東京工業大学
Tokyo Institute of Technology

Memory-Efficient Object-Oriented Programming on GPUs

Matthias Springer

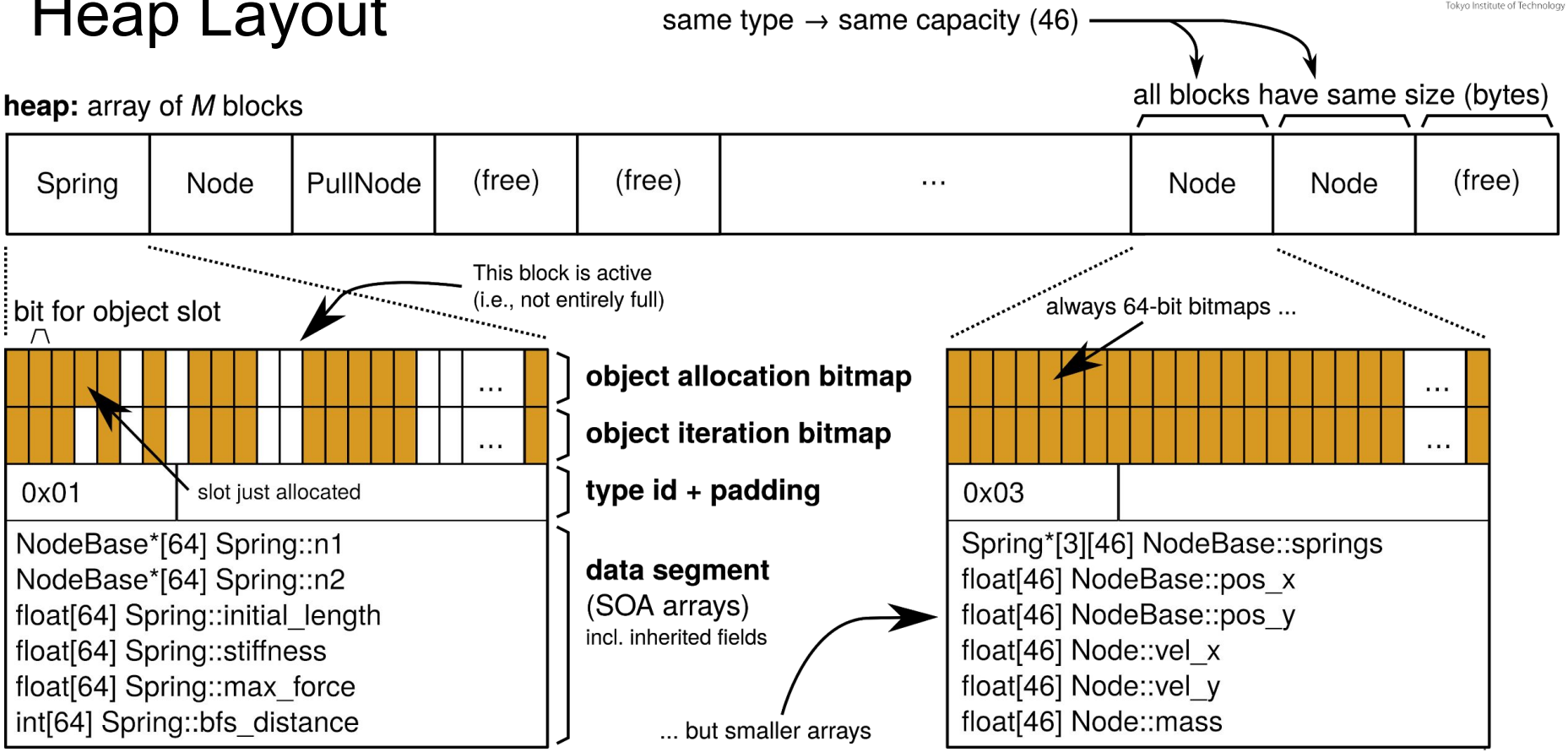
Thesis Exam, 07/30/2019

Outline

1. Details of DynaSOAr: **Allocation/Deallocation by Example**
2. **Comparison** of DynaSOAr with other (Lock-free) GPU Allocators
3. **Overhead of Ikra-Cpp** / DynaSOAr Data Layout DSL
4. Integration of **DynaSOAr with OpenMP**/...

“Explain Details of the DynaSOAr Algorithm”

Heap Layout


same type → same capacity (46)
all blocks have same size (bytes)
heap: array of M blocks
Spring
Node
PullNode
(free)
(free)
...
Node
Node
(free)
bit for object slot
This block is active
(i.e., not entirely full)
always 64-bit bitmaps ...
object allocation bitmap
object iteration bitmap
type id + padding
0x01
slot just allocated
0x03
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance
data segment
(SOA arrays)
incl. inherited fields
... but smaller arrays
Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

Object Allocation

1. **Select** *active[T]* block for allocation.
   Initialize a new active[T] block if none found.
2. **Reserve** object slot in selected block.
3. **Update** block state bitmaps (*indices*).

1

2

3

Algorithm 1: DAllocatorHandle::allocate<T>() : T* GPU

```
repeat                                                          ▷ Infinite loop if OOM
    bid ← active[T].try_find_set();          ▷ Find and return the position of any set bit.
    if bid = FAIL then                                                          ▷ Slow path
        bid ← free.clear();          ▷ Find and clear a set bit atomically, return position.
        initialize_block<T>(bid);                    ▷ Set type ID, initialize object bitmaps.
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                     ▷ Reserve an object slot. See Alg. 7.
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;                                                   ▷ Volatile read
        if alloc.state = FULL then  active[t].clear(bid) ;
        if t = T then  return ptr ;
        deallocate<t>(ptr);                          ▷ Type of block has changed. Rollback.
until false;
```

Walk through **allocation** with two concurrent threads.

① Object Allocation by Exam

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve a
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is active
(i.e., not entirely full)

object allocation bitmap

object iteration bitmap

type id + padding

0x01 — slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)

free

allocated[Node] active[Node]

allocated[PullNode] active[PullNode]

allocated[Spring] active[Spring]

② Object Allocation by Exam

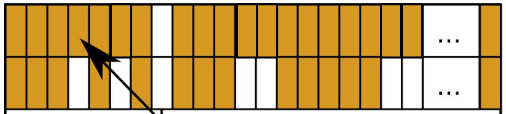
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

This block is active
(i.e., not entirely full)

bit for object slot



object allocation bitmap

object iteration bitmap

0x01 — slot just allocated

type id + padding

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

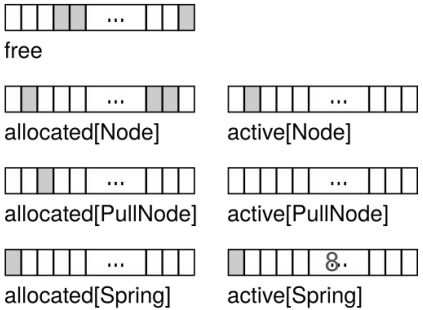
0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



③ Object Allocation by Exam

same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

alloc

This block is now full!
No longer active.

bit for objects

object allocation bitmap

object iteration bitmap

type id + padding

0x01 — slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

0x03

float[46] Node::mass

Inconsistency between block state index and actual block state!

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

Atomic semantics

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)

free

allocated[Node] — active[Node]

allocated[PullNode] — active[PullNode]

allocated[Spring] — active[Spring]

④ Object Allocation by Exam

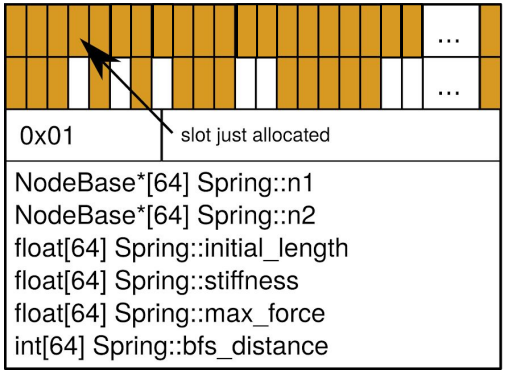
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is now full!
No longer active.



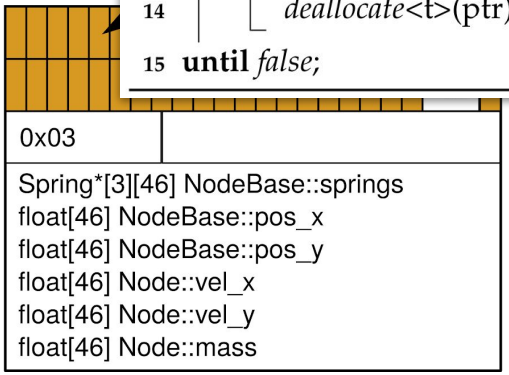
object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

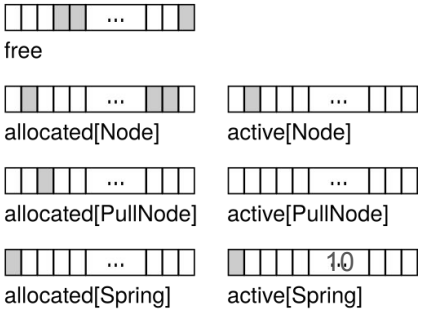
… but smaller arrays



Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve a
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



⑤ Object Allocation by Exam

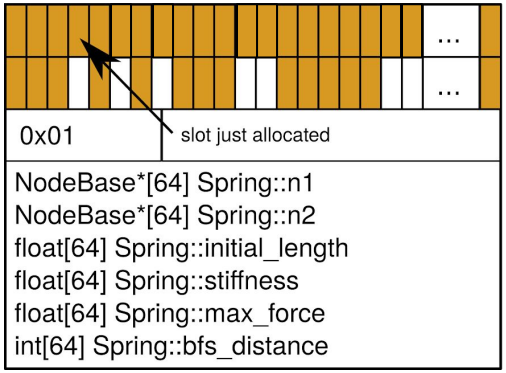
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is now full!
No longer active.



