Applications of Cache--conscious Data Layout to Copying Garbage Collection

V. Benjamin Livshits
David Louie

Section 1. Introduction

A typical copying garbage collector reorders the heap data when it copies it from one semi-space to the other. As a result, most of the time, the original data layout is completely lost and a new layout is introduced, which can completely change the locality of reference. Most copying collectors make use of the Cheney algorithm, which implements breadth-first copying using just two pointers. In this paper we are considering other possible approaches to copying and their effect on locality.

Section 2. Problem statement

Clearly, breadth-first copying is not always the best solution; depending on the data structures and on how they are being used, breadth-first layout may not always be desirable. Ideally, for every data structure in the heap we would like to know what layout we should use and copy that data structure in accordance with that information. In this study, we decided to adopt the level of granularity used by some shape analysis algorithms[1]. We have considered the following shapes of data structures:

- Trees.
- DAGs.
- Cyclic structures.

In this paper we will be looking at the following copying strategies:

- Cheney, standard breadth-first copying.
- Hierarchical decomposition, an improvement to Cheney which breaks the toSpace into multiple chunks and performs local traversals on each one.
- Depth-first copying. This form of copying does recursive copying on each chunk. This approach was considered by some early researches, but when applied to all heap objects, it didn't yield much of an improvement and was actually performing worse in some cases.

Each of these strategies is described in detail below.

We would like to build a system that would allow the user to specify what copying policies to use for different shapes of data structures. We want to use static shape analysis at the level of classes to determine how to deal with each heap object[2]. We also want to do some experimentation with the system to determine what policies are efficient in different settings and whether we can come up with a general set of policies that works well in most cases.

The rest of the paper is organized as follows. In section 3 we discuss our implementation, in section 4 we describe some of the

[1] Clearly, this classification is too coarse-grained. For instance, both a linked list and a balanced tree will be deemed trees by our analysis, but clearly, a list is better laid out depth-first, whereas hierarchical decomposition generally works well for balanced trees. Also see the description of DaveTest below.

[2] A possible extension of this would allow the user to specify the layout to use on a per-class basis using pragmas.

benchmarks we have run and attempt to explain the results. In section 5 we present our conclusions and provide directions for further exploration and section 6 discusses some related work.

Section 3. Implementation

For our implementation we adopted Marmot, a research Java compiler from Microsoft. Many common optimizations have been implemented in Marmot. The compiler takes the Java bytecode, then converts it into Java intermediate representation (JIR). The first step is to convert the bytecode to a temporary-variable-based intermediate representation. Next, it takes this intermediate representation and converts it into static single assignment (SSA) form. One of the big advantages of SSA form is that there are no nested pointer calls of the form `(p.q).f`, which makes the shape analysis easier. The final step in creating the JIR is to perform type elaboration using type information implicitly left in the bytecode. After that, various optimizations are performed, as specified in [9]. To implement these optimizations, Marmot first performs the global optimizations, then it performs the local optimizations. The shape analysis algorithm was designed to be intraprocedural, so it was implemented as a local optimization.

Section 3.1: Static shape analysis

For the shape analysis, we've decided to implement the algorithm provided in [5]. This algorithm uses direction and interference relationships to determine the shape of a structure. The paper outlines certain rules for how direction and interference relationships change depending on the type of statement seen. For example, when the statement `p = new…` appears, we know that all of the existing relationships for pointer `p` get killed. The most difficult statement to handle is of the form `p->f = q` because this is where the shape of the structure can change.

At a high level, shape analysis was performed for every function within recursive data structures found in the Java bytecode. We assume that class variables are private so that any modifications to the data structure will be done in these class methods. This is not such an unreasonable assumption to make good object-oriented style commands the user to use class methods to manipulate the class fields.

The basic principle behind our shape analysis implementation was that we assume that the structure was a tree as a precondition and see if there was any code within a class method which modified the data structure so that it was no longer a tree. If there is no evidence in the function to prove that it is not a tree, then we can assume that it is.

After performing the shape analysis on every function within a given recursive data structure, we determine the shape of the data structure to be the weakest shape found. For example, if we had a recursive data structure with 3 functions and after performing shape analysis on each function, 2 functions tell us that the structure is a tree and 1 function tells us that the structure is a DAG, then we decide that the shape of the structure is a DAG. To deal with control flow, we process all reachable code. Since we are trying to see if there is any code that can change the shape of the structure,

we must explore all possible paths of control flow. Also, we assume that any function calls within a method which return structures of this class type return trees that are disjoint from the pointers within the method. One quick observation made during testing of the algorithm is that since our analysis is intraprocedural, it only works well when each method is sufficiently large.

Section 3.2: Runtime (or collection-time) support

Marmot comes with three collectors, a "null" collector, which only does allocation and doesn't do any collection at all, a mark-and-sweep conservative collector similar to the Boehm-Weiser collector, and a copying type-accurate collector. For our experiments we used the latter.

We changed the copying collector to use a hierarchical decomposition approach explored by Wilson et al [2]. The algorithm breaks the toSpace into chunks referred to as pages (we should emphasize that these are not necessarily virtual memory pages, in our experiments we chose smaller "page" sizes most of the time). In our implementation we maintain an array of scan pointer and an array of free pointers, `scan[]` and `free[]`. The current toSpace page that is being processed is denoted by `majorScan` and the page that has free space into which objects are currently copied is called `majorFree`.

In each iteration of a loop we look at the first unscanned location on the page we are currently processing and copy it to the toSpace incrementing the `toFree` pointer and adjusting `majorFree` if necessary. Then we look at the page the current object was copied to and do some local processing.

