The Garbage Collection Advantage: Improving Program Locality

Xianglong Huang
The University of Texas at Austin
xlhuang@cs.utexas.edu

Stephen M Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S McKinley*
The University of Texas at Austin
mckinley@cs.utexas.edu

J Eliot B Moss
The University of Massachusetts, Amherst
moss@cs.umass.edu

Zhenlin Wang
Michigan Technological University
zlwang@mtu.edu

Perry Cheng
IBM T.J. Watson Research Center
perryche@us.ibm.com

ABSTRACT

As improvements in processor speed continue to outpace improvements in cache and memory speed, poor locality increasingly degrades performance. Because copying garbage collectors move objects, they have an opportunity to improve locality. However, no static copying order is guaranteed to match program traversal orders. This paper introduces *online object reordering* (OOR) which includes a new dynamic, online class analysis for Java that detects program traversal patterns and exploits them in a copying collector. OOR uses runtime method sampling that drives just-in-time (JIT) compilation. For each *hot* (frequently executed) method, OOR analysis identifies the hot field accesses. At garbage collection time, the OOR collector then copies referents of hot fields together with their parent. Enhancements include static analysis to exclude accesses in cold basic blocks, heuristics that decay heat to respond to phase changes, and a separate space for hot objects. The overhead of OOR is on average negligible and always less than 2% on Java benchmarks in Jikes RVM with MMTk. We compare program performance of OOR to static class-oblivious copying orders (e.g., breadth and depth first). Performance variation due to static orders is often low, but can be up to 25%. In contrast, OOR matches or improves upon the best static order since its history-based copying tunes memory layout to program traversal.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management (garbage collection)*

General Terms

Languages, Performance, Experimentation, Algorithms

Keywords

adaptive, generational, compiler-assisted, locality

*This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, ARC DP0452011, and IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

1. Introduction

The goals of software engineering and high performance are often at odds. Common wisdom holds that garbage collected languages such as Java offer software engineering benefits of reduced errors and development time, but at a cost: the collector must periodically scavenge for unused memory. Common wisdom also holds that explicitly managed languages such as C offer performance benefits at a software engineering cost. In theory, programmers can free memory as soon as possible, and use specialized allocators for memory efficiency and speed. However, C has a hidden performance cost. Because it cannot move objects without violating language semantics, it requires a non-moving allocator, such as a free list. A free-list allocator places contemporaneously allocated objects in locations free at that time, but that are not necessarily adjacent or even nearby in memory. Java does not have this restriction, and can thus use contiguous allocation to attain locality for contemporaneously allocated objects, and copying collection to place objects with temporal locality closer together.

Prior research on improving locality with generational copying collectors uses a priori static orders [13, 22], static class profiling [7, 18], and online object instance sampling [9]. *Static orders* are problematic when traversal patterns do not match the collector's single order. Although for many benchmarks the locality of static copying orders has negligible impact, we show large differences of up to 25% in total time for some benchmarks. In a Java just-in-time (JIT) compiler, the generality of *static profiling* is limited because it conflicts with dynamic class loading. *Instance based reordering* is potentially more powerful than the class based orders we introduce here, since objects with locality are not necessarily connected. However, sampling space and time overheads just for the mature objects are significant (6% in time for Cecil [9]) and miss the opportunity to improve locality when the collector promotes young (*nursery*) objects.

We introduce *online object reordering* (OOR) which includes a low cost dynamic class analysis that drives a generational copying collector [15, 21]. Previous copying collectors choose an a priori order such as breadth first to copy reachable (live) objects. This order may not match program traversal patterns, and thus exposes program performance to copying order jeopardy. The OOR collector instead uses copying orders that match program access patterns to provide locality.

OOR compiler analysis finds *hot* (frequently executed) field accesses and is low cost (at most 1.9% of total time). It piggybacks on method sampling in an adaptive JIT compiler. The adaptive compiler in Jikes RVM uses timer-driven sampling to identify hot methods, and recompiles them at higher optimization levels. At compile time, OOR analysis enumerates the field accesses in each method. During execution, when the adaptive compiler identifies a hot method (regardless of its optimization choice), OOR analysis marks as hot the fields the method accesses. At garbage collection time, the OOR collector preferentially copies referents of hot fields first together with their parent.

We further improve the OOR system by decaying heat to respond to phase changes, by exploiting Jikes RVM static analysis to exclude cold basic blocks from the reordering analysis, and by using a separate copy space to group objects of hot classes together.

In addition to the SPECjvm98 benchmarks, we use five Java programs from the DaCapo benchmark suite [6] that vigorously exercise garbage collection. Experimental results show that many programs use data structures with only one or two pointer fields, and copying order does not influence their performance much. However, a number of programs are very sensitive to copying order. For example, depth-first order consistently improves upon breadth first performance by around 25% on a wide range of heap sizes for one program. OOR protects user programs from this source of potentially adverse effects, i.e., copying order jeopardy. Running time and hardware performance counter experiments show that OOR matches or improves upon the best of the class-oblivious orders for each program by improving mutator locality. Additional experiments show that our algorithm is robust across architectures, and not very sensitive to the policy knobs.