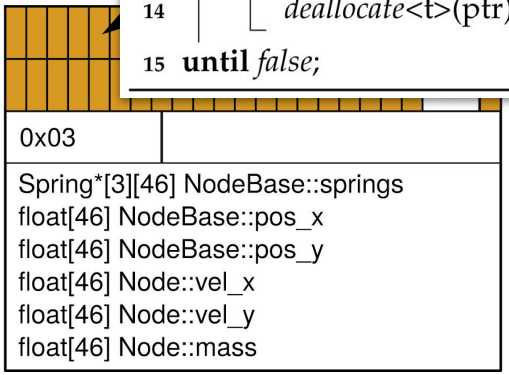
object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

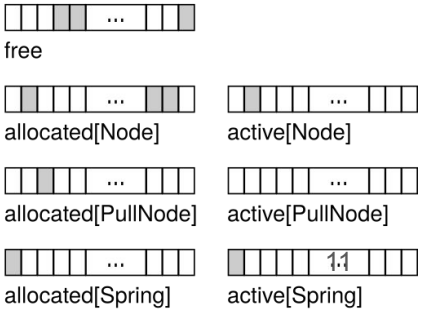
… but smaller arrays



Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve a
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



⑥ Object Allocation by Exam

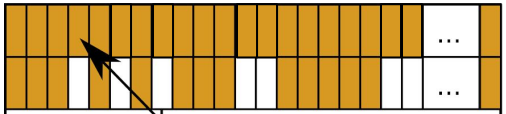
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

This block is now full!
No longer active.

bit for object slot



object allocation bitmap

object iteration bitmap

0x01 | slot just allocated

type id + padding

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

0x03

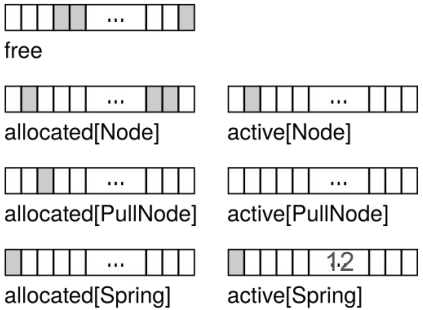
Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reser                        ▷ Reserve a
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

FAIL!

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



⑦ Object Allocation by Exam

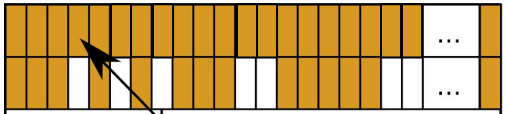
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is now full!
No longer active.



object allocation bitmap

object iteration bitmap

0x01 | slot just allocated

type id + padding

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

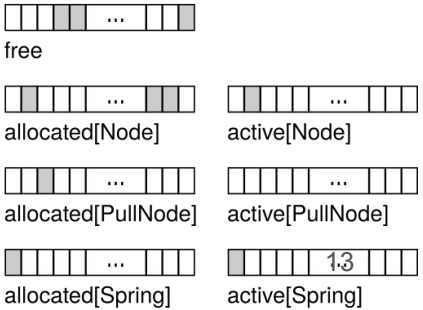
Algorithm 1: DAllocatorHandle:

```
repeat
    bid ← active[T].try_find_set();                    return t
    if bid = FAIL then
        bid ← free.clear();                            ▷ Find a clear a set bit a
        initialize_block<T>(bid)                       type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.size
        t ← heap[bid].type;
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                            ▷ Type of bloc
until false;
```

Retry. Select new block.

… but let's focus on the other thread.

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



⑧ Object Allocation by Exam

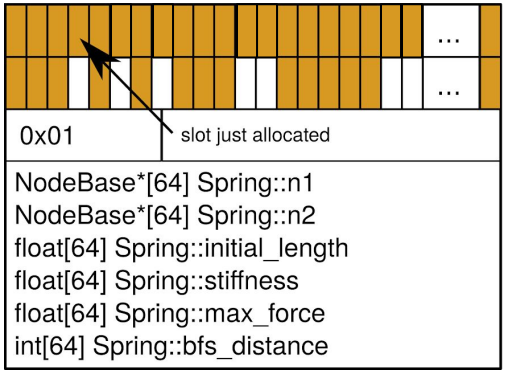
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is now full!
No longer active.



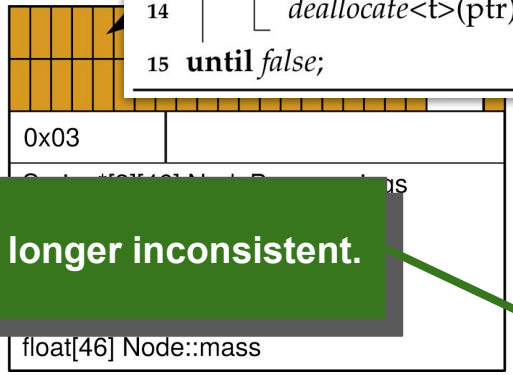
object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays



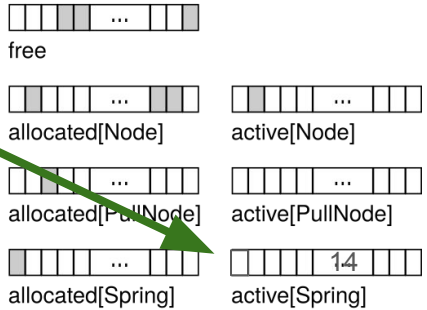
Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat
    bid ← active[T].try_find_set();                ▷ Find and return t
    if bid = FAIL then
        bid ← free.clear();                        ▷ Find and clear a set bit a
        initialize_block<T>(bid);                  ▷ Set type ID
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve a
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, a
        t ← heap[bid].type;
        if alloc.state = FULL then  active[t].clear(bid) ;
        if t = T then  return ptr ;
        deallocate<t>(ptr);                        ▷ Type of bloc
until false;
```

Block is now full!

No longer inconsistent.

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



⑨ Object Allocation by Exam

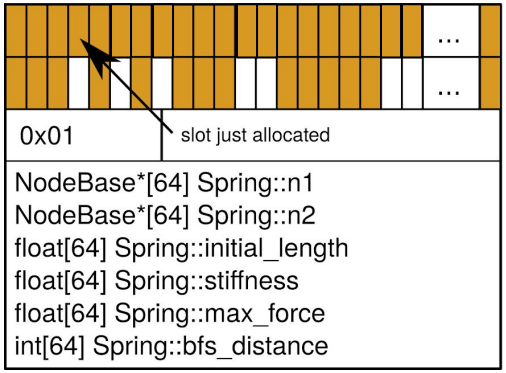
same type → same capacity (46

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…

bit for object slot

This block is now full!
No longer active.



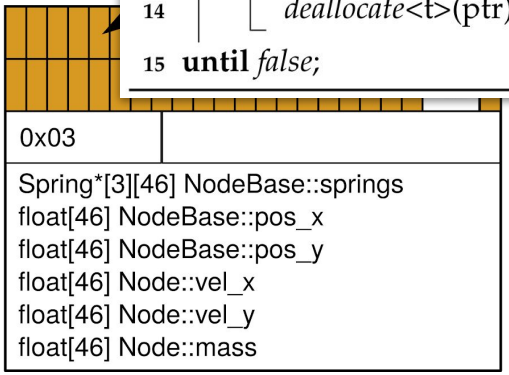
object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays



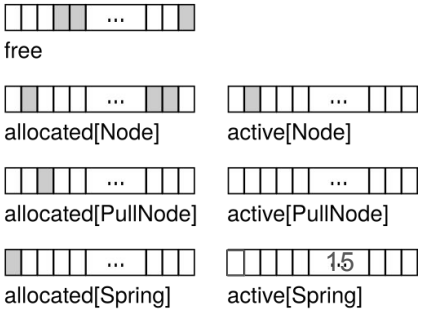
Algorithm 1: DAllocatorHandle::allocate<T>() : T*

1 **repeat**
2 bid … d return t
3 …
4 … set bit a
5 … Set type ID
6 …
7 active[T].set(bid);
8 alloc ← heap[bid].*reserve*(); ▷ Reserve a
9 **if** alloc ≠ *FAIL* **t**
10 ptr ← *make_*
11 t ← heap[b
12 **if** alloc.sta
13 **if** $t = T$ **then return** ptr ;
14 *deallocate*<t>(ptr); ▷ Type of bloc
15 **until** *false*;

Block could have been deleted and reinitialized to another type *t != T* before Line 8.

Double check if block type is still *T*.

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)

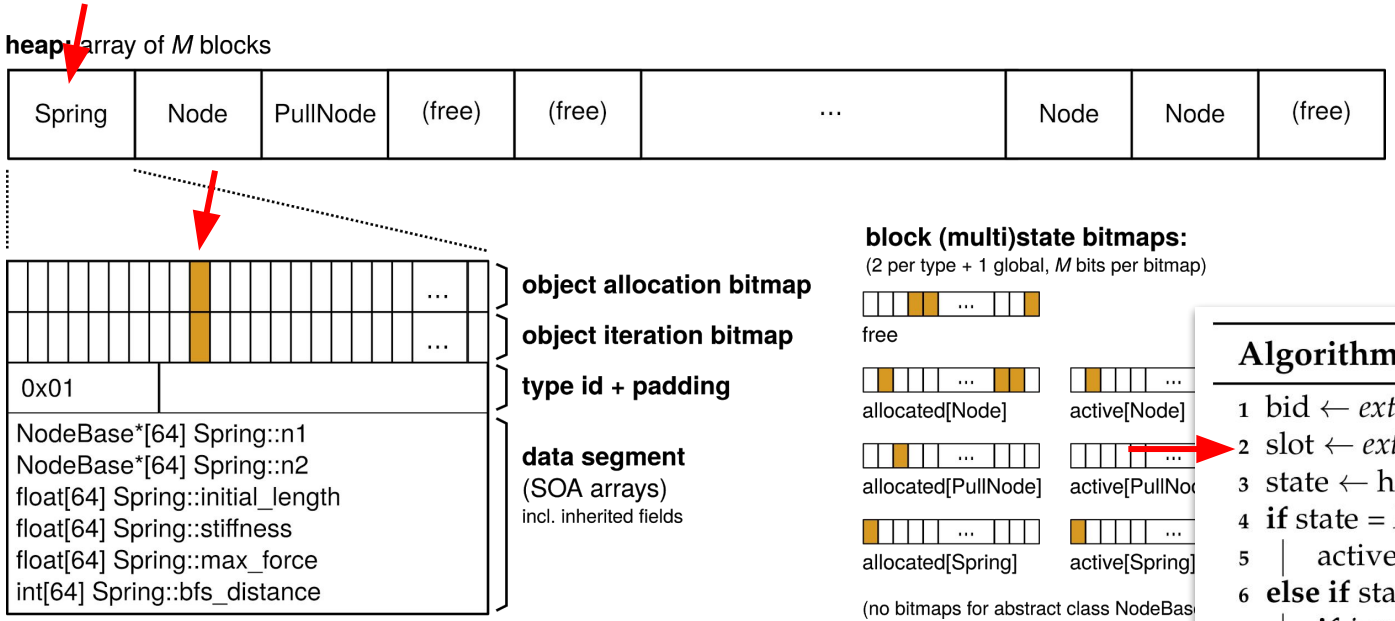


Challenges in Object Allocation

- We use block state bitmaps for finding active blocks, but those **bitmaps may be (temporarily) inconsistent**.
  - *Source of truth:* Values stored inside block.
  - Bitmaps are only **indices** and they may not always be correct.
  - *Solution:* Use bitmaps for finding blocks quickly, then double check by looking at block.
  - Slot reservation is **optimistic**.
    - Assuming that block state has not changed. Otherwise, we have to **rollback**.
- *Block selection* and *block reservation* **together are not atomic**.
  - E.g.: Two threads may select the same block with only one free object slot. Only one thread can succeed with slot reservation.
  - *Assumption in* `Block::reserve()`*:* Block has at least 1 free object slot and is of type *T*.
  - This assumption may sometimes be violated, in which case we retry.

Walk through **deallocation** with one thread.

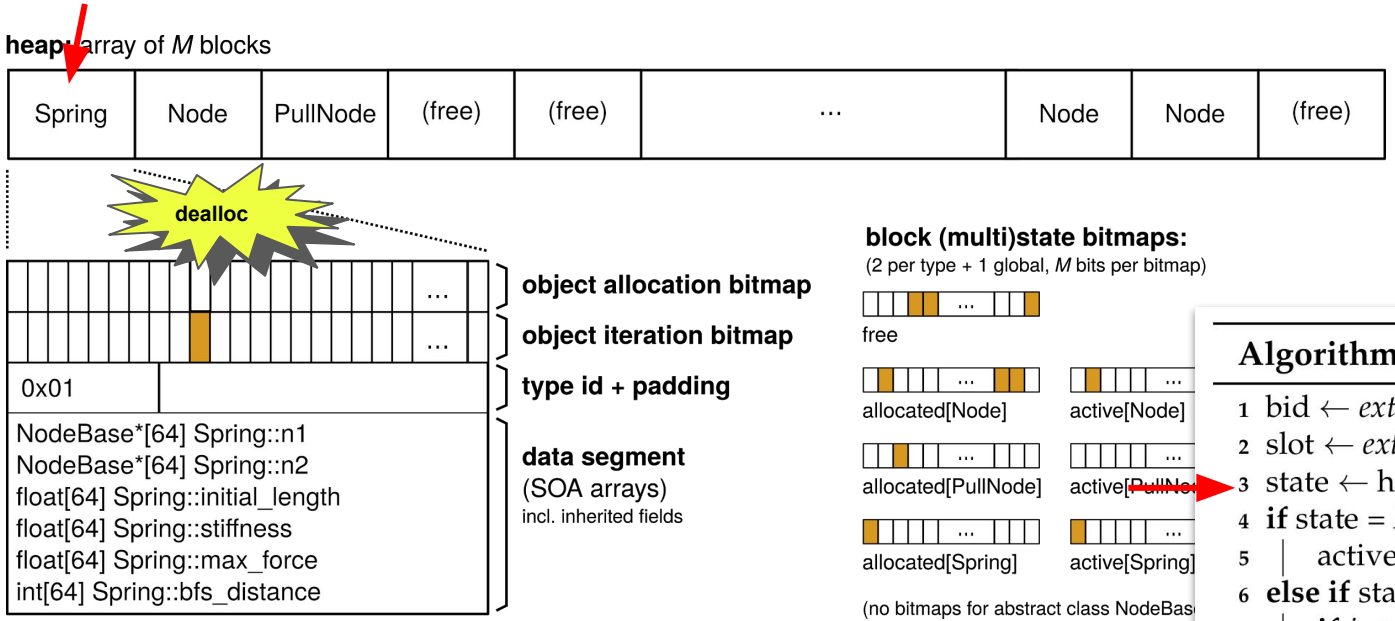
① Block Deletion by Example



Algorithm 2: DAllocatorHandle::dealloca

```
bid ← extract_block(ptr);
slot ← extract_slot(ptr);
state ← heap[bid].deallocate(slot);
if state = FIRST then
    active[T].set(bid);
else if state = EMPTY then
    if invalidate(bid) then
        t ← heap[bid].type;
        active[t].clear(bid);
        allocated[t].clear(bid);
        free.set(bid);
```

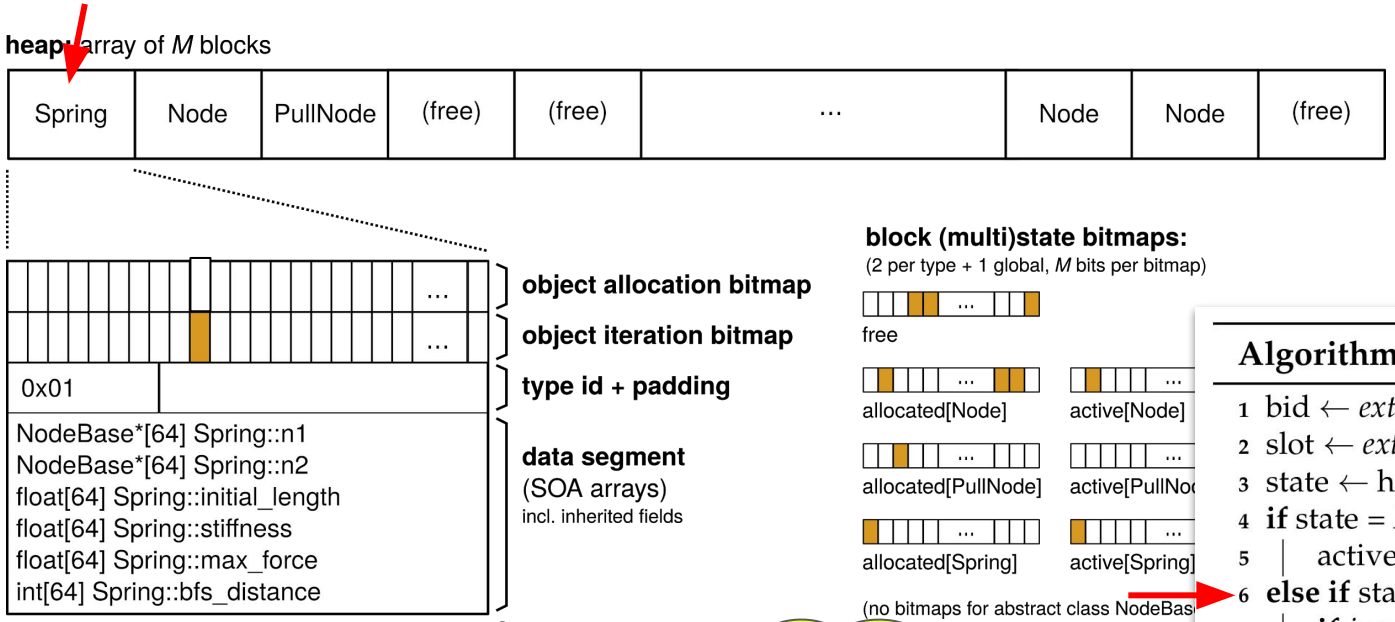
② Block Deletion by Example



Algorithm 2: DAllocatorHandle::dealloca