We made it so it was possible to invoke different local processing procedures for each object; each copying procedure just has to update the common data structures.

Suppose we have a set of policies, which we can view as a map from the set of shapes to the set of copying strategies: $f\colon S \to C$. Then the algorithm we have implemented can be expressed in pseudo-code as follows:

```
collect:
forward(root_set);

/* Initialize toFree, scan[],
free[],majorScan, and
majorFree appropriately */

while (majorScan<majorFree){
  object=Scan[majorScanPage];
  foreach field (object){
    forward(field);
    scanLocal(field);
  }
}

scanLocal (object):
shape = get_shape(object);
run(f(shape), object,
           PAGE(object));

//hierarchical decomposition
HD (object, page):
/* perform standard breadth-
first Cheney traversal on
page using scan[page] and
free[page] as breadth-first
queue delimiters */
while(scan[page] < free[page]
   && PAGE(free[page])==page)
{
  object = scan[page];
  foreach field (object){
    forward(field);
  }
}
```

```
//depth-first recursive
//copying
DF (object, page):
forward(object);

if(!full(page)){
  object = scan[page];
  foreach field (object){
   DF(field, page);
  }
}

//breadth-first copying.
BF (object, page):
no_op

forward(object):
t = size(object);
mem_copy(toFree, object, t);
toFree += t;
/* adjust the data structures
if necessary */[3]
```

Note that breadth-first copying strategy in this case effectively reduces to Cheney. The only difference is that Cheney works with the whole heap, whereas BF can be applied to some object types selectively. There are many design decisions this pseudo-code doesn't illustrate. For instance, how do we deal with objects that cross the page boundary? Wilson only allows objects to cross the page boundary if there's little room left on the previous page, since we allow (and use) very small pages, all objects can span multiple pages, we just need to update the data structures carefully to reflect that.

In our implementation, the copying strategy is passed in an environment variable the runtime system has to parse, that's probably the easiest thing to do, since we don't have easy access to the program's command line parameters at runtime. The shape determined by static analysis is read from a *vtable* field, where it was written by static analysis[4].

Section 4. Discussion of experimental results

Our goal was to see how various layout strategies affect the program runtime. Moreover, we decided to concentrate on cache performance issues. Wilson et al [2,4] looked at virtual memory performance, but they measured the performance of their Scheme system as a whole whereas in Java we typically have one application that doesn't allocate more memory than there is physically present on the machine.

We decided to run two rounds of experiments. The first round would use various allocation strategies to lay out trees and see how the performance changes. The second round would take some real programs and run them with different collection strategies.

The two rounds are pretty much independent in terms of implementation. The micro-benchmarks we were using were written in C and ran on Linux, whereas the macro-benchmarks were written in Java and ran on NT under Marmot. However, they serve the same purpose. In a sense, the results we get from the first round are an upper bound on what we can get in real program doing GC. Also, micro-benchmarks are much easier to control and analyze.

[3] We read [4] after implementing this algorithm and were surprised to see that it's exactly the same as what Wilson et al have come up with.

[4] Each object has a pointer to the *vtable* for its class as its first field.

Section 4.1.1: Round 1: micro-benchmarks

For our experiments we used a Pentium II with 128Mb of RAM running Linux. Both L1 and L2 caches are 4-way set associative with line size of 32 bytes. The size of L1 and L2 cache is 16K and 512K, respectively. The system we used for benchmarking had a Linux kernel patch installed, which allowed us to query hardware performance counters for the number cache misses. Unfortunately, the Pentium chip has only two counters and we needed to record two data point to measure the miss ratio (the number of misses and the total number of requests) for each cache, so we had to run each program twice with each input to get miss ratios for both caches.

There were two C benchmarks we used: `full_walk` and `random_walk`. Each benchmark consists of two stages. The first stage is the same for both programs:

- Allocation stage: Allocate a balanced binary tree of depth *d* in
  - depth-first
  - breadth-first[5] or
  - depth-first right-to-left fashion
  - using hierarchical decomposition

The second stage is different for the two benchmarks:

For `full_walk`:

- Traversal stage: Walk the whole tree it depth first (left-to-right)

For `random_walk`:

- Traversal stage: Perform a random walk down the tree

Allocation is performed using a big contiguous block of memory we `malloc()` in the very beginning of the program to make sure allocation is linear. Node size is 16 bytes, so exactly two objects fit on a cache line.

Traversal stage is repeated *n* times in a loop. We are only timing stage two of our benchmarks because different allocation strategies can potentially take different times to run. We also turn the hardware counters on only for the duration of the traversal stage.

Section 4.1.2: Discussion of Results

We expected to see a very distinct difference in performance between the copying strategies. Indeed, `full_walk` should heavily favor depth-first traversal over breadth first traversal and depth-first right-to-left should create a very high number of cache misses because the traversal order is just opposite the allocation order.

However, to our astonishment, we saw that there is *no significant difference* between the running time of different methods! Figure 1 shows how the runtime changes for different values of tree depth. In our experiments, the tree depth ranges from 5 to 21. For tree depth bigger than 21, tree doesn't fit into the memory and the system starts doing a lot of swapping and we don't really want to consider page behavior.

Since we don't have any paging in our benchmarks, not counting initial page faults at allocation time, we need to analyze

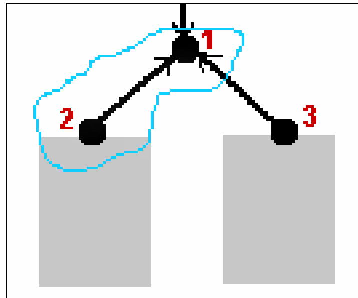
[5] We are using a next pointer in each node pointing to its right sibling to avoid using auxiliary data structures in the heap which would interleave with the tree.

cache behavior to find out why different allocation methods don't affect the runtime.

