The OHMM framework

Juliana Franco
Martin Hagelin
Tobias Wrigstad
Sophia Drossopoulou

Imperial College London

UPPSALA UNIVERSITET

Low-Level Memory Optimisations at the High-Level with Ownership-like Annotations

Do you want fast programs?

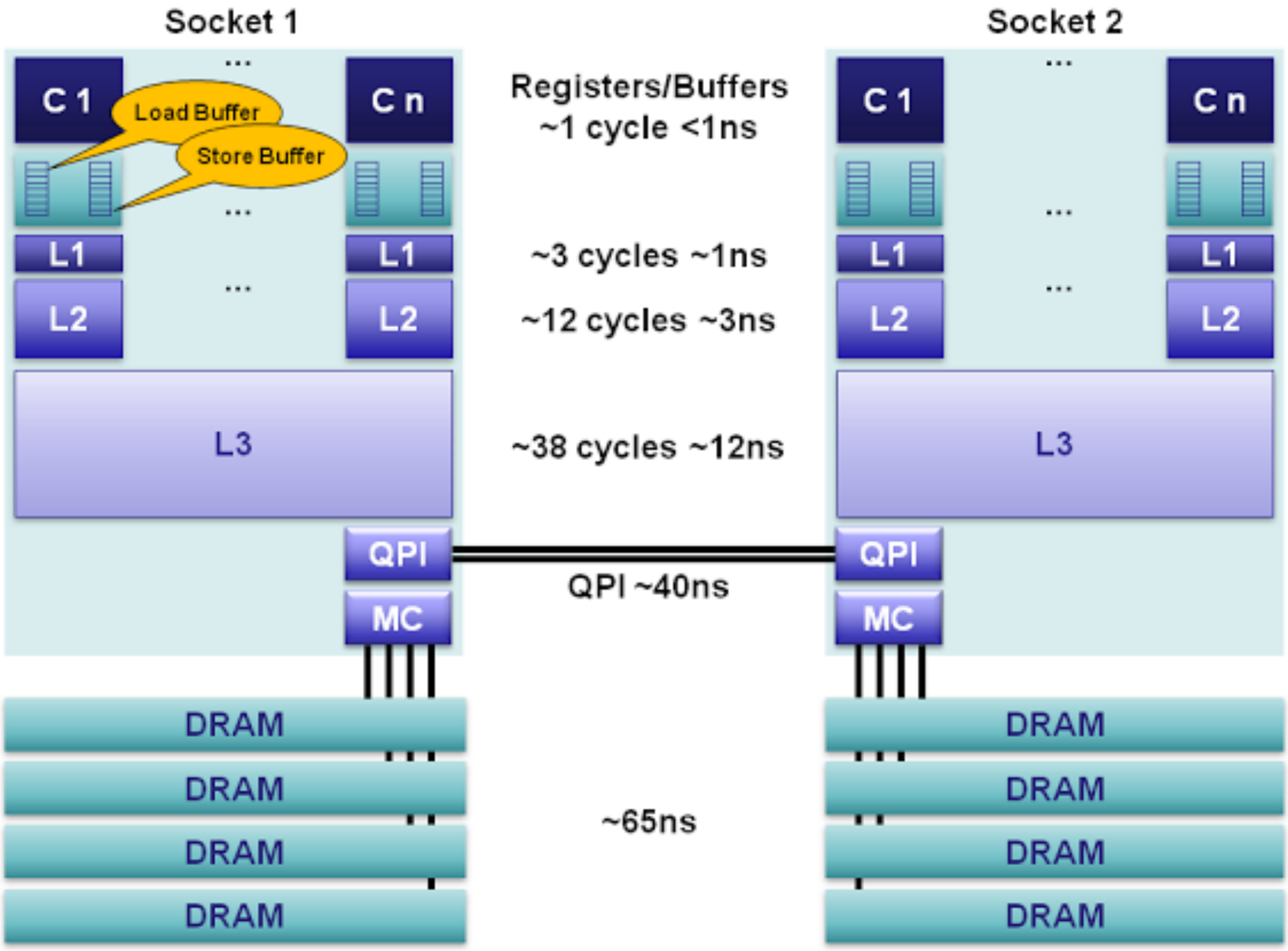
- More cores? More threads? Write better parallel and concurrent code?

- Data layout in memory can have a great impact in your program's performance!
  - Reduce cache misses
  - or help the prefetcher

Example: array[N] of arrays[N] vs array[N*N]

```
1,325 * 10⁶ cache-misses        833 * 10⁶ cache-misses
      28.04 seconds                   20.49 seconds
```

A little bit of context on hardware



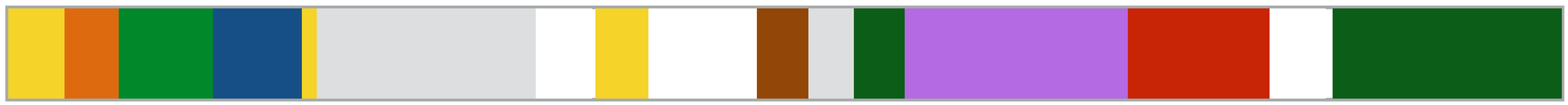
http://mechanical-sympathy.blogspot.co.uk/2013/02/cpu-cache-flushing-fallacy.html

A little bit of context on hardware

Core: `read` **`purple`** `data`

Cache:

Memory:



A little bit of context on hardware

Core: `read purple data`

Cache miss 65ns

Cache:

Memory:

A little bit of context on hardware

Core: `read purple`
`fetch purple data from memory`

Cache miss 65ns

Cache:

Memory:

A little bit of context on hardware

Core:

```
read purple
 fetch purple data from memory
read purple again
```

Cache miss	65ns
Cache hit	3ns

Cache:

Memory:

A little bit of context on hardware

Core:

```
read purple
 fetch purple data from memory
read purple again
read red data
```

Cache miss	65ns
Cache hit	3ns
Cache hit	3ns

Cache:

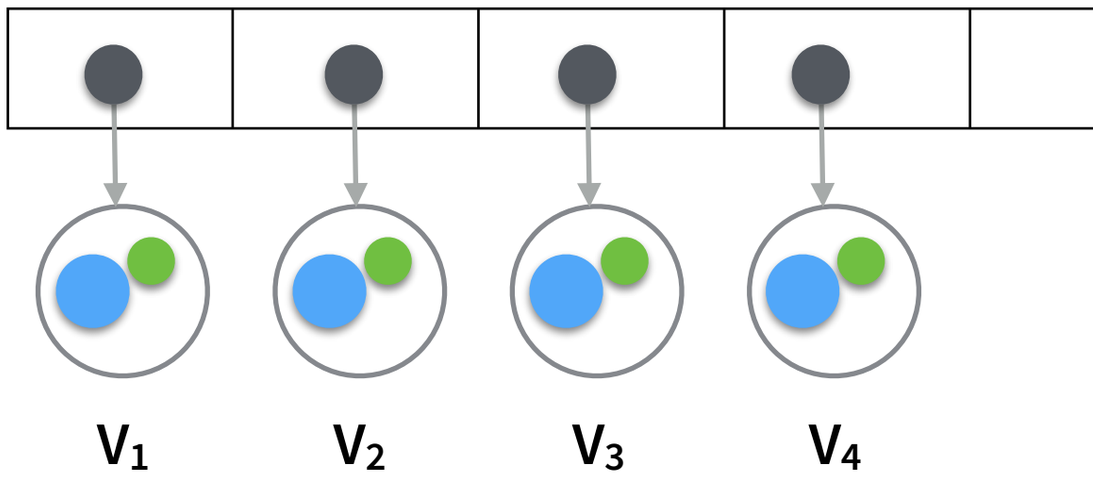
Memory:

