Comparing our sequential fits with the (also) cache-set guided allocations to `PRADA` and `CAMA`,

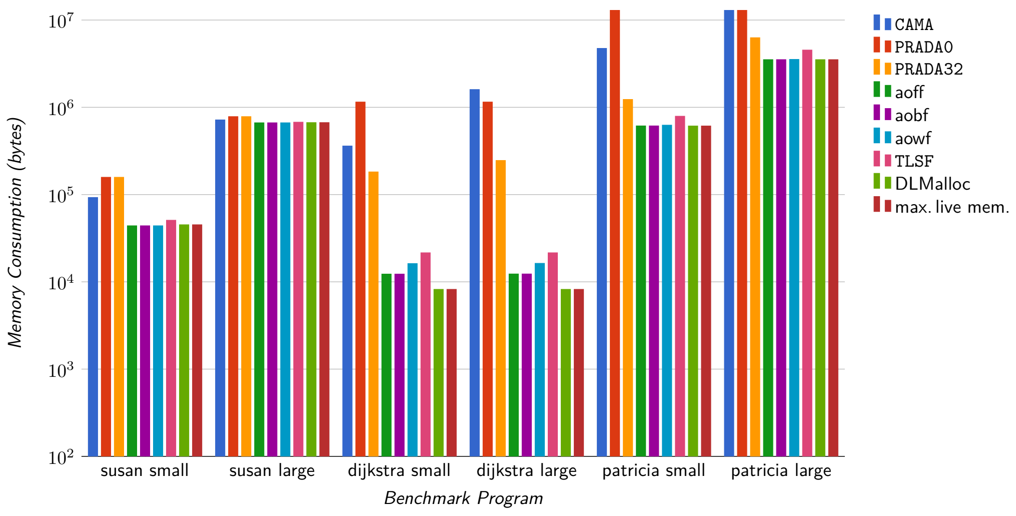

Figure 4 Memory consumption on (de-) allocation traces generated from the MiBench benchmarks.

we observe a significant jump in memory consumption. While we expect a small increase due to internal fragmentation from the employed segregated-list approach as well as the additional memory needed for `CAMA`'s descriptors, those allocators also introduce yet another kind of fragmentation. What kinds of fragmentation do exist and why do our constant-time cache-aware allocators exhibit those? General purpose dynamic memory allocators suffer from two kinds of fragmentation: *internal fragmentation* and *external fragmentation.* Internal fragmentation denotes the memory overhead when the allocator returns blocks larger than requested. This may be due to round-up block sizes, memory alignment, the allocator's inability to manage the remaining part of the block, etc. External fragmentation occurs when there is enough free memory to satisfy a request, but there is not a single block large enough. I.e. the free memory is interspersed with allocated memory. `PRADA` and `CAMA` additionally suffer from an *incomplete memory use.* This denotes the inability to find a suitable, large enough block that could be split in order to serve an allocation request simply because this block is assigned to another cache set's free lists. The term *incomplete memory use* was coined by Ogasawara to describe a similar problem of Half-Fit [12]. In Half-Fit, free blocks larger than the base size of their respective free list will not be used to serve requests for blocks of sizes larger than this base size; even if they are just one byte larger. Analytically, this leads to a worst-case memory consumption of roughly twice the maximal live memory just due to Half-Fit's incomplete memory use. While `TLSF`, `CAMA`, and `PRADA` use a similar segregated-list approach as Half-Fit, they do not *inherit* this problem. `TLSF` introduced finer grained segregated lists and always rounds up block sizes to the next segregated list base. While this simply transforms incomplete memory use into internal fragmentation, much lower analytical worst-case bounds can be given, depending on the number of second level size classes. Unfortunately, simply rounding up block sizes does not help counteracting the type of incomplete memory use occurring in `PRADA` nor `CAMA`. Still, the incomplete memory use of the cache-aware allocators can be counteracted by increasing their WCET for allocations. Currently, both allocators maintain a bit sequence indicating whether the segregated lists corresponding to the bits contain free blocks or are empty. This sequence is sorted in

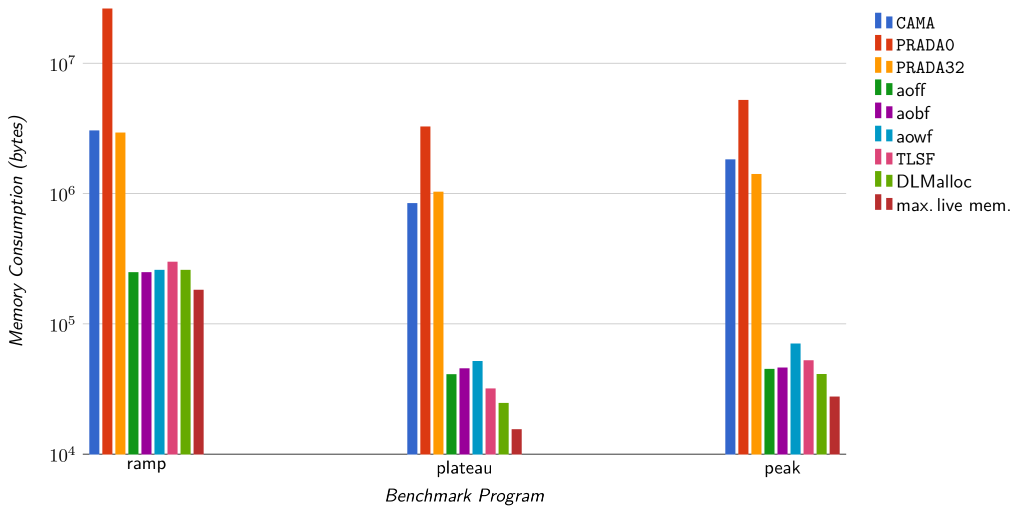

Figure 5 Memory consumption on our synthetic (de-) allocation traces.

