Custom Object Layout for Garbage-Collected Languages

Gene Novark Trevor Strohman Emery D. Berger
University of Massachusetts Amherst
{gnovark,strohman,emery}@cs.umass.edu

Abstract

Modern architectures require data locality to achieve performance. However, garbage-collected languages like Java limit the ability of programmers to influence object locality, and so impose a significant performance penalty. We present *custom object layout*, an approach that allows programmers to control object layout in garbage-collected languages. Custom object layout cooperates with copying garbage collection. At collection time, the garbage collector invokes programmer-supplied methods that direct object placement. Custom object layout is particularly effective at improving the locality of classes with well-known traversal patterns, such as dictionary data structures. We show that using custom object layout can reduce cache misses by 50%–77% and thus improves the query performance of dictionary data structures by 20%.

1. Introduction

Recent processor speed improvements have far outpaced memory speed improvements. At the same time, application development is shifting from languages like C and C++ that allow programmers considerable flexibility in determining object layout, to garbage-collected languages like Java and C#. Garbage-collected languages impose a considerable performance penalty by preventing programmers from influencing object placement. Past work has shown that careful placement of objects in memory can yield performance improvements as high as 42% in C and C++ applications [5, 11, 26].

This paper makes the following contributions:

- It introduces **custom object layout**, an efficient way for programmers to control object placement in garbage-collected languages. Custom object layout cooperates with copying garbage collection: the garbage collector invokes programmer-supplied methods that direct object placement. Section 2.2 shows how custom object layout can be implemented safely.
- It demonstrates that custom object layout substantially improves performance over automatic layout strategies for a variety of large dictionary data structures. This increased performance stems directly from reducing cache misses.

The rest of this paper is organized as follows. Section 2 describes the design of our custom object layout approach and its implementation in Jikes RVM [1, 2] using the MMTk toolkit [6]. Section 3 presents the experimental methodology and our benchmark suite, which measures the impact of custom layout on the query performance of a number of dictionary data structures. Section 4 describes the specific custom layout policies we employ for these data structures, and presents empirical results. Section 5 describes related work, Section 6 discusses planned future work, and Section 7 concludes.

2. Custom Object Layouts

Programmers specify a custom object layout for a particular class by implementing a `CustomLayout` interface. The interface methods provide hints to the runtime system about how to arrange objects in memory. Programs written using this strategy run unmodified on other virtual machines, but enable higher performance when used in conjunction with a virtual machine modified to use these hints. This section describes the `CustomLayout` interface and provides details on its communication with the garbage collector.

2.1 `CustomLayout` Design & Implementation

In order to override the default strategy and implement custom layout, we modified Jikes RVM to look for user classes that implement the `CustomLayout` interface. This interface (shown in Figure 1) specifies two methods which the programmer must implement. The `arrangeBegin` method initializes the `CustomLayout` state. The `arrange-Next` method's semantics are similar to `Iterator.next()`. Upon each invocation, the method returns the next item to be copied. Figure 2 shows a simple implementation that the

```
public interface CustomLayout {
  /// called at start of GC.
  public void arrangeBegin();

  /// return next object to place.
  public Object arrangeNext();
}
```

Figure 1. The `CustomLayout` interface

```
// next list node to lay out
private static Node next = null;
// current data object
private static Object obj = null;

public void arrangeBegin() {
  // start at list head
  next = first;
}

public Object arrangeNext() {
  if (next == null && obj == null)
    // done with list
    return null;

  if (obj == null) {
    // just copied the node, copy its data
    obj = next.data;
    return next;
  } else {
    // copy data, move to next node
    Object prev = obj;
    next = next.next;
    obj = null;
    return prev;
  }
}
```

Figure 2. `CustomLayout` implementation for `LinkedList`

collector uses to linearize a linked list. Section 4 describes the implementations for the other data structures we examine here.

During garbage collection, collection proceeds until the collector copies an object that implements the `CustomLayout` interface. The collector then invokes the object's `arrangeBegin` method, allowing it to initialize its state. The collector then repeatedly calls the `arrangeNext` method and places the returned objects into contiguous memory until the method returns `null`.

2.2 Safety

The methods implementing custom object layout and the copying collector must cooperate to ensure that they preserve consistency of the heap. There are four requirements that, when satisfied, guarantee heap consistency:

1. All reachable objects must be copied.
2. The collector must update all pointers to point to the correct targets.
3. The custom object layout methods must not corrupt the heap.
4. The custom code must not mutate the object graph.

The collector ensures requirements (1) and (2)—copying and updating pointers—by copying objects not placed by the custom layout code, and by updating pointers in all affected objects. As collection proceeds, the collector places each object on a scan queue. This queueing corresponds to coloring the object grey in the tricolor abstraction [17], and ensures that the collector will later find and copy all objects that are part of the data structure. Scanning also ensures that each pointer field is updated to point to the correct target. This behavior protects against incorrectly-written layout code and also allows the programmer to ignore less important data structure objects.

