CAMA

A Predictable Cache-Aware Memory Allocator

Jörg Herter, Peter Backes, Jan Reineke, Florian Haupenthal

Department of Computer Science
Saarland University

Current Situation

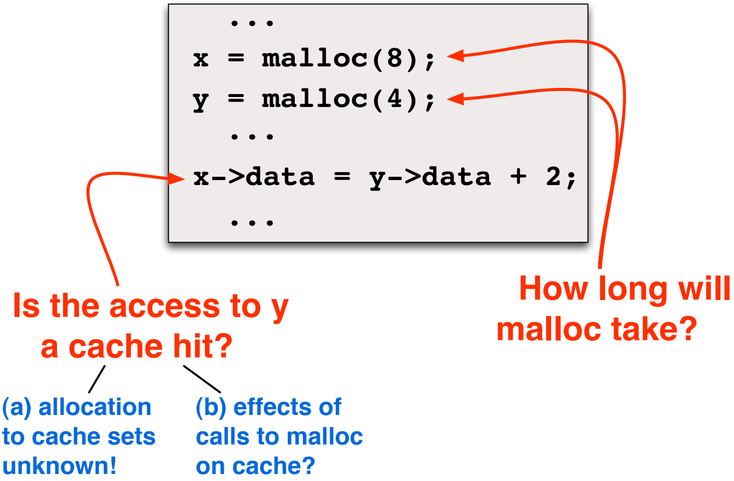
What we have ...

1. Precise WCET analysis
2. Dynamic Memory Allocation
    - often clearer program structure
    - easy memory reuse (e.g. in-situ transformations)

... but can we have both together?

Dynamic Memory Allocation & WCET Analysis

What are the challenges?



```
   ...
x = malloc(8);
y = malloc(4);
   ...
x->data = y->data + 2;
   ...
```

Cache-Aware Memory Allocation

```
   ...
x = camalloc(8,2);
y = camalloc(4,32);
   ...
x->data = y->data + 2;
   ...
```

1 **How long will camalloc take?**

constant response times!

2 **Is the access to y a cache hit?**

allocation to cache sets known!

effects of calls to camalloc on cache known!

Constant Time Allocators

Constant time allocators:

- (One level) Segregated list allocators
  - Idea:
    - manage free blocks in segregated free lists
    - blocks within the same free list fall into the same size class
  - Drawbacks: potential for high fragmentation
- TLSF[1] (two-level segregated fit)
  - Idea:
    - manage free blocks in segregated free lists
    - use two-level approach to building size classes to decrease the potential for fragmentation
  - Drawbacks: no cache predictability

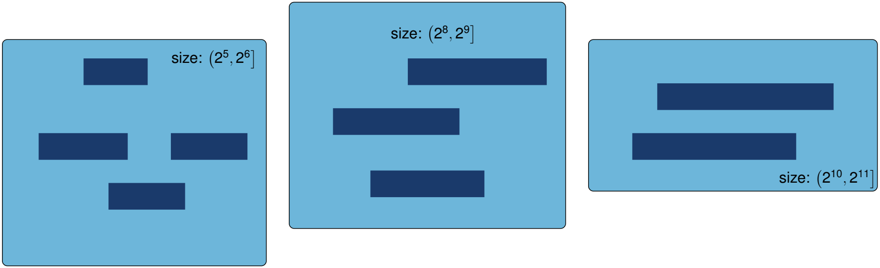
---

[1] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "*TLSF: A new dynamic memory allocator for real-time systems*," ECRTS ’04

One-Level Segregated List Allocators

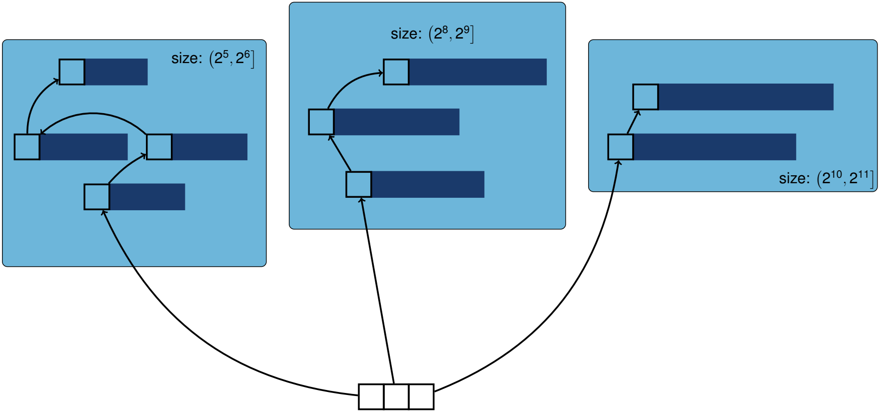
Take set of all free blocks . . .

