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

Juliana Franco Tobias Wrigstad Sophia Drossopoulou

ABSTRACT

In modern architectures, due to the huge gap between CPU performance and memory bandwidth, an application's performance highly depends on the speed at which the system is able to deliver data to operate on. The placement of data in memory affects the number of cache misses, and thus the overall speed of the application. To address this, pooling and splitting are two techniques that allow to group or split data in memory, according to whether they are usually accessed together or separately. However, these are either low-level optimisations, or outside the control of the programmer.

We propose OHMM, an object-oriented programming language that uses ownership-like annotations to express high-level constraints on how objects should be placed in memory. These annotations will allow the runtime to allocate objects using pooling and splitting, and thus lead to efficient data accesses. In this short paper, we explain OHMM through an example, show how the objects will be laid out, and informally argue the benefits in terms of cache performance.

Keywords

Ownership types, memory allocation, data-layout

1. INTRODUCTION

Most modern programming languages are designed with the mindset that memory accesses are "for free". When the speed of a memory access rivalled that of the CPU, this abstraction was valid, however in reality, the gap between the speed of CPU's and main memory is steadily increasing to the point where computation is almost for free, and the real cost of execution, both in terms of speed and power consumption, is in accessing memory (c.f. Memory Wall problem [29]).

Cache memories, or hierarchies of cache memories have been part of modern architectures to hide this latency for long time, exploiting the temporal and spatial locality inherent in most programs. In this type of architectures, a core interacts as much as possible with the top most cache level in order to get the needed data, and when it fails to do so, it tries to access the data in the next cache level, and so on, until it finds that the required data is not in cache (the so called *cache miss*) and needs to be loaded from main memory. Accessing data from the different memory levels has different costs. Table 1 shows an example of costs. These numbers show how important it is to avoid cache misses. When a program execution results in too many cache misses, it may suffer poor performance and high energy cost. Moreover, if the program runs on a NUMA machine, these costs will be even more evident, as a cache miss in these machine can imply accessing memory from a different NUMA node, which costs more CPU cycles than accessing the local memory [22].

Sub sytem	Latency	Slowdown	Bandwidth
L1 Cache	4 cycles	x 1	365GB/s
L2 Cache	12 cycles	x 3	204GB/s
L3 Cache	21 cycles	x 5	119GB/s
DRAM	250 cycles	x 62	20GB/s

Table 1: Latency and Bandwitdh of the different memory levels, in an Intel i7-4600U CPU. Numbers taken from [16].

In order to reduce the number of *cache misses*, the programmer needs to understand "what goes into cache" when data is loaded from memory. This is at odds with mainstream programming abstractions[1]. Ultimately, memory is an array of bytes, and unless the high-level data is carefully mapped to this array of bytes, there is no control over what will cause cache misses.

Pooling and *splitting* are two existing techniques to tackle this problem, when dealing with large data-structures: Pooling means that objects are created in separate memory pools depending on their type or time of allocation. The rationale behind this is that objects that are frequently used together should be placed together for better cache utilisation. There is already substantial work on pooling, most of it for unmanaged languages, such as C or C++ [6, 20, 21, 13, 25, 30, 26, 19]. Object splitting splits composite objects up into different parts that ideally are not used together often. Splitting can have a significant performance impact as it allows to bring into cache more useful data (parts of objects that are not needed are not fetched from memory). Franz and Kistler were among the first to explore the subject of automatic object splitting [15]. Several researchers have

[1]In Java, for instance, the size of objects is generally not even known by programmers.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

combined both pooling and splitting in recent works [13, 25, 7, 26, 27]. In applications where performance is critical, programmers manually transform an array of structs into a structure of arrays in order to obtain similar behaviour to pooling and splitting. However, this approach makes the code more complex, error-prone and not suitable for object-oriented programming, as the reasoning about an object, or struct, as a single unit is not longer valid.

Even though there has been a great amount of research on pool allocation and object splitting, to the best of our knowledge, there has been no work on designing a front end that allows the programmer to express object pooling and splitting.