We have not yet implemented static checks to satisfy requirements (3) and (4), but describe how to do so below.

To avoid heap corruption (3), the `CustomLayout` implementation cannot allocate memory. To avoid allocating memory, our implementation uses a preallocated context to store state. This context is shared per class and allocated at class load time. In a single-threaded collector, this approach is safe because the collector traverses only one data structure at a time. A parallel or concurrent collector would need either one context per object, or context stored in thread-local memory. It would be straightforward to check at class load-time that none of the `CustomLayout` methods allocate memory by checking for direct or indirect calls to *new*.

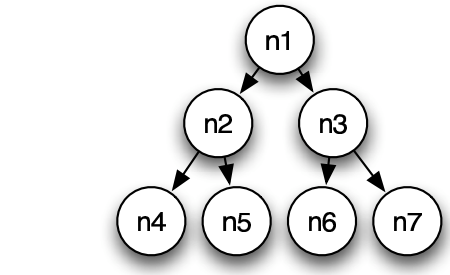
Finally, to satisfy requirement (4), the implementation must ensure that the custom layout code does not modify the structure of the object graph. Pointer modifications to heap objects may result in stale pointers being stored into new objects which are already scanned. We cannot simply forbid custom layout code from performing pointer mutations, as these operations are required to keep state for object traversal. Static analysis can be used to verify that the pointers mutated by custom layout code are never accessed by mutator code.

2.3 Pseudo-depth-first Layout

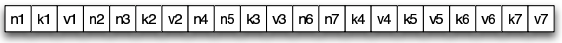
We use Jikes RVM [1, 2] and its MMTk [6] memory management toolkit as our experimental infrastructure. MMTk supports several different copying garbage collectors. The default copy order in MMTk is *pseudo-depth-first*. This traversal is used over pure depth-first because it can be implemented more efficiently. Pseudo-depth-first copies an object's children immediately after each other.

The key difference in resulting layouts between the two policies is that pseudo-depth-first separates parents from their children. For example, in a binary tree, pure depth-first copies all of the left subtree before copying any of the right subtree. However, pseudo-depth first copies the roots of both subtrees before recursively copying the rest of the left subtree. Figure 3 shows the effect of different layouts on a binary tree.

Because `CustomLayout` only affects annotated data structures, it requires a default layout algorithm. Because of its effectiveness at placing general-purpose data structures, we use pseudo-depth-first. The choice of default layout al-



(a) Example binary tree



(b) Breadth-first layout

(c) Pure Depth-first layout

(d) Pseudo-DF layout

(e) Custom layout

Figure 3. Effects of layout on binary search tree

gorithm has little impact on our microbenchmarks, which rely primarily on custom object layout.

3. Methodology

We perform our experiments on two architectures. The first is a 3.0 GHz Pentium 4 with Hyperthreading disabled. It has a 64 byte DL1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 512KB unified 8-way set associative L2 cache, and 1GB of main memory. We run Linux kernel version 2.6.13.2 with the *perfctr* patch version 2.6.16. We use the *perfctr* patch and libraries to access the Pentium 4's on-chip performance counters to measure the number of L1 and L2 cache misses for each run.

The second architecture is a 1.8 GHz PowerPC 970. This processor has 128 byte cache line size, a 32KB 2-way set associative L1 data cache and a 512KB unified 8-way set associative L2 cache. The machine has 2GB of physical memory. It runs Linux 2.6.14.6 with *perfctr* version 2.7.19.

Our approach requires a copying garbage collector to implement custom object layouts. We use a *SemiSpace* collector, a standard full-heap copying garbage collector [17]. Semispace collectors are generally less efficient than generational collectors for overall program performance. However, we perform our experiments after a full-heap garbage collection, and no objects are allocated during measurements. We compare custom object layouts to pseudo-depth-first (Section 2.3) and breadth-first layouts.

Structure	Per-query cost
Red-black trees	$O(\log N)$ [3]
Splay trees	amortized $O(\log N)$ [25]
Skip lists	expected $O(\log N)$ [22]
Linked lists	$O(N)$

Table 1. Implemented data structures

Our runs use a 512MB maximum heap size. However, because no objects are allocated during the timing portion of the benchmarks, no garbage collections may occur. The reported times are thus insensitive to heap size, as long as the data structures built can fit.

Our virtual machine infrastructure is Jikes RVM version 2.4.0, configured to compile all methods with the optimizing compiler. We use a *second run* methodology, in which we report performance metrics for a second run of the benchmark. The first run allows the system to compile all needed methods, so the reported results do not include compilation overhead. This methodology removes variability due to the default adaptive compilation system in Jikes RVM.