ascending cache set numbers and (per cache set) ascending size classes. The allocator handles an allocation request for *size* bytes mapped to cache set k by computing which segregated list L would contain 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 sub-string 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 no such bit is found, we would like to also consider other cache sets and search for larger block to split to prevent incomplete memory use. We can do this in constant time by having a second sequence of bits with the same semantics but different order. This sequence is sorted by descending size classes and (per size class) descending cache sets. On this sequence, we again search for the first bit set and take the first block from the free list corresponding to this bit and check whether it can be split to yield a block suitable to serve the original request. In other words, if the allocator's constant time *good fit* approach finds no suitable block, it reverts to a slower (in the worst-case the whole bit sequence is read), but still constant time *bad fit* approach. This fall-back mechanism is, however, not implemented yet.

Figure 5 shows the actual memory use for our randomly generated traces. On these synthetical traces, an even larger impact on the memory consumption is observable when splitting and merging is disabled. We also observe that enforcing a statically predefined cache set mapping raises memory consumption more than ensuring constant response times on these traces. Also, incomplete memory use turns out to be again the greatest source of memory waste.

4 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 bounded by a constant. However, it may suffer from a relatively high internal fragmentation. The first dynamic memory allocation algorithm especially aiming at satisfying the requirements of real-time applications, `Half-Fit`, was proposed by Takeshi Ogasawara in 1995 [12]. His segregated lists approach was further refined in `TLSF` [11]. The first real-time allocator also considering its cache effects was CAMA [8, 6].

Chilimbi et al. proposed a so-called cache-conscious memory allocator (ccmalloc), however, they aimed to improve program execution times [1]. Chilimbi's ccmalloc also takes an additional argument like CAMA and PRADA. However, instead of a fixed cache set, ccmalloc takes a pointer to an existing object that is likely to be accessed contemporaneously with the object 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 allocated storage is often located in the same cache set as the referenced one.

Besides efforts to make memory allocators more predictable, automatically transforming dynamic memory allocation into static memory allocation was proposed as a means to allow programmers to employ dynamic memory allocation in real-time applications. Approaches to algorithmically find suitable static allocations schemes for a given program with dynamic allocation are proposed in [7] and [5].

5 Conclusions

The contributions of this paper are twofold. We propose PRADA, an alternative approach to cache-aware dynamic memory allocation. We also present a small case study investigating the sources of fragmentation and general spatial costs of dynamic memory allocation in real-time applications.
PRADA overcomes the disadvantages of general purpose dynamic memory allocators in hard real-time systems. Its implementation is simpler than that of CAMA. For the proposed allocator, a tight bound on the WCET for allocations and deallocations can be derived. The effect of allocations and deallocations on the cache state is bounded to a single cache set. This introduces predictable cache behaviour that does not hinder a static cache analysis to derive precise information about an application's cache performance. Which, again, can be used by a timing analysis to derive tight WCET bounds for the application. PRADA achieves this predictability by deferring a fixed amount of actions (splits/merges) which would cause unpredictable behaviour if always directly executed. This bound can be configured. Lower bounds may yield higher fragmentation, but lower WCET estimates. Resources on embedded hardware are restricted. And the possibility of configuration may widen the space of possible applications of our allocator.
With respect to memory consumption, we make several observations:

- Enforcing a cache set mapping on dynamically allocated objects does not necessarily significantly increase the application's memory consumption.
- CAMA's most general approach to immediately execute splits and merge (and never *drop* such an action) may not be needed in current real-time applications to keep memory consumption low. However, generally, deferred splits and merges cause higher memory usage[14], so once real-time applications become more and more complex, this may change.
- Completely disabling splitting and merging does significantly increase the memory consumption, even for (generally simpler) current real-time applications.
- Current approaches pay for strong guarantees about their cache influence with potentially drastic increases in memory consumption due to *incomplete memory use*. However, there is a potential trade-off to reduce this type of fragmentation at the price of increased WCET bounds and increased, although predictable cache usage.

References

1 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. *IEEE Computer*, 33(12):67–74, 2000.

2 Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. Ieee, 2001.

3 Florian Haupenthal. PRADA: Predictable Allocations by Deferred Actions. Bachelor's thesis, Saarland University, 2012.

4 Reinhold Heckmann, Christian Ferdinand, Absint Angewandte, and Informatik Gmbh. Worst-Case Execution Time Prediction by Static Program Analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society, 2004.

5 Jörg Herter and Sebastian Altmeyer. Precomputing Memory Locations for Parametric Allocations. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 125–136. Austrian Computer Society, July 2010.

6 Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS '11)*. IEEE Computer Society, July 2011.

7 Jörg Herter and Jan Reineke. Making Dynamic Memory Allocation Static To Support WCET Analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

8 Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-Aware Memory Allocation for WCET Analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

9 Daniel S. Hirschberg. A Class of Dynamic Memory Allocation Algorithms. *Commun. ACM*, 16(10):615–618, October 1973.

10 Doug Lea. A Memory Allocator. unix/mail, 1996.

11 Miguel Masmano, Ismael Ripoll, Jorge Real, Alfons Crespo, and Andy J. Wellings. Implementation of a Constant-Time Dynamic Storage Allocator. *Software: Practice and Experience*, 38(10):995–1026, 2008.

12 Takeshi Ogasawara. An Algorithm with Constant Execution Time for Dynamic Storage Allocation. *Real-Time Computing Systems and Applications, International Workshop on*, 0:21, 1995.

13 James L. Peterson and Theodore A. Norman. Buddy Systems. *Commun. ACM*, 20:421–431, June 1977.

14 Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Henry G. Baker, editor, *IWMM*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer, 1995.