Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times

Yefim Shuf [¶] [§] Manish Gupta [¶] Hubertus Franke [¶] Andrew Appel [§] Jaswinder Pal Singh [§]

[¶] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
{yefim, mgupta, frankeh}@us.ibm.com

[§] Computer Science Department
Princeton University
Princeton, NJ 08544
{yshuf, appel, jps}@cs.princeton.edu

ABSTRACT

The growing gap between processor and memory speeds is motivating the need for optimization strategies that improve data locality. A major challenge is to devise techniques suitable for pointer-intensive applications. This paper presents two techniques aimed at improving the memory behavior of pointer-intensive applications with dynamic memory allocation, such as those written in Java. First, we present an allocation time object placement technique based on the recently introduced notion of *prolific* (frequently instantiated) types. We attempt to co-locate, at allocation time, objects of prolific types that are connected via object references. Then, we present a novel locality based graph traversal technique. The benefits of this technique, when applied to garbage collection (GC), are twofold: (i) it improves the performance of GC due to better locality during a heap traversal and (ii) it restructures surviving objects in a way that enhances locality. On multiprocessors, this technique can further reduce overhead due to synchronization and false sharing. The experimental results, on a well-known suite of Java benchmarks (SPECjvm98 [26], SPECjbb2000 [27], and jOlden [4]), from an implementation of these techniques in the Jikes RVM [1], are very encouraging. The object co-allocation technique improves application performance by up to 21% (10% on average) in the Jikes RVM configured with a non-copying mark-and-sweep collector. The locality-based traversal technique reduces GC times by up to 20% (10% on average) and improves the performance of applications by up to 14% (6% on average) in the Jikes RVM configured with a copying semi-space collector. Both techniques combined can improve application performance by up to 22% (10% on average) in the Jikes RVM configured with a non-copying mark-and-sweep collector.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection), run-time environments, optimization, compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies, garbage collection*

General Terms

Languages, Algorithms, Performance, Experimentation

Keywords

Java, JVM, locality, memory management, run-time systems, memory allocation, garbage collection, object co-allocation, object placement, heap traversal, locality based graph traversal, prolific types

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

1. INTRODUCTION

The growing disparity between processor and memory speeds is motivating a strong need to optimize the memory behavior of programs. Numerous researchers have studied data locality, its importance and its effect on performance, and have investigated interactions of programs with processor architecture as well as the memory system. While work on optimizing array-based applications with regular access patterns has yielded excellent results, improving the performance of pointer-intensive workloads with dynamic memory allocation still represents a serious challenge. Although much progress has been made in the latter area, researchers seem to agree that more work is needed to achieve desirable levels of performance. Our work is an attempt to contribute to this important effort.

The growing popularity of languages like Java [15] on a wide variety of platforms, ranging from embedded systems to servers, presents a particular challenge from two points of view. First, Java programs tend to make extensive use of heap-allocated memory and typically have significant pointer chasing, which puts pressure on the memory subsystem [23]. Second, the dynamic nature of Java (due to features like dynamic class loading) makes it impractical to apply optimizations requiring whole program analysis and other extensive analyses.

In this paper, we present two software-based techniques for creating and preserving data locality in pointer-based applications that make heavy use of dynamically allocated memory and rely on automatic memory management, such as applications implemented in Java. Our techniques are inherently simple and efficient, in that they rely on a "macroscopic" view of the program rather than on detailed program analysis. The first technique relies on identification of frequently instantiated types, termed *prolific types* [22], of the given program, and tries to co-locate objects of prolific type together *at allocation time*. The second technique relies on a novel memory-friendly traversal algorithm *during garbage collection* (GC),

which improves GC performance. When used with a copying collector, this technique tends to further improve the data locality for the remainder of program execution by bringing closely related objects even closer in memory. In spite of the apparent imprecision of the analyses guiding these optimizations, we show that our techniques are quite effective.

Thus, our techniques cover two natural points during program execution when the placement of data can be improved: allocation time and GC time. Rearrangement of data at GC time is inherently useful only for relatively long-lived objects that survive GC. Improving the locality of references to short-lived data clearly needs to be done at or close to the object allocation time. Interestingly, our allocation time technique targets objects of prolific types, which tend to be short-lived [22].

Allocation time approach Our first technique is a novel object co-allocation scheme, which relies on the notion of *prolific types* [22] to place related objects close to each other in memory. It has been observed that for most programs, a small number of object types, referred to as *prolific types* for that program, account for a large fraction of objects allocated at run time. Objects of prolific types tend to have short lifetimes, which makes them a suitable target for frequent garbage collections. While the main focus of [22] was to demonstrate applications of this concept for garbage collection, *write barrier elimination*, and reducing application memory requirements, in this paper, we exploit the idea of prolific types for making simple and effective *object co-allocation* decisions, which benefit data locality.

Given the differences in the number of instances of objects of prolific and non-prolific types, we postulate that it is more natural to co-allocate two objects of prolific types that point to each other, than to co-allocate an object of a non-prolific type with an object of a prolific type. Since most allocated objects are those of prolific types, focusing co-allocation efforts on objects of those types seems to be a prudent choice.

Consequently, we devise a *type affinity* based *object co-allocation* scheme in which related objects of prolific types are partitioned into clusters of parents and children. We create and (with the help of simple compiler analysis) inline specialized allocation routines such that when a parent prolific object is created, enough space is reserved right next to that object for its child (or children). When it is time for a child object to be created, it is placed right next to its parent rather than into some arbitrary memory block. Co-allocation of parent objects and children objects that cross-reference each other has a number of important benefits:

- It improves the spatial locality of reference by ensuring that related objects that are connected via object references are stored next to each other. Objects that point to each other are often accessed contemporaneously [17]. Hence, this technique reduces the negative impact of pointer chasing.
- It reduces GC time, since co-allocated live objects form bigger clusters. Marking clustered objects increases the locality of garbage collectors, whose locality is known to be worse than that of applications [19].
- It often reduces memory fragmentation, particularly when used in a system with a non-copying collector. At allocation time, a space within a memory block that would otherwise go unused can be taken by an object from a cluster of related objects. At garbage collection time, since objects that are born together tend to die together [17], and since memory blocks occupied by dead co-allocated objects are considerably larger than those occupied by individual objects, freeing those larger blocks would leave fewer small fragmented blocks.

This co-allocation heuristic can potentially waste memory if the corresponding child object is never allocated (we attempt to limit such memory fragmentation, as discussed in Section 2.4). Also, this approach would not improve locality if the parent and child objects are not accessed contemporaneously. However, it seems to work well in practice. Experimental results demonstrate that our object co-allocation technique improves application performance by up to 21% (10% on average) for the Jikes RVM [1] configured with a non-copying mark-and-sweep collector. No improvements are observed for the configuration with a copying collector, since the memory allocator used in that configuration already leads to good locality.

Garbage collection time approach Our second technique is a locality-based algorithm for traversing reachable objects at GC time, which aims to achieve shorter GC delays (a critical requirement for some applications). A key observation is that since all reachable objects (which can be large in number) that reside in the region of the heap being collected have to be visited, a traversal algorithm that exploits the *inherent locality* of reachable objects will improve the performance of the GC itself and that of the application. Since there is little data reuse during a traversal (many reachable objects will be visited only once), it is essential to ensure that the traversal algorithm interacts well with the memory subsystem (e.g., exploits locality with respect to a data cache, a data TLB, and hardware prefetching).

Consequently, our *locality-based traversal* algorithm visits objects that reside close to each other in memory first before visiting their distant children and siblings. At each step, the algorithm picks a region of objects to traverse. When all objects within that region are marked, it moves on to another region, again performing a local traversal. This radically different traversal technique has several useful properties:

- It will have inherently better memory behavior than other existing algorithms, which blindly traverse the graph of reachable objects (e.g., even if objects reside far apart) while ignoring the principles of both spatial and temporal locality.
- When combined with a copying collector, it can also improve locality of the resulting data layout by ensuring that objects that used to reside closely to each other stay or get closer to each other in memory. It does not rely on monitoring memory accesses, nor does it incur overheads associated with construction and sorting of a temporal relationship graph.
- It is well-suited for efficient traversal of both young and old objects.
- In a parallel GC, by directing each GC thread to work on a different region, it can substantially reduce overheads associated with synchronization and false sharing during garbage collection.