3.1 Benchmarks

We study the average query cost on several data structures. Our benchmarks consist of multiple queries applied to a static data structure. They maintain an array of the values known to be in the data structure, and select only those values as query parameters. We choose elements using a uniform distribution.

We investigate four different linked data structures: **skip lists** [22], **red-black trees** [3], **splay trees** [25], and **doubly-linked lists**. Table 1 presents their algorithmic characteristics. The linked list and red-black tree implementations are from GNU Classpath [14], modified to implement `CustomLayout`. The splay tree implementation is based on the red-black tree from Classpath. We implemented skip lists to adhere to a limited form of the `java.util.Map` interface.

Each tree-based benchmark performs 500,000 uniformly-distributed queries on data structures of varying sizes. The skip list benchmark performs 100,000 queries, while the linked list benchmark performs 10,000 queries. To emulate the effects of unrelated data that would exist in a real program, the benchmark builds 4 different data structures of each size. Omitting this detail can have a significant effect on locality. For example, a breadth-first collector will linearize a single linked list in isolation, but will interleave nodes from multiple linked lists.

We run each benchmark 10 times and report the mean for each.

4. Results

We present the results of our experiments on each of the data structures described above. We describe the custom layout policy we implemented for each. Each experiment reports

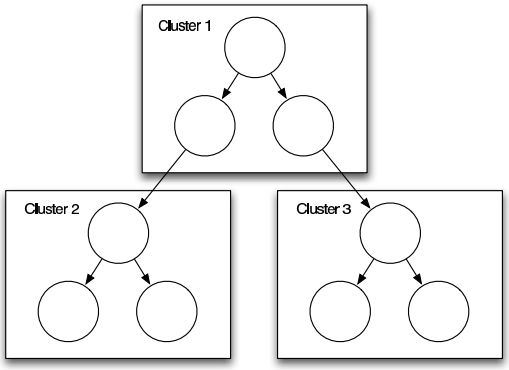


Figure 4. Hierarchical Clustering

Benchmark	Execution time	L1	L2
Red-Black, P4	18.8%	20.2%	51.5%
Red-Black, PPC970	22.0%	12.6%	77.2%
Splay, P4	17.9%	21.0%	50.0%
Splay, PPC970	23.3%	19.3%	62.0%

Table 2. Performance improvements over baseline

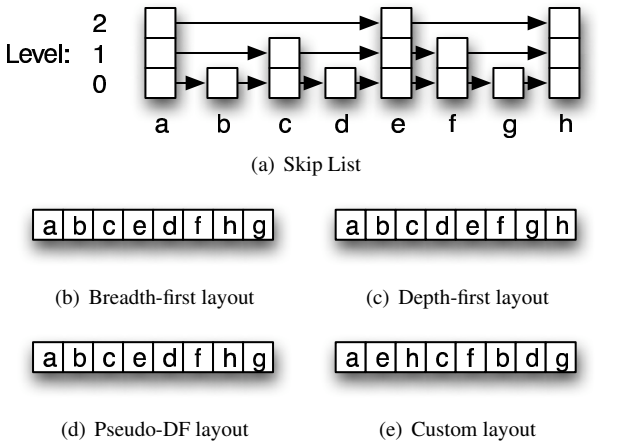
the total execution time, L1 data cache misses, and L2 data cache misses.

4.1 TreeMap Results

We implemented a `TreeMap` class using both red-black trees and splay trees. The implementation of the `CustomLayout` interface in `TreeMap` breaks the tree into hierarchical clusters, which it arranges in depth-first order. Figure 4 depicts hierarchical clustering. Each cluster consists of a node, its key, and its children, their keys, and so on recursively up to the cluster size. Because the cache line sizes on the architectures we examine here are relatively small with respect to object size, we use a cluster size of 3. We segregate nodes and keys from values, because values are not needed during traversal. This organization prevents unneeded value objects from polluting cache lines containing keys and nodes, which queries require. Figure 3 shows the resulting layout for a small binary tree.

While we use a depth-first hierarchical decomposition here, it is important to note that this layout is not optimal. Bender et al. show that the *van Emde Boas* layout is an optimal cache-oblivious layout policy for binary trees [4]. However, it is unclear how to write custom layout code to implement the van Emde Boas layout without imposing excessive space overhead.

Figures 6–7 show the performance results for these two structures. Table 2 summarizes the performance improvement that custom layout achieves over pseudo-depth-first, which is better than breadth-first on all of our benchmarks.