Figures 2 and 3 show the number of cache misses for `full_walk` for L1 and L2 cache, respectively, for different depths of the tree. The size of L1 cache is 16K, so approximately 10 top levels of the tree will fit into the cache completely. After that we see that there's a rapid increase in the cache miss ratio, which stabilizes at 3.5%. The basic question is, however, why are the miss ratios the same for all allocation strategies? Let's break our analysis into several parts.

DF left-to-right vs DF right-to-left

Consider the picture below. Assume for this analysis that exactly two objects fits onto a cache line[6].



We will show that the number of cache misses should be the same for both traversal orders.

Suppose we are walking this tree left-to-right. Then 1 and 2 are on the same line and 3 is on a different one. We will get a miss when we are coming to the root of the tree, node 1. 1 and 2 are on the same cache line, so we are not going to get a miss when we access 2. We are also going to get a miss when we access 3, since it's on a different cache line. We can count the precise number of misses:

$$T_{LR}(d) = 1+(T_{LR}(d-1)-1) + T_{LR}(d-1) = 2T_{LR}(d-1)$$

The first term corresponds to the initial miss, terms two and three correspond to the number of misses in the left and right subtree, respectively. With an appropriate initial condition, this yields $T(d)=2^d$. But the derivation is exactly the same for right-to-left traversal! The formula becomes:

$$T_{RL}(d) = 1+ T_{RL}(d-1) + (T_{RL}(d-1)-1) = 2T_{RL}(d-1),$$

yielding $T_{RL}(d) = 2^d$. This explains why the number of misses for both traversals is exactly the same even though their *order* is different.

We are using several assumptions here: first, we are assuming that both subtrees have the same kind of layout as the whole tree, which they don't. Consider, for instance, the subtree when the tree has been allocated left-to-right. Its root, node 2, shares a cache line with node with 1 instead of sharing a line with its left child, like 1. But this shouldn't really matter at least for big trees, since, in principle, the layout is the same. The formula for T(d) above is approximate, but since the number of memory requests is large most of the time[7], we don't see much of a difference between left-to-right and right-to-left allocation.

The second assumption is that when we are done traversing the left subtree, node 1 is still going to be in the cache when we access it to reference node 3. This is so for small trees that completely fit into

[6] We can perform similar analysis for other object sizes. It is impossible to fit more than five objects on a line, which corresponds to one word per object. It is not clear why the cache lines on Pentium are so small.

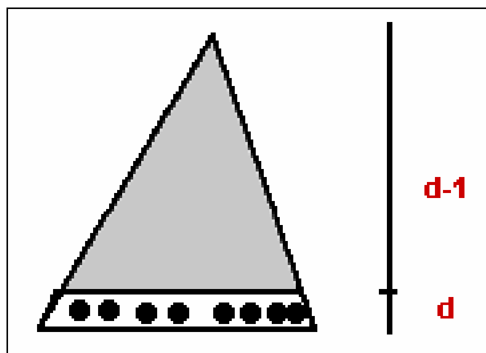
[7] The number of memory references in L2 cache is small for small values of *d*, that's probably why we are seeing some discrepancy in the first zone of the graph for L2 misses in fig. 4.

the cache. If the tree is bigger than that, this is going to be the case when the left subtree fits into the cache, which is the case for subtrees of size 10 or smaller (for L1 cache). So all nodes located at level *d-10* or lower where *d* is the depth of the tree fits into the cache. But an easy calculation shows that the top 10 levels are only a small portion of the whole tree (about a thousandth), so the misses occurring there shouldn't really change the situation significantly.

Another way to see that the number of misses should generally be the same is to note that *every other* node should create a miss independent of traversing order. This is because every node is going to be on the same line with one of its children (taking into account the assumption above). That's why the number of misses is approximately half the size of the tree in our derivation of *T(d)* above.

DF vs BF

We can compute the number of misses created after a BF allocation inductively. We can write an inductive expression for the number of misses:



T(d) = T(d-1)+ (size of the bottom level)/2

This is because when allocation is breadth-first, *every other* node at the bottom level causes a miss. For a binary tree, the size of the bottom level is the same half the size of the tree, so

$$T(d)=T(d-1)+2^d,$$

which means that $T(d) = 2^d$. So the number of misses is the same as in the case of depth-first allocation.

The formula for *T(d)* is exponential in the depth of the tree, which explains why the miss ration becomes essentially constant. Indeed, the number of memory accesses in also exponential (it's proportional to the number of nodes) and the ratio must be constant. Counting the number of references for each node in the generated assembly code helps explain why the miss ration is around 3.5% (for L1 cache).

Similar analysis applies to L2 cache and we can make the same sort of conclusions.

Other issues

It is not clear, however, why hierarchical decomposition gives the same number of cache misses for `full_walk` as the other two strategies.

In the case of `random_walk` we see that hierarchical decomposition gives us the best performance both in terms of overall runtime (fig. 4) and cache performance (figs. 5 and 6). Depending on the size, the difference in miss ratios can be as much as 20%. It is not quite clear, however, why breadth-first copying performs *better* than depth-first.

To summarize, if we look at tree traversal and walk as two basic

patterns of tree-like data access[8], changing the copying order is likely to help a bit in the case of random walk, but doesn't really matter for full travesal. This is consistent with runtime data we are presenting below.

Section 4.2.1: Round 1: Macro-benchmarks

Next, we tested different copying strategies on different Java programs to see if there would be one particular copying strategy that would perform better in most cases[9]. Runtime data is summarized in fig. 7 and 8 in appendix B.