Our scheme is simple and can be used as a plug-in replacement for any exhaustive graph traversal algorithm where the order of traversal does not matter. Experimental results are very encouraging. The overhead of additional processing of pointer targets is more than compensated by the improvement in locality during traversal. The locality-based traversal scheme is shown to reduce GC times by up to 20% (10% on average) and improve application performance by up to 14% (6% on average) in the Jikes RVM

configured with a copying semi-space collector. The combination of object co-allocation and memory-friendly traversal can improve application performance by up to 22% (10% on average) in the Jikes RVM configured with a non-copying mark-and-sweep collector.

Organization The rest of the paper is organized as follows. Section 2 discusses the allocation time technique for co-allocating objects of prolific types. Section 3 discusses our new technique for traversing reachable heap objects at GC time. Experimental results obtained from an implementation of both techniques are described and analyzed in Section 4. Section 5 presents comparisons with related work. Finally, we summarize the results of our work and present ideas for future research in Section 6.

2. CREATING LOCALITY AT ALLOCATION TIME

In this section, we present the details of our object co-allocation scheme. Pointer-intensive applications, such as those written in Java, tend to have poor locality of accesses to heap-allocated objects. Java objects are linked to each other via references and many memory accesses traverse such links [23] (this behavior is called *pointer chasing*). A program whose linked objects are scattered around the heap will interact poorly with the memory system (e.g., it may cause data TLB thrashing). Co-allocating linked objects will improve both temporal and spatial locality.

A major challenge is to find an efficient and effective heuristic that will produce reasonably good co-allocation decisions. Our technique is using the notion of *prolific types* to decide what objects are to be co-allocated.

2.1 Prolific types

It has been observed that although Java applications may have a large number of object types, a fairly small number of object types produce a disproportionally large number of instances. Such types are referred to as *prolific types* [22] and can be efficiently identified via on-line or off-line profiling of allocation requests or sampling[1]. The remaining object types, which do not produce many instances, are termed *non-prolific*.

Prolific types have a number of useful properties and applications to memory management (e.g., garbage collection, *write barrier elimination*, reduction of heap space requirements, object recycling) [22]. The *object co-allocation* scheme presented here is yet another application of prolific types.

While our experiments have been done on middle-sized programs, we believe the notion of prolific types is also applicable to large programs. Recently, Chevalier et al. [7] have demonstrated that it does indeed apply to large-scale Java applications (each application had over 20K lines of code, not counting comments) and large programs written in Haskell. What is even more important is that the applications used in their study were very sensitive to input. They also empirically validated various properties of prolific types discussed in [22] (e.g., the number and the fraction of prolific types is small, instances of prolific types tend to have short lifetimes – the *prolific hypothesis*, instances of prolific types tend to be small, and children of prolific types tend to be prolific). Further, they showed that programmers can identify which types in a program are prolific and non-prolific (without using profiling) and use that information for program annotation – the prolificity annotation hypothesis. In other words, their experiments suggest that the notion of prolific types is quite intuitive to programmers.

[1]Misclassification of object types does not create a correctness problem.

2.2 Type affinity based co-allocation

Considering our classification of objects into instances of prolific and non-prolific types, we argue that objects of prolific types should be allocated together with other objects of prolific types. First, in looking for instances of objects which are likely to be accessed together (such as objects connected via reference fields), there is a greater chance of a *one-to-one* relationship between objects of prolific types than between prolific and non-prolific objects (since there are many more prolific objects than non-prolific objects). Second, given the larger number of objects of prolific types, greater performance benefits may be achieved by focusing the effort on co-allocating prolific objects.

2.3 The co-allocation algorithm

We formulate the problem of co-allocating objects of prolific types as a graph partitioning (or node clustering) problem. Consider a directed graph where nodes are object types labeled with their names and their prolificacy (P or NP, corresponding to prolific and non-prolific, respectively)[2]. A directed edge in this graph corresponds to a reference field and is directed towards a node corresponding to the type of that field. An edge is labeled as P if it connects two prolific types or as NP if it connects two non-prolific types; otherwise, an edge is not labeled.

To solve the co-allocation problem, we partition this graph by clustering together the related nodes of prolific types chained together via P-edges. Each such cluster is then treated as a unit of allocation. The node in each cluster which is allocated first is designated as the "representative" node. When an object corresponding to this representative type is created, enough space is reserved for all objects in the cluster. The allocator keeps a record of where the rest of the "constituent" objects are to be allocated.

Although we apply our algorithm only for co-allocation of objects of prolific types, objects of non-prolific types can also be co-allocated together similarly, if desired (if they are found to be frequently accessed and cause many data cache and data TLB misses). In this case, we partition the graph by clustering the nodes of non-prolific types connected to each other by NP-edges.

The empirical data we collected, by profiling more than a dozen Java applications, suggest that because (i) scalar[3] objects in Java programs tend to be small (around 16-20 bytes without an object header), i.e., have a small number of instance fields, and (ii) only a small number of those fields are reference fields (generally less than 3), there is usually a small number of choices regarding the types of objects that can be co-allocated. These characteristics of applications make it possible to use simple heuristics for graph partitioning.

2.4 Discussion and implementation issues

There are several issues that have to be dealt with.

- In some cases, a child object may be allocated before its parent object. To handle this, an allocation routine can reserve space for other objects in a cluster and record this fact. Because it happens rather infrequently, we do not optimize for this case and do not reserve space for other objects.

- The size of a child object may not be known in advance. For example, different instances of arrays of the same type

[2]The prolificacy of types is determined by on-line or off-line profiling.

[3]Note that *scalar* object types (i.e., non-array object types) should not be confused with *primitive* types such as `boolean`, `char`, `int` and others which do not have object instances.

may have different sizes (in contrast, all instances of a scalar type have the same size). A heuristic can be used to identify the most common size of an array of the type in question (this can be accomplished by on-line or off-line profiling). This identification can be coarse grained (e.g., at the program level) or fine grained (e.g., at the call site granularity). In our implementation, we use the coarse grained heuristic. Note that small differences between expected and actual object sizes may not necessitate a switch to the fall back allocation strategy. Memory allocators tend to reserve memory blocks at a certain granularity, and minor differences in object sizes are "absorbed" by extra space within a memory block.

- In Java, object types form a hierarchy. Given the type of a reference, the true type of an object pointed to by that reference may not be known at compile time; a subclass of some prolific class may have more instance fields than its superclass. Consequently, instances of that subclass would require more space, making the optimization ineffective, although it does not create a correctness problem. The problem is alleviated by the fact that prolific types tend to be leaves in a class hierarchy[4]. Consequently, a field whose type is some prolific type T is likely to point to an object whose true type is T.

- Preserving the optimization in the presence of dynamic class loading may require recompilation of methods using specialized allocation routines, in order to account for changes in the sizes of object instances. However, recompilation is likely to be infrequent and, in some cases, unnecessary (e.g., due to the minimum granularity at which a memory allocator allocates memory blocks). Again, dynamic class loading does not create a correctness problem for this optimization. Hence, we do not address this issue in our current implementation.

- Allocation patterns in applications may change with time and may occasionally depart from those predicted by our analysis A naive approach for pre-allocation of memory for objects in a cluster may not perform well as it may cause noticeable memory fragmentation. We limit such memory fragmentation in our implementation by employing an adaptive strategy. We monitor whether pre-allocated space has been consumed by its intended consumers (i.e., objects in some cluster) and if it was not, no memory is preallocated until an application begins to exhibit "exploitable" allocation patterns.

- The information about the allocation sites is used to refine the contents of clusters: only types whose allocation sites are reasonably close to each other (e.g., one object is created in a constructor of another) remain in a cluster.