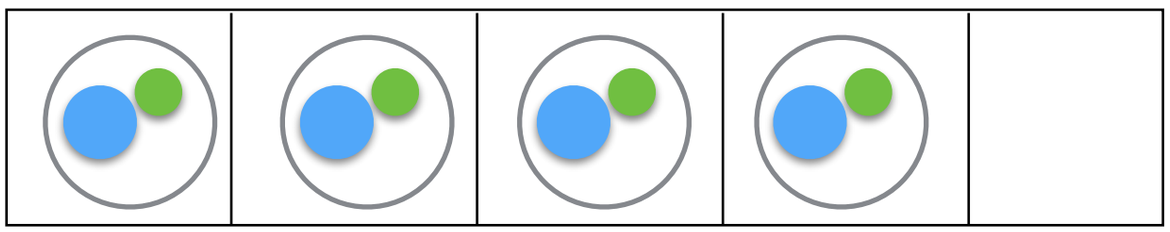
Existing techniques

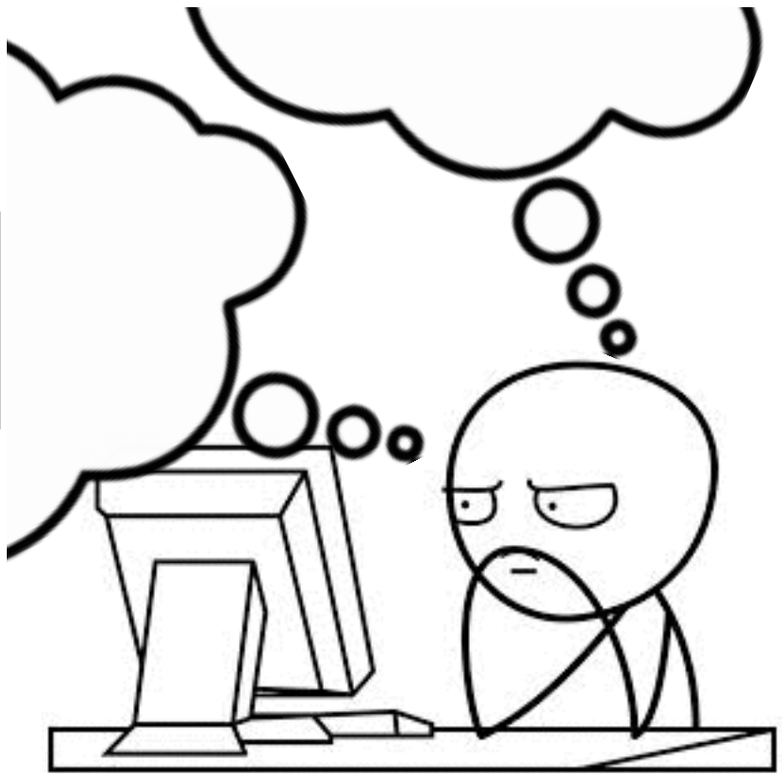
```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

def popularVideos(pivot: int): void
// iterates over all videos
```