(a) Skip List

(b) Breadth-first layout

(c) Depth-first layout

(d) Pseudo-DF layout

(e) Custom layout

Figure 5. Skip list layout

Red-Black Trees

Custom object layout significantly reduces runtime and cache miss rates on both architectures. On red-black trees, the custom layout achieves up to a 28.1% reduction in average query time when compared to the next best strategy, pseudo-depth-first. This decrease is mostly due to a reduced L2 miss rate. This reduction is especially noticeable for smaller trees, where custom object layout eliminates practically all L2 cache misses. For these sizes, custom object layout allows all relevant tree data to fit into the L2 cache because it segregates values from keys and nodes. For trees significantly larger than the L2 cache, the custom layout reduces the L2 miss rate by around 33% on the P4 and 52% on the PPC970.

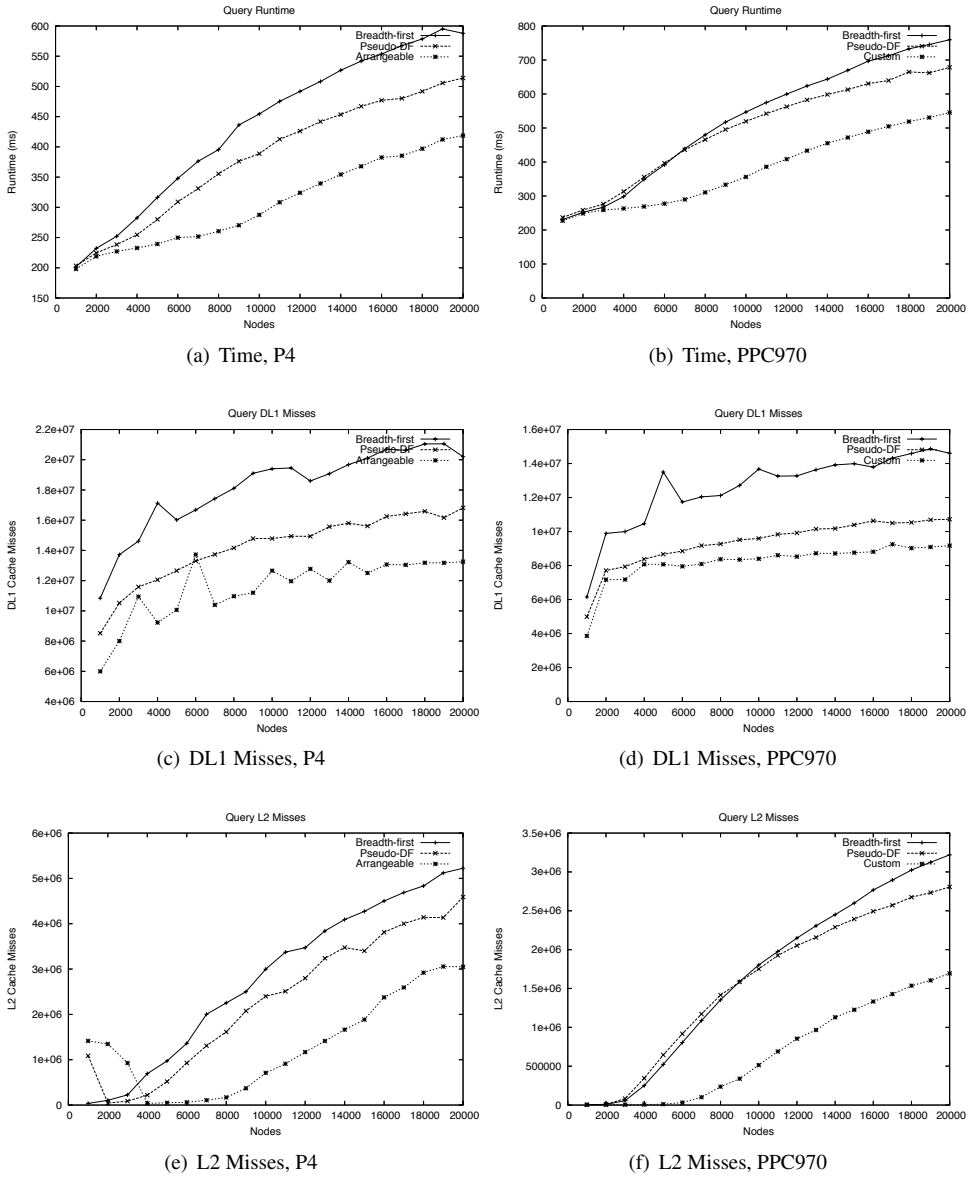
Splay Trees

Unlike red-black trees, whose structure is unchanged by queries, splay trees dynamically reorganize themselves. The most-recently accessed node is "splayed" to become the new root. Because our benchmark performs random queries, the resulting tree structure is effectively shuffled.

Despite this restructuring, the custom object layout results in similar performance improvements. We attribute these improvements to the segregation of values from keys and nodes.