- It is possible that class A has a pointer to class B, but during the lifetime of an object of type A, it points to several different instances of objects of type B. Such a dynamic behavior does not create a correctness problem for our scheme. Note that we perform object co-allocation as opposed to object inlining. Therefore, we do not remove a pointer from an object of type A to an object of type B. We also leave individual object headers intact. Consequently, object co-allocation (while offering less benefits) is less restrictive than object inlining, because it has to meet fewer constraints to ensure program correctness.

[4]We verified this trend by profiling more than a dozen applications written in Java.

2.5 Implementation

We have implemented this technique in the Jikes RVM [1]. We modified the VM and implemented the capability to collect allocation profiles, store them into files, and retrieve allocation profiles during future executions to perform object co-allocation.

We also implemented a module that analyzes the allocation profile of an application and determines a co-allocation strategy (based on the clusters of prolific types). In a large number of cases, clusters consist of only a few objects of prolific types (usually two, because of a small number of fields in objects of prolific types). Also, even an L2 cache line in our target machine can usually accommodate no more than two objects. These observations greatly simplify the implementation of the co-allocation technique. The heuristic we implemented performs partitioning by building clusters of two prolific types such that a prolific type A has a field that points to a prolific type B (for simplicity, we chose to ignore cases in which type A is the same as type B).

We have made changes to the optimizing compiler, which implements a chosen co-allocation strategy and inlines appropriate memory allocation routines. A different routine is inlined for a different member of a cluster: a parent vs. a child object, a scalar object vs. an array. For a non-copying GC, we had to make changes to the mark-and-scan routines and relevant data structures to ensure that all co-allocated objects within a memory block get marked and scanned. The original mark-and-sweep non-copying GC was not designed to support object co-allocation (i.e., its implementation assumes that only a single object can reside in a memory block).

The core memory allocation routines were left intact, and are invoked for non-prolific objects or whenever our scheme chooses to fall back to the standard allocation method (e.g., when a child object cannot fit into the space reserved for it). The memory allocator used with the non-copying mark-and-sweep GC partitions the heap space into equally sized meta-blocks, which are further partitioned into smaller blocks of various fixed sizes. Block sizes are determined empirically by profiling allocation requests of a reasonably large set of applications. Memory is allocated from the smallest memory block that is big enough to satisfy a request. With this allocator, object co-allocation can place objects of the same or different sizes into the same memory block (as opposed to two different memory blocks residing far apart from each other in the address space).

The allocator used with the semi-space copying GC allocates memory by incrementing a pointer. Therefore, it is thought to have better locality. With both GCs, large objects are placed in a separate portion of the heap (termed a large object space), which is managed in a mark-and-sweep fashion. Since most objects are small, memory allocations from the large object space are rare.

3. PRESERVING LOCALITY AT GC TIME

In this section, we describe our *locality-based* traversal technique. Since a traversal algorithm has to visit all reachable objects that reside in the region of the heap being collected and since during that traversal there will be little data reuse (many reachable objects are likely to be visited only once), it is crucial to take into consideration interactions between a GC and a memory system. The basic underlying principles behind our traversal scheme are the following: (i) processing is relatively cheap compared to performing data transfers and (ii) accessing memory sequentially (and access-

ing memory within a fixed-size region) is cheaper than accessing random memory locations. Consequently, our technique trades off a small amount of computation to uncover and exploit inherent locality of data structures during a traversal (by ensuring that surviving objects are accessed in some regular order), thereby reducing memory transfers and improving memory behavior of GC.

When used with a copying collector, this technique also improves the locality of surviving objects. Many existing systems attempt to improve locality of objects by bringing those objects closer to each other at GC time. These systems typically collect information about memory accesses of an application and construct a temporal relationship graph reflecting past application access patterns to predict future access patterns. In contrast, we simply exploit the *inherent locality* of data structures. This leads to shorter GC pauses, which are important in many application environments.

3.1 The traversal algorithm

The pseudocode for the locality-based traversal algorithm is shown in Figure 1. (In the discussion below, the terms *pointer* and *reference* will be used interchangeably.) We partition the heap space into contiguous pieces of memory referred to as *chunks*. Each chunk has a *chunk ID* associated with it. The first step is to determine the set of *roots*, in other words, live objects through which the rest of the reachable objects can be found. Given the set of roots, the first chunk to be collected is selected. This chunk is termed the *current chunk*. All the root pointers into the current chunk are identified. The set of these local pointers is referred to as LP. The set of all other (non-local) pointers is referred to as NLP.

The next step is to take a pointer from LP. The object pointed to by this local pointer is examined. If the object is marked as visited, another pointer is taken from LP. Otherwise, the object is marked as visited. All the pointers in that object are examined and are classified as either local (if they point into the current chunk) or non-local (otherwise). Depending on the category, they are added to LP or NLP.

The above procedure is repeated until all pointers in LP are exhausted. In that case, the next chunk to be collected is selected. That chunk becomes the new current chunk.

The process is repeated until there are no more pointers in NLP (this implies that there are no more pointers in LP). At this point, all reachable objects have been visited and marked as such.

```
LBT() {
   Roots = RuntimeSystem.IdentifyRoots();
   CurrentChunk = Roots.SelectTheFirstChunk();
   LP = Roots.identifyLP(CurrentChunk);
   NLP = Roots.identifyNLP(CurrentChunk);
   done = LP.empty();

   while(!done){

      while(!LP.empty()) {
        object = LP.next();
        if(!object.visited()) {
           object.visit();
           LP.extractLP(object);
           NLP.extractNLP(object);
        } // if(!object.visited())
      } // while(!LP.empty())

      if (NLP.empty())
        done = true;
      else {
        CurrentChunk = NLP.SelectTheNextChunk(LP);
        LP = NLP.identifyLP(CurrentChunk);
      } // else

   } // while(!done)

} // LBT
```

Figure 1: Pseudocode for the *locality-based traversal* algorithm

3.2 Discussion and implementation issues

For the most part, the discussion below follows the flow of the algorithm shown in Figure 1.

- The chunk size determines the unit of data and the level at which locality and performance is to be optimized. For example, the chunk size can be set to the size of the address space covered by the data translation look-aside buffer (TLB)[5]. To optimize for virtual memory performance, the chunk size should not exceed the size of physical memory that can be used by a process. For simplicity and efficiency, all chunks have the same size, which is a power of two. In such a case, given an address A, determining a *chunk ID* for address A can be done simply by shifting the address $log_2(ChunkSize)$ bits to the right.

- When collecting roots, it is important to note those roots through which objects may have been accessed most recently. Those are likely to be pointers in the top-most stack frame and pointers in the processor registers. It may also be important (for performance) to differentiate between pointers to old objects and pointers to recently allocated objects (e.g., much of recently allocated data may still reside in a processor cache).

- Selecting which chunk to collect first can be done in a number of ways. For example, if the data still in a cache when a GC is started are thought to be live, then the first chunk to be collected could be the last chunk from which an application allocated memory. Pointers into that chunk are likely to be found in the top-most stack frame or among recently updated or created static (class) fields. We have found that this heuristic works well in practice and used it in our implementation. Alternatively, a fixed number of root pointers can be sampled and the chunk into which most sampled pointers point to is selected. Of course, in the simplest case, any root pointer can be used to determine a chunk to be collected first.

- The next pointer to be selected from LP should be a pointer to an object that is close to the one visited recently. This will ensure that co-allocated objects (e.g., those sharing a cache line or a page) will be visited by a garbage collector at about the same time.

- Selecting a chunk to be collected next can be done in a similar manner (to the way the first chunk is selected) by inspecting pointers in NLP. For example, sampling a small number of pointers in NLP can give a good estimate about the chunk which contains many reachable objects. Alternatively, a pointer into the chunk closest to the current chunk can be used. We have found that the simplest approach, which is to

[5] We used this heuristic in our implementation.

select any non-local pointer, works fairly well and used it in the implementation.

- It is expected that during a traversal, most pointers will be local pointers, and the number of pointers in NLP will be small. Our measurements indicate that this is indeed the case.