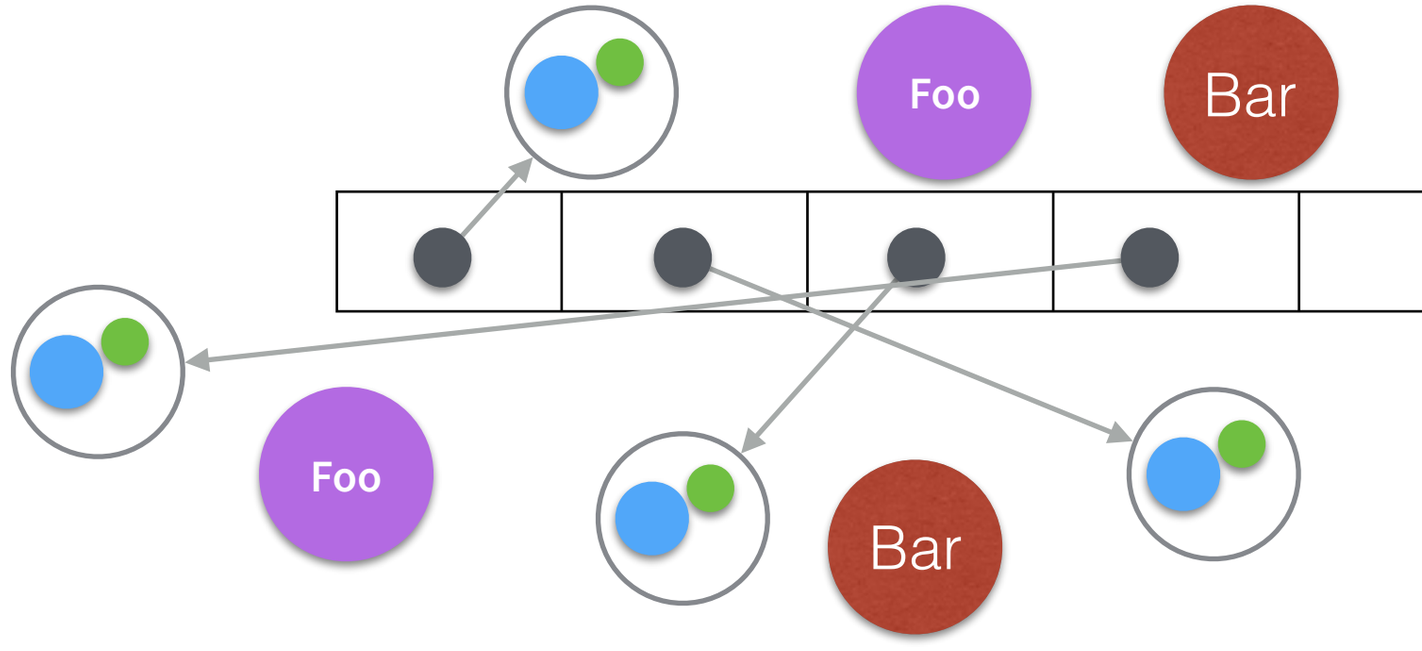


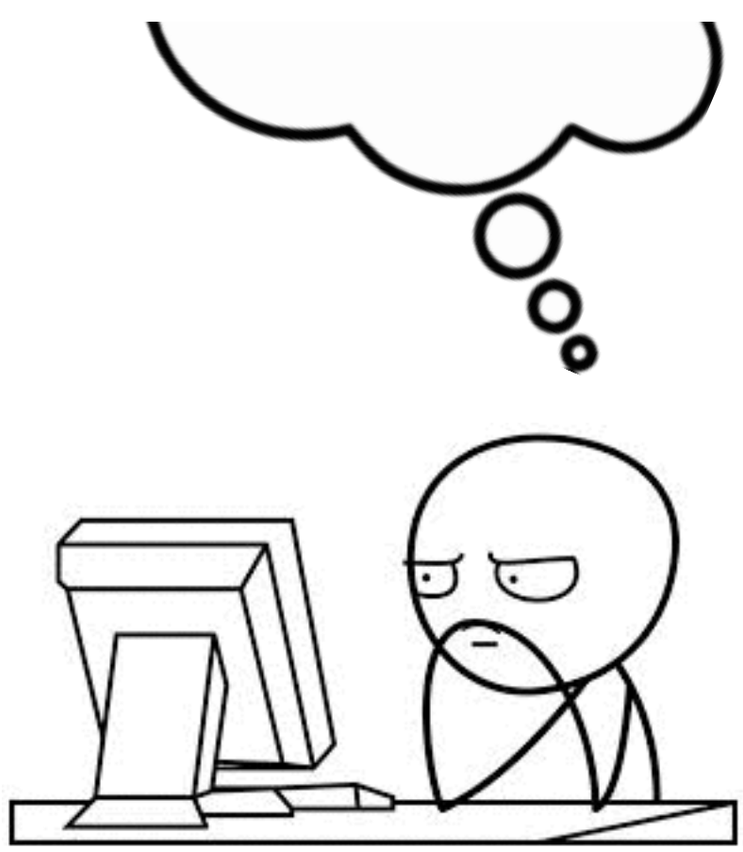
Existing techniques

```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

def popularVideos(pivot: int): void
// iterates over all videos
```



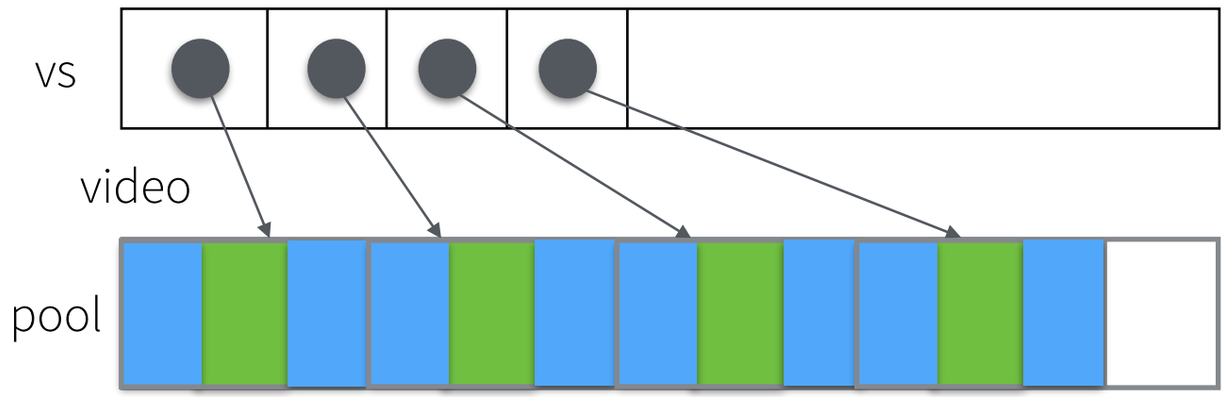


Existing techniques

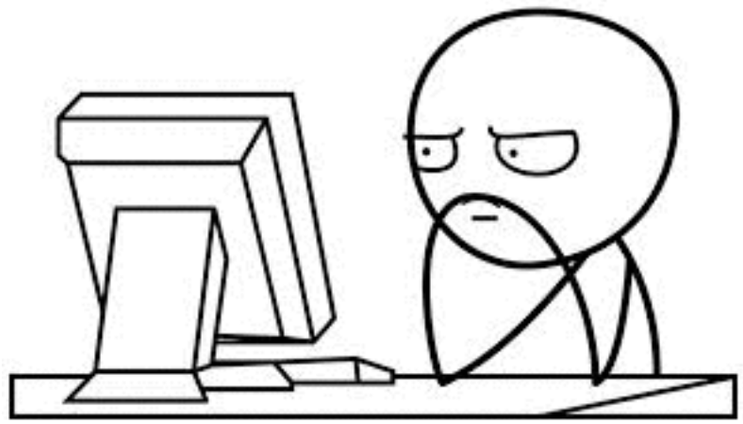
```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

def popularVideos(pivot: int): void
    // iterates over all videos
```