Performance Anomalies

We observe several performance anomalies in these experiments, especially on the Pentium 4. In particular, Figures 6 and 7 show that custom object layout causes erratic L2 cache miss behavior on both types of binary search trees at small tree sizes. We cannot currently explain why the absolute number of L2 misses is so much greater at these sizes, as the entire tree should fit easily within L2 cache. Other anomalies appear at isolated tree sizes. For example, for DL1 cache miss rates in Figure 7, breadth-first has a large anomaly at 17,000 nodes. This result is consistent over many runs and is



(a) Time, P4

(b) Time, PPC970

(c) DL1 Misses, P4

(d) DL1 Misses, PPC970

(e) L2 Misses, P4

(f) L2 Misses, PPC970

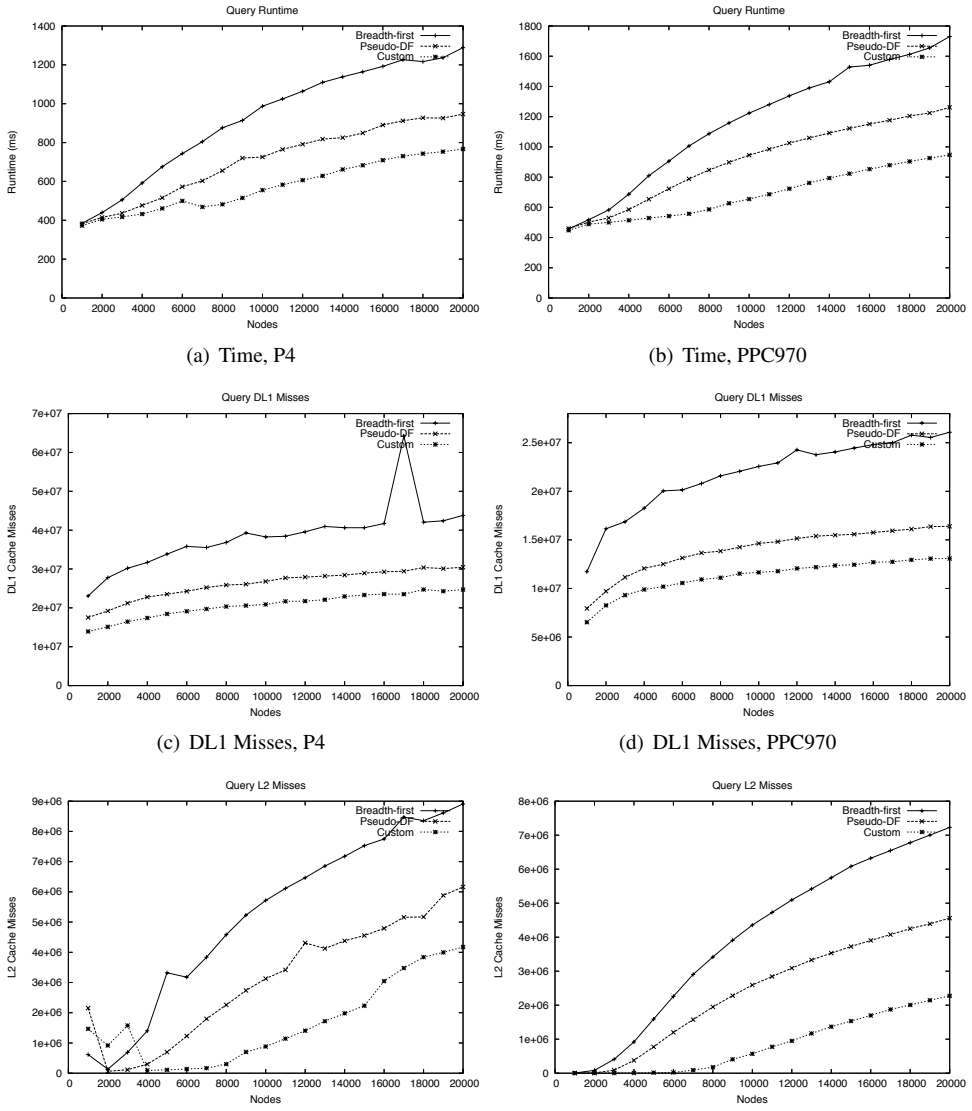
Figure 6. Red-Black Tree Performance

not the result of a single outlier in the data. We suspect that at this size, the collector places some commonly-accessed data such that cache conflicts occur.

4.2 Skip List Results

Skip lists are a probabilistic dictionary data structure with expected $O(\log n)$ operation time [22]. The data structure is parameterized by p, which determines the expected *level* of the node. Each node has a probability $1/p^k$ of being level k. A node of level k contains pointers for every level between 1 and k. Nodes on each level are ordered by their keys. A pointer of level k *skips* an expected p^{k-1} nodes in the graph. A query proceeds by searching down one level of the list until it finds an object greater than the specified key. The search algorithm then searches in the same manner on the next lower level of the list. The algorithm terminates when it reaches the end of the lowest level.