We used a combination of medium-size Java programs we wrote and existing real benchmarks. Below are descriptions on each.

JLex is a lexical analyzer generator. It takes in a grammar and outputs a Java source file for the lexer. To increase the runtime, the main analysis is performed 100 in a loop. The program allocates a good deal of data and collects in every iteration of the loop.

Jess: Jess is a theorem proving system. It has several hundred classes and makes extensive use of vectors, hash tables lists and trees.

DaveTest is a Java test designed to test whether different data structures should be copied differently during the garbage collection time using different strategies. We specifically designed DaveTest to have all possible kinds of shapes[10].

First, a tree of depth 20 is created in a depth-first fashion, the garbage collector is called, then a full depth-first traversal is made. Next, a DAG of depth 20 is created in a tree-like manner in which a particular node has a pointer to two children, but it also has a pointer to its right sibling. Again, the garbage collector is called, then the DAG is then traversed along the sibling pointers, so it is essentially traversing all of the nodes for a particular height. Finally, a cycle of 300,000 elements is created so that it is in the shape of a daisy. The structure contains local cycles of size 1000 and each of the local cycles has a pointer back to a central node. When we do a traversal of this cycle, we walk along the leaves of this daisy, then move onto a different local cycle. First, we ran DaveTest using the same copying strategy for all the structures, then we tested using different copying strategies for different structures.

The size of a tree node in this test is 24 bytes, while the size of a DAG node is 28 bytes, and the size of a daisy node is 24 bytes.

LinkIt is a test specifically designed to show that in some cases, breadth first copying should perform much worse than depth first copying or hierarchical decomposition. First, an array of linked lists is created. The array is 600 elements long and each linked list contains 800 elements. After the creation, the garbage collector is called, then the program randomly traverses a particular linked list and

[8] These access patterns correspond to basic database operations if the database index is organized as a tree.

[9] We used the same sort of hardware for these macro-benchmarks as before.

[10] Most real programs that we tried only had trees, so we needed an artificial benchmark for our tests.

this traversal is repeated 1000 times. The size of each node is 20 bytes.

Full traversal and **random walk** are Java version of the C micro- benchmarks described above. They create trees of size 18 and 20, respectively. After tree allocation, the collector is invoked and then we perform either a full depth-first traversal or a random walk and time it.

Section 4.2.2: Discussion of Results

JLex: We see some pretty significant variation among different strategies for this benchmark – 16%. Looking at the source code reveals that most of the data structures used are arrays, Vectors (also implemented as arrays) and hash tables. We think that depth-first copying may produce a better layout of hash tables.

Jess: There is very little difference in runtimes for Jess – about 2.7%. The fact that Cheney wins in this case doesn't have that much significance.

DaveTest: In this case if we use the same strategy for all types of structures, depth-first traversal wins. This could be because daisy-traversal favors depth-first traversal over the other two strategies. The variations in data are on the range of 20%.

DaveTest was the only program we have experimented with that has all three kinds of data structures. We see that picking a mixed strategy (strategy 2 in this case, see fig. 8) can result in an improvement over a fixed strategy (depth-first), but the improvement in our case is only 2.7%. Variation in runtime among different policies is about 18%.

LinkIt: We see that hierarchical decomposition performs best for this test. We know that breadth-first performs the worst, which is what we expected because subsequent nodes in each of the lists are located far from each other.

It is not clear, though, why depth-first performs worse than hierarchical decomposition. We also saw that increase in page sizes yielded better performance. It's also worth noticing that applying different strategies really makes a difference in runtimes – the change in non-GC runtimes between HD (the best) and Cheney (the worst) is 27%!

Full traversal: We see that Cheney is the best approach in this case, but that doesn't have much of a significance, because the variation in non-GC times for this test are just 1.5%, so the differences can be caused by many factors we are not taking into account or just random noise. This matches our conclusions for `full_walk` described above.

Tree search: hierarchical decomposition is the best method to use in this case and variation in runtime is about 8%. The fact that hierarchical decomposition is the best approach is consistent with the cache miss data in appendix A.

Section 5. Conclusions

While code optimization techniques have been studies extensively, as the processors ate getting faster and faster, it's memory hierarchy that will produce significant differences in program runtimes. We discovered that using different copying strategies can make a significant difference for some kinds of programs. In our benchmarks we got up to 27% improvement in non-GC runtimes using various copying stratgies. The

overhead of a more complicated copying algorithm may be pretty high, however[11], especially for depth-first copying.

There is no copying strategy that wins most of the time. Therefore, implementation of an efficient copying collector would allow using different strategies for each object, which is what we implemented.

Shape analysis can yield some measurable performance improvements in runtime. In our experiments (see DaveTest above) we get about 18% of improvement using one copying policy over another. This implies that experimentation with policies is generally beneficial. We must remark, however, that shape analysis is a pretty crude tool to use is this context. For most real programs the analysis deems all structures to be trees. In addition to this, it may well be the case that in a big program different tree-like structures will be used completely differently. This suggests that we should have a finer-grained specification of policy. One approach would be to use per-class pragmas telling the system how objects that class should be copied. That would avoid the need for runtime profiling used by Larus [6,7,8], which if, of course, another way to find out those pragmas.

[11] Looking at GC times in appendix B reveals that depth-first copying is quite time-consuming in most cases. This is probably due to the fact that we have a lot of recursive invocations in our implementation and, since we didn't turn on the optimizations, the compiler couldn't use tail recursion elimination. A better implementation would use an explicit stack to keep track of recursion.