- If reachable objects are located sparsely in the heap (e.g., if the number of objects in each of several recently processed chunks is smaller than some threshold), a switch to the default traversal scheme can be made. However, the need for such an adaptive strategy did not arise frequently.

3.3 Scalability issues in multiprocessors

In a multiprocessor system, it is important, from a performance point of view, to ensure that only one processor is working on a given chunk at a time. This has to be done at the beginning, when each processor takes the first chunk, and later, whenever a new current chunk is selected. If the system uses processor-local heaps, the last chunk from which memory was allocated before GC should be used as the first chunk to be processed, since some data from that chunk may still be available in the processor local cache.

For performance, in a multiprocessor system, the set NLP is partitioned and distributed among chunks (i.e., one subset of NLP per chunk); this distributed subset of NLP is then used to feed the LP set of a respective processor when the LP set of the corresponding processor becomes empty. More specifically, each GC thread will have one queue associated with each chunk (one queue is used for processing by a thread and others are used to distribute work for other threads). Objects that are reachable but residing outside a currently processed chunk are added to an appropriate queue for later processing. A GC thread that empties its local queue associated with a currently processed chunk steals work from non-local queues associated with that chunk (i.e., queues filled by other GC threads). In this scheme, a GC thread needs to use synchronization primitives only when it is accessing a queue associated with a chunk other than the one it is processing (i.e., "work stealing"). The GC threads terminate when all work queues are empty.

Partitioning GC work in this manner reduces the overhead of false sharing during traversal of reachable objects since no two processors can be visiting objects within the same chunk and within the same cache line. In addition, synchronization is not necessary when visiting objects within a current chunk. This leads to lower cost for marking and, especially, for copying reachable objects (because it eliminates contentions).

Increasing the page size or the number of data TLB entries in a processor would lead to wider TLB coverage. On a uni-processor system with wide TLB coverage, the benefits from the locality based traversal may be small. However, this technique is still applicable for systems with multiple processors (each processor having wide TLB coverage) due to the ability of the proposed technique to reduce the overhead of synchronization and false sharing.

3.4 Implementation

The Jikes RVM implements a version of Cheney's traversal with a work queue (used in the base configuration). The work queue is designed to ensure load balancing among parallel GC threads. Each thread has local buffers (termed *get* and *put* buffers) from which it takes references to objects that need to be scanned and into which it stores references to objects that need to be scanned later. Empty *get* buffers are refilled from the global work queue; full *put* buffers are placed onto the work queue. We used a single GC thread in our experiments.

We enhanced the standard work queue implementation with our locality-based traversal technique. We added support for buffers containing local and non-local pointers. The first chunk from which a traversal is started is the last one from which memory was allocated. During the traversal, the buffer with local pointers is used until it is exhausted, at which point we select the next chunk that needs to be scanned. Sampling of pointers to select the next chunk to be scanned seemed to work only slightly better than picking the first available non-local pointer (perhaps, because it was a good choice anyway). The chunk size was set to the area covered by the data TLB.

4. EXPERIMENTAL RESULTS

In this section, we present results on the performance impact of the type-based object co-allocation (OC) scheme and the locality-based traversal (LBT) technique in the Jikes RVM for two configurations: one with a non-copying mark-and-sweep GC, and the other with a copying semi-space GC (along with their respective allocators). We decided to start with these collectors because they were simple and easy to work with. Most importantly, by studying our techniques in the context of these basic collector, we hope to get a better understanding of a potential performance impact of our techniques. The allocators used in the VM were described in Section 2.5. Because of its short allocation sequence, a copying semi-space GC is a popular default configuration of the Jikes RVM.

Our intention is to demonstrate the advantages of the proposed techniques in their "relatively pure" forms.[6] The performance impact in systems that are hybrids of the schemes with which our techniques are being compared (e.g., a generational system with a copying young generation and mark-sweep-compact old generation) may be smaller. However, we believe that the mark-and-sweep old generation can also benefit from object co-allocation. In some systems, the use of moving collectors, such as copying generational collectors, is not possible. In such cases, non-moving (mark-sweep or reference counting) collectors are used. The non-moving collectors use free list pointer allocation, as opposed to bump pointer allocation, and can benefit from object co-allocation. In summary, the extent of performance benefits from our techniques with other more complex garbage collectors, such as generational collectors, is an open question, which will be a subject of future work.

4.1 Benchmarks and the experimental setup

To evaluate the effectiveness of object co-allocation and locality-based traversal techniques, we selected Java applications from the industry standard benchmark suites (SPECjvm98 [26] and SPECjbb2000 [27]) and from a suite of pointer-intensive applications, jOlden [4].

The SPECjvm98 applications ran for four iterations. (Note that during a single iteration, many SPECjvm98 applications allocate over 100MB and up to 300MB of heap space [19].) Most of the compilation activity was done during the first iteration. In the tables below, we report the best execution times (in seconds). After the first run, performance variations from one execution to another were insignificant. SPECjbb2000 ran for two minutes after the initial ramp up period. This application performs a variable amount of work during a fixed period of time, which is reported as throughput in terms of transactions per second. This application was run four

[6]The Jikes RVM collectors implement a separate large object space (LOS) for objects that are bigger than 2KB. In programs that we are considering, the number of large objects is not significant.

Table 1: A VM configured with a non-copying GC. (Note: OC - Object Co-allocation; LBT - Locality-Based Traversal.)

Base configuration				
Benchmark	Run time	Av. GC time	# of GCs	GC time
compress	40.687	.433	19	8.233
db	78.642	.570	7	3.992
jack	40.123	.467	34	15.905
javac	37.077	.806	45	36.296
jess	29.102	.510	76	38.795
mpegaudio	25.821	.442	7	3.099
mtrt	19.304	.624	36	22.482
jbb	1390.1	.820	15	12.309
bh	32.069	.485	8	3.883
health	43.987	.979	1	.979
voronoi	9.846	.908	2	1.816
Configuration with OC				
Benchmark	Run time	Av. GC time	# of GCs	GC time
compress	40.671	.429	19	8.151
db	65.251	.554	7	3.878
jack	38.017	.423	32	13.536
javac	35.239	.741	43	31.863
jess	27.915	.485	72	34.920
mpegaudio	25.798	.439	7	3.073
mtrt	17.458	.602	34	20.468
jbb	1481.3	.791	15	11.865
bh	26.617	.475	7	3.325
health	35.053	.935	1	.935
voronoi	8.341	.886	2	1.772
Configuration with LBT				
Benchmark	Run time	Av. GC time	# of GCs	GC time
compress	40.781	.435	19	8.265
db	78.649	.574	7	4.018
jack	40.378	.478	34	16.252
javac	36.985	.796	45	35.820
jess	29.013	.505	76	38.380
mpegaudio	25.829	.453	7	3.171
mtrt	19.317	.631	36	22.716
jbb	1383.4	.829	15	12.435
bh	32.112	.491	8	3.928
health	43.757	.996	1	.996
voronoi	9.821	.903	2	1.806
Configuration with OC + LBT				
Benchmark	Run time	Av. GC time	# of GCs	GC time
compress	40.673	.430	19	8.170
db	65.228	.536	7	3.752
jack	37.974	.404	32	12.946
javac	35.185	.694	43	29.871
jess	27.907	.472	72	33.984
mpegaudio	25.796	.438	7	3.066
mtrt	17.406	.575	34	19.581
jbb	1497.8	.784	15	11.760
bh	26.072	.434	7	3.038
health	34.441	.803	1	.803
voronoi	8.173	.773	2	1.546

Figure 2: Normalized run times (non-copying GC = 100%).