```
bid ← extract_block(ptr);
slot ← extract_slot(ptr);
state ← heap[bid].deallocate(slot);
if state = FIRST then
    active[T].set(bid);
else if state = EMPTY then
    if invalidate(bid) then
        t ← heap[bid].type;
        active[t].clear(bid);
        allocated[t].clear(bid);
        free.set(bid);
```

③ Block Deletion by Example



Can I delete this block now?

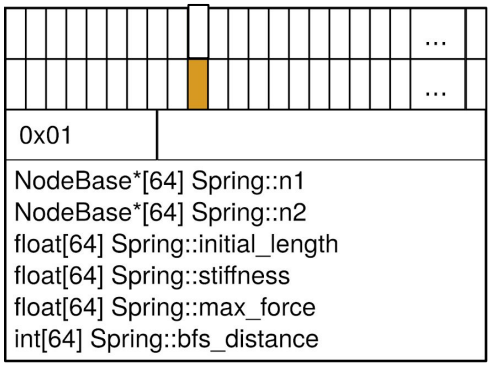
Algorithm 2: DAllocatorHandle::dealloca

```
bid ← extract_block(ptr);
slot ← extract_slot(ptr);
state ← heap[bid].deallocate(slot);
if state = FIRST then
    active[T].set(bid);
else if state = EMPTY then
    if invalidate(bid) then
        t ← heap[bid].type;
        active[t].clear(bid);
        allocated[t].clear(bid);
        free.set(bid);
```

④ Block Deletion by Example

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…	Nod



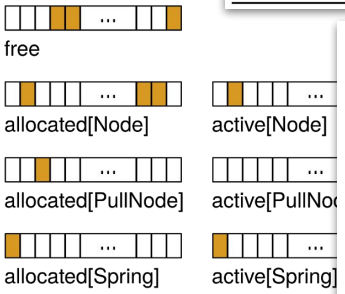
object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

block (multi)state
(2 per type + 1 global, *M* b



(no bitmaps for abstract class NodeBas

Can I delete this block now?

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
repeat                                                  ▷ Infinit
    bid ← active[T].try_find_set();                     ▷ Find and return the position
    if bid = FAIL then
        bid ← free.clear();                             ▷ Find and clear a set bit atomically, r
        initialize_block<T>(bid);                       ▷ Set type ID, initialize c
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                        ▷ Reserve an object sl
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;                             ▷
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                             ▷ Type of block has chan
until false;
```

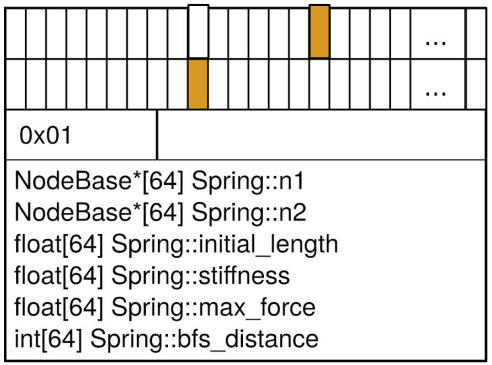
Algorithm 2: DAllocatorHandle::dealloca

```
bid ← extract_block(ptr);
slot ← extract_slot(ptr);
state ← heap[bid].deallocate(slot);
if state = FIRST then
    active[T].set(bid);
else if state = EMPTY then
    if invalidate(bid) then
        t ← heap[bid].type;
        active[t].clear(bid);
        allocated[t].clear(bid);
        free.set(bid);
```

⑤ Block Deletion by Example

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…	Nod



object allocation bitmap

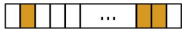
object iteration bitmap

type id + padding

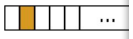
data segment
(SOA arrays)
incl. inherited fields

block (multi)state
(2 per type + 1 global, *M* b

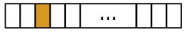
free



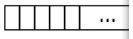
allocated[Node]



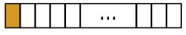
active[Node]



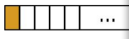
allocated[PullNode]



active[PullNo

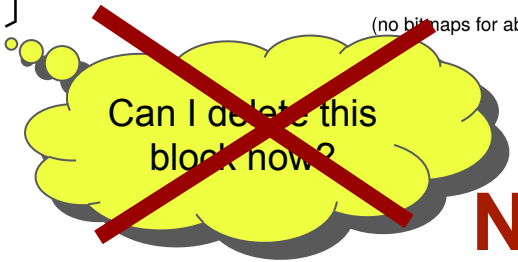


allocated[Spring]



active[Spring]

(no bitmaps for abstract class NodeBas



NO !

Algorithm 1: DAllocatorHandle::allocate<T>() : T*

```
1  repeat                                                    ▷ Infinit
2      bid ← active[T].try_find_set();                       ▷ Find and return the position
3      if bid = FAIL then
4          bid ← free.clear();                               ▷ Find and clear a set bit atomically, r
5          initialize_block<T>(bid);                         ▷ Set type ID, initialize c
6          allocated[T].set(bid);
7          active[T].set(bid);
       alloc ← heap[bid].reserve();                          ▷ Reserve an object sl
9      if alloc ≠ FAIL then
10         ptr ← make_pointer(bid, alloc.slot);
11         t ← heap[bid].type;                                ▷
12         if alloc.state = FULL then active[t].clear(bid) ;
13         if t = T then return ptr ;
14         deallocate<t>(ptr);                               ▷ Type of block has chan
15 until false;
```

Algorithm 2: DAllocatorHandle::dealloca

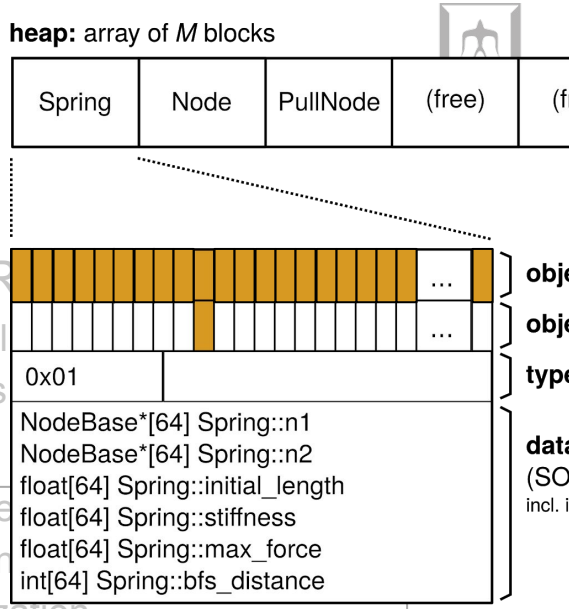
```
1  bid ← extract_block(ptr);
2  slot ← extract_slot(ptr);
3  state ← heap[bid].deallocate(slot);
4  if state = FIRST then
5      active[T].set(bid);
6  else if state = EMPTY then
7      if invalidate(bid) then
8          t ← heap[bid].type;
9          active[t].clear(bid);
10         allocated[t].clear(bid);
11         free.set(bid);
```

Challenges of Object Deallocation

- The basic problem is **Safe Memory Reclamation** (SMR).
  - A notoriously different problem in lock-free algorithms with lots of literature.
  - *Common solutions:* Epoch-based reclamation [1], hazard pointers [2].
- DynaSOAr's approach: **Block invalidation**
  - Set all object slots to "1", so that the block appears to be completely full to other threads.
  - Remove block from the active[T] index, so that other threads will no longer find it.
  - Reinitialize object allocation bitmap to all zeros upon block initialization.
  - Although unlikely, some allocating threads may sleep during the above points and resume allocation in a newly initialized block of now different type. They can detect such problems by checking the type of the block. **Rollback** if necessary.
  - *Crucial design choice:* **All blocks have the same structure.** Same #bytes and object allocation bitmaps are always at the same offset, regardless of block type.

[1] K. Fraser. **Practical Lock-Freedom**. PhD thesis, University of Cambridge Computer Laboratory. 2004.
[2] M. Maged. **Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects**. In: IEEE Transactions on Parallel and Distributed Systems. 2004.

Challenges of Object Deallocation



- The basic problem is **Safe Memory Reclamation** (SMR
  - A notoriously different problem in lock-free algorithms with lots of l
  - *Common solutions:* Epoch-based reclamation [1], hazard pointers
- DynaSOAr's approach: **Block invalidation**
  - Set all object slots to "1", so that the block appears to be complete
  - Remove block from the active[T] index, so that other threads will n
  - Reinitialize object allocation bitmap to all zeros upon block initialization.
  - Although unlikely, some allocating threads may sleep during the above points and resume allocation in a newly initialized block of now different type. They can detect such problems by checking the type of the block. **Rollback** if necessary.
  - *Crucial design choice:* **All blocks have the same structure.** Same #bytes and object allocation bitmaps are always at the same offset, regardless of block type.

[1] K. Fraser. **Practical Lock-Freedom**. PhD thesis, University of Cambridge Computer Laboratory. 2004.
[2] M. Maged. **Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects**. In: IEEE Transactions on Parallel and Distributed Systems. 2004.

