To appear in International Symposium on Memory Management (ISMM'98), October, 1998.

Using Generational Garbage Collection To Implement Cache-Conscious Data Placement

Trishul M. Chilimbi and James R. Larus[1]
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, Wisconsin 53706
chilimbi@cs.wisc.edu

1. ABSTRACT

The cost of accessing main memory is increasing. Machine designers have tried to mitigate the consequences of the processor and memory technology trends underlying this increasing gap with a variety of techniques to reduce or tolerate memory latency. These techniques, unfortunately, are only occasionally successful for pointer-manipulating programs. Recent research has demonstrated the value of a complementary approach, in which pointer-based data structures are reorganized to improve cache locality.

This paper studies a technique for using a generational garbage collector to reorganize data structures to produce a cache-conscious data layout, in which objects with high temporal affinity are placed next to each other, so that they are likely to reside in the same cache block. The paper explains how to collect, with low overhead, real-time profiling information about data access patterns in object-oriented languages, and describes a new copying algorithm that utilizes this information to produce a cache-conscious object layout.

Preliminary results show that this technique reduces cache miss rates by 21–42%, and improves program performance by 14–37% over Cheney's algorithm. We also compare our layouts against those produced by the Wilson-Lam-Moher algorithm, which attempts to improve program locality at the page level. Our cache-conscious object layouts reduces cache miss rates by 20–41% and improves program performance by 18–31% over their algorithm, indicating that improving locality at the page level is not necessarily beneficial at the cache level.

1.1 Keywords

Garbage collection, cache-conscious data placement, object-oriented programs, profiling

2. INTRODUCTION

Since 1980, microprocessor performance has improved 60% per year, while over the same period, memory access time decreased only 10% per year [20]. This discrepancy has produced a large processor-memory imbalance. Memory caches are the ubiquitous hardware solution to this problem [29, 23]. In the beginning, a single level of cache sufficed, but the increasing imbalance (now almost two orders of magnitude) demands a memory hierarchy, which produces a large range of memory-access costs. As a result, many programs' performance is dominated by memory references.

A variety of hardware and software techniques—such as prefetching [19, 2], multithreading [16, 24], non-blocking caches [14], dynamic instruction scheduling, and speculative execution—have been developed and implemented to reduce or tolerate memory latency. These techniques require complex hardware and compilers, but have proven ineffective for many programs [21].

The fundamental problem with these techniques is that they attack the manifestation (memory latency), not the source (poor reference locality), of the bottleneck. Prior research has focused on improving cache locality in scientific programs that manipulate dense matrices through program (loop) transformations [31, 3, 12]. Recently, Calder *et al.* described a profile-driven, compiler-directed approach to cache-conscious data placement [1]. Other work has demonstrated that programs which manipulate pointer-based structures can benefit greatly from cache-conscious structure layouts [9, 22].

This paper describes how a copying garbage collector can produce a cache-conscious object layout using real-time data

[1]Current address: Microsoft Research, One Microsoft Way, Redmond, WA 98052, larus@microsoft.com.

profiling information. The copying phase of garbage collection offers a invaluable opportunity to reorganize a program's data layout to improve cache performance. A cache-conscious data layout places objects with high temporal affinity near each other, so that they are likely to reside in the same cache block. This paper is a preliminary investigation into a new area with many possible design choices. The paper does not claim to explore the choices, rather it demonstrates that a reasonable set of parameters leads to significant performance improvements.

In our approach, a program is instrumented to profile its data access patterns. The profiling data gathered during an execution is quickly used to optimize that execution, rather than a subsequent one. Our technique relies on a property of object-oriented programs—most objects are small (< 32 bytes)—to perform low overhead (< 6%) *real-time* data profiling. The garbage collector uses the profile to construct an object affinity graph, in which weighted edges encode the temporal affinity between objects (nodes). A new copying algorithm uses the affinity graph to produce cache-conscious data layouts while copying objects during garbage collection. Experimental results for five Cecil programs [4, 5] show that our cache-conscious data placement technique reduces cache miss rates by 21–42% and improves program performance by 14–37%.

Other researchers have attempted to improve a program's virtual memory (page) locality by changing the traversal algorithm used by a copying garbage collector [18, 30, 15, 10]. We compare our cache-conscious copying scheme against one such algorithm (the Wilson-Lam-Moher algorithm [30]). The results show that our cache-conscious object layout reduces cache miss rates by 20–41% and improves program performance by 18–31% over their technique, indicating that page-level improvements are not necessarily effective at the cache level.

The rest of the paper is organized as follows. Section 3 provides brief background information on generational garbage collection as well as an overview of the most common traversal algorithm for copying objects. Section 4 describes the design and implementation of our low overhead real-time data profiling system for object-oriented programs. Section 5 describes how this profiling information is used to construct object affinity graphs. Section 6 describes our copying algorithm. Section 7 presents experimental results that illustrate the benefits of our technique. Finally, Section 8 briefly discusses related work.