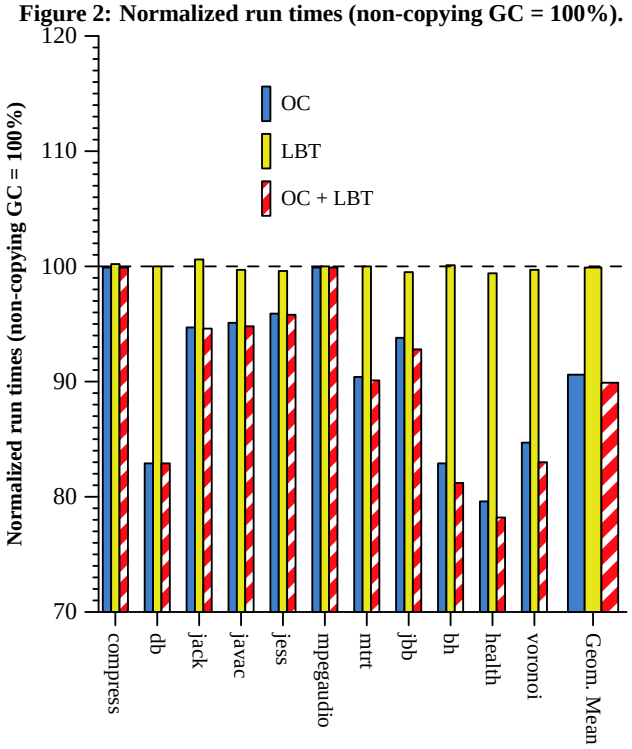


Figure 3: Normalized GC times (non-copying GC = 100%).

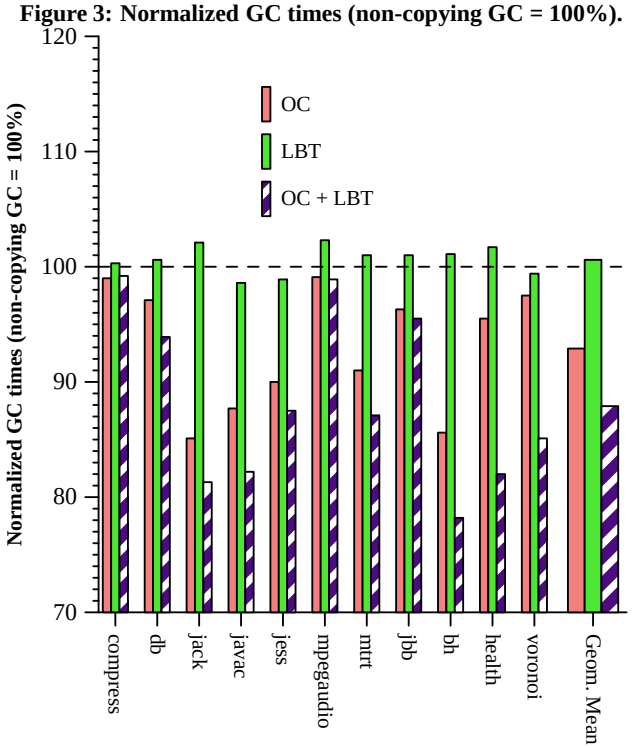


Table 2: A VM configured with a copying GC. (Note: OC - Object Co-allocation; LBT - Locality-Based Traversal.)

Base configuration				
Benchmark	Run time	av. GC time	# of GCs	GC time
compress	40.565	.379	20	7.580
db	81.118	.719	12	8.628
jack	37.587	.425	48	20.400
javac	37.766	.914	63	57.582
jess	25.615	.489	92	44.988
mpegaudio	25.024	.404	8	3.232
mtrt	23.798	.782	61	47.702
jbb	1513.7	1.039	25	25.975
bh	28.279	.469	9	4.221
health	35.016	.649	1	.649
voronoi	15.655	.968	4	3.872
Configuration with OC				
Benchmark	Run time	av. GC time	# of GCs	GC time
compress	40.621	.381	20	7.620
db	81.235	.715	12	8.580
jack	37.992	.424	49	20.776
javac	37.814	.918	64	58.752
jess	25.702	.485	93	45.105
mpegaudio	25.037	.412	8	3.296
mtrt	23.829	.779	61	47.519
jbb	1497.4	1.043	25	26.075
bh	28.317	.471	9	4.240
health	35.112	.644	1	.644
voronoi	15.687	.970	4	3.881
Configuration with LBT				
Benchmark	Run time	av. GC time	# of GCs	GC time
compress	40.498	.368	20	7.360
db	74.347	.683	12	8.196
jack	37.014	.413	48	19.824
javac	34.021	.765	63	48.195
jess	25.595	.472	92	43.424
mpegaudio	25.006	.402	8	3.216
mtrt	21.539	.654	61	39.894
jbb	1543.2	.996	25	24.900
bh	25.443	.405	9	3.647
health	30.251	.514	1	.514
voronoi	14.397	.788	4	3.152
Configuration with OC + LBT				
Benchmark	Run time	av. GC time	# of GCs	GC time
compress	40.512	.371	20	7.420
db	74.829	.687	12	8.244
jack	37.263	.409	49	20.041
javac	34.843	.772	64	49.408
jess	25.605	.468	93	43.524
mpegaudio	25.011	.403	8	3.224
mtrt	21.758	.661	61	40.321
jbb	1531.5	1.015	25	25.375
bh	25.507	.412	9	3.708
health	30.348	.510	1	.510
voronoi	14.441	.793	4	3.172

Figure 4: Normalized run times (copying GC = 100%).

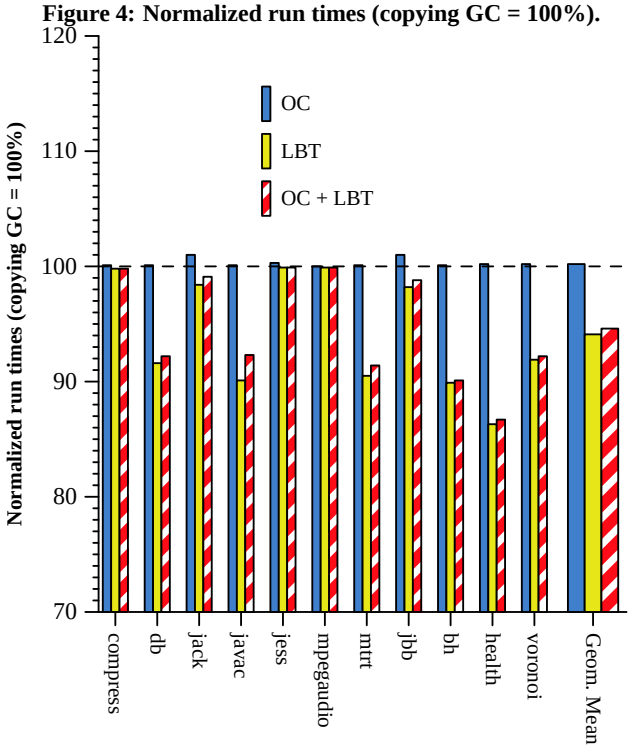
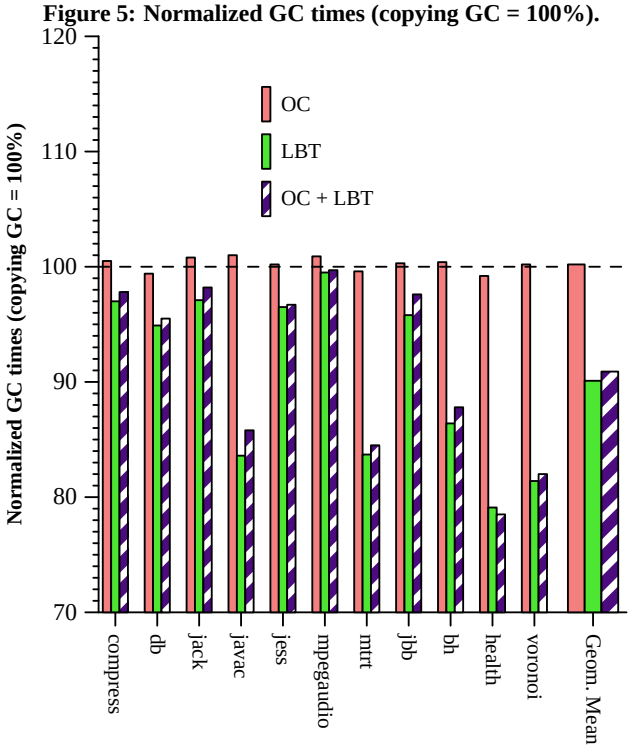


Figure 5: Normalized GC times (copying GC = 100%).



times and the best result was reported. Programs from the jOlden suite were run four times; again, the best execution times were reported.