Object Pooling

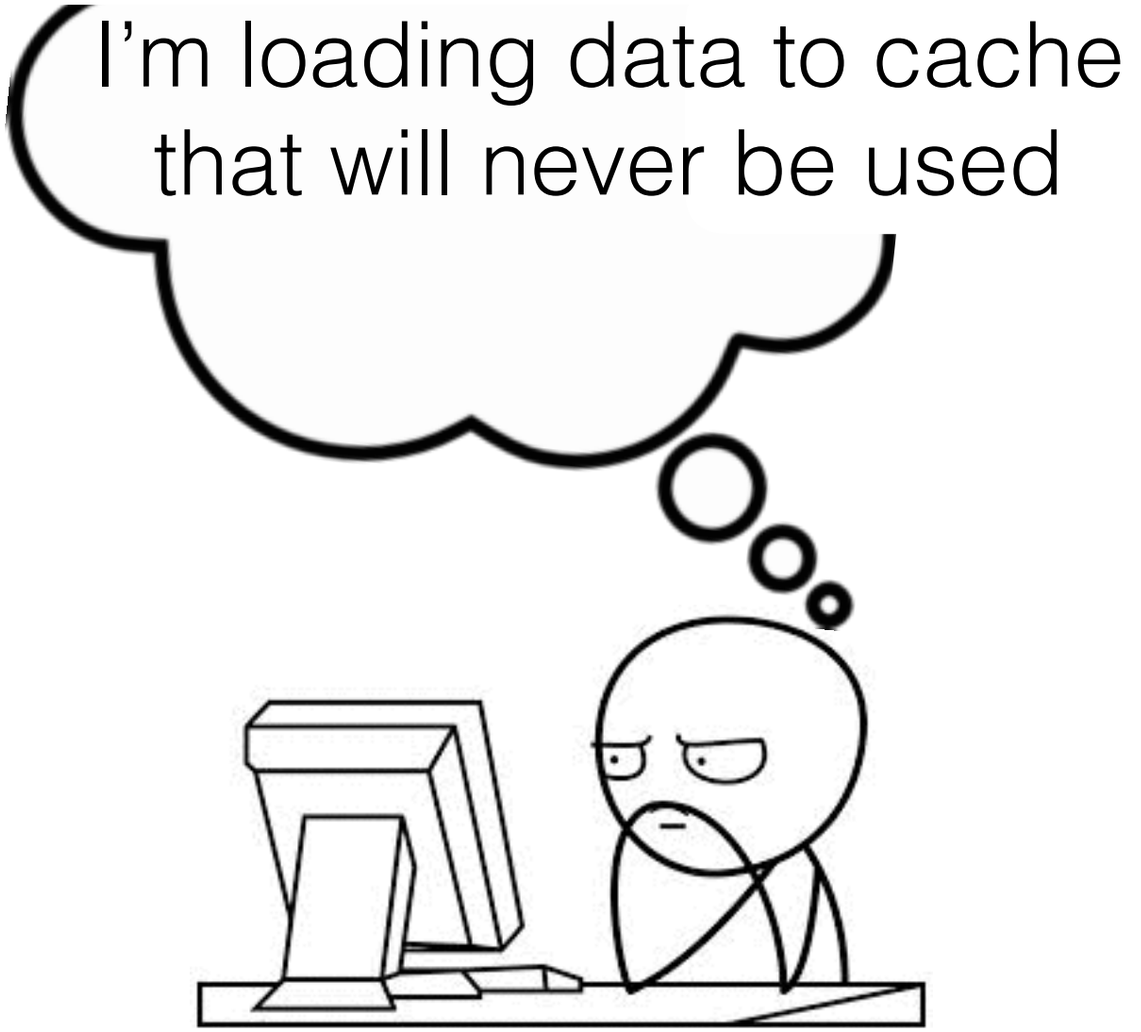


Existing techniques

```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

def popularVideos(pivot: int): void
    foreach v in this.vs do
        if v.views > pivot then
            print(v.id, v.views, v.likes)
```

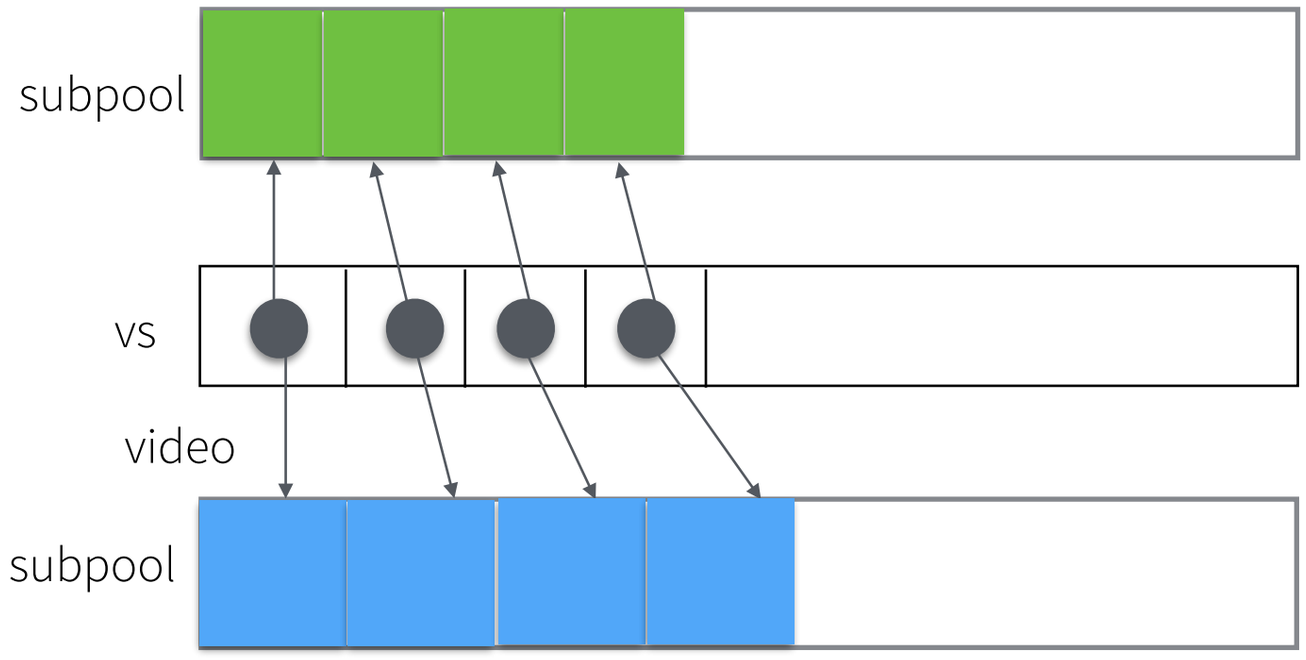


Existing techniques

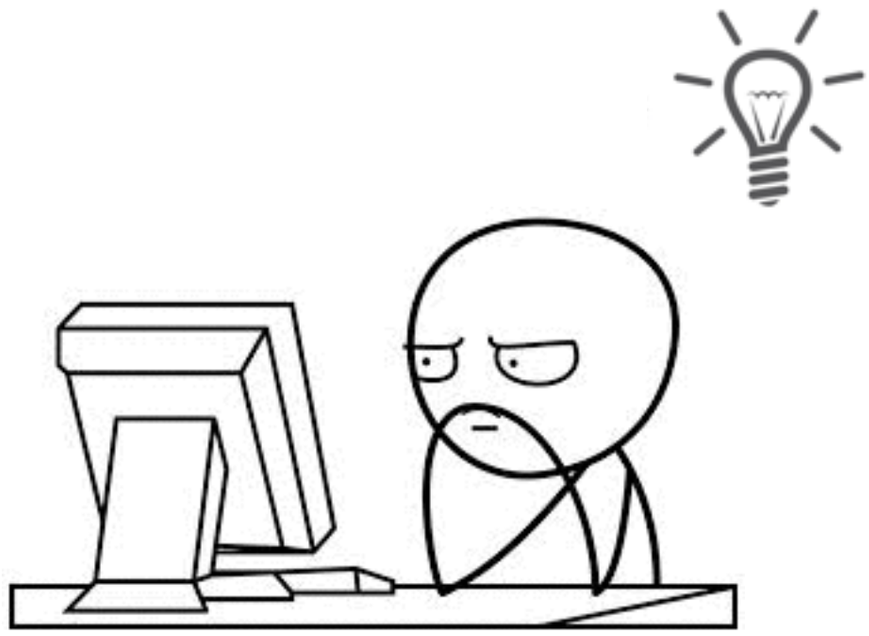
```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

def popularVideos(pivot: int): void
    foreach v in this.vs do
        if v.views > pivot then
            print(v.id, v.views, v.likes)
```



Object Splitting



- It is known that these techniques can improve performance
- And programmers use it a lot
  - Ex: array of structs vs struct or arrays
- However:
  - they are too low level
  - the concept of *struct* or *object* is lost
  - the code becomes difficult to write and to modify