3. BACKGROUND: GENERATIONAL GARBAGE COLLECTION

Our system uses the University of Massachusetts language-independent garbage collector toolkit [13]. The toolkit implements a flexible generation scavenger [17, 26] with support for a time-varying number of generations of time-varying size. Figure 1 illustrates the heap organization from the garbage collector's viewpoint. The garbage collected heap is divided into a number of generations. The youngest (first) generation holds the most recently allocated objects. Objects that survive repeated scavenges are promoted to older (higher) generations. Garbage collection activity focuses on young objects, which typically die faster than old

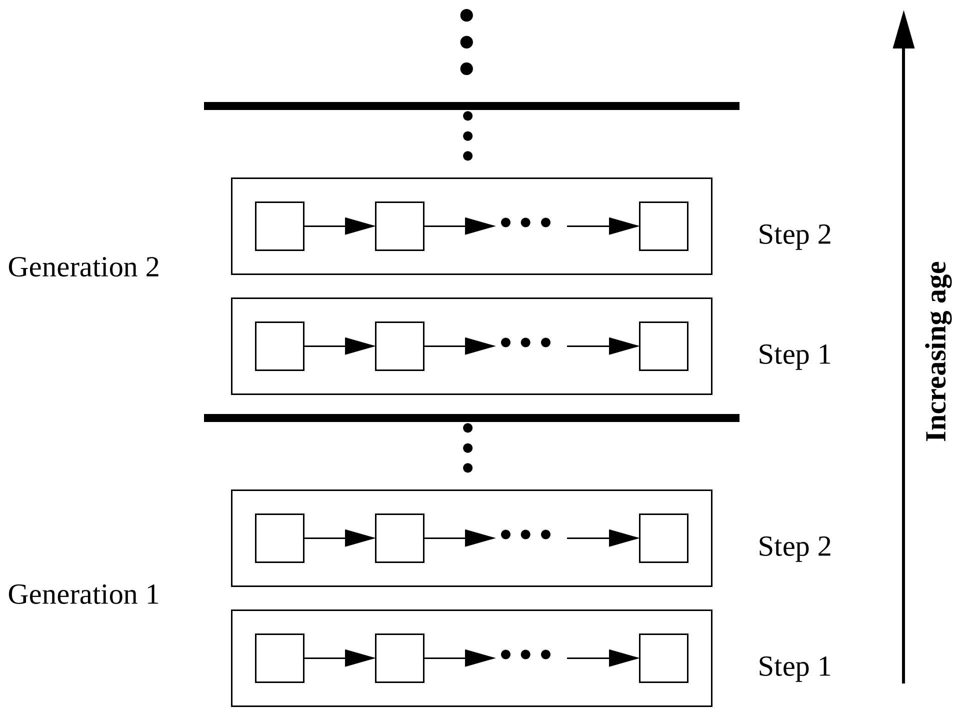

Figure 1. Garbage collected heap layout.

```
scavenge()
{
   Flip roles of Fromspace, Tospace;
   unprocessed = free = Tospace;
   for R in root set
      R = Tospace_copy(R);
   while unprocessed < free
   {
      for P in children(unprocessed)
         *P = Tospace_copy(*P);
      unprocessed +=
         sizeof(*unprocessed);
   }
}
```

```
Tospace_copy(P)
{
   if forwarded(P)
      return forwarding_addr(P);
   else
   {
      addr = free;
      copy(P, free);
      free += sizeof(P);
      forwarding_addr(P) = addr;
      return addr;
   }
}
```

Figure 2. Cheney's copying algorithm.

objects. Each generation is divided into one or more steps, which encode objects' age. The first step of a generation is the youngest. Objects that survive scavenges are moved to the next step. Objects in the oldest step of a generation are promoted to the youngest step of the next generation. Each step consists of a collection of fixed size blocks, which are not necessarily contiguous in memory. To simplify our implementation, our generations contained a single step.

Ungar and Jackson [27] demonstrated performance advantages from not copying large objects. The UMass garbage collector toolkit also provides a separate *large object space* (LOS) as part of the collected area. Each step has an associated set of large objects ($\geq$ 256 bytes) that are of the same age as the small objects in the step. A step's large objects, though logically members of the step, are never physically moved. Instead, they are threaded onto a doubly linked list and moved from one list to another. When a large object survives a collection, it is unlinked from its current step's list and added to the *TO space* list of the step to which it is promoted. The toolkit does not compact large object space.

The scavenger always collects a generation *g* and all generations younger than *g*. Collecting a generation involves copying all objects in the generation that are reachable from the roots (objects in the generation pointed to by objects in older generations) into free blocks. The blocks that previously held the generation can be reused. The new space to which generation objects are copied is called *TO space* and the old space is called *FROM space* [11].

A common traversal algorithm for copying objects into *TO space* is Cheney's algorithm [8] (the toolkit uses this algorithm, see Figure 2). Starting with the root set, objects are traversed in breadth-first order and copied to *TO space* as they are visited. Breadth-first traversal requires a queue. Objects to be processed are extracted from the head of the queue, while their children (if any) are added to the tail of the queue. The algorithm terminates when the queue is empty.