All applications ran with a 64MB heap (except for SPECjbb2000: 128MB; compress, jess, mpegaudio, mtrt: 32MB). SPEC applications are self-timed. The timing results reported by SPEC applications include GC time. It was difficult to accurately separate GC times due to each iteration. Therefore, for the sake of consistency, garbage collection times are reported for the entire execution of these applications (e.g., all four executions of SPECjvm98 and the entire execution of SPECjbb2000).

Experiments were performed on a system with a 333MHz PowerPC 604e processor running AIX 4.3.3. Both on-chip L1 instruction and data caches in the processor were 32K, 4-way associative, with 32-byte cache blocks. The off-chip unified L2 cache had 256K, was direct-mapped, and had 64-byte cache blocks. Split instruction and data TLBs had 128 entries each and were 2-way associative. The page size on this system was 4KB. The system was configured with 3GB of RAM. There was no significant paging during our experiments.

4.2 Non-copying GC

For the first set of experiments (Table 1), we configured the VM with a non-copying mark-and-sweep GC (the first base configuration) and ran all the benchmarks. For each application, we recorded the best execution time in seconds (throughput for SPECjbb2000, in terms of ops/second), GC time, and the number of GCs. Then we experimented with three variations of this base configuration: (i) with object co-allocation (OC) enabled; (ii) with locality-based traversal enabled (LBT); and, finally, (iii) with both OC and LBT enabled. For each variation, we computed normalized execution time and GC time (normalized with respect to the base configuration). These data are shown in Figures 2 and 3. We also computed the geometric means of normalized times, shown in the last three bars.

Configuration w/OC It appears that OC by itself has a noticeable positive impact on application run times. The performance improvements are likely to be due to the fact that the base configuration has poor locality to start with. Note that in the base configuration, objects that point to each other but have different sizes are likely to be allocated from different chunks (partitioned into small blocks of fixed sizes); thereby they are likely to be far apart from each other in memory. With OC, objects of prolific types are placed into the same block and are likely to reside on the same page, which can reduce data TLB misses (and page faults). Because prolific objects are small, there is also a high chance of two or more objects of prolific types sharing the same L2 cache line, which leads to lower cache miss rates if those objects are accessed contemporaneously.

The applications that benefit a lot from OC are those that have poor memory behavior. Among those, the applications that allocate many objects show the greatest improvements. In programs with these characteristics, our technique can accurately identify prolific types and perform OC optimizations for objects of those prolific types more successfully.

It is notable that for some applications (e.g., jack, javac, jess, mtrt, bh), OC also reduces the number of GCs. Object co-allocation reduces memory fragmentation, which is an inherent characteristic of non-copying GCs. Lower memory fragmentation leads to fewer cases when a GC has to run to reclaim space occupied by unreachable objects.

Also, OC has some impact on the GC time. Performing mark-and-sweep over clusters of co-allocated objects improves locality even with a built-in traversal algorithm.

Configuration w/LBT Since the locality of objects is poor in the original configuration, LBT, which is designed to exploit available inherent locality during a traversal, has difficulty to find enough locality (which would otherwise reduce GC time). Also, because LBT is only used at GC time (which it does not improve in this configuration), it does not noticeably impact overall execution times of applications.

Configuration w/OC+LBT It seems that the synergy of both OC and LBT has the greatest impact. As discussed earlier, OC by itself creates better data locality which leads to reduced run times, lower memory fragmentation, and shorter GC pauses. LBT appears to be exploiting locality created by OC which is demonstrated by noticeably shorter GC pauses. Shorter GC pauses lead to further reductions in run times of applications.

The bottom line Poor data locality of the non-copying configuration is easily improved with help of OC by creating good locality at allocation time. Indeed, in a non-copying configuration, the OC technique is approximating the allocation behavior of a configuration with a copying GC. Namely, it tries to ensure that objects created temporally together are also placed spatially close to each other.

Further performance improvements are obtained by utilizing LBT along with OC. Thereby, locality created by one technique (namely, OC) is exploited later by another technique (i.e., LBT).

4.3 Copying GC

For the second set of experiments (Table 2), we configured the VM with a copying semi-space GC (the second base configuration), repeated all experiments, and computed normalized execution times and GC times (shown in Figures 4 and 5) with respect to the second base configuration. The computed geometric means of normalized times are shown in the last three bars.

Configuration w/OC OC by itself does not seem to impact the execution times of applications. The already fairly good locality of new objects, most of which are of prolific types, delivered by the memory allocator used with the copying GC (which places all new small objects contiguously in memory), is perhaps the reason for such a behavior. Particularly, since we co-allocate objects many of which probably would have been close to each other anyway.

Any changes to locality introduced by the OC technique are not "visible" to the built-in traversal algorithm. Hence, the GC times are not impacted either.

Configuration w/LBT Given a fairly good locality of objects in a copying configuration, the LBT technique is able to pick up on that and exploit this inherent locality. As a consequence, GC times are noticeably reduced. It appears that those applications that have a lot of live data to be traversed at GC time tend to benefit a lot from LBT.

In addition to direct minor impact on application run times by performing a better traversal, LBT has a major indirect impact on performance by creating a better data layout. By copying objects (many of which are, perhaps, old ones and some of which are recently created young ones) in a such a way so that objects that used to reside near each other, still reside close to each other in another semi-space, it is able to preserve locality present in the original layout.

Configuration w/OC+LBT Since OC does not improve performance of this configuration (the locality of new objects is fairly good), it is expected that adding OC to the configuration that implements LBT does not reduce application run times and GC times. To the contrary, the work done for OC shaves off some of the per-

formance benefits of LBT.

The bottom line Increasing locality of new objects in a copying configuration is hard: the locality of new objects is fairly good and OC does not seem to help. However, LBT is capable of exploiting the locality inherently present in this configuration by traversing objects in the *local-first order* and preserving the original layout while copying objects. The end results are reduced GC times and application run times.

5. RELATED WORK

The related work falls into two categories: (i) research on object co-allocation and placement and (ii) traversal algorithms.

5.1 Object Co-allocation and Placement

Due to the difficulty of making good allocation decisions at compile-time, much of the work described in the literature has been devoted to profile-based approaches.

Profile-driven approaches Calder et al. [5] used a rich set of profiling information to create cache-conscious data layout and to reduce conflict misses. The placement of data was guided by a temporal relationship graph (TRG) constructed from profiling data (collected off-line).

Chilimbi et al. [9] demonstrated that a generational copying collector can be successfully used to implement cache-conscious placement of "old" long-lived objects and improve locality of pointer-based data structures. Their system collected profiling information about data access patterns to old objects (in programs written in Cecil), constructed a TRG of data accesses, and then used the TRG to reorganize those objects during major collections with the help of their copying algorithm.

Yardimci and Kaeli [31] presented two approaches for *profile-guided allocation* of heap objects in the context of C programs. In the first approach, cache miss *profiling* information was used along with a modified *malloc* library to place objects in appropriate regions in such a way so as to reduce conflict misses in the L1 data cache. In the second approach, they used *profiling* information to build TRGs for accesses to heap objects. The TRGs were then used to guide allocation of contemporaneously accessed objects into neighboring regions in the heap and data cache so as to increase spatial locality (and to avoid mapping those objects to the same portion of a cache).

Seidl and Zorn [21] used off-line profiling and call site information to predict the reference frequency and lifetime of objects. Their object placement strategy was shown to decrease the usage of virtual memory pages in several allocation-intensive C programs. They used off-line profiling to classify objects into frequently- and infrequently-referenced (as well as into short-lived and other) and allocate objects belonging to the same category into the same heap segment.