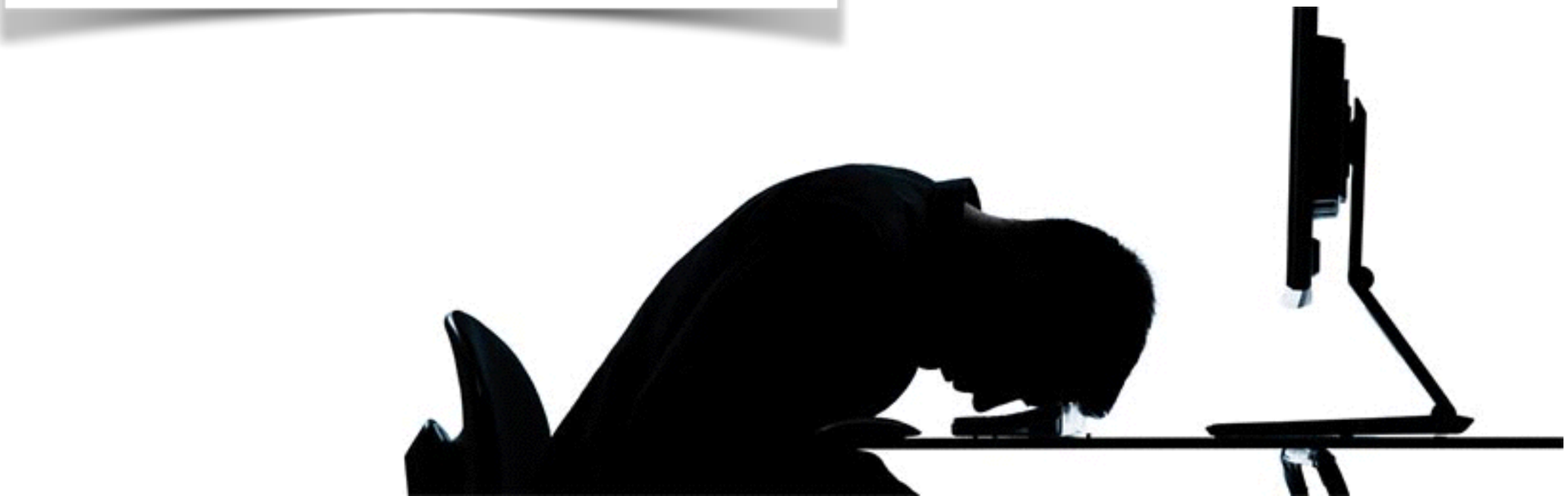
```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

    def popularVideos(pivot: int): void
        foreach v in this.vs do
            if v.views > pivot then
                print(v.id, v.views, v.likes)
```

```
class VideoList
    ids: int[N]
    views: int[N]
    likes: int[N]

def popularVideos(pivot: int): void
    for (int i = 0; i < N; i++) do
        if this.views[i] > pivot then
            print(this.ids[i], this.views[i], this.likes[i])
```



```
class VideoList
    id_likes: (int, int)[N]
    views: int[N]

def popularVideos(pivot: int): void
    for (int i = 0; i < N; i++) do
        if this.views[i] > pivot then
            print(this.id_likes[i].fst, this.views[i], this.id_likes[i].snd)
```

Our solution

We want to provide a high-level way of specifying the data structures which does not affect the way they are used

Martin

This code for…

```
class Video
    id: int
    views: int
    likes: int

class VideoList
    vs: Array[Video]

    def popularVideos(pivot: int): void
        foreach v in this.vs do
            if v.views > pivot then
                print(v.id, v.views, v.likes)
```

```
class VideoList
    ids: int[N]
    views: int[N]
    likes: int[N]

def popularVideos(pivot: int): void
    for (int i = 0; i < N; i++) do
        if this.views[i] > pivot then
            print(this.ids[i], this.views[i], this.likes[i])
```

… this behaviour

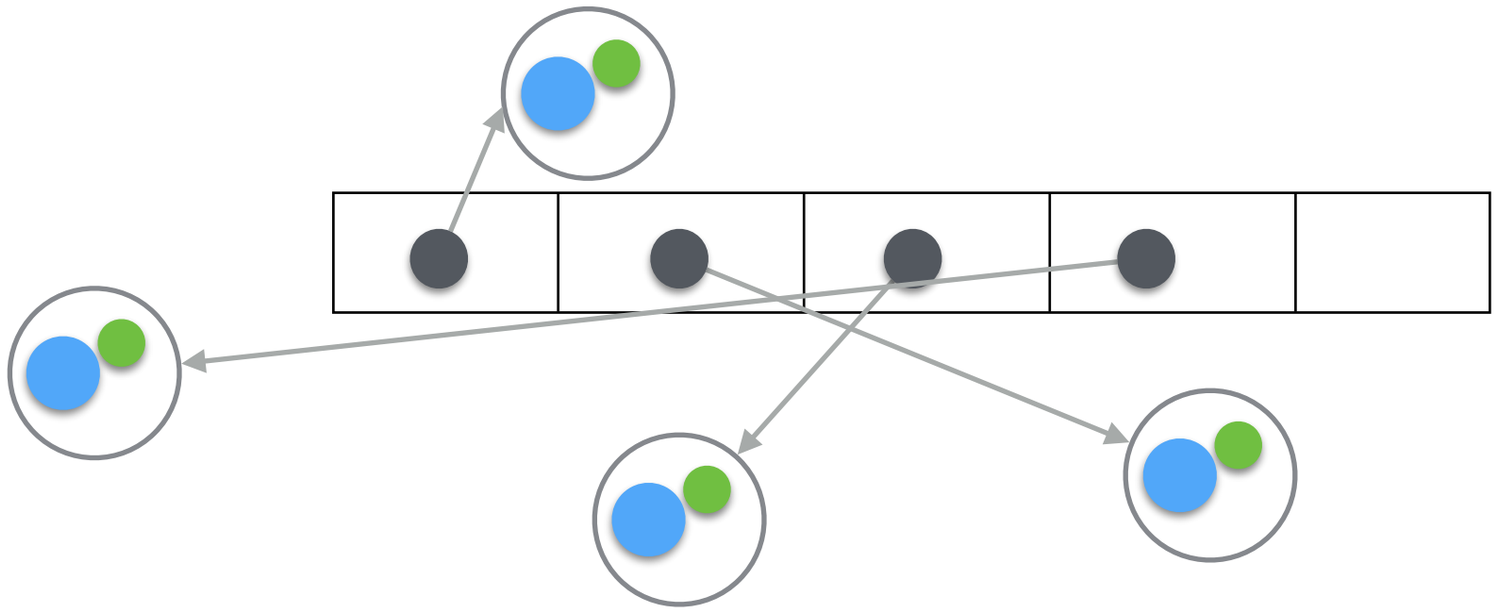
Layout annotations

```
class Video<o>
    id: int
    views: int
    likes: int

class VideoList<o, o'>
    vs: Array[Video<o'>]
```