Our custom layout exploits these properties of the search algorithm. When the parameter p is greater than 2, the search is most likely to look at the next node on the level, rather than moving to the next level. We therefore first linearize the highest level of the list, then the next level, and so on. As with binary trees, we place the key and node in adjacent memory, but segregate values to prevent cache line pollution. Figure 5 shows the resulting layout for a small skip list.



(a) Time, P4

(b) Time, PPC970

(c) DL1 Misses, P4

(d) DL1 Misses, PPC970

(e) L2 Misses, P4

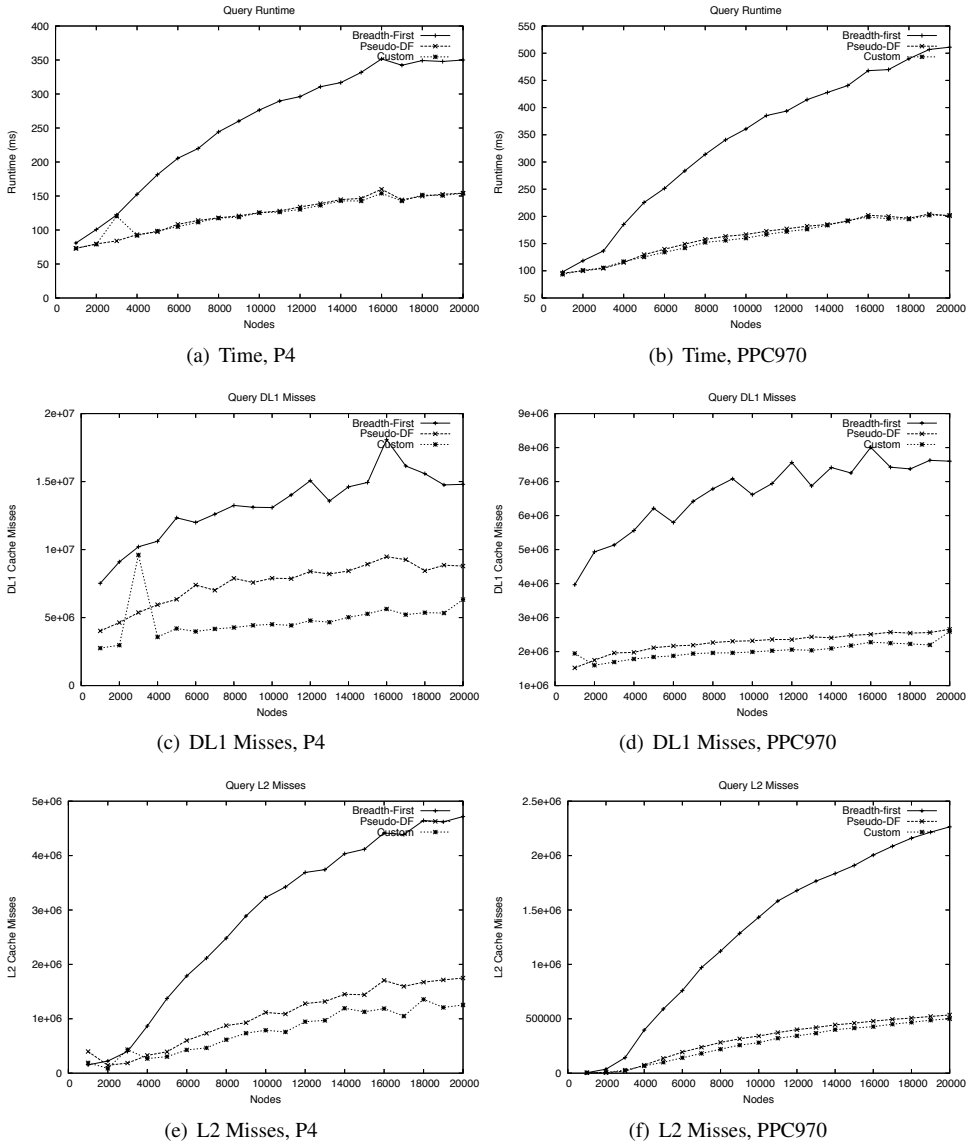
(f) L2 Misses, PPC970

Figure 7. Splay Tree Performance

The actual implementation of skip lists in Java requires two objects per node. Since each node contains a variable number of pointers, we use a separate array. Our layout policy places this array adjacent to the node object (which contains pointers to the key and value).

Pugh shows that the expected operation cost of a skip list is optimal when the parameter $p \approx e$. However, our layout linearizes the skip list along single levels, making the expected cost of a forward traversal lower than that of moving to the next level of the list. We varied this parameter to determine the effect of locality and found that optimal performance on the Pentium 4 was achieved with p set to 5; we use the same value of p for the PowerPC experiments.