As a final experiment, we explore the question: *How significant is the locality benefit of copying collection compared to non-moving explicit memory management?* We first demonstrate that indeed copying collectors offer a locality advantage over a high performance mark-sweep implementation [4, 5]. However, mark-sweep's space efficiency yields better total performance when heap space is limited. To examine roughly the difference between explicit mark-sweep memory management and copying collection, we compare copying total time to *only* the mutator time of mark-sweep, thus excluding the direct cost of freeing. This measure is imperfect since continuous, immediate freeing should reduce the total memory footprint and improve the locality of free-list allocations as compared to the more periodic 'inhale/exhale' pattern of mark-sweep collection. Nonetheless, the total (mutator + collector) time using a copying collector is sometimes *better* than the mutator time alone of mark-sweep. In small heaps, the idealized mark-sweep does much better, because copying collection triggers more frequent collections and this cost dominates. However, in moderate and large heaps, the mutator locality benefits of contiguous allocation and copying collection can offer total performance improvements, *including the cost of GC*, of up to 15%. Since future processors will demand locality to achieve high performance, copying garbage collection may soon combine software engineering with performance benefits. Furthermore, an OOR system will provide growing benefits with its adaptive online mechanisms that match object layout to past program usage patterns.

2. Related Work

The key investigations of this work are (1) exploiting the object reordering that happens during copying generational garbage collection [15, 21], and (2) using online profiling to collect information for controlling the copying order. Much previous research in this area considers non-garbage collected languages (such as C) [7, 8, 22], or does not address the effects of copying collectors [12]. In other words, it neither considers nor exploits the *moving* of heap objects.

The related work most pertinent to ours falls into two categories: techniques that group objects to improve *inter*-object locality [7, 9, 13, 22], and those that reorder fields within an instance to improve *intra*-object locality [7, 12]. This prior work relies on static analysis or offline profiling to drive object layout decisions and is class-oblivious for the most part, i.e., it treats all classes the same.

One can improve inter-object locality by clustering together objects whose accesses are highly correlated. The work in this area differs in how to define correlation and specific methods to cluster objects. Wilson et al. describe a hierarchical decomposition algorithm to group data structures of LISP programs using static-graph reorganization to improve locality of memory pages [22]. They found that using a two-level queue for the Cheney scan groups objects effectively. Lam et al. later conclude that hierarchical decomposition is not always effective [13]. They suggest that users supply object type information to group objects. We automatically and adaptively examine fine-grained field accesses to generate such class advice.

Chilimbi and Larus use a continuously running online profiling technique to track recently referenced objects and build a *temporal affinity graph* [9] based on the frequency of accesses to pairs of objects within a temporal interval. The object pair need not be connected by a pointer, but must lie in the same non-nursery generation to reduce overhead. Their dynamic instance-level profiling records in a buffer most pointers fetched from the heap. They report overheads of 6% for Cecil. Exploiting the timer-driven sampling, already in the adaptive optimization system of Jikes RVM, is much cheaper, while copying cannot guarantee to improve every program by at least 6% so as to overcome instance profiling costs. Their algorithm copies together objects with high affinity only during collection of the old generation whereas our system reorders objects during both nursery and old generation collections.

Chilimbi et al. split objects into hot and cold parts to group the hot parts together [7]. This technique is not fully automated and requires substantial programmer intervention. Chilimbi et al.'s clustering and coloring methods also rely on manual insertion of special allocation functions [8]. Our technique is automatic.

Intra-object locality can be improved by grouping hot fields together so that they will usually lie in the same cache line, and is most useful for objects somewhat bigger than a cache line. The size of hot objects in Java benchmarks is close to and rarely exceeds 32 bytes [10], whereas typical L1 cache line sizes are 32 or 64 bytes and L2 line sizes are 64 to 256 bytes. Thus the performance improvement offered by field reordering alone is usually small. Kistler and Franz use an LRU stack to track the temporal affinity of object fields, and they partition and reorder fields based on their affinity graph [12]. They use a mark-sweep collector, where field reordering has no effect on the object order after collection. Chilimbi et al.'s field reordering depends on profiling to generate reordering advice [7]. The programmer then follows the advice to rewrite the code and reorder fields.

Rubin, Bodik, and Chilimbi developed a framework that attempts to pull together much prior work in this area [18]. Their approach involves the following steps. (1) Produce an access trace with instance and field labels. (2) Compress the trace to fit in main memory and include only accesses relevant to a specific cache size and configuration. (3) Compute the objects with the most accesses and misses. (4) Use object properties (e.g., size, field access frequencies, field access correlations) to select optimizations. (5) Perform a hill-climbing search of possible field and object layout schemes, and model misses for each scheme on the compressed trace. Their framework would need significant changes to address moving collectors, and is practical only as an offline tool. In contrast, we exploit the reordering of objects inherent in copying collection and our online analysis is inexpensive and robust to phase behavior.

3. Background

We first describe how the adaptive compilation system in Jikes RVM works, and the generational copying collector from the Memory Management Toolkit (MMTk) that we use, to set the stage to explain the online object reordering system.

Jikes RVM [1] is an open source high performance Java virtual machine (VM) written almost entirely in a slightly extended Java. Jikes RVM *does not have* a bytecode interpreter. Instead, a fast templatedriven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then judiciously optimizes the most frequently executed methods [3]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, it then selects frequently executing methods to optimize. Finally, the optimizing compiler thread re-compiles these methods at increasing levels of optimizations.

MMTk is a composable Java memory management toolkit that implements a wide variety of high performance collectors that reuse shared components [4, 5]. It provides the garbage collectors of Jikes RVM, and we use its generational copying collector (*GenCopy*).