Challenges of Object Deallocation

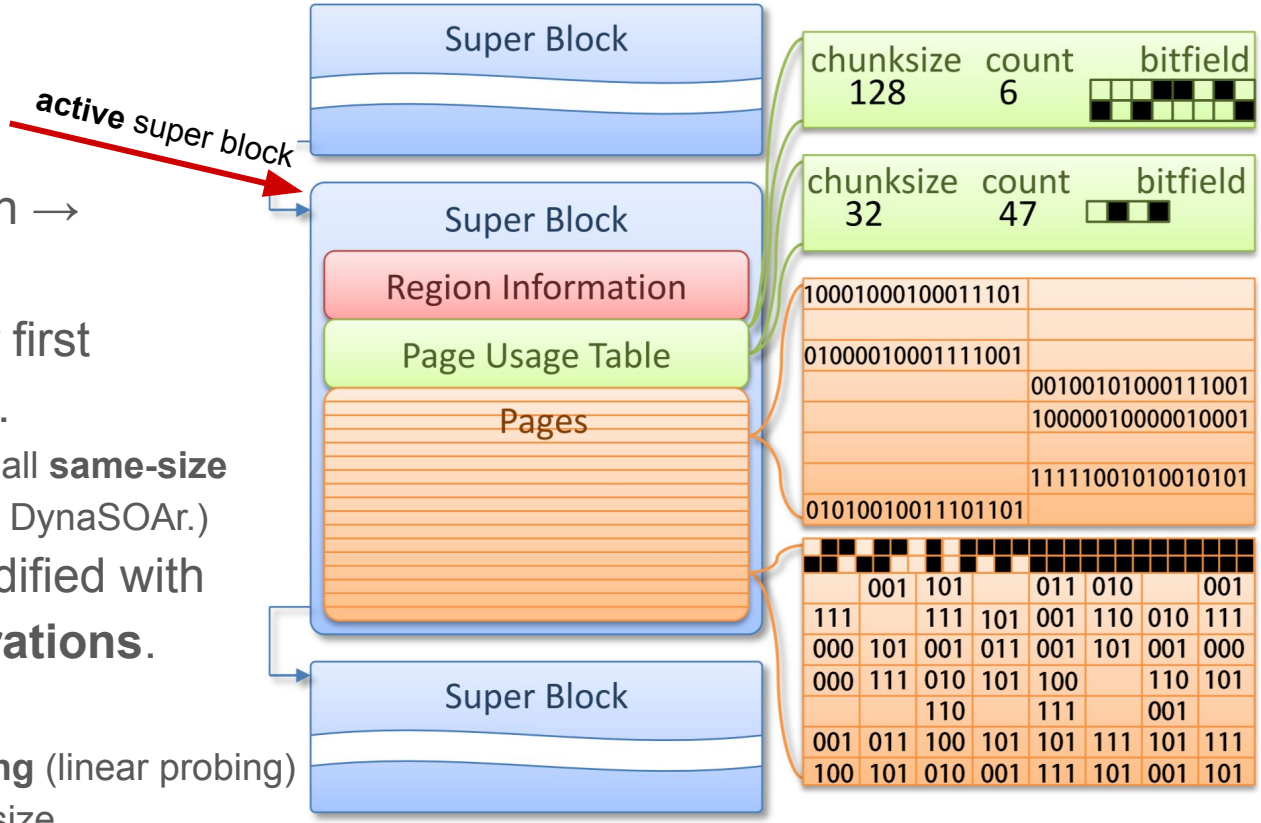
- The basic problem is **Safe Memory Reclamation** (SMR).
  - A notoriously different problem in lock-free algorithms with lots of literature.
  - *Common solutions:* Epoch-based reclamation [1], hazard pointers [2].
- DynaSOAr's approach: **Block invalidation**
  - Set all object slots to "1", so that the block appears to be completely full to other threads.
  - Remove block from the active[T] index, so that other threads will no longer find it.
  - Reinitialize object allocation bitmap to all zeros upon block initialization.
  - Although unlikely, some allocating threads may sleep during the above points and resume allocation in a newly initialized block of now different type. They can detect such problems by checking the type of the block. **Rollback** if necessary.
  - *Crucial design choice:* **All blocks have the same structure.** Same #bytes and object allocation bitmaps are always at the same offset, regardless of block type.

[1] K. Fraser. **Practical Lock-Freedom**. PhD thesis, University of Cambridge Computer Laboratory. 2004.
[2] M. Maged. **Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects**. In: IEEE Transactions on Parallel and Distributed Systems. 2004.

“Compare DynaSOAr with other Lock-free Allocators”

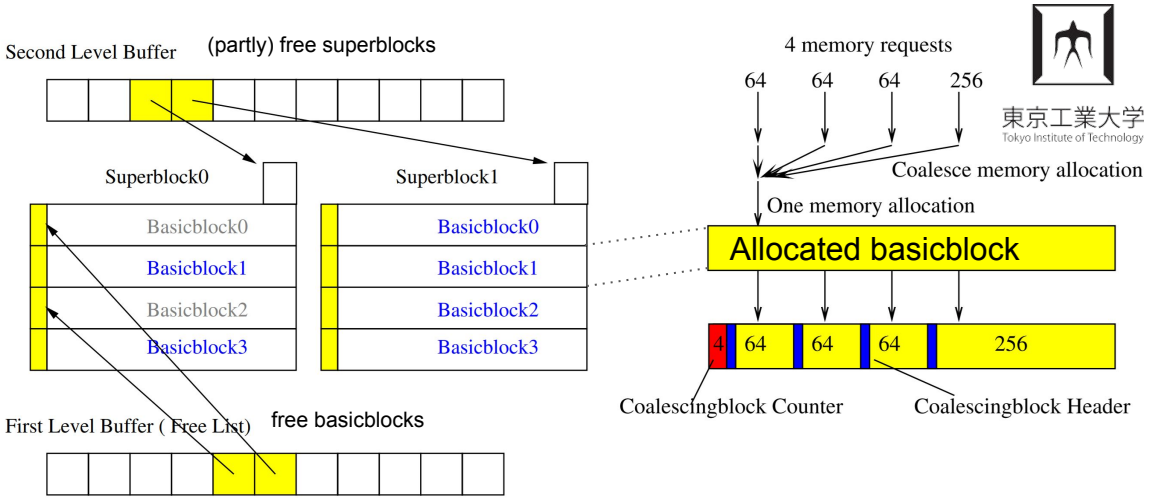
ScatterAlloc [1]

- Super block → Region → Page → Chunk
- Chunk size fixed after first allocation within page.
  - *Assumption:* Many small **same-size allocations**. (Same in DynaSOAr.)
- Page usage table modified with **atomic bit-wise operations**.
- Allocation algorithm
  - Select page by **hashing** (linear probing) SM ID and allocation size.
  - Skip regions with high fill level.
  - **Trade higher fragmentation for faster allocation**. (Opposite of DynaSOAr.)
- Deleting a page requires a **lock** (similar to invalidation in DynaSOAr).



[1] M. Steinberger, et. al. **ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU**. In: InPar 2012.

XMalloc [2]



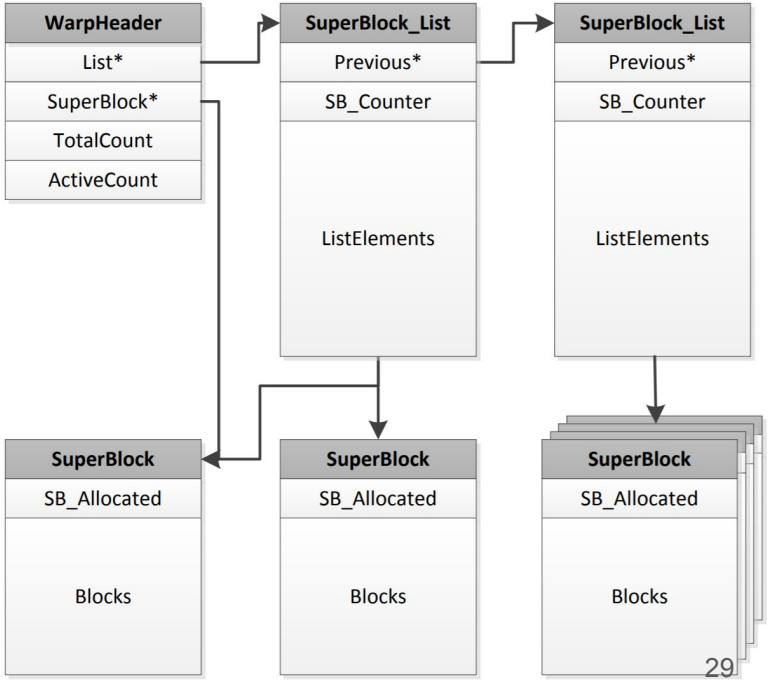
- Memoryblk. → Superblk. → Basicblk. → Coal.blk.
- **Lock-free free lists** for empty *basicblocks* (for **pre-determined** sizes).
- Simultaneous alloc. requests of the same warp are **combined**: Request one *basicblock* and subdivide into *coalescingblocks* to deliver to threads.
- Unclear how SMR is solved.

[2] X. Huang, et. al. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In: CIT 2010.

using same technique in DynaSOAr

FDGMalloc [3]

- **Private heaps**: One heap per warp. (Similar to *Hoard* [4].)
- Programming Interface
  - `malloc`: Allocate memory in private heap. (Less contention/competition among threads.)
  - No `free` operation. Can only **free an entire private heap**.
  - Efficient memory allocation via **bump pointer allocation**.
  - SMR is trivial (delete everything).
- Not expressive enough for SMMO.



[3] S. Widmer, et. al. **Fast Dynamic Memory Allocator for Massively Parallel Architectures**. In: GPGPU-6.