Figure 8 shows the performance of skip list queries for $p = 5$. As with binary search trees, custom object layout improves cache performance, especially on the Pentium 4. L1 misses decrease by an average of 42.6%. L2 misses show a similar decrease, dropping by approximately 40.0% for skip lists that do not fit in L2 cache and reducing misses by 32.1% over the range of sizes tested. The PowerPC results show similar trends, although the magnitude of the difference between the baseline and the custom layout is smaller. Nonetheless, these miss rate reductions do not translate into a significant performance improvement over pseudo-DFS on either platform. We are currently investigating why the substantial reduction in cache misses does not result in a runtime improvement on the Pentium 4.



(a) Time, P4

(b) Time, PPC970

(c) DL1 Misses, P4

(d) DL1 Misses, PPC970

(e) L2 Misses, P4

(f) L2 Misses, PPC970

Figure 8. Skip List Performance, $p = 5$

4.3 Linked List Results

Figure 9 shows performance for the linked list microbenchmark. Pseudo-depth-first matches the performance of our custom layout, because it also linearizes the list. Because the traversal requires both the node and the data object, no other objects cause cache pollution. L2 misses are negligible, since the lists span contiguous memory and fit within the 512K L2 cache.

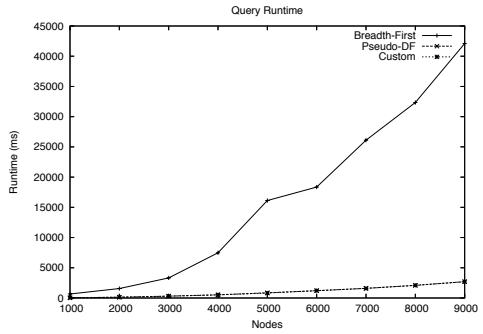
For this benchmark, the breadth-first collector exhibits poor performance. Recall that each benchmark creates four instances of each data structure. Breadth-first interleaves objects from the four different lists together. As a result, any cache line can contain data from at most one node, causing cache residency to drop.

5. Related Work

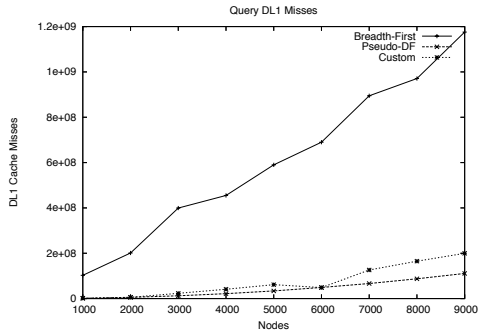
We describe in this section the most closely related work to custom object layout. We divide these into three categories: *oblivious layout policies* that ignore type or access information, *adaptive layout policies* that respond to such information, and other approaches.

5.1 Oblivious Layout

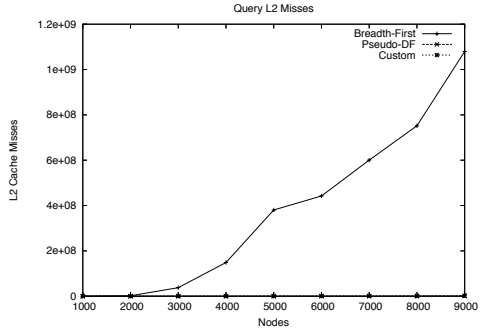
The canonical copying algorithm is Cheney scan [8], which copies objects in breadth-first order. Previous work [16, 21, 27] has shown that the breadth-first ordering resulting from Cheney scan has poor locality for many common data structures. A breadth-first collector may separate objects from all of their children. They cluster unrelated objects together. In



(a) Time



(b) DL1 Misses



(c) L2 Misses

Figure 9. Linked List Performance

particular, breadth-first ordering does not linearize lists, one of the earliest locality optimizations noted in the context of copying garbage collection.

Unlike breadth-first layouts, a depth-first layout linearizes lists. However, pure depth-first traversal has important drawbacks. Because it is recursive, the call stack grows when traversing a deep structure such as a linked list. While this overhead can be avoided [23], most collectors avoid it by implementing less expensive variants like the pseudo-depth-first approach described in Section 2.3.

Wilson et al. propose hierarchical decomposition which groups the upper nodes of a tree together. They show that using hierarchical decomposition over breadth-first layout significantly reduces the page fault rate for several LISP programs [27]. We use a specialized form of hierarchical decomposition to improve the locality of binary search trees, described in section 4.1.

5.2 Adaptive Layout

Wilson et al. report that hierarchical decomposition impairs performance for data structures which are not tree-like [19]. They propose instead that the layout be type-directed. Their system identifies trees and association lists by scanning objects a few levels ahead and using a recognition heuristic. It then chooses depth-first or hierarchical decomposition as appropriate. Rather than attempting to infer the best copying order, custom object layouts allow the programmer to specify it directly.