Pool and Object Allocation

new VideoList<**none, none**>



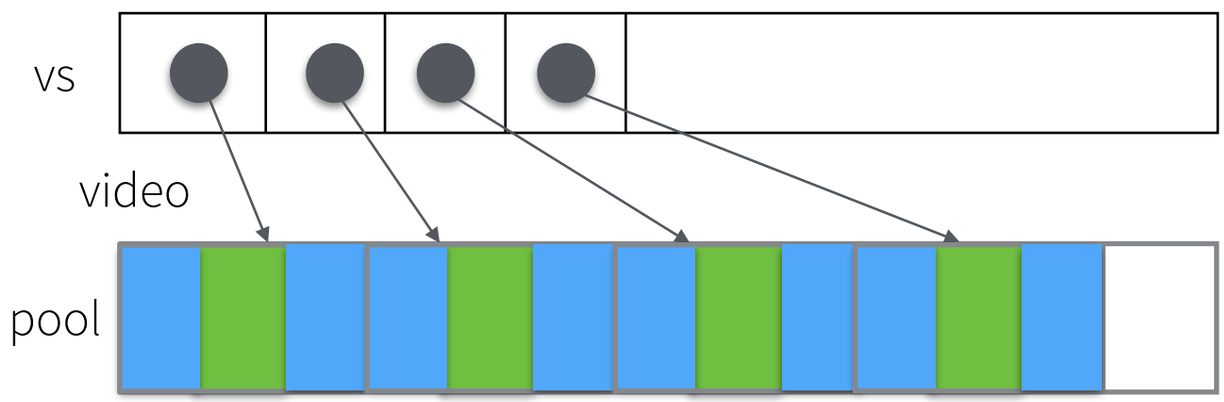
Layout annotations

```
class Video<o>
    id: int
    views: int
    likes: int

class VideoList<o, o'>
    vs: Array[Video<o'>]
```

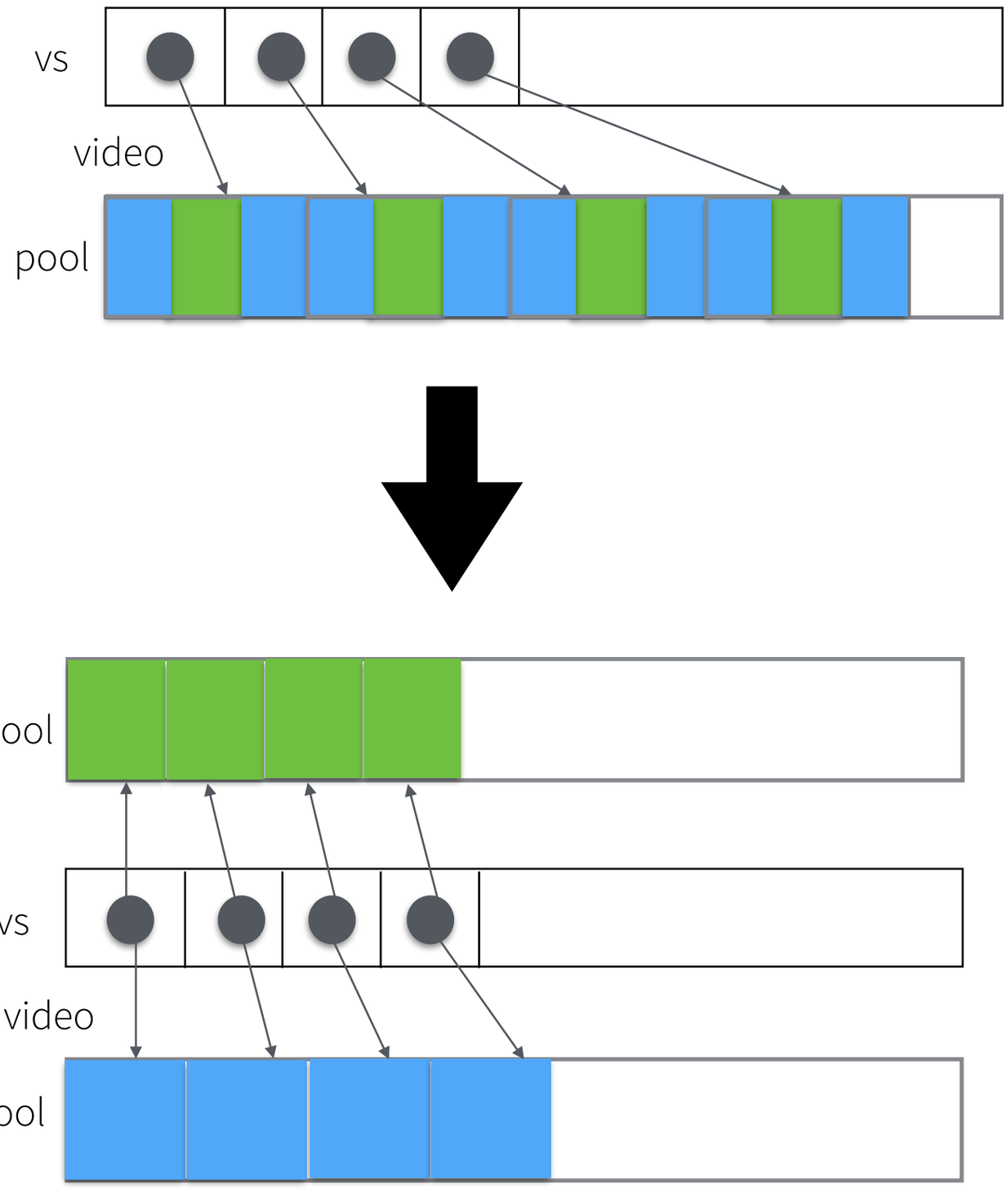
Pool and Object Allocation

```
Pool pool of Video in
    new VideoList<none, pool>
```



Clustering annotations

Pool pool **of** Video **in**
new VideoList<**none,** pool>



Pool pool **of** Video =
cluster {id, likes}
+ **cluster** {views}
in
new VideoList<**none,** pool>

How do we use this data structure?