Section 6. Related work

The first series of papers we have looked at was based on work by Paul Wilson. In [2] he outlines many important issues that the collector has to face. However, his studies were performed in a somewhat different context. Let me outline some of the major differences.

He considers the effect of virtual memory behavior, i.e., paging. The (relatively small) programs one would typically write in Java wouldn't create so much live data as to overflow the main memory. As noted above, our emphasis was on cache performance rather than paging.

Wilson is looking at the system image, that is, "the relatively persistent heap data that make up applications, development tools, and the systems that support them." He remarks that for programs that have fast-cycling data the access pattern is quite different. In fact, he even suggests looking at the cache behavior for such programs.

Systems images considered in [2] are typical of big Lisp systems and don't have to bear any resemblance to what (even reasonably large) Java programs are producing. In particular, Wilson gets a big performance gain from special treatment of upper-level hash tables.

Wilson concludes that his scheme can decrease the number of faults by up to a third, for smaller programs. It is not always clear, however, how much of this improvement is due to changing the order of traversal and how much to reorganizing top-level data structures.

In [4] Wilson considers the effect of some basic runtime type analysis on page locality.

Unfortunately, we ran across this paper pretty late in the term; we discovered that his collector is using basically the same algorithm we were using. He considers a database benchmark, which creates a huge balanced tree with many of the leaves linked together. If the number of accesses between the leaves is large, hierarchical decomposition is going to perform worse than Cheney that would just place all the leaves sequentially.

Despite this, his analysis, which looks at runtime organization of the data structures concludes that the structure *is* a tree (the algorithm only looks at the first several levels of indirection)! However, he only considers these three database operations: lookup, traversal and insert (lookup and insert shouldn't be much different) and the effects of different strategies on the number of page faults. These experiments show that we can see any significant difference only for small memory sizes. On lookup and insert hierarchical decomposition performs *worse* than depth first and better than breadth first traversal. On database traversal breadth-first is worse than depth-first and hierarchical decomposition, which perform identically. Unfortunately, the authors don't give a satisfying explanation as to why this is the case.

In [5], there are various points to consider in relation to our implementation. The algorithm described in this paper was designed for parallelism, but we adopted it to copying garbage collection. Also, their shape analysis algorithm is interprocedural and uses Ghiya and Hendren's "Points-to Analysis" work [5] in order to do it. For us to implement "Points-to Analysis", it would have required a lot of extra work, and we believe that the benefits from this additional work are not going to be great. The authors note that their algorithm is conservative. If a program builds a tree-like data structure by appending at the beginning or end of the existing structure, then shape analysis will successfully infer the shape of this data structure is a tree. If a tree or DAG-like data structure is built by inserting new nodes between existing nodes, shape analysis provides conservative estimates and reports the shape to be DAG-like or cyclic. One criticism of the paper is that the authors mention that this is used to help parallelism, but they don't provide any results to that effect.

Also, the authors note that shape analysis is only useful to them when the algorithm detects that a structure was a tree. When it was a DAG or cycle, they noted that shape information was either only slightly useful or not useful at all. They probably intended to use shape analysis as a check to see whether a program could be made parallel. If the structure was a DAG or cycle, then they could not parallelize because different threads might try to modify the same place in memory. Also, they note that the direction and inference relationships can be useful on their own. To identify if data structures accessible from two pointers, say `p` and `q`, share a node, one needs to simply check if the interference matrix entry `I[p,q] = 1`. Given this data, we can attempt to parallelize a program. Also, direction matrix information can aid the programmer in safely deallocating memory. At a call-site like `free(p)`, if any pointer can have a path to p, then it may not be a safe deallocation. In Java,

this could be useful in garbage collection in that we could use the direction matrix information in addition to the root set information to figure out what is live and isn't live.

We also looked at many of James Larus' papers for more background information on cache-conscious data structure research. In [6], Larus goes over the implementation of two semi-automatic tools for improving cache performance. The paper notes that cache-conscious trees outperform their native counterparts by a factor of 4-5[12]. Semi-automatic cache-conscious data placement improves performance 28-194%, and even outperforms state-of-the-art prefetching by 3% - 194%. One of the tools, `ccmorph`, reorganizes a data structure using clustering and coloring. Clustering attempts to pack in a cache block data structure elements likely to be accessed contemporaneously. Coloring is used because caches have finite associatively, which means that only a limited number of concurrently accessed data elements can map to the same cache line without incurring conflict misses. `ccmorph` operate on trees and the programmer must supply it a pointer to the root of a data structure, a function to traverse the structure, and cache parameters. `ccmorph` affects correctness in that if the structure is not a tree, it may get restructured incorrectly. The other function, `ccmalloc`, attempts to locate the new data item in the same cache block as the existing item. It takes an additional parameter that points to an existing data structure element likely to be accessed contemporaneously.

In [7], Larus explores the usefulness of splitting large data structures and reordering fields within a large data structure. If the size of a data structure is comparable to the size of a cache block, splitting structure elements into a hot and cold portion can produce hot structure pieces smaller than a cache block, which allows for better caching. To implement structure splitting, the first step is to profile a program to determine member access frequency. These counts identify class member fields as hot or cold. A compiler extracts cold fields from the class and places them in a new object, which is referenced indirectly from the original object. Accesses to cold fields require an extra indirection to the new class, while hot field accesses remain unchanged. Class splitting combined with Chilimbi and Larus' cache-conscious object collocation reduced L2 cache miss rates by 29-43%, wich class splitting accounting for 26-62% of this reduction, and improved performance by 18-28%, with class splitting contributing 22-66% of this improvement. When structure elements span multiple cache blocks, reordering structure fields to place those with high temporal affinity in the same cache block can also improve cache block utilization.