Chilimbi et al. [12] dynamically build a *temporal affinity graph* and use it to colocate hot objects that are referenced together. In this graph, nodes are objects which are connected by an edge if the mutator references them together frequently within a short period of time. Their technique does not require these objects to be connected. Though this system requires a read barrier, the recent LO system uses sampling to decrease the overhead to a few percent [7]. Our system guarantees good locality for annotated containers, while LO may suffer if the set of hot objects changes rapidly. On the other hand, LO can improve locality for any structure in the heap, while ours can only improve annotated ones.

Huang et al. use method sampling to identify connected objects which the mutator is likely to access together [16]. Their system analyzes "hot" methods and uses their field access patterns to identify hot fields on a per-class basis. Their collector then copies hot fields of an object first, adjacent to the parent. This technique ensures that parents and commonly-accessed children are close together. Their system generally matches the best static layout, rather than improving on it. Our system allows much more flexible policies, allowing it to beat either static layout.

Chilimbi et al. [11] present a tool called *ccmorph* that allows C programmers to specify an ordering for tree-like data structures. The tool reorganizes the data structure on request based upon this ordering. Unlike custom object layout, *ccmorph* can only reorder data structures when there are no incoming pointers to objects from outside of the data structure itself. Our system exploits copying garbage collection to ensure the safety of its reorganization. The program must invoke *ccmorph* explicitly, while our system automatically rearranges the structure at each garbage collection.

Shuf et al. propose a *locality-based* traversal ordering, in which pointers local to a section of the heap are traversed first [24]. This approach ensures that connected objects maintain good locality over multiple collections. They combine this approach with an allocation colocation scheme to ensure good initial locality, which their layout policy maintains.

5.3 Other Approaches

Other approaches to improving cache performance have included reordering fields within a structure, splitting a structure into hot and cold parts [10, 18], allocating objects into contiguous regions or reaps [5, 9, 15, 20], and non-moving allocators designed to improve locality [11, 13, 26]. Structure splitting is complementary to the work presented here. Our approach improves upon regions by grouping objects by traversal order rather than by allocation order.

6. Future Work

Our system currently works with a semispace collector. We would like to support generational copying collection, but generational collectors present several complications.

First, the current collector identifies the root of the data structure, rather than the contents. A generational collector may separate a data structure between two generations so that the mature space contains the root, and the nursery contains the contents. In this case, a minor collection will not organize the portion of the data structure within the nursery.

Second, the implementation of layout annotations as actual Java code causes problems. Objects must maintain state during custom object layout such as the current cursor location. If this code contains accesses to objects or arrays, these will invoke the write barrier in a generational (or other incremental copying) collector, which is generally unsafe during collection.

We are considering the use of declarative annotations rather than Java code as a way around these problems. A declarative language could be compiled to low-level code that would be guaranteed to be safe, i.e., that would satisfy the requirements given in Section 2.2. Generating safe code would also eliminate the need for static analyses to check the safety of user-written code.

7. Conclusion

We designed and implemented a system for specifying custom layout algorithms for data structures in garbage collected languages. Our system is easy to use, because the annotations are in the language itself. It is efficient, because the mutator phase runs unaltered, and the performance impact during garbage collection is negligible.

Our results show that custom object layout can substantially improve locality and thus performance. For a variety of large dictionary data structures, custom object layout improves query performance by around 20%, with larger reductions in L2 cache misses. We plan to extend this work with static checks to ensure that custom object layout directives are safe.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[2] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, Nov. 1999.

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, (1):290–306, 1972.

[4] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, Nov. 2002. ACM Press.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[7] W.-K. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[8] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[9] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the 2004 International Symposium on Memory Management*, Vancouver, Canada, October 2004.

[10] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

[11] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.

[12] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ACM International Symposium on Memory Management*, pages 37–48, Vancouver, BC, Oct. 1998.

[13] Y. Feng and E. Berger. A locality-improving dynamic memory allocator. In *MSP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Memory systems performance*, Chicago, IL, USA, 2005.

[14] GNU Classpath. http://www.classpath.org/.

[15] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan. 1990.

[16] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, Oct. 2004. ACM Press.

[17] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[18] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22(3):490–505, 2000.

[19] M. S. Lam, P. R. Wilson, and T. G. Moher. Object type directed garbage collection to improve locality. In Y. Bekkers and J. Cohen, editors, *ACM International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 404–425, St. Malo, France, Sept. 1992. Springer-Verlag.

[20] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, Chicago, IL, USA, 2005.

[21] D. A. Moon. Garbage collection in a large lisp system. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 235–246, 1984.

[22] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[23] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[24] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 13–25, Seattle, Washington, USA, 2002.

[25] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[26] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 322, Washington, DC, USA, 1998. IEEE Computer Society.

[27] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage collected systems. *ACM SIGPLAN Notices*, 26(6):177–191, 1991.