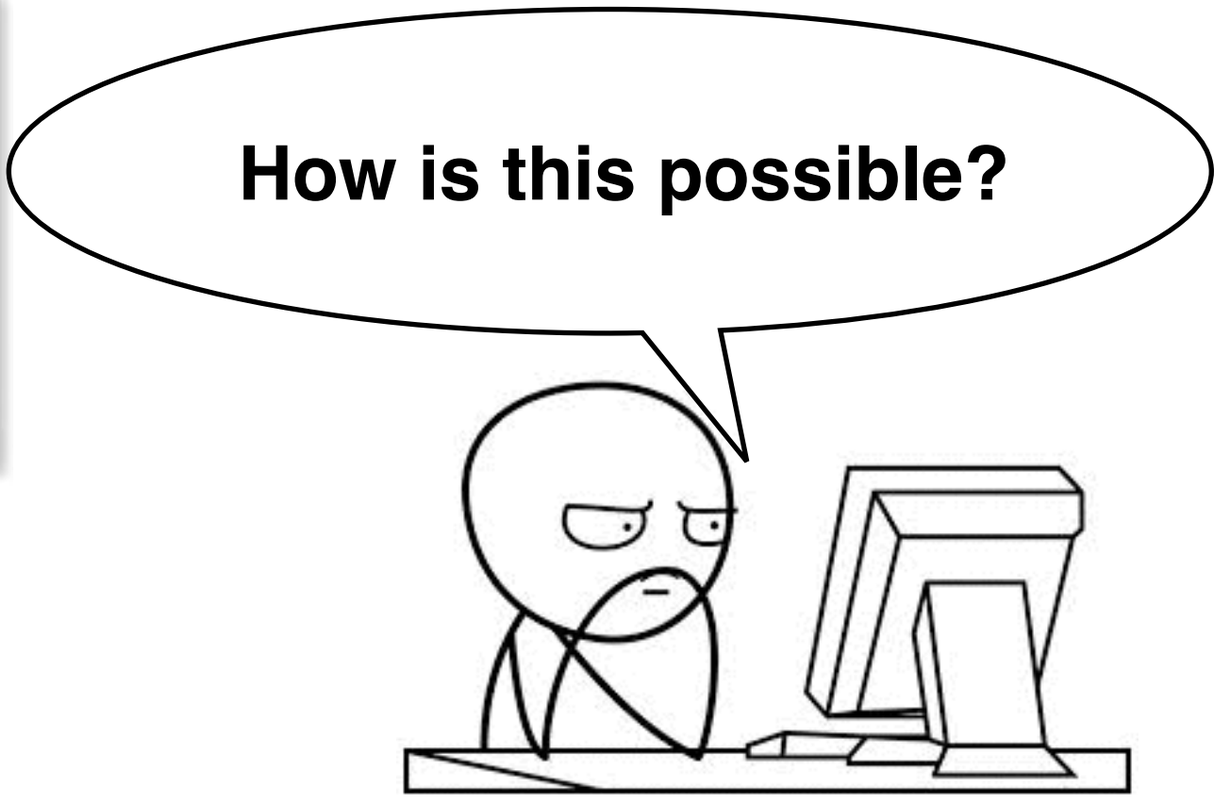
```
def popularVideos(pivot: int): void
    foreach v in this.vs do
        if v.views > pivot then
            print(v.id, v.views, v.likes)
```

```
let vl = new VideoList<none, none> in
vl.vs[45678].likes ++
print(vl.vs[45678].views)
```

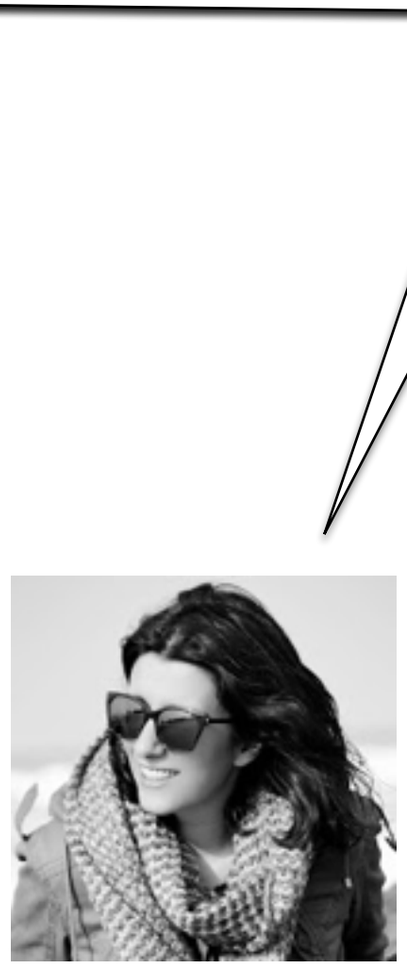
```
Pool pool of Video =
            cluster {id, likes, views}
let vl = new VideoList<none, pool> in
vl.vs[45678].likes ++
print(vl.vs[45678].views)
```

```
let vl = new VideoList<none, pool> in
vl.vs[45678].likes ++
```

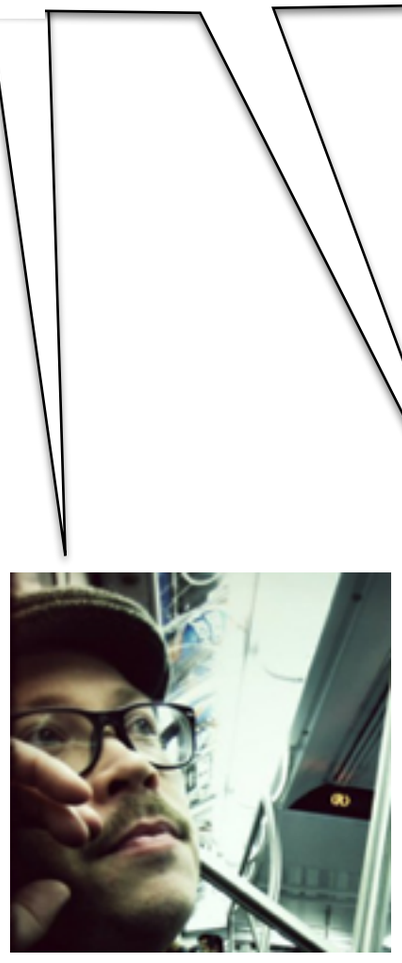
```
Pool pool of Video =
        cluster {id} + cluster {likes, views}
let vl = new VideoList<none, pool> in
vl.vs[45678].likes ++
print(vl.vs[45678].views)
```

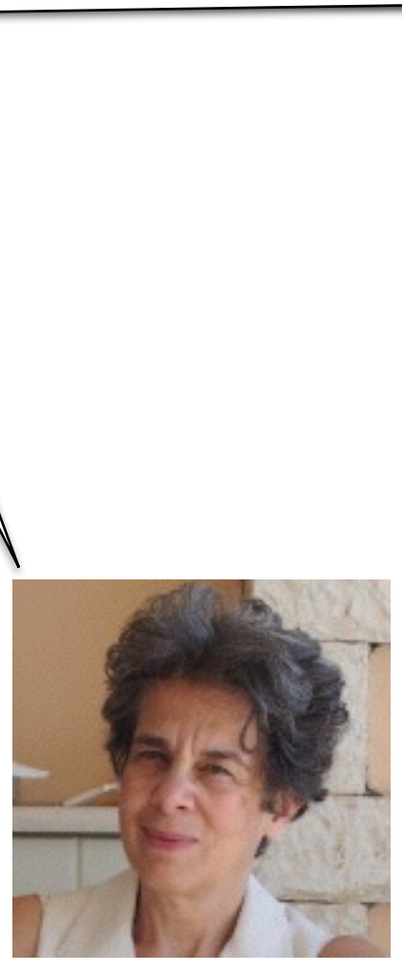


1. A low-level language that does all the hard work
2. A compiler that uses the annotations to compile HL code to equivalent LL code