One-Level Segregated List Allocators



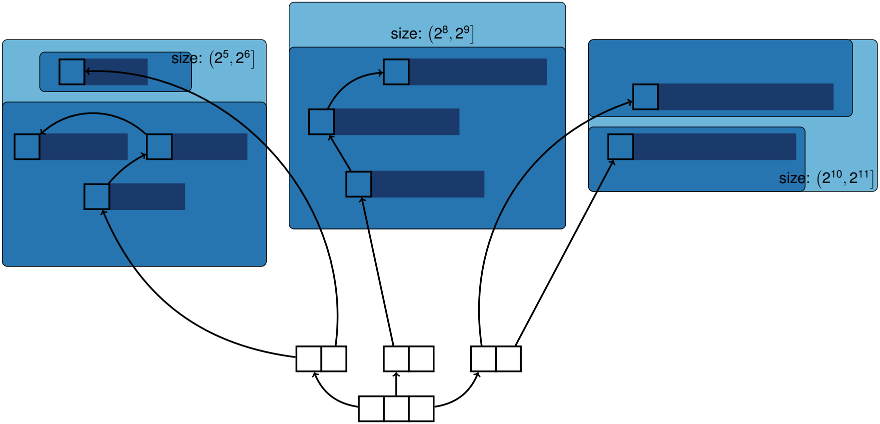
Partition this set into sets containing blocks of the same size class …

One-Level Segregated List Allocators



Finally, organize these subsets in segregated free lists.
List addressed by *i* contains blocks of sizes $\in \left(2^i, 2^{i+1}\right]$.

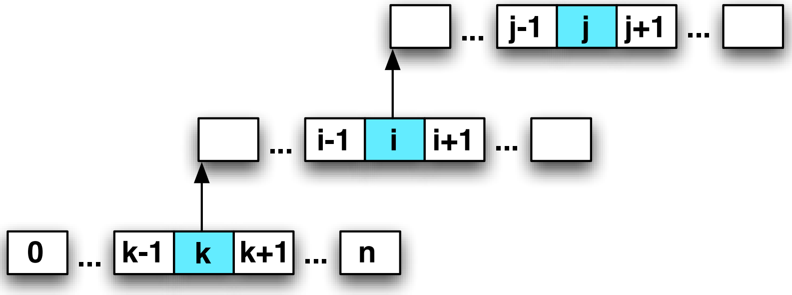
Two-Level Segregated Fit Allocator (TLSF)



Segregated list addressed by pair (i, j) contains blocks of sizes $\in \left(2^i + \frac{2^i}{L} \cdot j; 2^i + \frac{2^i}{L} \cdot (j+1)\right]$, L number of linear classes.

Cache-Aware Memory Allocation

CAMA adds a third layer to this scheme:



Segregated list addressed by (k, i, j) contains blocks starting in cache set k of sizes $\in \left(2^i + \frac{2^i}{L} \cdot j; 2^i + \frac{2^i}{L} \cdot (j+1)\right]$.

How are we doing so far?

Problems solved:

- constant execution times
- explicit cache set mapping of allocated blocks
- cache influence of (de)allocation routines predictable

Open issues:

- still potential for high fragmentation, cannot just copy TLSF's splitting and merge operations

Splitting & Merging

Constant-time, cache-aware splitting and merging?

1. splitting: split large free blocks to satisfy requests for smaller blocks
2. merging: merge consecutive free blocks to satisfy later requests for larger blocks

Splitting & Merging

Problem: Splitting/Merging has unknown effects on cache

Merging. During deallocation, we do not know:

- whether merging will occur,
- how large the block we merge are, and hence,
- at which cache set the merged blocks start.

Splitting & Merging

How to ’make splitting/merging cache-aware’?

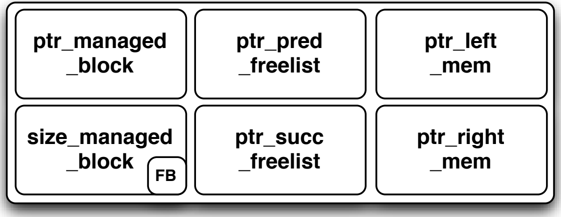
- Do not store free blocks directly in the segregated free list, but management units (*descriptors*) for these blocks!
- Store descriptors only in memory locations mapped to a known, bounded range of cache sets!

Descriptor Blocks

What information do we have to store in a descriptor?

ptr_managed_block	**ptr_pred_freelist**	**ptr_left_mem**
size_managed_block **FB**	**ptr_succ_freelist**	**ptr_right_mem**

Descriptor Blocks



ptr_managed _block

ptr_pred _freelist

ptr_left _mem

size_managed _block

FB

ptr_succ _freelist

ptr_right _mem

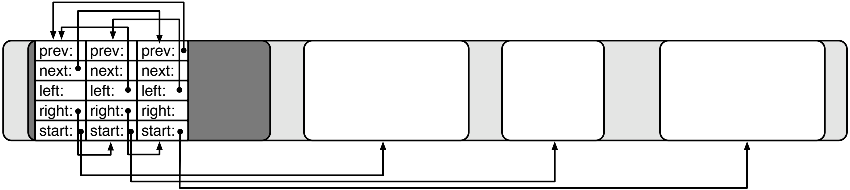
Splitting

1. update size of managed block,
2. update right memory neighbor,
3. add new descriptor for remainder.