A generational copying collector divides the heap into two portions, a *nursery* containing newly allocated objects, and a *mature space*, containing older objects [15, 21]. It further divides the mature space into two semispaces. It collects the nursery frequently (whenever it fills up), by copying reachable nursery objects into one of the mature semispaces. Once the mature semispace is full, at the next collection the whole heap is collected, all surviving objects are copied into the empty semispace and the roles of the semispaces are flipped.

Since the generational collector collects the nursery separately from the mature space, it must assume any pointers into the nursery are live. To find these pointers, the compiler inserts *write-barrier* code, which at run time records stores of pointers from mature to nursery objects in a *remembered set*. When the collector starts a nursery collection, the remembered set forms part of the set of *root pointers*, which also consists of the stacks, registers, and static variables. It copies any referents of the root pointers that lie in the nursery, and iteratively enumerates the pointers in newly copied objects, copying their nursery referents, until it copies all reachable nursery objects. Mature space collection proceeds similarly, except the remembered set is empty and the collector copies any uncopied object, not just nursery objects. This scheme generalizes to multiple generations, but we use two.

We use a bounded generational collector. It follows Appel's flexible nursery [2], which shrinks the nursery as mature space occupancy grows, except that the nursery never exceeds a fixed chosen bound (4MB). When mature space occupancy approaches the maximum total heap size, GenCopy shrinks the nursery, until it reaches a lower bound (256KB) that triggers mature space collection. We select this configuration because it performs almost as well as the Appel nursery [2], but has more regular behavior and lower average pause times.

MMTk manages large objects (8KB or bigger) separately in a non-copying space, and puts the compiler and a few other core elements of the system into the boot image, an immortal space. Blackburn et al. include additional implementation details [4, 5].

4. Online Object Reordering

The Online Object Reordering (OOR) system is class-based, dynamic, and low-overhead. OOR consists of three components, each of which extends a subsystem of Jikes RVM: (1) static compiler analysis; (2) adaptive sampling for hot methods in the adaptive optimization subsystem; and (3) object traversal and reordering in garbage collection. Figure 1 depicts the structure and interactions of the OOR system. When Jikes RVM initially compiles a method, we collect information about field accesses within that method. Later, the Jikes RVM adaptive compilation system identifies frequently executed (*hot*) methods and blocks using sampling (see Section 3). We piggyback on this mechanism to mark hot field accesses by combining the hot method information with the previously collected field accesses. We then use this information during garbage collection to traverse the hot fields first. The next three sections discuss each component in more detail.

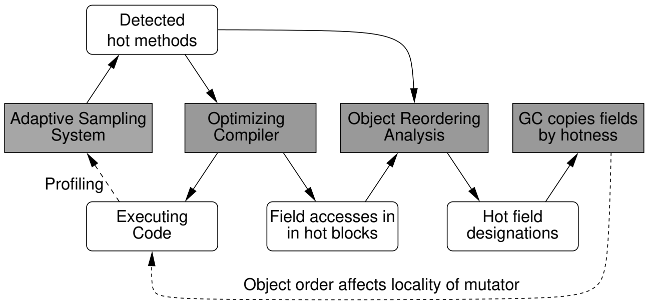


Figure 1: OOR System Architecture

4.1 Static Identification of Field Accesses

OOR analysis first identifies potentially hot fields by noting field accesses when first compiling each method. The Jikes RVM optimizing compiler uses a static analysis with a coldness threshold to mark cold basic blocks. OOR does not enumerate field accesses in cold blocks, and uses the compiler's default threshold (see Section 6.5). The compiler uses loop and branch prediction heuristics to estimate the execution frequency of basic blocks in a method. For example, it marks exception handler basic blocks as cold, and basic blocks in loops as potentially hot. For each method, OOR analysis enumerates all the field accesses in potentially hot blocks, generating tuples of the form <`class`, `offset`>. The tuples identify the class and offset of any potentially hot field, and OOR associates each tuple with the compiled method. This analysis thus filters out field accesses the compiler statically determines are cold and associates a list of all non-cold field accesses with each compiled method. At present, we do not perform any field access analysis in the Jikes RVM baseline compiler. Since the Jikes RVM adaptive compilation framework recompiles hot methods with the optimizing compiler, we use it to apply our analysis selectively to hot methods. Jikes RVM also collects basic-block dynamic execution frequencies using a counter on every branch. We believe this information can improve the accuracy of OOR analysis, although we have not implemented this feature here.

4.2 Dynamically Identifying Hot Fields

The Jikes RVM adaptive sampling system detects hot methods by periodically sampling the currently executing method. When the number of samples for a method grows beyond a threshold the adaptive system invokes the optimizing compiler on it. OOR analysis piggybacks on this mechanism. The first time it identifies a hot method, it marks all the potentially hot field access for the method as hot. Each time the sampling mechanism re-encounters a hot method (regardless of whether the adaptive system recompiles it), it updates the heat metric for the corresponding hot fields.

Figure 2 shows OOR's decay mechanism for adapting to phase changes. Other policies are possible of course. The high and low heat thresholds, *HI* and *LO* (default values of 100 and 30 respectively) indicate the hottest field with heat *HI*[1] Any field cooler than *LO* is regarded as cold. Initially all fields are cold, with heat 0. When the timer goes off, the heuristic records the current sampling time, NOW(), and updates one or more heat fields in *class* for the method.