User-driven approach Chilimbi et al. [8] demonstrated that good data organization and layout can improve the locality and performance of pointer-intensive programs. They describe a cache-conscious heap allocator (called *ccmalloc*), which uses programmer-supplied hints (inserted into the source code of C programs) and attempts to co-allocate contemporaneously accessed data on the same cache line. A hint is a pointer to a data structure that is likely to be accessed contemporaneously with a data structure to be created. Insertion of a "good" hint requires detailed knowledge of the program's code and its behavior, and is impractical for large-scale programs.

Compiler-driven approach Dolby and Chien [11, 12, 13] have introduced *object inlining* for C++ programs, an optimization which uses whole program analysis to inline-allocate objects (with certain properties) within their respective containers. Object inlining eliminates many object references and reduces object allocation costs. When applicable, object inlining is more effective than object co-allocation. However, object co-allocation is a more widely applicable and general technique, which (unlike object inlining) does not require legality checks based on complex compiler analysis because object co-allocation does not eliminate inter-object references and individual object headers.

Our work Unlike [9], our scheme does not make a direct use of a GC (generational or non-generational ones) to implement cache-conscious object placement and co-allocation, nor does it involve copying objects to perform reorganization. We focus the efforts on all newly created objects at the time of their birth and not on old long-lived objects [9]. Consequently, our scheme may be well-suited for environments without a GC or with a non-copying GC, which cannot move objects around. Also, unlike that scheme, our scheme does not "preserve" garbage and collects all unreachable objects. Although we collect profiling information (which was also done in [5, 9, 21, 31]), our profile is, in fact, not as rich and is very inexpensive; its size is independent of the number of memory accesses and other program characteristics. In our scheme, object placement is not driven by temporal affinity. Hence, we do not construct a TRG, which may be big. Consequently, we do not sort a TRG which can be an expensive operation. Instead, we use *type affinity* and the notion of *prolific* types [22], and base object placement decisions exclusively on the prolificacy of types. Unlike [8], our technique is fully automatic and does not require any involvement from a programmer; this feature makes our scheme attractive for optimizing large-scale software systems. Unlike [11, 12, 13], which use an expensive whole program analysis, which is somewhat impractical for runtime compilation in Java, the compiler analysis employed by our scheme is simple, which broadens the applicability of our scheme. Compared to object inlining, object co-allocation does not need to satisfy the same set of stringent constrains to ensure program correctness because it does not change the underlying structure and the layout of individual objects. To the best of our knowledge, this is the first reported effort on locality-conscious allocation-time object placement in the context of Java; most previous work was performed in the context of languages like FORTRAN, C, C++, and Cecil.

5.2 Traversal Algorithms

Traversal algorithms for finding reachable objects can be classified into basic and hybrid ones. To assure that no live object (residing in the heap region being collected) is reclaimed, a traversal algorithm has to find all reachable objects. The key differences among the algorithms are in the order they visit objects. The order of traversal has implications for storage requirements and data locality.

Basic traversal algorithms A traversal based on a *depth-first search* (DFS) is one of the simplest ones to implement. However, because it requires an additional data structure to maintain a potentially large stack of objects to be scanned, it has not been popular. Copying objects using a DFS-based traversal has been shown to improve data locality of shallow and wide data structures (albeit only slightly according to [30]). For deep data structures, this traversal tends to place siblings far apart from each other, which may not be desirable; thereby reducing locality.

A *breadth-first search* (BFS) or a *level-order* traversal is one of

the most popular algorithms for finding reachable objects. A BFS-based algorithm tends to place siblings and distant cousins closer to each other (which may improve spatial locality) but may separate them from their parent nodes especially if the root set is very large. In addition, it places upper-level objects, which may be used for indexing into the rest of the data structure, closer to each other, which may have a positive performance impact. Cheney [6] described a version of a BFS traversal that does not require auxiliary storage space for the queue. In his scheme, objects copied into the *To-space* act as a queue. The head and the tail of the queue are denoted with two pointers: *scan* and *free*.

Hybrid traversal algorithms Moon [20] discussed several GC techniques in a Lisp system with a large virtual memory, including one technique which improves the locality of objects copied by a GC into a target space. Moon's algorithm is a modification of Cheney's scheme [6] and works in an "approximately depth-first fashion." It always scans a partially-filled page at the end of the *To-space* first and tries to bring a parent and a child objects closer to each other (as a DFS-based algorithm would) even if they reside far apart (e.g., on different pages), which may cause more paging activity during a GC. When a page is filled, it continues scanning from an object pointed to by the *scan* pointer. Unfortunately, this algorithms may need to rescan some objects more than once, which may not be desirable. According to Wilson et al. [30], a true DFS traversal is likely to be better than Moon's approximate DFS.

Wilson et al. [30] presented a technique for improving page-level locality of persistent runtime-related objects (i.e., the system heap image which included a compiler and development tools) in a large Lisp system with a generational copying GC. The focus of their study was page locality of "old" live objects as opposed to the locality of objects created by executing programs. They noted that the cache-level locality is different and is dominated by references to young objects. Their technique is based on static-graph restructuring and aims to group data structures according to their hierarchical organization. First, they group upper-level objects on the same page and then apply the same grouping recursively to the objects in the remaining subgraphs. Their technique also takes into consideration poor locality effects of linear traversal of hash tables and treats hash tables specially during a traversal. Their hierarchical decomposition algorithm (based on Cheney's [6] algorithm) is related to the one developed by Moon [20] but avoids the redundant scanning. Their technique yields the most benefit with tree-like data structures and when most objects are pointed to by only one other object. Although this may be true for functional languages like Lisp, it is not always the case for programs written in Java where objects tend to cross-reference one another. They note that the locality of data "outside the system image, created during program execution" is an area for future work. Our paper addresses exactly this problem.

Chilimbi et al. [9] presented a cache-conscious copying algorithm which utilizes profiling information in the form of a TRG and relies on Cheney's [6] copying algorithm to accomplish its task. First, they pick an edge in a TRG with the highest weight; then, they perform a greedy DFS traversal of an entire TRG by following edges with the highest affinity weights. During this traversal, they copy objects associated with the selected affinity edges into the *To-space*. Once all objects corresponding to nodes in the TRG have been copied, Cheney's algorithm is invoked to process all pointers into the *From-space*. Finally, all root objects that still reside in the *From-space* are copied and processed by Cheney's algorithm (the last step is necessary as not all root objects may be part of the TRG or reachable from the TRG). As noted in [9], the downside of this scheme is that by copying objects in the TRG first, it will retain unreachable garbage objects until the next major collection. A major collection usually occurs after several minor collections; this implies that some dead objects may remain in a system for an extended period of time.

Appel and Bendiksen [2] show that a traversal of live objects can be done in a "vector" mode on vector supercomputers. They explain that non-copying and copying collectors can be expressed as breadth-first searches in which a queue of objects to be scanned (such as the one in Cheney's scheme [6]) is processed in parallel. This processing is accomplished with the help of vector instructions performing *scatter* and *gather* operations.

Boehm [3] describes a *prefetch-on-grey* technique (to be used during heap traversal) and shows the benefits of prefetching for reducing GC pauses.

Our work Our algorithm can be viewed as a distant cousin of Cheney's algorithm [6], but unlike his algorithm, ours takes into account the locality of objects during traversal and preserves this locality should the objects have to be copied. Unlike Moon's algorithm [20], ours tries to avoid chasing pointers between objects residing far apart from each other and traverses closely-residing objects first, and does not perform a DFS-like traversal the way it is described in his paper. Unlike the scheme developed by Wilson et al. [30], ours does not treat hash tables specially and applies a locality-based traversal to all objects (not just those residing in the system heap). In contrast to the scheme proposed by Chilimbi et al. [9], ours does not rely on a TRG, does not retain garbage, and can work with non-generational collectors. Also, our scheme, unlike the one developed by Courts [10], is purely software-based and does not require any special-purpose hardware. Our traversal has good scalability properties for parallel GCs. Finally, our work is the first attempt to exploit the synergy and interactions of allocation time object placement and a locality-conscious heap traversal.

6. CONCLUSIONS

We presented two simple yet effective software-based techniques for improving and preserving data locality in pointer-intensive applications, such as those written in Java. Given current technological trends and growing memory latencies, such techniques, which let applications utilize hardware caches and TLBs more effectively, will become increasingly important.