Merging

1. update size of managed block,
2. update right memory neighbor,
3. remove descriptors of merged blocks.

Cache-Aware Memory Allocation

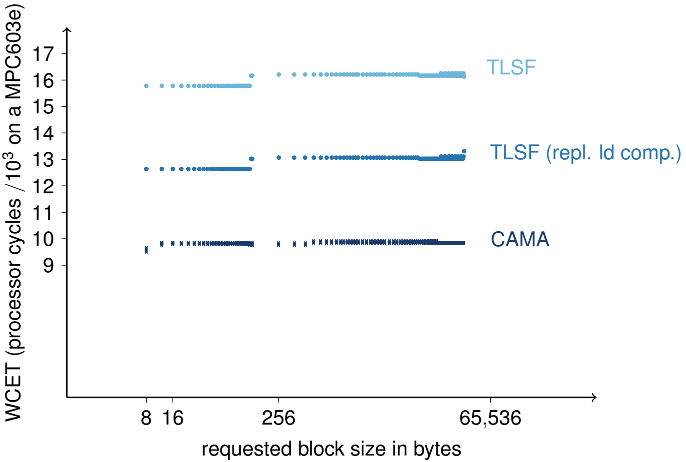


Summary:

- Manage not free blocks but descriptors in segregated free lists.
- ’All’ accesses go to descriptor blocks.
- Descriptor blocks mapped to dedicated cache sets.
- Results in known number of accesses to known cache sets.
- Third cache set level.

Benchmark Results—WCET Bounds for CAMA & TLSF

Provable[2] WCET of the allocation routines on a MPC603e:



---

[2] Derived by AbsInt's a^3; http://www.absint.de/ait/

Benchmark Results—WCET Bounds for CAMA & TLSF

Provable WCET of the allocation routines on a MPC603e can be bounded by:

- CAMA: 9,935 cycles
- TLSF: 13,026 cycles[3]

Provable WCET of the deallocation routines on a MPC603e:

- CAMA: 6,891 cycles
- TLSF: 5,703 cycles

---

[3] 16,260 cycles for the unmodified version of TLSF.

Benchmark Results—Potential to Lower WCET Bounds?

Assume a simple task scheduler with segregated task lists and a main loop body:

```
struct task_descr* lowPriority = low;
struct task_descr* highPriority = high;

// loop bound: 16
for(i = 0; i < LP_LIST_SIZE; i++) {
   // loop bound: 4
   for(j = 0; j < HP_LIST_SIZE; j++) {
      // high prioritized tasks waiting?
      ...
      high = high->next;
   }
   high = highPriority;
   // next lower prioritized task waiting?
   ...
   low = low->next;
}
low = lowPriority;
```

1. allocate all objects with CAMA s.t. high and low priority objects map to disjoint cache sets
2. allocate all objects with some constant-time allocator without explicit/known cache set mapping

Benchmark Results—Potential to Lower WCET Bounds?

Assume a simple task scheduler with segregated task lists and a main loop body:

```
struct task_descr* lowPriority = low;
struct task_descr* highPriority = high;

// loop bound: 16
for(i = 0; i < LP_LIST_SIZE; i++) {
   // loop bound: 4
   for(j = 0; j < HP_LIST_SIZE; j++) {
      // high prioritized tasks waiting?
      ...
      high = high->next;
   }
   high = highPriority;
   // next lower prioritized task waiting?
   ...
   low = low->next;
}
low = lowPriority;
```

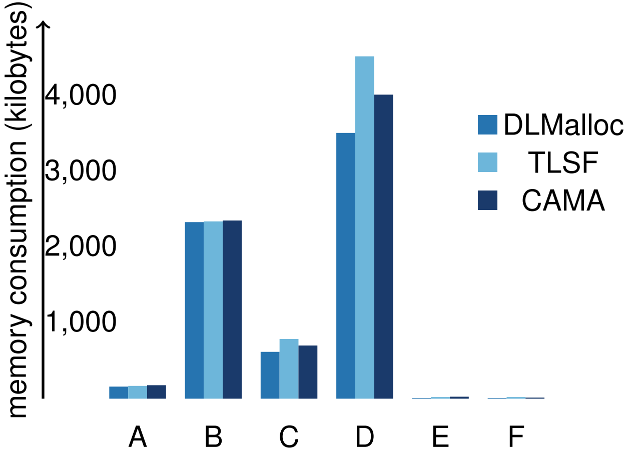
1. provable WCET using CAMA to segregate lists in cache: 6,505 cycles
2. provable WCET otherwise: 10,915 cycles

Memory Consumption/Fragmentation

How to benchmark fragmentation?

- Random (de)allocation traces?
- Traces from (hard) real-time applications?

Benchmark Results—Fragmentation



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).

Conclusions

- Cache-awareness does not necessarily nor overly increase fragmentation compared to other real-time allocators.
- Predictable, cache-aware allocators do have potential do drastically decrease WCET bounds, and …
- …enable dynamic memory allocation for hard real-time applications.