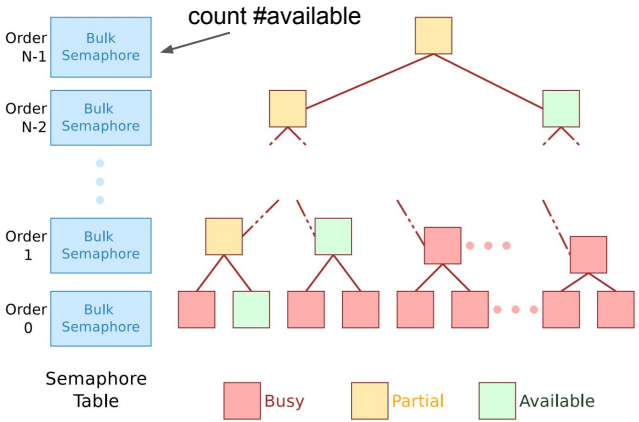
[4] E. D. Berger, et. al. **Hoard: A Scalable Memory Allocator for Multithreaded Applications**. In: ASPLOS 2000.

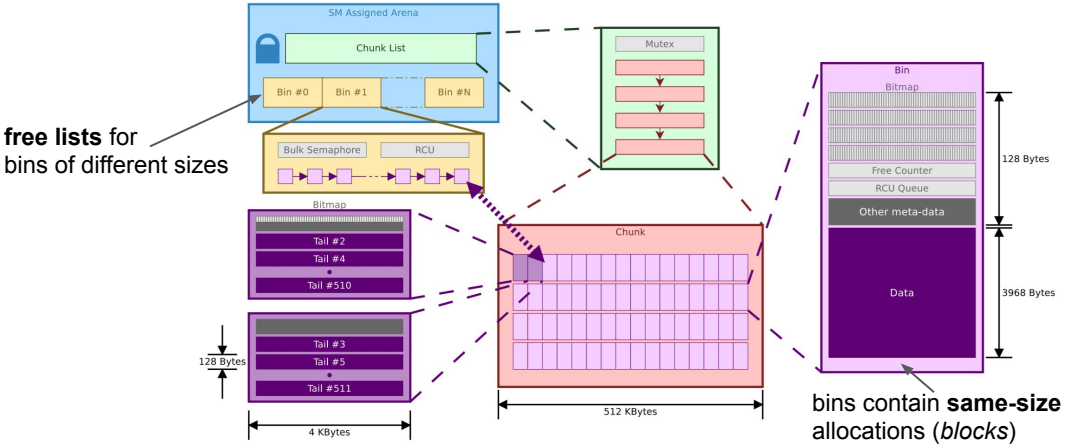
TBuddy/UAlloc [5]

[5] I. Gelado, M. Garland. **Throughput-Oriented GPU Memory Allocation**. In: PPoPP 2019.

東京工業大学
Tokyo Institute of Technology

- Large allocations: *TBuddy*, Small allocations: *UAlloc*





- Check semaphore (thread-safe counter) to see if block available.
- Select block of suitable size and maybe split a higher-order block.
- Updating the tree requires **locking**.

- Arena (per-SM) → Chunk → Bin → Block
- **Bitmaps** to keep track of chunk/bin usage.
- *Alloc.:* Find bin in free list. If none, init. from chunk list.
- Chunks are allocated with TBuddy.
- Unclear how SMR is solved.

hierarchical bitmaps in DynaSOAr are **lock-free**!

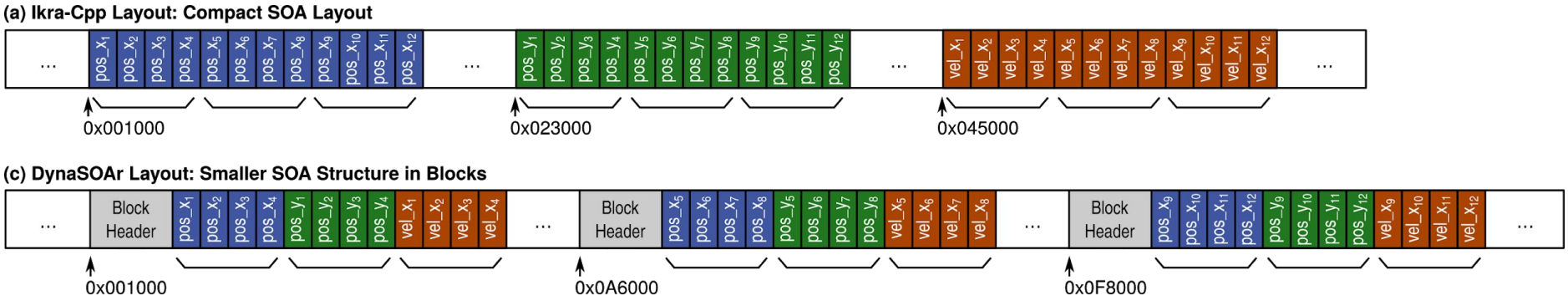
Conclusion

- Other allocators have a **hierarchy of containers** (different kind of blocks) to find free memory fast. DynaSOAr has a **hierarchical index** instead!
  - This simplifies the design.
- Other allocators are **memory allocators**, DynaSOAr is an **object allocator**.
  - Therefore, they cannot apply data layout optimizations (such as SOA).
- Other allocators trade higher fragmentation for faster (de)allocations. **DynaSOAr does the opposite**!
- W.r.t. lock freedom: All GPU allocators based on **atomic operations and retry loops**. Some allocators use a technique similar to **block invalidation**.
- Many different designs for CPU allocators. **Private heaps** are common.
  - E.g.: [6] uses privates heaps, hazard pointers for SMR, blocks states similar to DynaSOAr.

[6] M. M. Michael. **Scalable Lock-Free Dynamic Memory Allocation**. In: PLDI 2004.

“Explain the Overhead of Ikra-Cpp”

Overview



- Ikra-Cpp is a data layout DSL for SOA.
  - Combines **SOA performance** characteristics and notation of **object-oriented programming**.
- DynaSOAr is an extension of Ikra-Cpp with a dynamic memory allocator.
  - DynaSOAr and Ikra-Cpp have different layouts and different overheads.
- There are two kinds of overhead:
  - *Compiler overheads:* DSL makes core more complex, **compiler fails to optimize**.
  - *Address computation overhead:* DSL does some sort of **memory address translation**. In Ikra-Cpp, this translation is free. **In DynaSOAr, it is not free**!

Data Layout DSL: Example

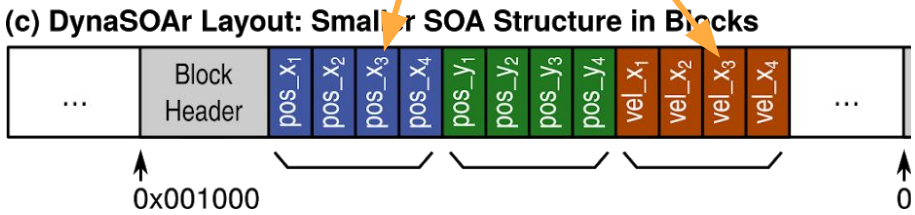
```
class Body : public IkraSoaBase<Body> {
 public:
  declare_field_types(Body, float, float, float,
                            float, float, float, float)

  Field<Body, 0> pos_x = 0.0;
  Field<Body, 1> pos_y = 0.0;
  Field<Body, 2> vel_x = 1.0;
  Field<Body, 3> vel_y = 1.0;
  Field<Body, 4> force_x;
  Field<Body, 5> force_y;
  Field<Body, 6> mass;

  void move(float dt) {
    pos_x = pos_x + vel_x * dt;
    pos_y = pos_y + vel_y * dt;
  }
};
```

proxy type

Can be used like a normal C++ class:

```
Body* b = new Body();
b->pos_x = 1.5f;
b->vel_x = 0.9f;
```

address translation
(in software/C++ code)

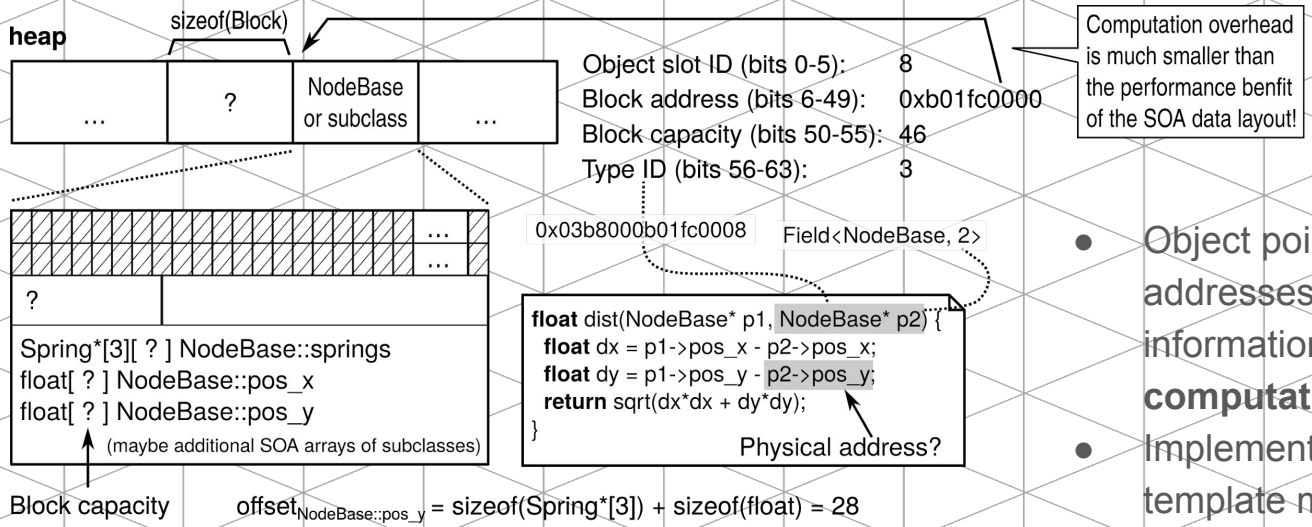


Fake Pointers

```
Body* b = new Body();
printf("%p\n", b);   // e.g.: 0x03b8000b01fc0008 -- Not a valid memory address.
printf("%c\n", *reinterpret_cast<char*>(b));   // Probably crashes
```

- Object pointer **does not point to an actual memory location** (*fake pointer*), but encodes various information that is required for address translation.
- The main job of the data layout DSL is **address translation**.
  - Implemented entirely in C++.
  - Template metaprogramming: `Field<...>` classes are *proxy types*.
  - Operator overloading: `Field<...>` references (*lvalues*) can be **implicitly converted** to base type references.