In this paper we give an outline of our ideas on OHMM[2], an object-oriented programming language that relies on a static type system, specifically on a variant of ownership types, to control how data is placed in memory. This language allows the programmer to write modular code (in the sense that the same class declarations can be used for different layouts), high-level and type safe object oriented code, while benefiting from the low-level advantages of pooling and splitting.

Paper structure. Section 2 demonstrates the problem using an example written in OO style; Section 3 describes how to solve the problem, by adding annotations to the same code; Section 4 briefly explains how OHMM will take advantage of the garbage collector; Section 5 discusses related work and Section 6 finishes the paper with conclusions and future work.

2. DELVING INTO THE PROBLEM

In this section we demonstrate how mainstream object-oriented languages lack the means to express data placement, and why they suffer from bad cache utilisation, using a running example. We consider the following VideoList that is a typical list of nodes linked by a next field. Each object of type Node points to a further object of type Video, which contains three fields of type **int**: an identifier (id), the number of times the video has been played (views) and the number of likes (likes) the video has gotten.

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

class Node
  video: Video
  next: Node

class VideoList
  head: Node
  def popularVideos(pivot: int): void
    let cur = this.head in
    while (cur != null) {
      let v = cur.video in
      if cur.views > pivot then
        print(v.id + ": " + v.views + ", " + v.likes);
      cur = cur.next
    }
```

Note that the colour of the fields in the code corresponds to their colours in all the diagrams of the paper.

[2]stands for Optimised Heaps for Memory Management

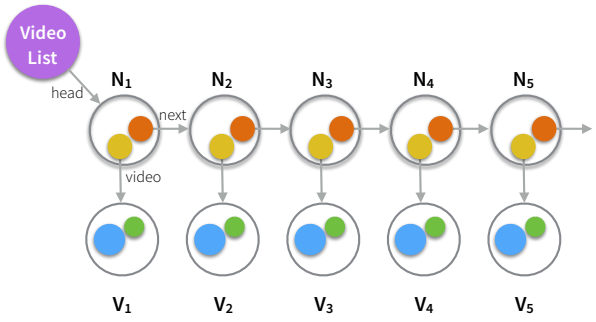

Figure 1: VideoList representation in the programmer's mind.

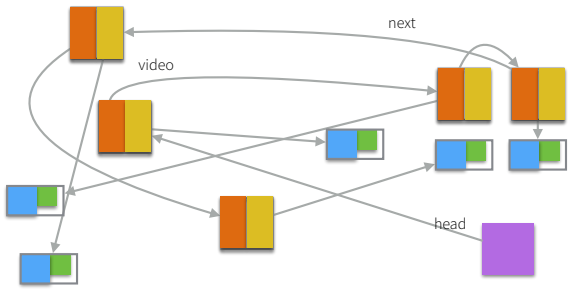

Figure 2: Actual VideoList representation in memory.

The method popularVideos measures popularity by iterating over the whole list and checking for each video, if the number of views is greater than a given pivot. If it is, then it prints its id, views and likes.

How many programmers "visualise" this list in memory and how it is actually allocated often differs. Figure 1 shows how usually data structures are taught and depicted in text books. However, this neat representation does not reflect the reality in memory. As we see in Figure 2, all the nodes and videos are likely to be scattered all over memory with no ordering, depending on the allocator and garbage collector used.

Whenever the processor requires in-memory data, a chunk of memory (of the same size of a cache line), containing this data and what is adjacent in memory to it, is fetched to cache. Given this, and with such "random" allocation, each time a Video object is fetched to cache, other (useless) data will potentially be fetched to the same cache line, thus occupying precious cache space. Moreover, in the worst case, each video access will result in an expensive cache miss.

3. SOLVING THE PROBLEM WITH OHMM

A possible solution to good data layout and consequent good program locality is to allocate all the objects of type video in consecutive memory, so that, when a Video is read from memory, a few more videos (depending on the cache line size) will be loaded as well. This brings to cache useful data for the next loop iterations, thus reducing cache misses. This optimisation can be refined by splitting objects so that only the useful part of the object is loaded into cache, allowing to fit more data.

In this section, using the example from the previous sec-

tion, we informally describe how we intend to extend an object oriented programming language with annotations that describe how and where objects should be allocated in memory. As basis for this work we use a small sequential OO language, which features class declarations, and field and method declarations inside classes, as expected.

We extend such a language with two kinds of annotations:

Ownership annotations that identify which objects must be allocated close to each other. The rationale behind these annotations is that objects that will be often used together, should be allocated together, if possible following their access order, in order to fetch useful data in advance.

Cluster annotations that define which object fields must be together or separated in memory. The idea behind these annotations is that fields of the same object that are often used together should be placed together, while fields that are not likely to be used together should be allocated in a different places. Splitting of object fields allows to keep more "important" data in cache.

3.1 Ownership Annotations

In this section we explain how we use annotations based on ownership types to describe where to place objects. For example, the class declaration for the VideoList can be extended as follows:

```
class VideoList⟨o1, o2, o3⟩
  head: Node⟨o2, o3⟩
  /** etc **/
```

This means that an instance of VideoList will be located in o1, and can reach objects located in o2 and o3 through the head field, similarly to other ownership type systems [8, 9].

The remaining class declarations are extended in the following manner:

```
class Node⟨o1, o2⟩
  video: Video⟨o2⟩
  next: Node⟨o1, o2⟩
  /** etc **/

class Video⟨o⟩
  id: int
  views: int
  likes: int
  /** etc **/
```

The class declaration Node takes two ownership parameters and has two fields. The object referred by next is an instance of Node which is in the same location o1, as **this** node. The object referred by video is an instance of Video which is in some other location o2. The Video type is parameterised over a single pool parameter denoting its containing location; all its three fields are of primitive type, and have value semantics—they do not take additional parameters. With this OHMM code, all the nodes of this list are allocated close to each other—all of them are allocated in the same contiguous space, the same pool, as well as, all the videos are allocated all together in some other pool.

One of the main motivations to choose ownership types to describe where objects are allocated was that they support modular class declarations. We want to allow different instances of the same class to be allocated in different places: objects can be *floating somewhere* in memory, or allocated

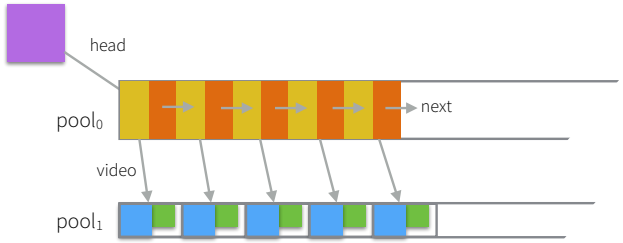

Figure 3: VideoList pointing to objects allocated in two different pools.

in pools, depending on their type. As examples, we consider three different allocating schemes, using the current VideoList example:

Scheme 1. All the objects should be allocated somewhere in memory, as in Figure 2.

Scheme 2. All the videos should be allocated in one pool while all the other objects are floating in memory.

Scheme 3. All the nodes and videos should be allocated in pools, and the VideoList object should be somewhere in memory, as in Figure 3

In order to achieve these different layouts we allow the programmer to use in her types the keyword **none** to identify objects that are not allocated in pools, and to create pools which can be referred in the types as well. The code to create a VideoList with these three schemes is below:

```
/** Scheme 1 **/
new VideoList⟨none, none, none⟩

/** Scheme 2 **/
Pool p of Video in
  new VideoList⟨none, none, p⟩

/** Scheme 3 **/
Pool
  pool0 of Node
  pool1 of Video
in
  new VideoList⟨none, pool0, pool1⟩
```

In the third layout, depicted in Figure 3, we can see that the head field of the list points to the first position of $pool_0$[3], and that all the next fields point to nodes in their next positions. As a consequence, each access to a node will either result in a cache hit, or in loading more nodes into cache for subsequent cache hits. All the videos are also allocated contiguously. Note that the order in which the objects are placed in a pool is very important: the wrong order could cost as much as if objects were not allocated in a pool. Currently we consider the allocation order and we intend to explore other kinds of ordering in future work.

3.2 Cluster Annotations

In this section, we explain how to use the **cluster** annotations to split pools into different subpools, that is, to split objects into different *records*.

[3]we can think of a pool as an array where each element contains an object (not a pointer as it is common in other languages).

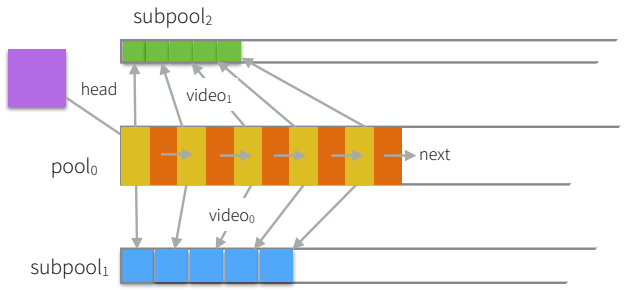

Figure 4: Splitting within a pool.

For instance, in order to iterate over all videos, in each loop-iteration, the programmer needs to read the video it points to and its next node (a common pattern when iterating over linked-lists), therefore it makes sense that instances of the type Node keep the references to their videos and to next nodes together. In order to group the fields video and next, we extend pool creation of $pool_0$, from the previous section, with the following clustering information:

```
Pool pool0 of Node = cluster {video, next} in //...
```

However, in other cases, it does not make sense to keep all the object fields together, as for instance in the Video objects. We extend $pool_1$ as below:

```
Pool pool1 of Video = cluster {id, likes}
                    + cluster {views} in //...
```

In order to understand this splitting decision we look at the popularVideos method, and in particular Lines 4 and 5, where these fields are used:

```
1 let cur = this.head in
2 while (cur != null) {
3   let v = cur.video in
4   if cur.views > pivot then
5     print(v.id + ": " + v.views + ", " + v.likes);
6   cur = cur.next
7 }
```

If the programmer passes a high number as pivot, the program is not likely to access the id and likes fields of the object referenced by cur. The lower it is the probability of these fields being accessed, the more we hurt program performance by reading the entire video. Therefore, given that the field views is accessed in each loop-iteration, we use a cluster for the single field views, allowing for more green data (respective to the views) to be fetched, thus reducing the number of cache misses. Moreover, because every time the video's id is read from memory, the video's number of likes is also read, it makes sense that these two fields are allocated together, in the same cluster. This allows now to split $pool_1$ of Figure 3 in $subpool_1$ and $subpool_2$. The result is in Figure 4.

These cluster annotations allow the programmer to allocate different instances of the VideoList data structure with different layouts. This is important because different parts of a program may use different fields of different instances of the same class. For example, we could extend the class Video from our running example with a toString method that returns its textual representation, reading all its fields. Then if we instantiated a different VideoList and invoked the toString method on all its videos, the best layout would be one where the objects' fields would be allocated together.

There are two problems with this approach that we intend to tackle in future work: 1) it breaks encapsulation of objects, in the sense that their fields are no longer hidden from the clients; and 2) it does not deal with the fact that the same data structure may be accessed with different iteration patterns.

3.3 Putting it all together

We are finally able to show the final code, with all the class declarations properly annotated and with a new Main class that creates the data structures. The code is shown below, and the respective layout in memory is in Figure 4.

```
class VideoList⟨o1, o2, o3⟩
  head: Node⟨o2, o3⟩

  def popularVideos(above: int): void
    // This code does not require any changes
    let cur = this.head in
    while (cur != null) {
      let v = cur.video in
      if cur.views > pivot then
        print(v.id + ": " + v.views + ", " + v.likes);
      cur = cur.next
    }

class Node⟨o1, o2⟩
  video: Video⟨o2⟩
  next: Node⟨o1, o2⟩

class Video⟨o⟩
  id: int
  likes: int
  views: int

class Main
  // it does not take any ownership information
  // it only has a single instance, therefore no need for
  // pooled allocation
  def main(): void
    Pool
      p1 of Node = cluster {video, next}
      p2 of Video = cluster {id, likes} + cluster {views} in
    let
      data = readFile("videos.txt")
      videos = new VideoList⟨none, p1, p2⟩
    in {
      this.populate(videos, data);
      videos.popularVideos(50000);
    }
```

4. POOL REORDERING, GARBAGE COLLECTION AND OHMM

Garbage collection will play an important role in OHMM. In this section we extend our running example in order to explain how we can use a garbage collector to optimise object layouts. We extend the main method as follows:

```
1 Pool
2   p1 of Node = cluster {video, next}
3   p2 of Video = cluster {id, likes} + cluster {views} in
4 let
5   data = readFile("videos.txt")
6   videos = new VideoList⟨none, p1, p2⟩
7 in
8   this.populate(videos, data);
9   videos.popularVideos(50000);
```

```
10    /** new code **/
11    this.iterate(videos);
12              // type of top: VideoList⟨none, none, p2⟩
13    let top = this.sortByLikes(videos)   in
14    this.iterate(top);
15    this.iterate(top);
16    this.iterate(top);
```

The method this.sortByLikes returns a new list (with new nodes) with aliases to the Videos of the list videos, where these videos are ordered by number of likes–both videos and top lists point to the same videos but with different orderings. The iterate method iterates over the videos of the list received as parameter.

By now, it should be clear that the iteration over the list videos on line 11 will have different performance from the iterations over the list top, on lines 14–16, even though they point to the same videos: the iteration over the top list does not follow the ordering of the pool of Videos (causing more cache misses) while the list videos does. In order to avoid cache misses in this kind of situations, we intend to use a moving collector. For instance in this case, pool p2 would see its objects ordered, so that they would follow the same ordering as in the top list.

Note however that this operation can be very expensive, depending on the size of the pool; if the resulting pool is never used again, or not used often enough, then it is not worth it to move the objects. Moreover, the garbage collector needs to find what is the order it should use. In order to solve these problems, we want to use static analysis to find the iteration patterns of programs.

5. RELATED WORK

As mentioned earlier, pooling and splitting are two techniques often used to improve programs' data layout, thus improving their performances. The concept of data placement to reduce cache misses was first introduced by Calder et al. [6], where the authors apply profiling techniques to find temporal relationships among objects. This work was then followed up by Lattner et al. [20, 21] where rather then relying on profiling, the authors apply static analysis to C and C++ programs, in order to find what layout to use. Huang et al. [19] explore pool allocation in the context of Java. Object Splitting was introduced by Franz and Kistler [15], where they classify fields as being hot (accessed frequently) and cold (accessed less frequently) and use this classification to decide how to split objects. Since then splitting has been combined with pooling [13, 25, 7, 26, 27].

Another interesting work is the one presented by Hirzel [18], that uses a copying garbage collector in order to implement several data layouts of object oriented programs and evaluate which layout present the best performance.

Tofte and Talpin introduced the concept of region-based memory management [23, 24]. They use region types, which divide memory in regions, in an ML language, where allocation and deallocation are inferred from type and effect analysis. This idea was then used in the Cyclone language [17], which is concerned with safety of C-like languages. Other language that also provides means to split data in the heap, is the Deterministic Parallel Java where code is annotated with regions information and it is possible to calculate the effects of reading and writing to data [5]. Note that these languages only divide the heap *conceptually*.

There are as well some programming languages that split the heap in several sub-heaps in order to simplify garbage collection or parallelism. Examples of these languages are Pony [10, 11, 12], Erlang [1, 2] and Loci [28]. None of these languages, however, share goals with OHMM, in the sense that they do not try to improve data locality, and particularly in Pony and Erlang, the programmer does not have any control on how to divide the heap.

It is not the first time that ownership types are used to express object layouts. In the context of NUMA systems, Franco and Drossopoulou [14] proposed a variant of ownership types in order to describe in which NUMA nodes the objects should be placed. The final goal of this work was to improve program performance by reducing memory accesses to remote nodes, ignoring any possible in-cache data accesses.

6. FINAL REMARKS