This heuristic decays heat for unaccessed fields based on the last time the analysis updated the instantiating class, *class.lastUpdate*. However, the heuristic does not decay field heat for all classes every sample period, since the cost would be prohibitive. Instead, it updates a class only when the adaptive compiler samples another method that uses a field instantiated by it. In the worst case of not strictly decaying field

[1]The units for these thresholds are sample intervals, which are approximately 10ms: *HI* $\approx$ 1 sec, *LOW* $\approx$ 0.3 sec.

```
DECAY-HEAT(method)
for each fieldAccess in method do
  if POTENTIALLYHOT(fieldAccess) then
    hotField ← fieldAccess.field
    class ← hotField.instantiatingClass
    class.hasHotField ← true
    for each field in class do
      period ← NOW() − class.lastUpdate
      decay ← HI/(HI + period)
      field.heat ← field.heat ∗ decay
      if field.heat < LO then
        field.heat = 0
    hotField.heat ← HI
    class.lastUpdate ← NOW()
```

Figure 2: Pseudocode for Decaying Field Heat

heat for all classes, the OOR collector will copy an old object using obsolete hot field information. Since none of the hot methods access this field, the order in which the collector copies these objects will simply be based on access orders further back in history and should not degrade performance. If these objects never become hot again, this mechanism does no harm. Otherwise, if their past accesses predict the future, program locality will benefit.

4.3 Reordering during Garbage Collection

The copying phase of the collector applies OOR advice. For each instance of a class, the collector traverses the hot fields (if any) first. At class load time, the OOR system constructs an array for each class, with one integer representing the heat of each field in the class. Initially all fields have a heat of zero. OOR analysis uses the algorithm in Figure 2 to set the heat value for each field and thus identify hot fields to the collector. The OOR collector then copies and enqueues the hot fields first. Figure 3 shows how the collector copies data. For a nursery collection, it begins by processing the remembered sets (these are empty in a full heap collection), and then processes the roots. ADVICE-PROCESS() places all uncopied objects (line 2) in the copy buffer, and updates the pointer for already copied objects. ADVICE-SCAN()) then copies all the hot fields first (line 3), and enqueues the remaining fields to process later. Without advice, all fields are cold.

We also experiment with using a *hot space* that segregates hot objects from the others to increase their spatial locality, which should improve cache line utilization, reduce bus traffic, and reduce paging. We refine hot objects to *hot referents*—instances referred to by hot fields, and *hot parents*—instances of classes that instantiate hot fields. When copying an object, it is identified as a hot parent if the *hasHotField* value of the object's class is true. Hot referents are discovered when traversing hot fields. The *hot space* contains all the hot objects and is part of the older space; during nursery garbage collection, the collector copies into the hot space all objects that contain hot fields and the objects to which the hot fields point. During older generation collection, the collector copies objects in the hot space to a new hot space. It always copies all other objects into a separate space in the older generation. We do not need to change the write barrier to add a hot space since we always collect it at the same time as other objects in the older generation. Therefore, this change does not influence write barrier overhead in the mutator.

An advantage of advice-directed traversal is that it is not exclusive. For those objects without advice, we can use the best static traversal order available to combine the benefit of both methods. In our current implementation, the default copy order is pure depth first for cold objects, last child first, because this static order generally generates good performance, as we will show in the following section.

5. Methodology

We now describe our experimental methodology, platforms, and relevant characteristics of the benchmarks we use. We use two methodologies for these experiments. (1) The *adaptive* methodology lets

```
ADVICE-BASED-COPYING()
 1  Objects ← EMPTYQUEUE()
 2  Cold ← EMPTYQUEUE()
 3
 4  for each location in Remsets do
 5    ADVICE-PROCESS(location)
 6  for each location in Roots do
 7    ADVICE-PROCESS(location)
 8  repeat
 9    while Objects.NOTEMPTY() do
10      ADVICE-SCAN(Objects.DEQUEUE())
11    while Cold.NOTEMPTY() do
12      ADVICE-PROCESS(Cold.DEQUEUE())
13  until Objects.ISEMPTY()

ADVICE-PROCESS(location)
 1  obj ← ∗location
 2  if NEEDSCOPYING(obj) then
 3    Objects.ENQUEUE(COPY(obj))
 4  if FORWARDED(obj) then
 5    ∗location ← NEWADDRESS(obj)

ADVICE-SCAN(obj)
 1  for each field in obj.fields() do
 2    if field.isHot(location)  // advice
 3      then ADVICE-PROCESS(obj.field)
 4      else Cold.ENQUEUE(obj.field)
```

Figure 3: Pseudocode for Advice Based Copying

the adaptive compiler behave as intended and is non-deterministic. (2) The *pseudo-adaptive* methodology is deterministic and eliminates memory allocation and mutator variations due to non-deterministic application of the adaptive compiler. We need this latter methodology because the non-determinism of the adaptive compilation system makes it a difficult platform for detailed performance studies. For example, we cannot determine if a variation is due to the system change being studied or just a different application of the adaptive compiler.

In the adaptive methodology, the adaptive compiler uses non-deterministic sampling to detect hot methods and blocks, and then tailors optimizations for the hot blocks. Thus on different executions, it can optimize different methods and, for example, choose to inline different methods. Furthermore, any variations in the underlying system induce variation in the adaptive compiler. We use this methodology only for measuring the overhead of our system.