Structure/Components of a Fake Pointer



- Object pointers do not point to memory addresses. Instead, we encode all information that is required for **address computation/translation**.
- Implemented with operator overloading, template metaprogramming, macros.

- Fields are defined with **proxy types**.
- Field address computation depends on the **runtime type** of an object. (Because the runtime type determines the object capacity of a block. The runtime type is not statically known.)

Address Computation Overhead: Hand-written SOA

```
struct SoaStruct {
  float pos_x[kNumObjects];
  float pos_y[kNumObjects];
  float vel_x[kNumObjects];
  float vel_y[kNumObjects];
  float force_x[kNumObjects];
  float force_y[kNumObjects];
  float mass[kNumObjects];
};

__global__ void codegen_test(SoaStruct* soa, int id) {
  soa->pos_y[id] = 1.2345f;
}
```

```
MOV R1, c[0x0][0x20];
MOV R2, c[0x0][0x148];
ISCADD R0.CC, R2.reuse,
c[0x0][0x140], 0x2;
SHR R2, R2, 0x1e;
IADD.X R2, R2, c[0x0][0x144];
IADD32I R4.CC, R0, 0x4000000;
MOV32I R0, 0x3f9e0419;
IADD.X R3, RZ, R2;
MOV R2, R4;
STG.E [R2], R0;
```

Address Computation Overhead: DynaSOAr

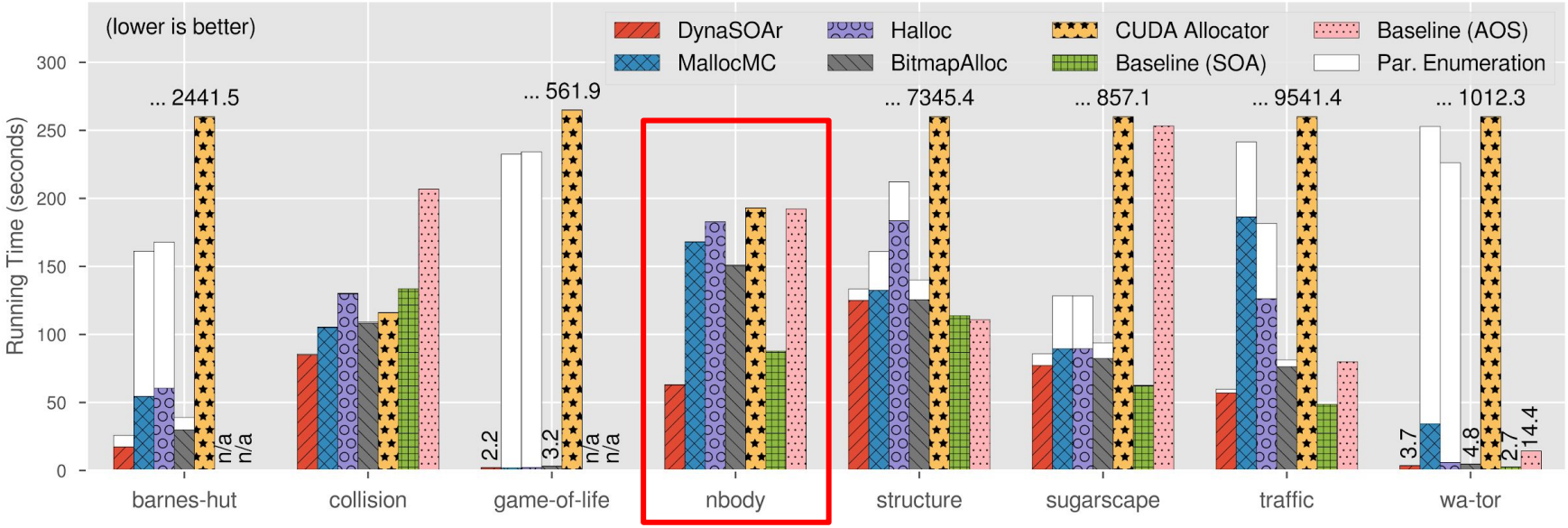
LISTING 5.2: Field address computation

```
// Impl. conv. operator: E.g., convert Field<NodeBase, 2> to float& in Figure 5.4.
// BaseType: N-th predeclared type in B (within declare_field_types).
template<typename B, int N>
Field<B, N>::operator BaseType&() {
  int offset = ...; // Computed with templ. metaprog. offset_{B::fieldname} in Figure 5.4.
  auto obj_ptr = reinterpret_cast<uint64_t>(this) - N;
  // Bits 0-49 and clear 6 least significant bits.
  auto* block_address = reinterpret_cast<char*>(obj_ptr & 0x3FFFFFFFFFFC0);
  int obj_slot_id = obj_ptr & 0x3F; // Bits 0-5
  int block_capacity = (obj_ptr & 0xFC000000000000) >> 50; // Bits 50-55
  auto* soa_array = reinterpret_cast<BaseType*>(
      block_address + field_offset * block_capacity);
  return soa_array[obj_slot_id];
}
```

```
__global__ void codegen_test(Body* b) {
  b->pos_y_ = 1.2345f;
}
```

```
MOV R1, c[0x0][0x20];
MOV R5, c[0x0][0x140];
MOV R2, c[0x0][0x144];
SHF.R.U64 R0, R5, 0x18, R2;
LOP32I.AND R0, R0, 0xff000000;
SHR R0, R0, 0x16;
LOP32I.AND R3, R5, 0xffffffc0;
IADD32I R0, R0, 0x40;
LOP32I.AND R2, R2, 0xffff;
LOP32I.AND R5, R5, 0x3f;
IADD R3.CC, R0.reuse, R3;
SHR R0, R0, 0x1f;
IADD.X R0, R0, R2;
LEA R3.CC, R5.reuse, R3, 0x2;
LEA.HI.X R0, R5, R0, RZ, 0x2;
LEA R2.CC, R3.reuse, RZ;
LEA.HI.X P0, R3, R3, RZ, R0;
MOV32I R0, 0x3f9e0419;
ST.E [R2], R0, P0;
```

Measuring the Overhead of DynaSOAr's DSL



Address Computation Overhead: Ikra-Cpp

```
Body* Body::get(int id) {
  return reinterpret_cast<Body*>(id)
}

template<typename T, int Index, int Offset, class Owner>
T* Field_<T, Index, Offset, Owner>::data_ptr() {
  Owner* obj = reinterpret_cast<Owner*>(this);
  return reinterpret_cast<T*>(Owner::storage
      + Offset * Owner::kMaxInst - sizeof(T)
      + sizeof(T) * reinterpret_cast<uintptr_t>(obj));
}
```

```
__global__ void codegen_test(Body* b) {
  b->pos_y_ = 1.2345f;
}
```

```
MOV R1, c[0x0][0x20];
MOV32I R2, 0x0;
MOV R0, c[0x0][0x140];
MOV32I R3, 0x0;
MOV R5, c[0x0][0x144];
LEA R2.CC, R0.reuse, R2, 0x2;
LEA.HI.X R3, R0, R3, R5, 0x2;
MOV32I R0, 0x3f9e0419;
STG.E [R2+0x138b0], R0;
```

Very similar to hand-written SOA assembly. Practically no overhead. For Ikra-Cpp CPU mode: Identical assembly code.

Measuring the Overhead of Ikra-Cpp's DSL

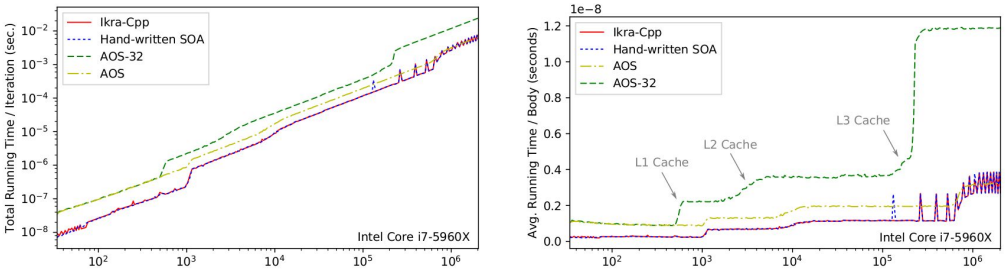


FIGURE 4.6: Host mode running time

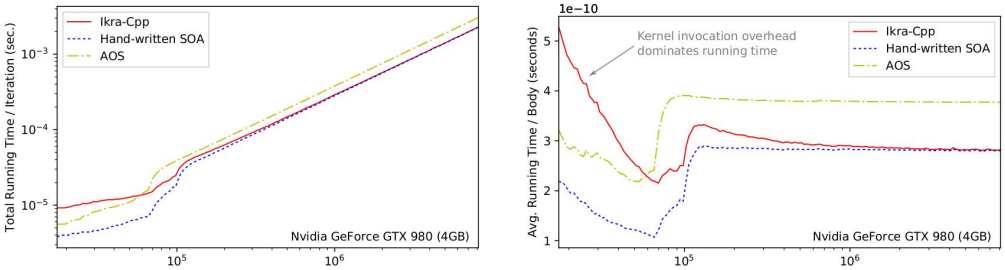


FIGURE 4.7: Device mode running time

- **No overhead** of Ikra-Cpp over hand-written SOA
- *CPU:* Same assembly code generated.
- *GPU:* Slightly different assembly code, but almost same performance.