Cheney's algorithm does not use extra space to maintain the queue. Rather, it uses an elegant technique illustrated in Figure 3 which utilizes two pointers (*unprocessed* and *free*). Since the algorithm copies objects as they are visited, it uses these *TO space* copies as queue elements for breadth-first traversal. The head and tail of the queue are marked by the *unprocessed* and *free* pointer, respectively. Once an object is processed, it is removed from the head of the queue by incrementing the *unprocessed* pointer, and any children it may have are added to the tail of the queue by copying them to *TO space* and incrementing the *free* pointer.

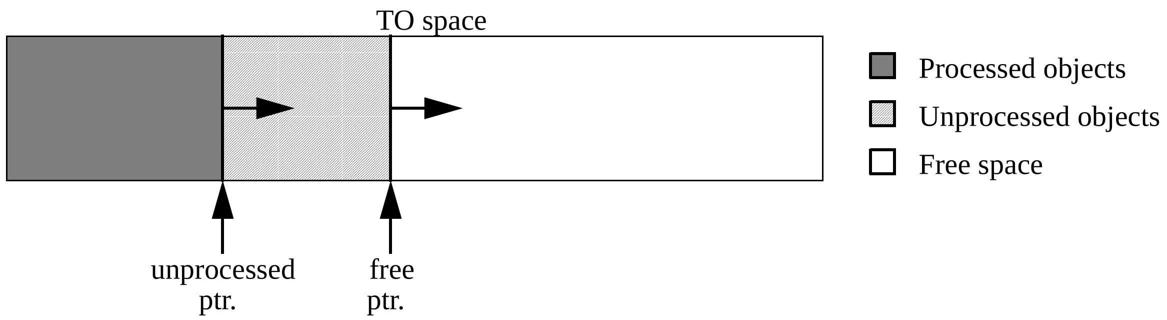

Figure 3. TO space during scavenging.

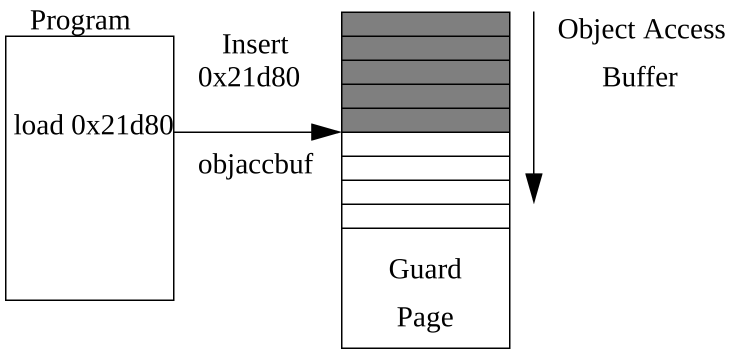

Figure 4. Object access buffer.

4. LOW OVERHEAD REAL-TIME DATA PROFILING

In the absence of programmer annotations or compiler analysis, cache-conscious data placement requires measurements of data access patterns to be effective. A profile of an earlier training run is commonly used to guide program optimizations. However, data access patterns require real-time profiling because of the difficulty of providing names for objects that are consistent and usable between runs of a program. Real-time profiling also spares a programmer an extra profile-execute cycle, as well as the difficulty of finding representative training inputs. However, the overhead of real-time profiling must be low, so the performance improvements are not outweighed by profiling costs. The rest of this section describes the design and implementation of a low-overhead, real-time data access profiler.

In the most general case, profile-guided data placement requires tracing every load and store to heap data. The overhead of such tracing (factor of 10 or more [1]) precludes its use in real-time profiling. However, two properties of object-oriented programs permit low overhead data profiling.:

- *most objects are small, often less than 32 bytes, and*
- *most object accesses are not lightweight.*

Section 7 provides experimental results to support these assumptions.

If most objects are small (< 32 bytes), then it is not necessary for data profiling to distinguish different fields within the same object, since cache blocks are currently larger (e.g., 64 bytes in the UltraSparc [25]) and growing. Profiling can be implemented at object, not field, granularity. Moreover, if most object accesses are not lightweight (i.e., multiple fields are accessed together or an access involves a method invocation), then profiling instrumentation (several instructions per object access) will not incur a large overhead.

```
ld baseobjptr, %reg
st %reg, [%objaccbuf]
add %objaccbuf, 4, %objaccbuf
```

Figure 5. Profiling instrumentation for base object address loads.

Our real-time data profiling system instruments loads of base object addresses, using information provided by a slightly modified compiler, which retains object type information until code generation to permit selective load instrumentation. The instrumentation enters the base object

```
construct_obj_affinity_graphs()
{
   limit = objaccbuf
   objaccbuf = OBJ_ACC_BUF_BASE;
   while(objaccbuf < limit)
   {
      insert_locality_queue(objaccbuf);
      if(!exists_obj_affinity_node(objaccbuf))
         create_obj_affinity_node(objaccbuf);
      increment_obj_affinity_edge_wts(objaccbuf);
      objaccbuf += 4;
   }
}
```

Figure 6. Constructing object affinity graphs.

```
insert_locality_queue(objaccbuf)
{
   if (in_locality_queue(objaccbuf))
   {
      move_to_queue_tail(objaccbuf);
   }
   else
   {
      if (is_queue_full())
         delete_queue_hd();
      insert_queue_tail(objaccbuf);
   }
}
```

Figure 7. Locality queue insertion.

address in an object access buffer, which is a sequential structure, similar to the sequential store buffer used in the garbage collection toolkit (Figure 4). This object access buffer records the temporal ordering of a program's object accesses. Figure 5 shows the instrumentation emitted for a base object address load (assuming the object access buffer pointer is stored in a dedicated register).