Martin





A little bit on the low-level language

Instructions:

$$
\begin{aligned}
rhs ::= {} & fn(rs) \mid \mathsf{null} \\
& \mid \mathsf{pread}(r, j, k) \mid \mathsf{read}(r, f) \\
& \mid \mathsf{pwrite}(r, j, k, r') \mid \mathsf{write}(r, f, r') \\
& \mid \mathsf{pcreate}(C) \mid \mathsf{palloc}(r) \mid \mathsf{alloc}(C)
\end{aligned}
$$

$$\ell \in ObjAddr \qquad \wp \in PoolAddr$$

$$r \in Register \qquad fn \in FunctionId$$

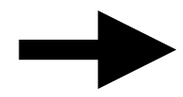
$$i, j, k, n \in \mathbb{N}$$

Example:

$r_1 := \mathbf{read}(r_0, \text{video});$
$r_2 := \mathbf{pread}(r_1, 1, 0);$
if $r_2 > r_x$ **then**
  $r_3 := \mathbf{pread}(r_1, 0, 0);$
  $r_4 := \mathbf{pread}(r_1, 1, 0);$
  $r_5 := \mathbf{pread}(r_1, 0, 1);$
  $r_6 := \text{print}(r_3, r_4, r_5);$
$r_0 = \mathbf{read}(r_0, \text{next})$

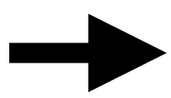
A little bit on the compiler

```
x = new Video<none>
y = x. likes
x.likes = y + 10
```



```
x = alloc(Video)
y = read(x, likes)
z = y + 10
write(x, likes, z)
```

```
Pool p1 of Video =
    cluster {id, likes} + cluster {views}
x = new Video<p1>
y = x. likes
x.likes = y + 10
```



```
p1 = pcreate(Video, [id, likes], [views])
x = palloc(p1)
y = pread(x, 0, 1)
z = y + 10
write(x, 0, 1, z)
```

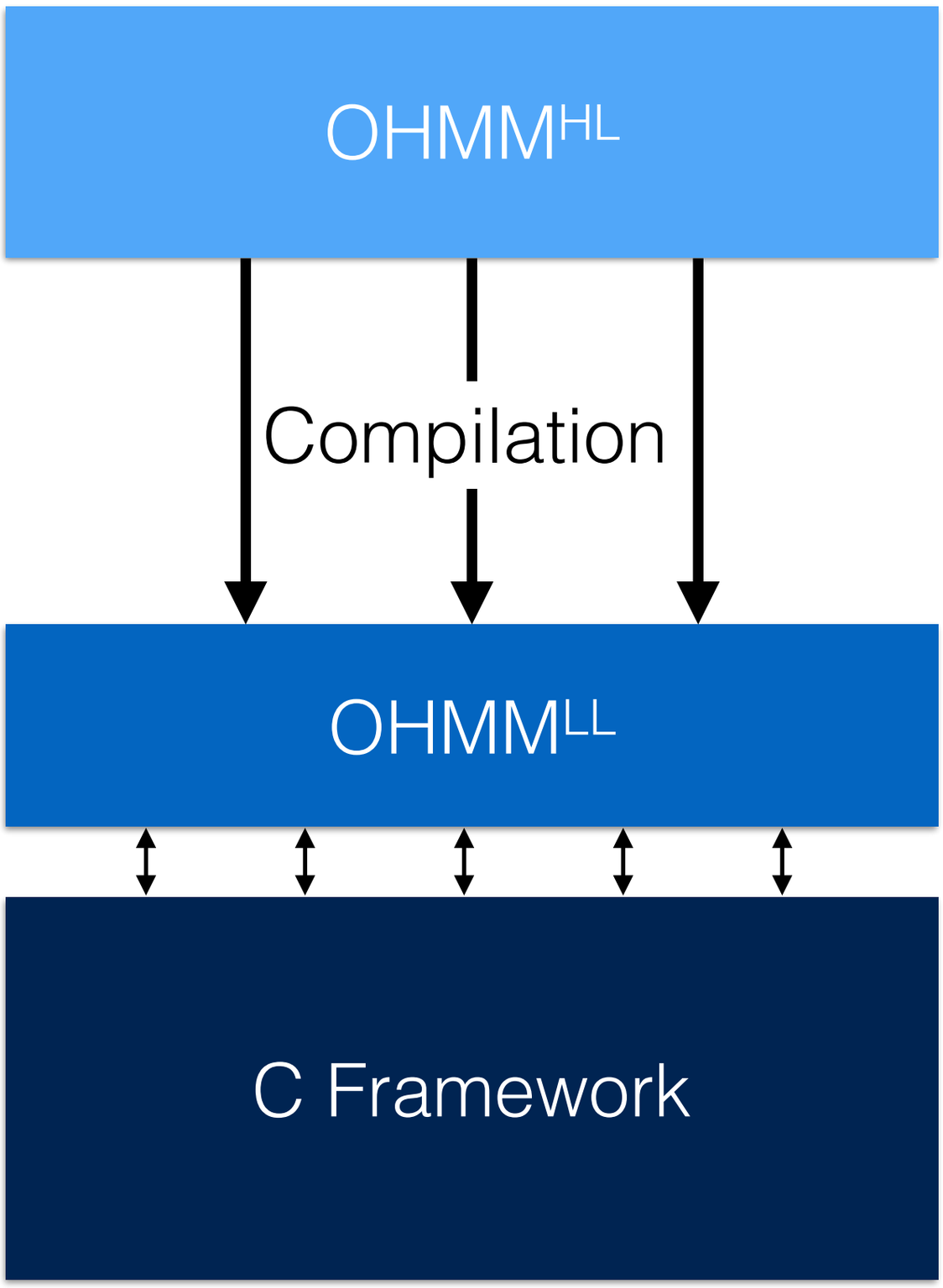
Contributions

- ***Separation*** of functional concerns from the layout concerns
  - At a higher-level: an ***object is still a single unit***, that is somewhere in memory.
  - Layout annotations describe how pools are organised but object access does not need to reflect that.
  - Therefore, the code ***easier to write and modify***, and also ***efficient***.
- But also much more:
  - The high-level language is ***type sound***, and given that we ***correctly*** compile it, we know that low-level program behaviour is equivalent to the high-level behaviour.

TO DO LIST

- [ ] Garbage Collection
- [ ] Sub-typing
- [ ] Value Semantics
- [ ] Iterators
- [ ] Concurrency and parallelism
- [ ] Benchmarks, benchmarks …

Conclusion



- OO sequential language
- Ownership-like annotations
- Splitting annotations

- Translation using the layout annotations

- Interface for the low-level framework with instructions to work with pools

- Pooling
- Splitting
- Pointer Compression
- Pool iterators
- Copying GC

Thank you!