For all other experiments, we use a deterministic methodology that holds the allocation load and the optimized code constant. The pseudo-adaptive methodology gives a mixture of optimized and unoptimized code that reflects what the adaptive compiler chooses, but is specified by an advice file from a previous run. We run each benchmark five times and profile the optimization plan of the adaptive compiler for each run. We pick the optimization plan of the run with best performance and store it in an advice file. For the performance measurement runs, we execute two iterations of each benchmark and report the second. We turn off the adaptive compiler, but not the adaptive sampling. In the first iteration, the compiler optimizes selected methods at selected level of optimization according to the advice file. Before the second iteration, we perform a whole heap collection to flush the heap of compiler objects. We then measure the second iteration. Thus we have optimized code only for the hot methods (as determined in the advice file). This strategy minimizes variation due to the adaptive compiler since the workload is not exposed to varying amounts of allocation due to the adaptive compilation. We measure only the application behavior and exclude the compiler in this methodology.

We report the second iteration because Eeckhout et al. show that measuring the first iteration, which *includes* the adaptive compiler, is dominated by the compiler rather than the benchmark behavior [11].

For each experiment we report, we execute the benchmark five times, interleaving the compared systems. We use the methodologies

	Adaptive				Fixed GC load														
	classes	methods	alloc	alloc:	alloc	alloc:	% nrs	% wb	alloc pointers			scan pointers			scan non-null pointers		
Benchmark	loaded	compiled	(MB)	min	(MB)	min	srv	take	0	1	many	0	1	many	0	1	many
jess	155	507	403	25:1	261	17:1	1	0.08	18%	40%	42%	1%	52%	47%	7%	49%	43%
jack	61	331	307	22:1	231	17:1	3	3.15	48%	31%	22%	21%	44%	35%	34%	53%	13%
javac	160	821	593	23:1	185	7:1	23	1.21	29%	34%	37%	5%	27%	68%	6%	34%	60%
raytrace	34	227	215	12:1	135	8:1	2	0.01	89%	1%	10%	55%	12%	33%	57%	14%	29%
mtrt	35	225	224	11:1	142	7:1	5	0.65	87%	2%	11%	55%	12%	33%	57%	14%	29%
compress	16	99	138	8:1	99	6:1	0	1.20	56%	34%	10%	41%	31%	29%	43%	37%	20%
db	8	92	119	6:1	82	4:1	9	1.21	4%	95%	1%	42%	53%	5%	42%	53%	5%
mpegaudio	59	270	51	4:1	3	1:1	0	0.00	76%	15%	10%	83%	5%	12%	83%	5%	12%
ps-fun	347	522	8602	410:1	8589	409:1	0	0.00	95%	2%	3%	25%	30%	45%	25%	30%	44%
ipsixql	120	381	1777	105:1	1739	102:1	31	1.17	40%	3%	56%	39%	2%	59%	39%	2%	59%
hsqldb	90	432	6804	76:1	6720	75:1	4	0.01	44%	41%	16%	50%	0%	50%	50%	0%	50%
jython	175	1050	796	47:1	722	42:1	1	0.103	0%	78%	22%	1%	62%	37%	2%	64%	34%
antlr	114	719	22	18:1	5	3:1	11	1.78	68%	23%	9%	25%	26%	48%	30%	41%	28%
pseudojbb	13	92	339	7:1	216	5:1	32	1.82	51%	26%	23%	36%	29%	35%	37%	29%	34%

Table 1: Benchmark Characteristics

above, and take the fastest time. The variation between these measurements is low. We believe this number is relatively undisturbed by other system factors. When measuring the system overhead in the adaptive compiler, we believe the low variation from the fastest time reflects a stable application of the adaptive compiler.

5.1 Experimental Platform

We perform our experiments on four platforms and find similarities across these. Section 7 reports on cross architecture results. For brevity and unless otherwise noted, we report experiments on a machine with the following characteristics:

3.2GHz P4 The machine is a 3.2 GHz Pentium 4 with hyper-threading enabled and user accessible performance counters. It has a 64 byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, 1GB main memory, and runs Linux 2.6.0.

We instrument MMTk and Jikes RVM to use the CPU's performance counters to measure cycles, retired instructions, L1 and L2 cache misses, and TLB (translation look-aside buffer) misses of both the mutator and collector, as we vary the collector algorithm, heap size, and other features. Because of hardware limitations, each performance counter requires a separate execution. We use version 2.6.5 of the *perfctr* Intel/x86 hardware performance counters for Linux with the associated kernel patch and libraries [17].

5.2 DaCapo Benchmarks

As part of an ongoing effort with our collaborators in the DaCapo project [16], we collected several memory intensive Java programs for the DaCapo benchmark suite [6].[2] These benchmarks are intended to exercise garbage collection vigorously in order to reveal collector and platform induced differences.

1. **antlr**: Language tool for constructing complier, language recognizer, and translators.
2. **hsqldb**: Database written in Java.
3. **ipsixql**: Persistent XML database.
4. **jython**: Python interpreter written in Java.
5. **postscript-fun**: A PostScript interpreter.

5.3 Benchmark Characteristics

Table 1 shows key characteristics of our benchmarks using the fixed workload and adaptive methodologies. We use the eight SPECjvm98 benchmarks, five DaCapo benchmarks, plus pseudojbb, a variant of SPECjbb2000 [19, 20] that executes a fixed number of transactions (70000), rather than running for a fixed time (for comparisons under a fixed garbage collection load). The *alloc* columns in Table 1 indicate the total number of megabytes allocated under adaptive and fixed GC loads respectively. The *alloc:min* column lists the ratio of total allocation to the minimum heap size in which the program executes in MMTk. Including the adaptive compiler substantially increases allocation and collector load (compare column four with six, and five with seven). This behavior can obscure program behaviors and further confirms Eeckhout et al. [11]. Notice that mpegaudio allocates only 3MB, and with a 4MB heap is never collected; hence we exclude it from the remaining experiments. Also notice that the DaCapo benchmarks place substantially more load on the memory management system than the SPECjvm98 benchmarks.