The object access buffer is normally processed just before a scavenge to construct object affinity graphs (Section 5). However, it may overflow between scavenges. Rather than include an explicit overflow check in the instrumentation, the virtual memory system causes a page trap on buffer overflow. The trap handler processes the buffer to construct object affinity graphs and restarts the application. Our experience indicates that setting the buffer size to 15,000 entries (60 KB) prevents overflow.

5. CONSTRUCTING OBJECT AFFINITY GRAPHS

As described in Section 3, generational garbage collection copies live objects to *TO space*. Our goal is to use data profiling information to produce a cache-conscious layout of objects in *TO space* that places objects with high temporal affinity next to each other, so that they are likely to be in the same cache block. The data profiling information captures the temporal ordering of base object addresses, which our system uses to construct object affinity graphs. An object affinity graph is a weighted undirected graph in which nodes represent objects and edges encode temporal affinity between objects.

Since generational garbage collection processes objects in the same generation together, we construct a separate affinity graph for each generation (except the first, see Section 6.2). This is possible because an object's generation is encoded in its address. Although this scheme precludes placing objects in two different generations in the same cache block, we choose this approach for two reasons. First, the importance of inter-generation object co-location is unclear. Second, the only way to achieve inter-generation co-location is to demote the older object or promote the younger object. Both alternatives have disadvantages. Since generational garbage collection copies all objects of a generation together, intra-generation pointers are not explicitly tracked. The only safe way to demote an object is to subsequently collect the generation it originally belonged to, in order to update any pointers to the demoted object, which can produce unacceptably long garbage collection times. The other option is to promote the younger object. Such promotion is safe since the younger object's generation is being collected (this will update any intra-

```
increment_obj_affinity_edge_wts(objaccbuf)
{
   queue_elem = NULL;
   init_locality_queue();
   do
   {
      queue_elem = next_queue_elem();
      if(exists_obj_affinity_edge(queue_elem, queue_tail())
         increment_affinity_edge_wt(queue_elem, queue_tail());
      else
         add_affinity_edge(queue_elem, queue_tail());
   }while (queue_elem != queue_tail())
}
```

Figure 8. Incrementing affinity graph edge weights.