This paper informally describes an object-oriented programming language, where the programmer is able to use a variant of ownership types in order to express how data structures and objects should be allocated in memory. This language was designed to be compiled to a low-level language that will function using pool allocation and object splitting techniques that are already well-studied and proved to improve significantly performance. The idea is that the annotated high-level programs are not meant to be executed. They should be translated instead to low-level code with the appropriated allocation primitives. This allows the programmer to separate the functional concerns from data-layout concerns, thus simplifying code.

This is a work in progress and there is still a great amount of work that we intend to do, such as: develop a formal model to prove correctness, and study the impact of pooling and splitting on cache coherency protocols as formalized in [4, 3]. Moreover, we want to add parallelism and concurrency to the language; add value semantics, so that we can reduce the amount of dereferencing; and add constructs that iterate on pools, rather than on data structures. We also want to develop a compiler, and benchmark OHMM's performance.

7. REFERENCES

[1] J. Armstrong. A history of erlang. In *HOPL*, pages 6–1. ACM, 2007.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in erlang. 1993.

[3] S. Bijo, E. B. Johnsen, K. I. Pun, and S. L. Tapia Tarifa. A maude framework for cache coherent multicore architecture. In *Proc. 11th International Workshop on Rewriting Logic and its Applications*. Springer, 2016. To appear.

[4] S. Bijo, E. B. Johnsen, K. I. Pun, and S. L. Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *SAC'16*. ACM, 2016. To appear.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. L. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, pages 97–116, 2009.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS VIII*,

pages 139–149. ACM, 1998.

[7] T. M. Chilimbi and R. Shaham. Cache-conscious Coallocation of Hot Data Streams. In *PLDI '06*, pages 252–262. ACM, 2006.

[8] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 15–58. Springer, 2013.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, 1998.

[10] S. Clebsch, S. Blessing, J. Franco, and S. Drossopoulou. Ownership and reference counting based garbage collection in the actor world. In *ICOOOLPS'2015*. ACM, 2015.

[11] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *OOPSLA'2013*, pages 553–570. ACM, 2013.

[12] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *AGERE15*, 2015.

[13] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. Mpads: Memory-pooling-assisted data splitting. In *ISMM '08*, pages 101–110. ACM, 2008.

[14] J. Franco and S. Drossopoulou. Behavioural types for non-uniform memory accesses. *PLACES 2015*, page 39, 2015.

[15] M. Franz and T. Kistler. Splitting Data Objects to Increase Cache Utilization. Technical report, University of California, Irvine, 1998.

[16] M. Hagelin. Optimizing memory management with object-local heaps. Master's thesis, Department of Information Technology, Uppsala University, 2015. Main advisor: T. Wrigstad.

[17] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *ISMM'04*, pages 73–84. ACM, 2004.

[18] M. Hirzel. Data layouts for object-oriented programs. In *ICMMCS*, pages 265–276. ACM, 2007.

[19] X. Huang, S. M. Blackburn, K. S. Mckinley, J. Eliot, B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *OOPSLA*, 2004.

[20] C. Lattner and V. Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Technical report, U. of Illinois, 2003.

[21] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI '05*, pages 129–142. ACM, 2005.

[22] Z. Majo and T. R. Gross. (mis)understanding the numa memory system performance of multithreaded workloads. In *IISWC'2013*, pages 11–22, 2013.

[23] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL '94*, pages 188–201. ACM, 1994.

[24] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[25] H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff. Automatic restructuring of linked data structures. In *LCPC'09*, pages 263–277. Springer, 2010.

[26] Z. Wang, C. Wu, and P.-C. Yew. On improving heap memory layout by dynamic pool allocation. In *CGO '10*, pages 92–100. ACM, 2010.

[27] Z. Wang, C. Wu, P.-C. Yew, J. Li, and D. Xu. On-the-fly structure splitting for heap objects. *ACM TACO*, 8(4):26:1–26:20, 2012.

[28] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009*, LNCS, pages 445–469. Springer, 2009.

[29] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[30] Q. Zhao, R. Rabbah, and W.-F. Wong. Dynamic Memory Optimization Using Pool Allocation and Prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005.