The *% nursery survival* column indicates the percent of allocation in the nursery that the collector copies. OOR can influence the subset of these objects with two or more non-null pointers. Notice that most programs follow the weak generational hypothesis, but that javac and ipsixql are memory intensive while not being very generational. However, generational collectors still improve their performance [4].

The *% wb take* column shows the percent of all writes that the write barrier records in the remembered set. The remaining columns indicate the percentage of objects with 0, 1, or many pointer fields. The *alloc pointers* column indicates these proportions with respect to allocated objects. The *scan pointers* column indicates the proportions with respect to objects scanned at collection time, and *scan non-null pointers* indicates the proportions with respect to non-null pointers in objects scanned at collection time. Since OOR influences only objects with two or more non-null pointers, the final column in Table 1 indicates the proportion of scanned (copied) objects to which OOR can be applied effectively.

We ran all the experiments we report here on all the benchmarks. For all but four of these benchmarks, performance variations due to copying orders are relatively small. For brevity and clarity, the results section focuses on programs that are sensitive to copy order, and just summarizes the programs where copy order has little effect.

6. Experimental Results

We now present evaluation of our online object reordering system. We begin with results that show that the overhead for the reordering analysis, including its use by the collector, adds at most 1 to 2% to total time. We then show some programs are sensitive to copying order. Comparisons with OOR show that it essentially matches or improves over the oblivious orders. A series of experiments demonstrates the sensitivity of OOR to the decay of field heat to respond to phase changes, the use of a hot space, cold block analysis, and hot method analysis. We also compare OOR with class-oblivious copying on three additional architectures. Static ordering performance is not

[2] A pre-release of these benchmarks may be downloaded at:
http://www.cs.utexas.edu/users/speedway/dacapo/DaCapo.html

Benchmark	Default	OOR	Overhead
jess	4.39	4.43	0.84%
jack	5.79	5.82	0.57%
raytrace	4.63	4.61	-0.59%
mtrt	4.95	4.99	0.7%
javac	12.83	12.70	-1.05%
compress	8.56	8.54	-0.2%
pseudojbb	13.39	13.43	0.36%
db	18.88	18.88	-0.03%
antlr	0.94	0.91	-2.9%
gcold	1.21	1.23	1.49%
hsqldb	160.56	158.46	-1.3%
ipsixql	41.62	42.43	1.93%
jython	37.71	37.16	-1.44%
ps-fun	129.24	128.04	-1.03%
mean			**-0.19%**

Table 2: OOR System Overhead

always consistent across architectures. However, OOR consistently attains essentially the same performance as the best static order across these platforms.

6.1 Overhead of Reordering Analysis

To explore the overhead of the analysis, we measure the first iteration of the benchmark (where the compiler is active) with the adaptive compiler on a moderate heap size (1.8 × maximum live) and pick the fastest of 5 runs. This experiment performs the additional run-time work to record hot class fields, and examines the results at collection time, but never acts on those results. Therefore, the system does all the work of class reordering, but obtains no benefit from it. Table 2 compares the original adaptive system with the augmented system. The table shows some improvements as well as degradations. At worst, OOR adds a 2% overhead, but this overhead is obscured by larger variations due to the timer-based sampling. For the exact same program, VM, and heap size, the timer-based sampling can cause variations up to 5% because of the non-determinism, and this variation is the dominant factor, not the OOR analysis.

6.2 Class Sensitive vs. Class Oblivious

This and all remaining sections apply the pseudo-adaptive methodology, reporting only application behavior. This section compares static and OOR copying orders. OOR uses a hot space (Sections 4.3 and 6.4), the decay function described in Section 4.2, and excludes field accesses from cold blocks (Sections 4.1 and 6.5). This configuration produces the best results across all architectures.

Most of the benchmark programs vary due to copy order by less than 4%. However, four programs (jython, db, jess, and javac) show variations of up to 25% due to copying order, so we focus on them. Figure 4 (jess) and Figure 5 (jython, db, javac) compare OOR with three static, class-oblivious orders: breadth first, depth first, and partial depth first using the first two children (a hierarchical order). The figures present total time, mutator time, mutator L2 misses (from performance counters), and garbage collection time. Notice that the total time of jess and javac and the mutator L2 misses of jython use scales different from the other benchmarks in the figures.

First consider variations due to a priori breadth or depth first on db and jython (Figure 5). In db, class-oblivious depth first and partial depth first using the first two children perform over 25% better in total time than breadth first copying order. For jess (Figure 4), partial depth first is more than 20% worse than breath first. For jython, depth first performs about 18% better than breadth first and partial depth first. Locality explains these differences as shown in the mutator time and L2 miss graphs. For a few other programs, partial depth first offers a minor improvement (1 to 4%) over the best of breadth or depth first. The wide variation in performance is a pathology of static copying orders, and is of course undesirable.