In [8], Larus discusses generational garbage collection and a new copying algorithm to use with it. In generational garbage collection, the heap is broken up into a number of generations. Youngest generations hold the most recently allocated objects. Objects that survive repeated scavenges are promoted to older generations. Promoting something to an older

[12] Surprisingly, this doesn't nearly match our experience with different layouts.

generation entails moving the object. We only move small objects. Large objects are not moved. Larus' copying algorithm uses object affinity graphs to determine which objects should be placed in the same cache block. The object affinity graphs are created using real-time profiling. There is a separate affinity graph for each generation. When it comes time to move objects from fromSpace to toSpace, copying is based on the object affinity graph.

Appendix A: Performance Data for Micro-benchmarks

`full_walk` performance data:

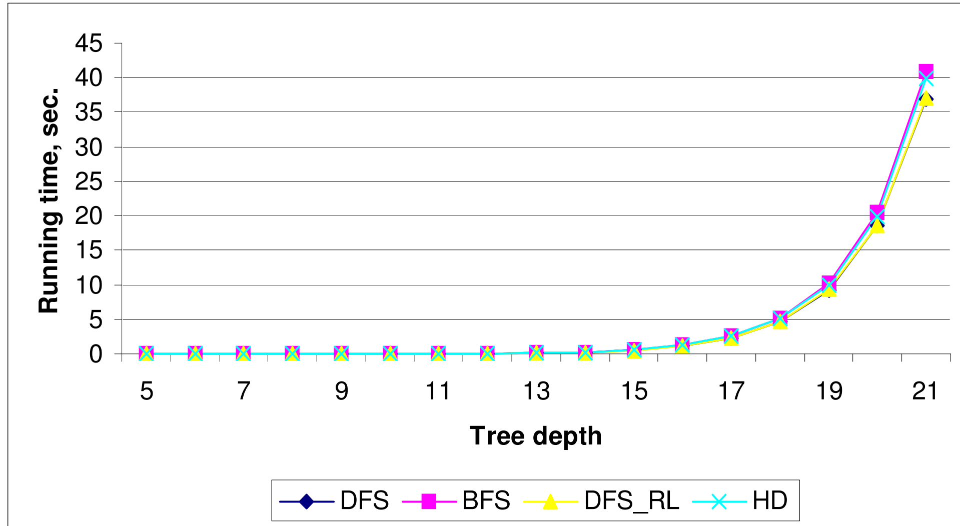


Figure 1. Running times for `full_walk`

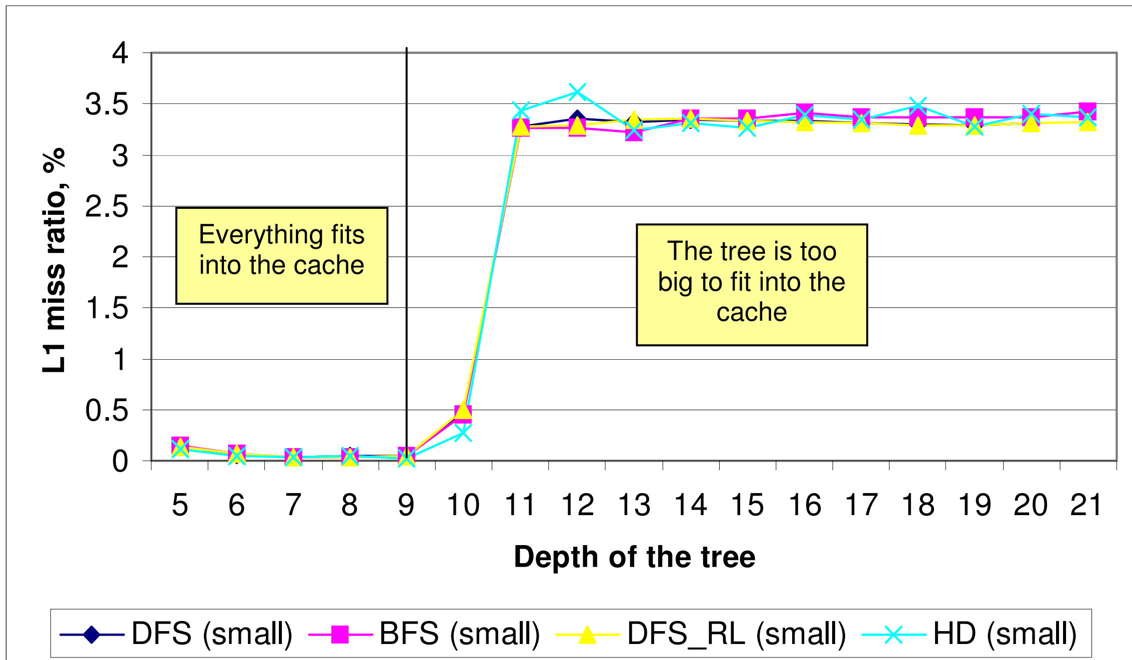


Figure 2. L1 miss ratios for `full_walk`

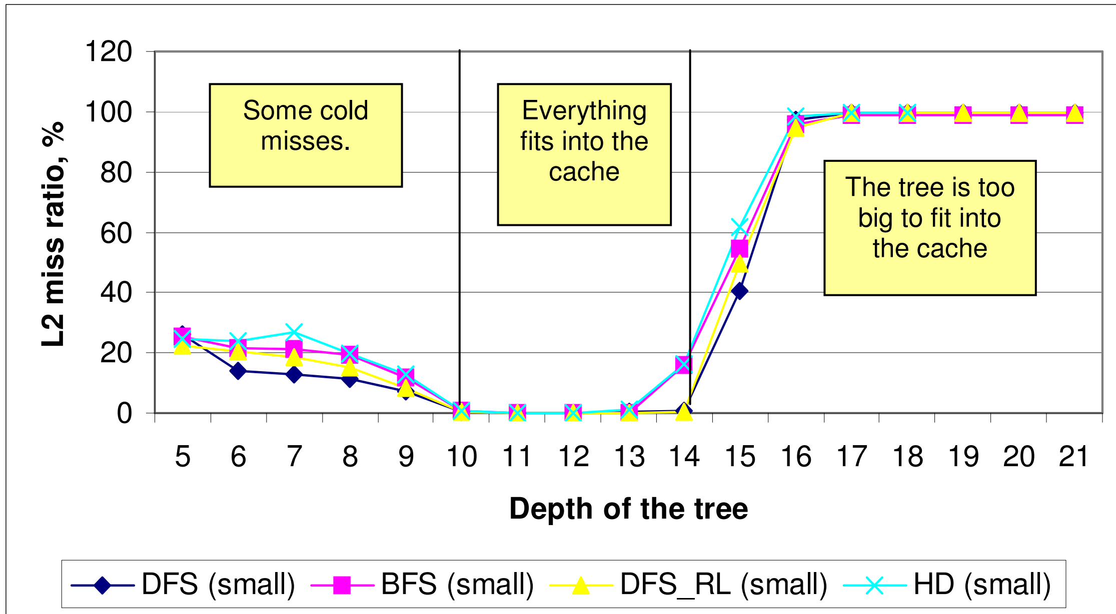


Figure 3. L2 miss ratios for `full_walk`

`random_walk` performance data:

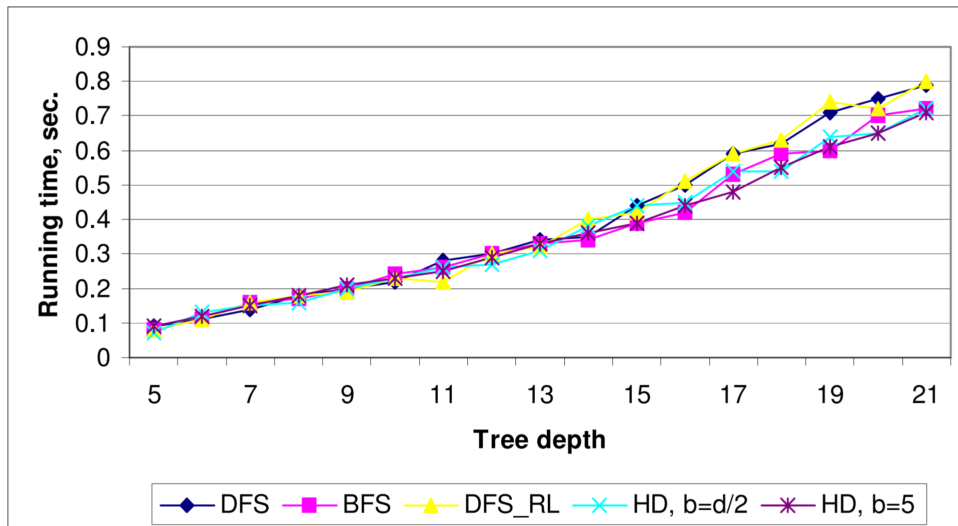


Figure 4. Running times for `random_walk`

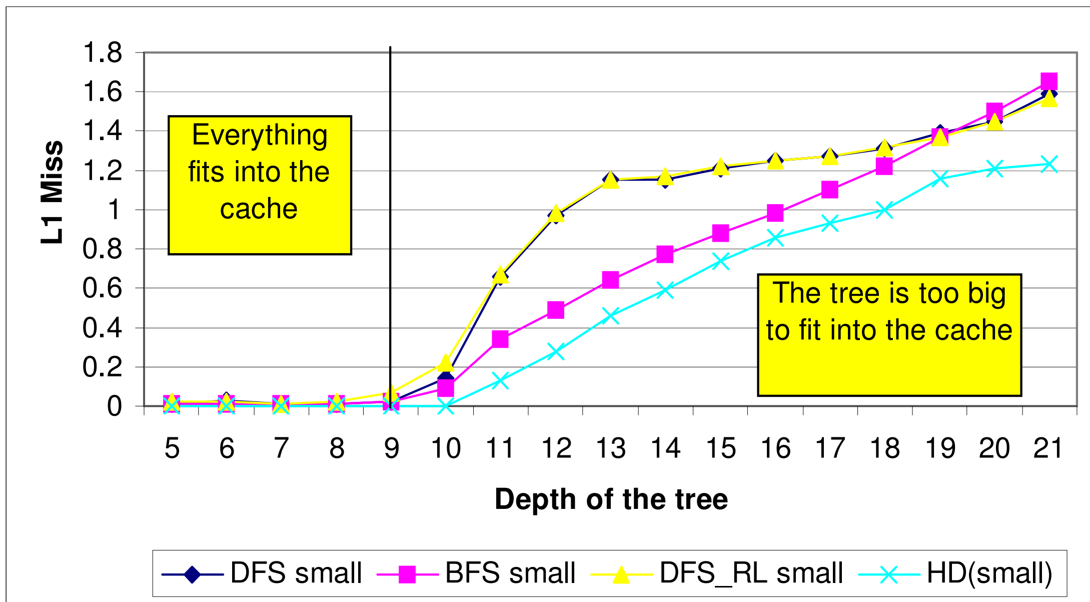


Figure 5. L1 miss ratios for `random_walk`

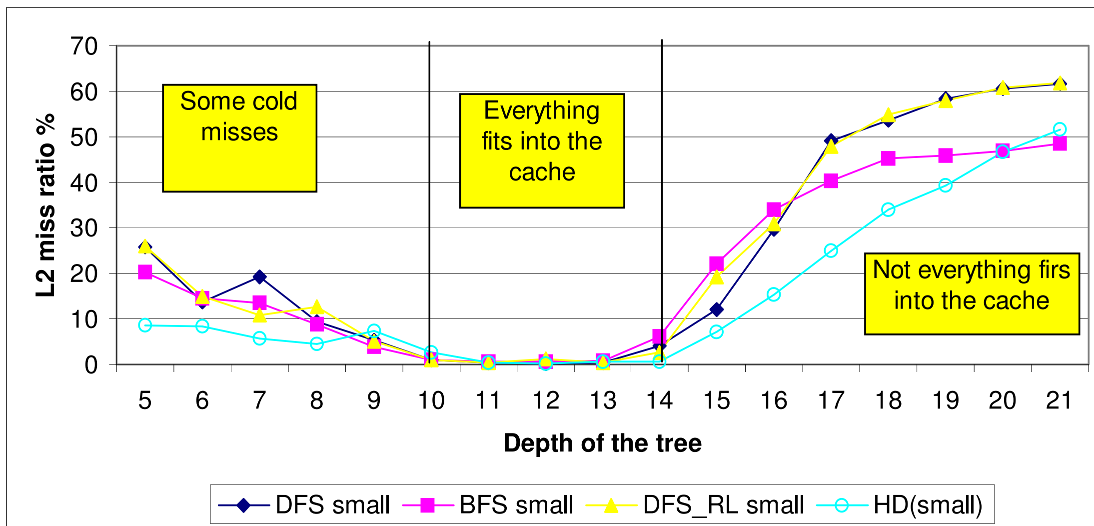


Figure 6. L2 miss ratios for `random_walk`

Appendix B: Performance data for macro-benchmarks.

Strategy / Test		jlex100	jess	DaveTest	LinkIt	full traversal	random walk
Cheney	Total RT	37.814	7.986	20.062	1.746	2.338	1.427
	GC-time	0.377	0.231	15.735	0.366	0.466	0.967
	Non-GC RT	37.437	7.755	4.327	1.38	1.872	0.46
DFS	Total RT	38.896	10.754	78.015	13.729	17.399	27.554
	GC-time	3.06	2.882	73.983	12.368	15.517	27.064
	Non-GC RT	35.82	7.872	4.032	1.361	1.882	0.49
HD	Total RT	38.789	9.543	35.424	3.92	9.533	14.06
Page Size = 128K	GC-time	1.997	1.638	30.393	2.809	7.631	13.605
	Non-GC RT	36.792	7.905	5.031	1.111	1.902	0.455
HD	Total RT	42.384	9.288	31.064	2.749	8.086	11.321
Page Size = 256K	GC-time	1.53	1.382	26.241	1.657	6.204	10.871
	Non-GC RT	40.854	7.906	4.823	1.092	1.882	0.45
HD	Total RT	42.817	9.168	25.323	2.168	6.854	14.025
Page Size = 512K	GC-time	1.276	1.196	20.646	1.087	4.977	13.56
	Non-GC RT	41.541	7.972	4.677	1.081	1.877	0.465

Figure 7. Times for various Java tests (in seconds). Best non-GC times for each test is highlighted in gray.

Strategy / Test		DaveTest
Cheney	Total RT	20.062
	GC-time	15.735
	Non-GC RT	4.327
DFS	Total RT	78.015
	GC-time	73.983
	Non-GC RT	4.032
HD	Total RT	35.424
Page Size = 128K	GC-time	30.393