The first technique is a new object co-allocation and placement technique based on the *prolificacy* of object types. Unlike existing object placement techniques, ours does not rely on a generational GC, application traces, sophisticated whole program compiler analysis (which is less practical in the context of Java due to dynamic class loading), or programmer involvement. In contrast, it is fully automatic and is simple to implement. This technique has the most impact in a configuration with a non-copying GC that tends to have poor locality to start with (compared to inherently better locality of configurations using a copying GC). Applied to a set of Java programs, our technique yields up to a 21% performance improvement (10% on average) in the Jikes RVM configured with a non-copying mark-and-sweep collector.

The second technique is a locality-based GC traversal algorithm, which attempts to do both (i) reduce GC time by taking into account object locality, and (ii) when incorporated into a copying collector, enhance the data locality of copied objects. Compared to other known techniques, ours does not rely on special-purpose hardware, does not retain garbage, can be applied to objects other than persistent objects in the system image, and can be used with non-generational collectors. For parallel collectors, our scheme will be able to reduce the overhead due to synchronization and false shar-

ing, thereby improving the scalability of multiprocessor GCs. Our algorithm is simple and easy to implement; it can be a plug-in replacement for an existing graph traversal algorithm. Experimental results are encouraging. We show that our traversal algorithm can reduce GC times in Java programs by 20% (10% on average) and improve application performance by up to 14% (6% on average) in the Jikes RVM configured with a copying semi-space collector.

Notably, the synergy of both object co-allocation and locality-based traversal techniques results in up to 22% (10% on average) performance improvement in the Jikes RVM configured with a non-copying mark-and-sweep collector. To the best of our knowledge, it is the first reported result on the benefits of combining allocation time object placement and a memory-friendly heap traversal.

Future work This work opens up a number of directions for future research. Given that co-allocation of objects of prolific types proved itself to be successful, the next logical step is to experiment with co-allocation of objects of non-prolific types and measure an additional performance impact. Similarly, it is worthwhile to explore co-allocation of frequently referenced virtual method tables, accesses to which may be responsible for a non-trivial fraction of data TLB misses [23]. Coarse grain information about "hot" frequently accessed reference fields in objects (e.g., on a per-class basis) can be used to refine and tune co-allocation decisions.

We plan to investigate performance benefits of our graph traversal algorithm for parallel applications from the domain of high-performance computing, such as Java Grande applications [16, 24] and measure the reduction in synchronization and false sharing overheads during garbage collection. We also plan to experiment with large applications using multi-megabyte heaps and asses the impact of our technique on virtual memory performance.

We would like to study the impact of our techniques on other collectors, such as copying generational and hybrid collectors (e.g., a copying collector in the young generation and a mark-and-sweep collector in the old generation). The bottom line performance improvements with these collectors may be lower than those with simple collectors. However, we still expect to see performance benefits. For example, the non-copying old generation can benefit from object co-allocation, which improves locality. In addition, generational schemes, whose goals are to reduce individual garbage collection pauses, can further benefit from the locality-based traversal.

There are indications that generational collectors employed in production JVMs can also benefit from the locality based traversal. For example, it has been commonly accepted that commercially important server-side workloads, such as the SPECjAppServer2001 application [25], formerly ECperf [14], need to run with a large young generation (0.7GB - 1.0GB) and large heap sizes (1.6GB - 3.5GB) to achieve high throughput [18]. Similarly, the SPECjbb2000 application [27], another important server workload, is often executed with a very large nursery (1.8GB - 3.1GB) and a large heap (3.8GB) to achieve the highest possible throughput [28, 29]. Clearly, the young generation of this size cannot fit into any modern L2 or even L3 caches. In addition, without large page support, the region of memory occupied by the young generation cannot be covered even by a large data TLB. Therefore, we believe that the locality based traversal would be beneficial in such a setting.

Acknowledgments

We would like to thank the members of the Jikes RVM team for providing us with an infrastructure for this work. We would also like to thank Dick Attanasio, Pratap Pattnaik, Steve Smith, and Marc Snir for valuable technical discussions.

7. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Burton, 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. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shephard, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. *IBM Systems Journal*, 39(1):194–211, 2000.

[2] A. W. Appel and A. Bendiksen. Vectorized garbage collection. *The Journal of Supercomputing*, 3:151–160, 1989.

[3] H.-J. Boehm. Reducing garbage collector cache misses. In *Proc. of the 2000 International Symposium on Memory Management (ISMM 2000)*, pages 59–64, Minneapolis, Minnesota, October 2000.

[4] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proc. of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*, Barcelona, Spain, September 2001.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 139–149, San Jose, California, October 1998.

[6] C. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677 – 678, November 1970.

[7] K. Chevalier, J. Kodumal, and X. Jiang. Memory subsystem optimization for functional languages: A case study. Technical Report, Computer Science Division, University of California, Berkeley, May 2002. Available at `http://www.cs.berkeley.edu/~xdjiang/cs252`.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 1 – 12, Atlanta, Georgia, May 1999.

[9] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proc. of the 1998 International Symposium on Memory Management (ISMM'98)*, pages 37 – 48, Vancouver, British Columbia, Canada, October 1998.

[10] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[11] J. Dolby. Automatic inline allocation of objects. In *Proc. of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, pages 7 – 17, Las Vegas, Nevada, June 1997.

[12] J. Dolby and A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 1 – 20, Vancouver, British Columbia, Canada, October 1998.

[13] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proc. of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 345 – 357, Vancouver, Britith Columbia, Canada, June 2000.

[14] ECperf. ECperf Home Page. http://java.sun.com/j2ee/ecperf/.

[15] J. Gosling, B. Joy, and G. Steele. *The Java(TM) Language*

Specification. Addison-Wesley, 1996.

[16] Java Grande Forum. The Java Grande Forum Benchmark Suite. Document available at URL `http://www.epcc.ed.ac.uk/javagrande/`, 2000.

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

[18] M. Karlsson, K. Moore, E. Hagersten, and D. Wood. Memory characterization of the ECperf benchmark. In *Proc. of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002), held in conjunction with the 29th International Symposium on Computer Architecture (ISCA29)*, Anchorage, Alaska, May 2002.

[19] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proc. of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 264 – 274, Santa Clara, California, June 2000.

[20] D. A. Moon. Garbage collection in a large LISP system. In *Proc. of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[21] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, San Jose, California, October 1998.

[22] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Proc. of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 295–306, Portland, Oregon, January 2002.

[23] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proc. of the Joint International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2001 / Performance 2001)*, pages 194–205, Cambridge, Massachusetts, June 2001.

[24] L. A. Smith and J. M. Bull. A multithreaded Java Grande benchmark suite. In *Proc. of the Third Workshop on Java for High Performance Computing*, Sorrento, Italy, June 2001.

[25] Standard Performance Evaluation Council. SPEC jAppServer Development Page. http://www.spec.org/osg/jAppServer/.

[26] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.

[27] Standard Performance Evaluation Council. *SPEC JBB2000 Benchmark*, 2000. `http://www.spec.org/osg/jbb2000/`.

[28] Sun Microsystems, Inc. SPECjbb2000 Results for Sun-Fire 6800 with HotSpot Server VM on Solaris/SPARC, version 1.3.1-02. `http://www.spec.org/osg/jbb2000/results/res2001q4/jbb2000-20011105-00092.html`, October 2001.

[29] Sun Microsystems, Inc. SPECjbb2000 Results for Sun-Fire 6800 with HotSpot Server VM on Solaris/SPARC, version 1.3.1-02 . `http://www.spec.org/osg/jbb2000/results/res2001q4/jbb2000-20011128-00097.html`, October 2001.

[30] P. Wilson, M. Lam, and T. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proc. of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 177–191, Toronto, Ontario, Canada, June 1991.

[31] E. Yardimci and D. Kaeli. Profile-guided tuning of heap-based memory access. In *Proc. of the 2nd Workshop on Memory Performance Issues*, Goteberg, Sweden, July 2001.