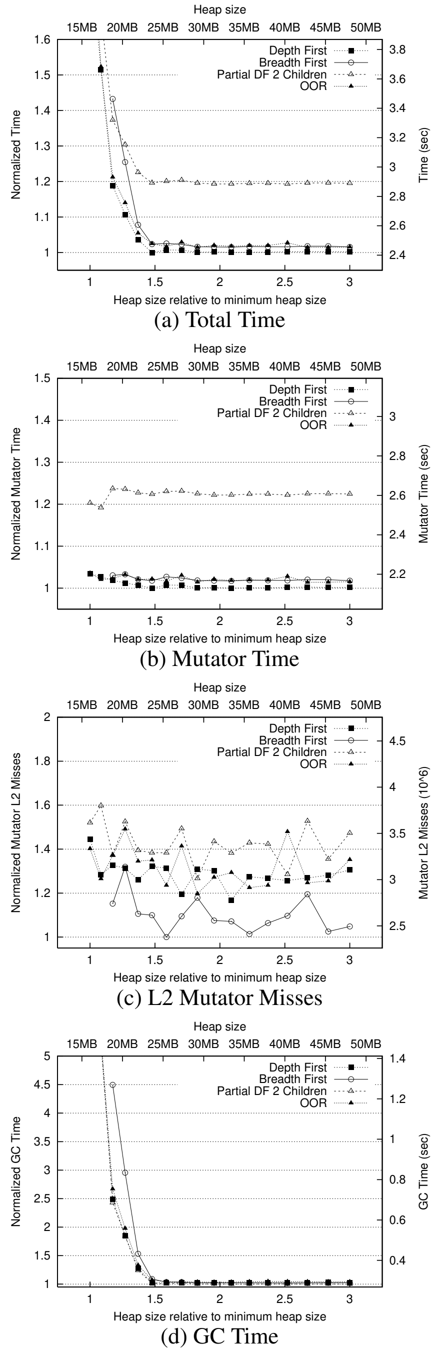
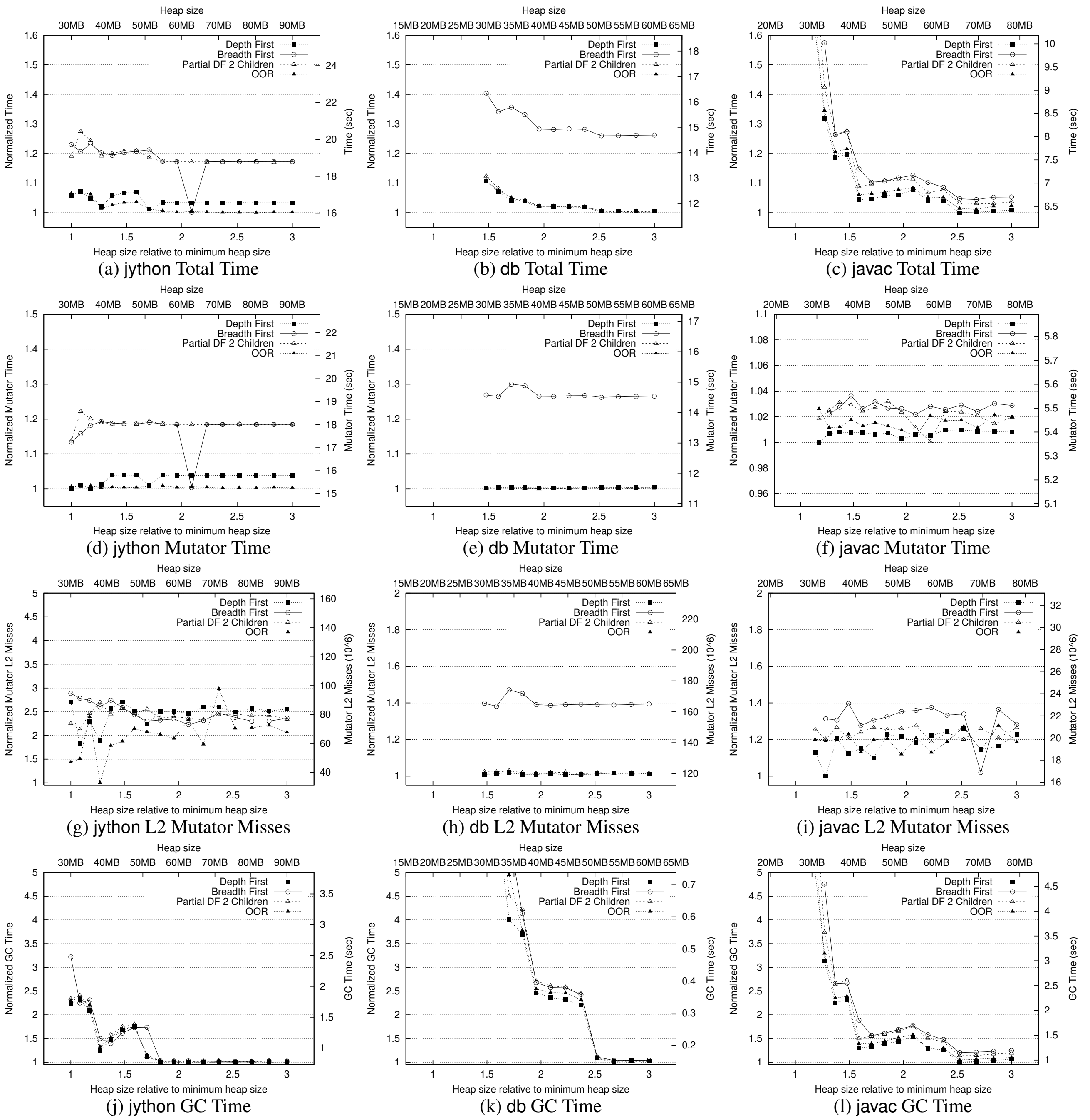


Figure 4: OOR vs. Class-Oblivious Traversals [jess]

Figures 4 and 5 show that OOR is not subject to this variation and matches or improves over the best static orders. In javac and jess, OOR sometimes degrades mutator time by 2 to 3% which degrades the total performance by 2%. The worst case for OOR on all benchmarks and platforms is 4% for ipsixql on the 3.2 GHz P4. For all other benchmarks, OOR matches or improves over the best mutator locality and total performance.