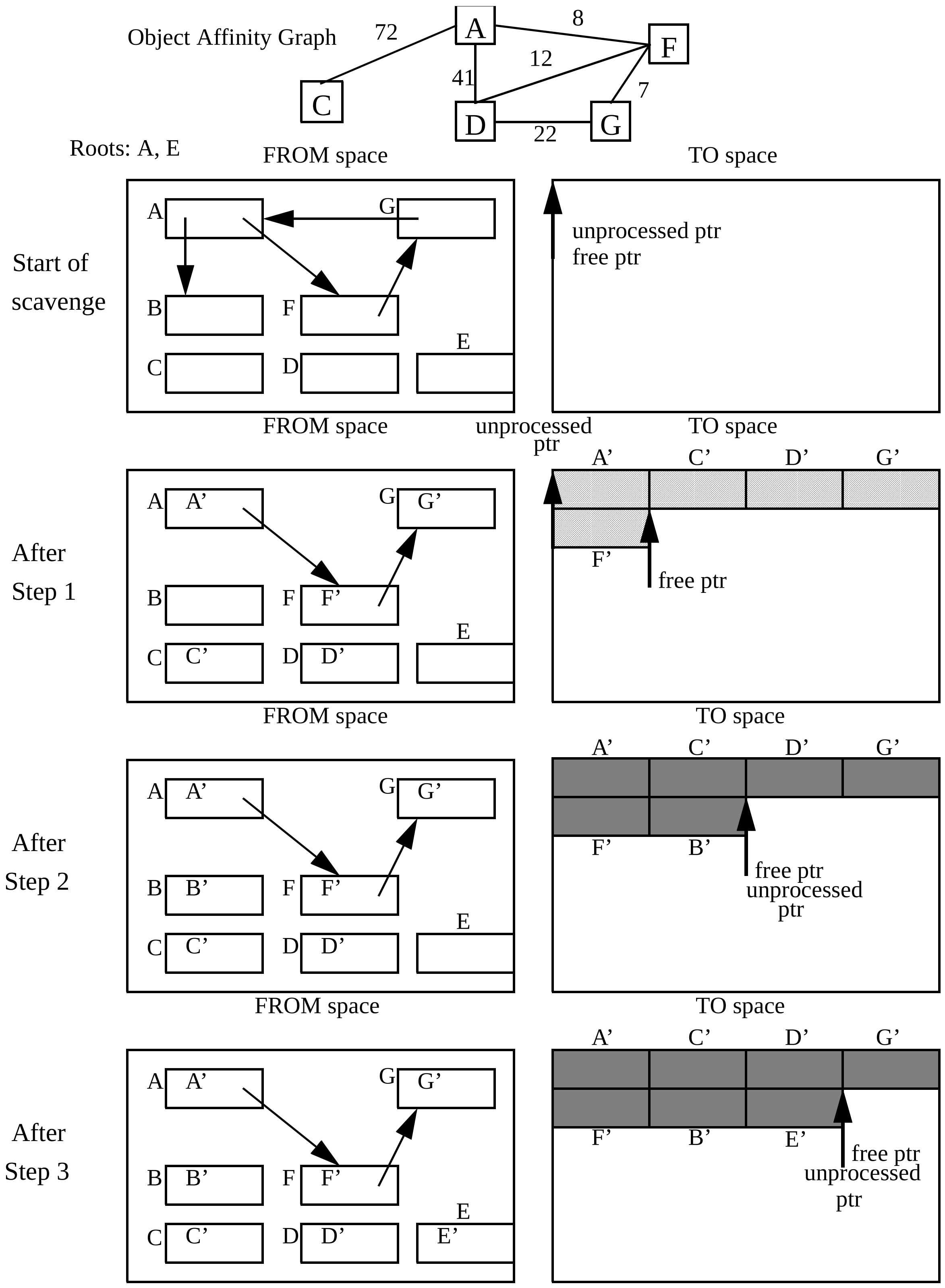

Figure 9. Combining cache-conscious data placement with garbage collection.

generation pointers to the object) Moreover, generational collectors track pointers from older objects to younger objects, so they could be updated (at a possibly high processing cost). However the locality benefit of this promotion will not start until the older generation is collected (since it cannot be co-located with the older object until that time), which may be much later. In addition, there is the danger of premature promotion if the younger object does not survive long enough to merit promotion.

Figure 6, Figure 7, and Figure 8 contain the algorithm used to construct object affinity graphs (one per generation) from profile information. The queue size used in the algorithm is an important parameter. Too small of a queue runs the risk of missing important temporal relationships, but a large queue can result in huge object affinity graphs and long processing times. We used a queue size of 3 for our experiments, since informal experiments indicated that it gave the best results. Prior to each scavenge, the object affinity graphs can either be re-created anew from the contents of the object access buffer, or the profiling information can be used to update existing graphs. The suitability of these approaches depends on application characteristics. Applications with phases that access objects in distinct manners could benefit more from re-creation (provided phase durations are longer than the interval between scavenges), whereas applications with uniform behavior might be better suited to the incremental approach. Our initial implementation re-creates the object affinity graph prior to initiating a scavenge. This permits demand-driven graph construction that builds graphs only for the generations that are going to be collected during the subsequent scavenge.

6. COMBINING CACHE-CONSCIOUS DATA PLACEMENT WITH GARBAGE COLLECTION

Cheney's algorithm copies objects to *TO space* in breadth-first order. Moon describes a modification to this algorithm that results in approximate depth-first copying [18]. Wilson et al. further refine the traversal to obtain hierarchical grouping of objects in *TO space* [30]. The copying algorithm (Figure 9) described in this section uses the object affinity graph to produce a cache-conscious layout of objects in *TO space*.

6.1 Cache-Conscious Copying Algorithm

Our cache-conscious copying algorithm can be divided into three steps.

STEP 1: Flip the roles of *FROM space* and *TO space*. Initialize the *unprocessed* and *free* pointers to the beginning of *TO space*. From the set of roots present in the affinity graph, pick the one with the highest affinity edge weight. Perform a greedy depth-first traversal of the entire object affinity graph starting from this node (i.e., visit the next unvisited node connected by the edge with greatest affinity weight). The stack depth for the depth-first traversal is limited to the number of nodes in the object affinity graph, and hence the object access buffer can be used as a scratch area for this purpose. In parallel with this greedy depth-first traversal, copy each object visited to *TO space* (increment the *free* pointer). Store this new object address as a forwarding address in the *FROM space* copy of the object. After this step all affinity graph nodes will be laid out in *TO space* in a manner reflecting object affinities (Figure 9), but will still contain pointers to objects in *FROM space*.

STEP 2: All objects between the *unprocessed* and *free* pointers are processed using Cheney's algorithm (except the copy roots portion).

STEP 3: This is a cleanup step where the root set is examined to ensure that all roots are in *TO space* (this is required as all roots may not be present in the object affinity graph or reachable from these objects). Any roots not present are copied to *TO space* and processed using Cheney's algorithm (Figure 9).

6.2 Discussion

The first step of our algorithm copies objects by traversing the object affinity graph, which may retain objects not reachable from the roots of the generation (i.e., garbage). However since the system recreates the object affinity graph from new profile information prior to each scavenge, such garbage will be incorrectly promoted at most once. In addition, we focus our cache-conscious data placement efforts on longer-lived objects and do not use our copying algorithm in the youngest generation (where new objects are allocated and most of the garbage is generated).