	Non-GC RT	5.031
HD	Total RT	31.064
Page Size = 256K	GC-time	26.24
	Non-GC RT	4.824
HD	Total RT	25.322
Page Size = 512K	GC-time	20.646
	Non-GC RT	4.676
Strategy 1	Total RT	45.728
Tree = DFS; DAG = HD; Cycle = HD;	GC-time	41.089
Page Size = 512K	Non-GC RT	4.639
Strategy 2	Total RT	73.755
Tree = HD; DAG = DFS; Cycle = HD;	GC-time	69.83
Page Size = 512K	Non-GC RT	3.925
Strategy 3	Total RT	55.759
Tree = DFS; DAG = HD; Cycle = CH;	GC-time	51.249
Page Size = 512K	Non-GC RT	4.51

Figure 8. Times for DaveTest using different copying strategies (in seconds). Best non-GC time is highlighted in gray.

Section 8. References.

1. R. Jones, R. Lins, “Garbage Collection, Algorithms for Automatic Dynamic Memory Management”
2. P. Wilson et al, “Effective “Static-graph” Reorganization to Improve locality in Garbage-Collected Systems.”
3. P. Wilson et al, “Caching Considerations for Generational Garbage Collection.”
4. P. Wilson et al, “Object Type Directed Garbage Collection to Improve Locality.”
5. Ghiya, Rakesh and Laurie J. Hendren. “Is it a Tree, a DAG, or a Cyclic Graph?”
6. Chilimbi, Trishul , James Larus , Mark Hill. “Tools for Cache-Conscious Data Structures.”
7. Chilimbi, Trishul , Bob Davidson, James Larus. “Cache-Conscious Structure Definition.”
8. Chilimbi, Trishul , James Larus. “Using Generational Garbage Collection To Implement Cache-Conscious Data Placement.”
9. Bob Fitzgerald, Erik Ruf, David Tarditi, “Marmot: an Optimizing Compiler for Java”