Conclusion

- Minimal overhead due to data layout DSL.
  - *Ikra-Cpp:* **No overhead** at all → Compiler can generate efficient code.
    (But **problems with vectorization** in mode CPU!)
  - *DynaSOAr:* Some overhead due to more complex address translation.
    - Overhead is much lower than the benefit of SOA.
    - N-Body is getting a bit faster due to cache associativity issues.
- Address translation is usually done at the compiler/OS/hardware level, but we do it in C++ due for engineering reasons.

“Discuss Integration with Mainstream Parallel Languages such as **OpenMP**”

Run-Time vs. Compile-Time Coalescing

- Vectorization on x86: SSE (Streaming SIMD Extensions)
- Generate vector assembly instructions: E.g.: `movdqa`

Source: https://stackoverflow.com/questions/56966466/memory-coalescing-vs-vectorized-memory-access

C++ Code:

```
alignas(128) int r[1024];
alignas(128) int a[1024];
alignas(128) int b[1024];

#pragma omp parallel for simd
for (int i = 0; i < 1024; ++i) {
    r[i] = a[i] + b[i];
}
```

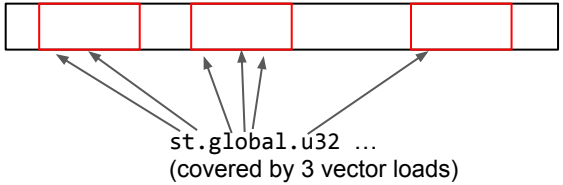
x86 Assembly:

```
movdqa 0x6020f0(%rax),%xmm0
```

Compile-time coalescing

CUDA Code:

```
__device__ int r[1024];
__device__ int a[1024];
__device__ int b[1024];

__global__ void kernel() {
    r[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```



PTX Assembly:

```
st.global.u32 [%rd7], %r4;
```

Run-time coalescing

No need to analyze access pattern at compile time!

Run-Time vs. Compile-Time Coalescing

東京工業大学
Tokyo Institute of Technology

Memory Coalescing vs. Vectorized Memory Access

Asked 18 days ago Active 18 days ago Viewed 127 times

2

I am trying to understand the relationship between **memory coalescing** on NVIDIA GPUs/CUDA and **vectorized memory access** on x86-SSE/C++.

It is my understanding that:

- Memory coalescing is a **run-time** optimization of the memory controller (implemented in hardware). How many memory transactions are required to fulfill the load/store of a warp is determined at run-time. A load/store instruction of a warp may be issued repeatedly unless there is perfect coalescing.
- Memory vectorization is a **compile-time** optimization. The number of memory transactions for a vectorized load/store is fixed. Each vector load/store instruction is issued exactly once.
- Coalescable GPU load/store instructions are more expressive than SSE vector load/store instructions. E.g., a `st.global.s32` PTX instruction may store into 32 arbitrary memory locations (warp size 32), whereas a `movdqa` SSE instruction can only store into a consecutive block of memory.
- Memory coalescing in CUDA seems to guarantee efficient *vectorized* memory access (when accesses are coalescable), whereas on x86-SSE, we have to hope that the compiler actually vectorizes the code (it may fail to do so) or vectorize code manually with SSE intrinsics, which is more difficult for programmers.

Is this correct? Did I miss an important aspect (thread masking, maybe)?

Now, why do GPUs have run-time coalescing? This probably requires extra circuits in hardware. What are the main benefits over compile-time coalescing as in CPUs? Are there applications/memory access patterns that are harder to implement on CPUs because of missing run-time coalescing?

cuda gpu cpu-architecture simd coalescing

share edit delete flag

edited Jul 10 at 8:37
Peter Cordes
151k • 21 • 239 • 385

asked Jul 10 at 8:26
Matthias Springer
20 • 4

I asked this question on StackOverflow and it sparked an interesting discussion…

https://stackoverflow.com/questions/56966466/memory-coalescing-vs-vectorized-memory-access

1 the addressing flexibility afforded by memory coalescing (as compared to your `movdqa` example) allows the CUDA programmer to write arbitrary thread code and expect functionally correct results. There is presumably some value to this. The programmer is allowed to do inefficient things for ease of programming, but has a roadmap to maximum performance/efficiency of use of the memory subsystem. Giving the programmer both options is considered to be of value. – Robert Crovella Jul 10 at 15:00

OpenMP SIMD Support

- SIMD-parallel `for` loops since OpenMP 4.0

```
float r[N];   float a[N];   float b[N];

void example() {
    #pragma omp parallel for simd
    for (int i = 0; i < N; ++i) {
        r[i] = a[i] + b[i];
    }
}
```

Note: Compilers with auto-vectorization do almost the same thing. (Apart from `__restict`.)

`for` loop must be in **canonical form**!

transform

Can only load consecutive (packed) floats. Otherwise, must use different instruction. Compiler must understand the memory access pattern!

```
#pragma omp parallel for
for (int i = 0; i < N; i += 8) {
  __m256 vec_a = _mm256_load_ps(&a[i]);
  __m256 vec_b = _mm256_load_ps(&b[i]);
  __m256 vec_r = _mm265_add_ps(vec_a, vec_b);
  _mm256_store_ps(&r[i], vec_r);
}
```

OpenMP SIMD Support

- SIMD-parallel for loops since OpenMP 4.0

```
float r[N];   float a[N];   float b[N];

void example() {
    #pragma omp parallel for simd
    for (int i = 0; i < N; ++i) {
        r[i] = func(a[i], b[i]);
    }
}

#pragma omp declare simd
float func(float p1, float p2) {
    return p1 + p2;
}
```

Functions OK!

OpenMP SIMD Support

- SIMD-parallel `for` loops since OpenMP 4.0

```
float r[N];   float a[N];   float b[N];

void example() {
    #pragma omp parallel for simd
    for (int i = 0; i < N; ++i) {
        r[1 + i - 1] = a[2*i - i] + b[atoi(sqrt(i*i))];
    }
}
```

OpenMP compiler must be able to **find SIMD-suitable access pattern**!

Note: Pretty sure, it will fail here…
(Yes, this technically, this is not the same as `i`.)

LISTING 5.2: Field address computation

```
// Impl. conv. operator: E.g., convert Field<NodeBase, 2> to float& in Figure 5.4.
// BaseType: N-th predeclared type in B (within declare_field_types).
template<typename B, int N>
Field<B, N>::operator BaseType&() {
  int offset = ...; // Computed with templ. metaprog. offset_{B::fieldname} in Figure 5.4.
  auto obj_ptr = reinterpret_cast<uint64_t>(this) - N;
  // Bits 0-49 and clear 6 least significant bits.
  auto* block_address = reinterpret_cast<char*>(obj_ptr & 0x3FFFFFFFFFFC0);
  int obj_slot_id = obj_ptr & 0x3F; // Bits 0-5
  int block_capacity = (obj_ptr & 0xFC000000000000) >> 50; // Bits 50-55
  auto* soa_array = reinterpret_cast<BaseType*>(
      block_address + field_offset * block_capacity);
  return soa_array[obj_slot_id];
}
```

… or here

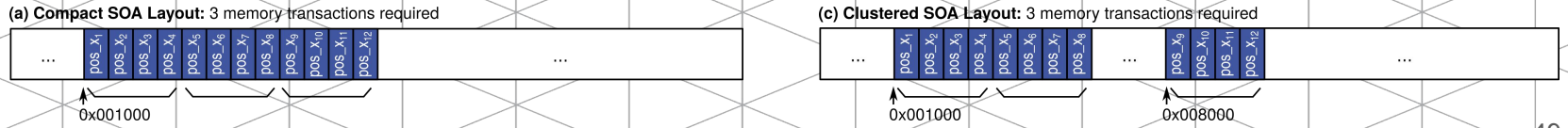
DynaSOAr `parallel_do` in OpenMP

- `parallel_do<T, &T::func>` is a parallel `for` loop, but it is **not in canonical form**! It is more like a parallel iterator.

```
int main() {
    auto* h_allocator = new AllocatorHandle<AllocatorT>();

    #pragma omp parallel for simd
    for (Body& b : h_allocator->get_objects<Body>()) {
        b.update(/*dt=*/ 0.5f);
    }
}
```

```
h_allocator->parallel_do<Body, &Body::update>(0.5f);
```

- *Problem:* DynaSOAr object space is **not an array**.



Conclusion

- Could DynaSOAr (`parallel_do`) be implemented in OpenMP? **Yes**
- But depends on the compiler to detect SIMD-suitable access patterns. In practice, **it will not work well!** (This is a general problem of SIMD.)

```
template<typename T, void (T::*func)()>
void parallel_do() {
    #pragma omp parallel for
    for (int i = 0; i < h_allocator->get_num_blocks<T>(); ++i) {
        Block<T>* block = h_allocator->get_ith_allocated_block<T>(i);

        #pragma omp parallel for simd
        for (int j = 0; j < 64; ++j) {
            (block->get_ith_object(j)->*func)();
        }
    }
}
```

thread parallelism

SIMD parallelism

OpenMP is **unlikely to generate efficient vector code** (due to SOA data layout DSL)