7. EXPERIMENTAL EVALUATION

This section presents experiments performed to support our assumption that object-oriented programs manipulate small objects (< 32 bytes), to demonstrate that our real-time data profiling technique incurs low overhead, and finally, to measure the impact of our cache-conscious object layouts on program performance.

7.1 Experimental Methodology

Our system uses the Vortex compiler infrastructure developed at the University of Washington [7]. Vortex is a language-independent optimizing compiler for object-oriented languages, with front ends for Cecil, C++, Java, and Modula-3. Unfortunately, generational garbage collection currently only works with Cecil, though efforts are underway to extend this functionality to Java [6].

Cecil [4, 5] is a dynamically-typed, purely object-oriented language. It combines multi-methods with a simple classless object model, a kind of dynamic inheritance, and modules. Instance variables in Cecil are accessed solely through messages, and can be replaced or overridden by methods. The Cecil benchmark programs used in the experiments are described in Table 1. The programs were compiled at the highest optimization level (o2), which applies techniques such as class analysis, splitting, class hierarchy analysis, class prediction, closure delaying, and inlining, in addition to traditional optimizations [7]. (We were unable to compile

.

Program	Lines of Code[a]	Description
richards	400	Operating system simulation
deltablue	650	Incremental constraint solver
instr sched	2,400	Global instruction scheduler
typechecker	20,000[b]	Typechecker for old Cecil type system
new-tc	23,500[b]	Typechecker for new Cecil type system

Table 1: Cecil benchmark programs.

a. Plus, an 11,000 line standard library.
b. The two Cecil typecheckers share approximately 15,000 lines of common support code, but the type checking algorithms are completely separate and were written by different people.

Program	# of heap allocated small objects	Bytes allocated (small objects)	Avg. small object size (bytes)	# of heap allocated large objects	Bytes allocated (large objects)	% small objects
richards	567,896	4,551,792	**8.0**	2	2,064	**100.0**
deltablue	4,575,532	40,173,296	**8.8**	2	2,064	**100.0**
instr sched	783,929	7,276,792	**9.3**	31	50,912	**100.0**
typechecker	14,095,598	118,520,372	**8.4**	1,821	1,676,104	**100.0**
new-tc	13,023,528	112,296,720	**8.6**	1,268	1,155,276	**100.0**

Table 2: Most heap allocated objects are small (< 32 bytes).

.

Program	Avg. # of live small objects	Bytes occupied (live small objects)	Avg. live small object size (bytes)	Large objects	% live small objects
richards	645	9,926	**15.4**	2	**99.7**
deltablue	16,567	305,637	**18.5**	2	**100.0**
instr sched	6,456	157,736	**24.4**	31	**99.5**
typechecker	51,627	1,114,865	**21.6**	1,821	**96.5**
new-tc	58,858	1,392,212	**23.7**	1,268	**97.9**

Table 3: Most live objects are small (< 32 bytes).

Program	Original execution time (secs)	Instrumented program execution time (secs)	% overhead of instrumentation
richards	0.202	0.213	**5.45**
deltablue	3.369	3.544	**5.19**
instr sched	3.518	3.683	**4.69**
typechecker	347.352	358.467	**3.20**
new-tc	391.250	403.378	**3.10**

Table 4: Overhead of real-time data profiling.

Program	L2 cache miss rate (base)	L2 cache miss rate (CCDP)	% reduction (L2 miss rate)	Execution time (base)	Execution time (CCDP)	% reduction (execution time)
richards	0.0131	0.0103	**21.4**	0.202	0.173	**14.4**
deltablue	0.0356	0.0240	**32.6**	3.369	2.578	**23.5**
instr sched	0.0543	0.0392	**27.8**	3.518	2.756	**21.7**
typechecker	0.0947	0.0591	**37.6**	347.352	238.179	**31.4**
new-tc	0.0979	0.0571	**41.7**	391.250	247.622	**36.7**

Table 5: Impact of our cache-conscious object layout.

Program	L2 cache miss rate (WLM)	L2 cache miss rate (CCDP)	% reduction (L2 miss rate)	Execution time (WLM)	Execution time (CCDP)	% reduction (execution time)
richards	0.0129	0.0103	**20.2**	0.211	0.173	**18.0**
deltablue	0.0341	0.0240	**29.6**	3.437	2.578	**25.0**
instr sched	0.0532	0.0392	**26.3**	3.621	2.756	**23.9**
typechecker	0.0925	0.0591	**36.1**	321.433	238.179	**25.9**
new-tc	0.0963	0.0571	**40.7**	358.512	247.622	**30.9**

Table 6: Comparison with the Wilson-Lam-Moher algorithm.

the typechecking programs at this optimization level due to a compiler assertion failure and used level o0 instead.)

The experiments were run on a single processor of a Sun Ultraserver E5000, which contained 12 167Mhz UltraSPARC processors and 2GB of memory running Solaris 2.5.1. The large amount of system memory ensures that locality benefits are due to improved cache performance and not paging activity.This system has two levels of data cache—a 16 KB direct-mapped level 1 cache with 16 byte cache blocks, and a unified (instruction and data) 1 MB direct-mapped level 2 cache with 64 byte cache blocks. The system has a 64 entry iTLB, as well as a 64 entry dTLB, both of which are fully associative. A level 1 data cache hit takes one processor cycle. A level 1 cache miss, followed by a level 2 cache hit, costs 6 additional cycles. A level 2 cache miss typically results in an additional 64 cycle delay. Each experiment was repeated five times and the average value reported (in all cases the variation between the smallest and largest values was less than 2%).

7.2 Experimental Results

Table 2 shows the results of our first set of experiments, which tested our assumption that most heap allocated objects are small. However, small objects often die fast. Since our cache-conscious layout technique is only effective for longer-lived objects, which survive scavenges, we are more interested in live object statistics. Table 3 shows the results of the next experiment, which measured the number of small objects that were live after each scavenge, averaged over the entire program execution. Once again, the results support our hypothesis that most objects are small (< 32 bytes).

The next set of experiments measured the overhead of our real-time data profiling (Table 4). The results indicate that the overhead of our real-time data profiling technique is low (< 6%).

We used the UltraSPARC’s [25] hardware counters to measure the effect of our cache-conscious object layouts on cache miss rates. Table 5 contains measurements of the overall execution time (including the instrumentation and processing overhead of our technique). Our cache-conscious layouts reduces cache miss rates by 21–42% (our technique had practically no impact on L1 cache miss rates, as L1 cache blocks are only 16 bytes), producing corresponding reductions in execution times ranging from 14–37%, despite the technique’s instrumentation and processing overhead. In addition, the data indicates that the L2 cache miss rate is correlated with total execution time.

Finally, we compared our approach against the Wilson-Lam-Moher algorithm [30], which uses a hierarchical decomposition algorithm for copying data between semi-spaces (instead of Cheney’s breadth-first traversal) to improve a program’s virtual memory (page) locality. This experiment (Table 6) investigated whether techniques designed to improve locality at the memory (page) level are effective at the cache level, and to ensure that the cache-miss rate reductions in Table 5 are not exaggerated by the poor locality of the base case (which uses Cheney’s breadth-first traversal algorithm). Comparing Table 5 and Table 6, we see that for three benchmarks (*richards*, *deltablue*, and *instr sched*), the Wilson-Lam-Moher algorithm performs worse than Cheney’s algorithm, while slightly outperforming it for *typechecker*, and *new-tc*. These surprising results are easily explained. Since the system has 2GB of memory, no application pages. In addition, the system has a 64 entry dTLB (which supports a 512KB working set), hence the only applications that might suffer dTLB misses are *typechecker*, and *new-tc* (see Table 3), which is our result. Since the Wilson-Lam-Moher algorithm is ineffective at reducing a program’s cache miss rate, and has a slightly higher overhead than Cheney’s algorithm, it performs worse for *richards*, *deltablue*, and *instr sched*.

8. RELATED WORK

White [28] first suggested using garbage collection to improve a program’s locality of reference. Researchers investigated two approaches to using a garbage collector to improve paging behavior of Smalltalk and LISP systems [18, 30, 15, 10]. Static regrouping [18, 30] uses the topology of heap data structures to rearrange structurally-related objects, while dynamic regrouping [10] clusters objects according to a program’s data access pattern. Moon [18] found that depth-first copying generally yields better virtual memory performance than breadth-first copying for LISP, because it is more likely to place parents and offspring on the same page, particularly if data structures tend to be shallow, but wide. Wilson et al. [30] treated hash tables, which group data in a pseudo-random order, specially, and ‘normal’ data structures were copied in depth-first order. Their results showed a significant reduction in the incidence of page faults. However, in a later study [15], the authors found that the optimal grouping of data structure elements was very dependent on the shape and type of the structure being copied. While hierarchical decomposition performed well for trees, it was disappointing for other structures. Court’s [10] dynamic regrouping technique takes advantage of specialized hardware to provide incremental garbage collection, which tends to move objects to *TO space* in program access order, and this can dramatically reduce the number of page faults. These studies focused on a program’s paging behavior, not its cache behavior. Our work differs, not only because of the different cost for a cache miss and a page fault, but also because cache blocks are far smaller than memory pages. As our results indicate, techniques that improve a program’s page locality, are not necessarily effective at the cache level. In addition, we attempt to lay out objects in program access order with real-time data profiling, rather than hardware support.

Recently, Calder et al. [1] applied placement techniques developed for instruction caches to data. They use a compiler-directed approach that creates an address placement for the stack (local variables), global variables, heap objects, and constants in order to reduce data cache misses. Their technique, which requires a training run to gather profile data, shows little improvement for heap objects, but significant improvement for stack objects and globals. By contrast, we use low overhead real-time data profiling and copying garbage collection to implement on-the-fly cache-conscious data placement, showing significant improvements for heap objects.

9. CONCLUSIONS

Extensive and expensive memory hierarchies require programmers to be concerned about the cache locality of their data structures. In general, properly laying out structures requires a deep understanding of a program's structures and operation. This paper describes an extremely attractive alternative for languages that support garbage collection. A generational garbage collector can easily be modified to produce cache-conscious data layouts of small objects. The paper demonstrates the feasibility of low-overhead, real-time profiling of data access patterns for object-oriented languages and describes a new copying algorithm that uses this information to produce cache-conscious object layouts. Measurements show that this technique reduces cache miss rates by 21–42% and improves program performance by 14–37%, as compared to the commonly used alternative. Techniques such as these may help narrow, or even reverse, the performance gap between high-level programming languages, such as Lisp, ML, or Java, and low-level languages such as C or C++.

10. ACKNOWLEDGEMENTS

The authors would like to thank Brad Calder, Jeffrey Dean, Patrick Dussud, Eliot Moss, Ben Zorn, and the anonymous referees for their useful comments. This research is supported by NSF NYI Award CCR-9357779, with support from Sun Microsystems, and NSF Grant MIP-9625558.

11. REFERENCES

[1] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. "Cache-conscious data placement." To appear in *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII),* Oct. 1998.

[2] David Callahan, Ken Kennedy, and Allan Porterfield. "Software prefetching." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, April 1991.

[3] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. "Compiler optimizations for improving data locality." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 252–262, Oct. 1994.

[4] Craig Chambers. "Object-oriented multi-methods in Cecil." In *Proceedings ECOOP'92, LNCS 615, Springer-Verlag,* pages 33–56, June 1992.

[5] Craig Chambers. "The Cecil language: Specification and rationale." *University of Washington Seattle, Technical Report TR-93-03-05*, Mar. 1993.

[6] Craig Chambers. "Personal communication." March 1998.

[7] Craig Chambers, Jeffrey Dean, and David Grove. "Whole-program optimization of object-oriented languages." *University of Washington Seattle, Technical Report 96-06-02*, June 1996.

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

[9] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. "Improving pointer-based codes through cache-conscious data placement." *University of Wisconsin-Madison, Technical Report CS-TR-98-1365*, Mar. 1998.

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

[11] Robert Fenichel and Jerome Yochelson. "A LISP garbage-collector for virtual-memory computer systems." *Communications of the ACM*, 12(11):611–612, 1969.

[12] Dennis Gannon, William Jalby, and K. Gallivan. "Strategies for cache and local memory management by global program transformation." *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[13] Richard Hudson, Eliot Moss, Amer Diwan, and Christopher Weight. "A language-independent garbage collector toolkit." *University of Massachusetts at Amherst technical report TR 91-47*, Sept. 1991.

[14] David Kroft. "Lockup-free instruction fetch/prefetch cache organization." In *The 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[15] M. S. Lam, P. R. Wilson, and T. G. Moher. "Object type directed garbage collection to improve locality." In *Proceedings of the International Workshop on Memory Management*, pages 16–18, Sept. 1992.

[16] James Laudon, Anoop Gupta, and Mark Horowitz. "Interleaving: A multithreading technique targeting multiprocessors and workstations." In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, San Jose, California, 1994.

[17] Henry Lieberman and Carl Hewitt. "A real-time garbage collector based on lifetimes of objects." *Communications of the ACM*, 26(6):419–429, 1983.

[18] D. A. Moon. "Garbage collection in a large LISP system." In *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pages 235–246, Aug. 1984.

[19] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. "Design and evaluation of a compiler algorithm for prefetching." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, October 1992.

[20] David Patterson, Thomas Anderson, Neal Cardwell,

Richard Fromm, Kimberly Keaton, Christoforos Kazyrakis, Randi Thomas, and Katherine Yellick. “A case for intelligent RAM.” In *IEEE Micro*, pages 34–44, Apr 1997.

[21] Sharon E. Perl and Richard L. Sites. “Studies of Windows NT performance using dynamic execution traces.” In *Second Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[22] Shai Rubin, David Bernstein, and Michael Rodeh. “Virtual cache line: A new technique to improve cache exploitation for recursive data structures.” *Submitted for publication*, Apr. 1998.

[23] Alan J. Smith. “Cache memories.” *ACM Computing Surveys*, 14(3):473–530, 1982.

[24] Burton J. Smith. “Architecture and applications of the HEP multiprocessor computer system.” In *Real-Time Signal Processing IV*, pages 241–248, 1981.

[25] Sun Microelectronics. *UltraSPARC User’s Manual*, 1996.

[26] David Ungar. “Generation scavenging: A non-disruptive high performance storage reclamation algorithm.” In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Apr. 1984.

[27] David Ungar and Frank Jackson. “An adaptive tenuring policy for generation scavengers.” *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[28] J. L. White. “Address/memory management for a gigantic LISP environment, or, GC considered harmful.” In *Conference Record of the 1980 LISP Conference*, pages 119–127, 1980.

[29] M. V. Wilkes. “Slave memories and dynamic storage allocation.” In *IEEE Trans. on Electronic Computers*, pages 270–271, April 1965.

[30] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. “Effective “static-graph” reorganization to improve locality in garbage-collected systems.” *SIGPLAN Notices*, 26(6):177–191, June 1991. *Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation*.

[31] Michael E. Wolf and Monica S. Lam. “A data locality optimizing algorithm.” *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN’91 Conference on Programming Language Design and Implementation*.