These results are consistent with cache and page replacement algorithms, among others, that use past access patterns to predict the future. OOR dynamically tunes itself to program behavior and thus protects copying garbage collection from the high variations that come from using a single static copying order that may or may not match program traversal orders.



(a) jython Total Time (b) db Total Time (c) javac Total Time

(d) jython Mutator Time (e) db Mutator Time (f) javac Mutator Time

(g) jython L2 Mutator Misses (h) db L2 Mutator Misses (i) javac L2 Mutator Misses

(j) jython GC Time (k) db GC Time (l) javac GC Time

Figure 5: OOR vs. Class-Oblivious Traversals [jython, db & javac]

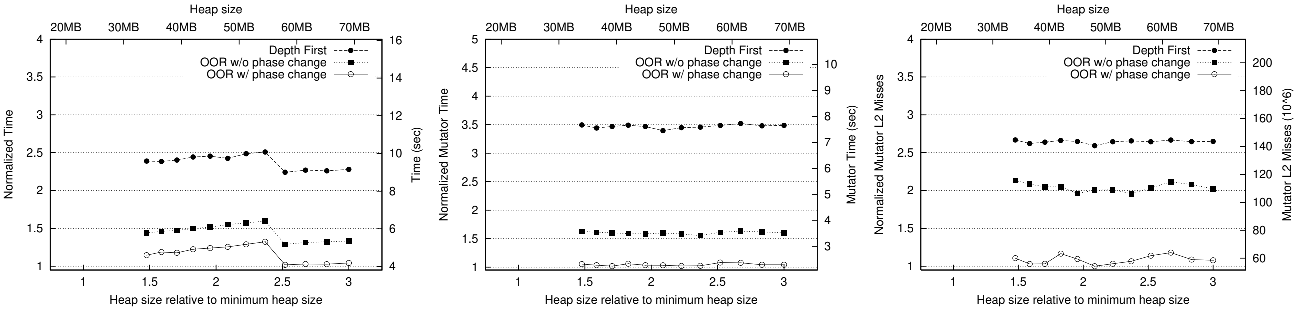
6.3 Capturing Phase Changes

OOR can adapt to changes *within* the execution of a given application. Section 4.2 describes how the decay model ensures that field heat metrics adapt to changes in application behavior. We now examine the sensitivity of this approach. We use a synthetic benchmark, phase, which exhibits two distinct phases. The phase benchmark repeatedly constructs and traverses large trees of arity 11. The traversals favor a particular child. Each phase creates and destroys many trees and performs a large number of traversals. The first phase traverses only the 4th child, and the second phase traverses the 7th child.

Figure 6 compares the default depth first traversal in Jikes RVM against OOR and OOR without phase change detection on the phase benchmark. Phase change detection improves OOR total time by 25% and improves over the default depth first traversal by 55%. Mutator performance is improved by 37% and 70% respectively (Figure 6(b)). Much of this difference is explained by reductions in L2 misses of 50% and 61% (Figure 6(c)). Figure 7 compares OOR with and without phase change detection on jess, jython, javac, and db. These and the other benchmarks are insensitive to OOR's phase change adaptivity, which indicates that they have few, if any, traversal order phases.

6.4 Hot Space

In order to improve locality further, OOR groups objects with hot fields together in a separate copy space within the mature space, as described in Section 4.3. Figure 8 shows results from four representa-



(a) phase Total Time (b) phase Mutator Time (c) phase Mutator L2 Misses

Figure 6: Performance Impact of Phase Changes Using a Synthetic Benchmark

tive benchmarks for OOR with and without a hot space. On average, these configurations perform similarly. However, in our experiments for other platforms, we found OOR with the hot space usually has slightly better results (see Figure 11(b) in Section 7). The hot space generally reduces the footprint of the hot objects but this benefit is not as significant as copying order.

6.5 Hot Field Analysis

We now explore the impact of the Jikes RVM static analysis thresholds for basic block heat on OOR (see Section 4.1). The Jikes RVM optimizing compiler assigns a heat value to basic blocks based on static loop iteration estimates (or counts if available) and branches. It then classifies them as hot or cold based on a run-time configuration threshold. OOR directly uses this classification to enumerate field accesses in hot basic blocks. The default configuration marks the fewest blocks cold (BB1 in Figure 9). BB20 through BB150 mark increasingly more basic blocks cold. Figure 9 presents the sensitivity of OOR to this threshold. Most of the benchmarks, including jess and javac, are fairly insensitive to it, but jython is particularly sensitive, with a worst case degradation of 20%. For db, when OOR marks only basic blocks with heat greater than 20 as hot, the program has the worst performance. One possible explanation is that this threshold causes OOR to distribute an important data structure between the hot and cold spaces. With thresholds higher and lower than 20, OOR probably tends to put the whole data structure in one space or the other. Based on these results, we use the Jikes RVM default and mark as hot any basic block with a heat greater than or equal to one.

6.6 Hot Method Analysis

Finally, Figure 10 examines the sensitivity of the sampling frequency for selecting hot methods. Hot methods are identified according to the number of times the adaptive optimization infrastructure samples them. Figure 10 shows OOR with sampling rates of 20ms, 10ms, and 5ms. More frequent sampling marks more methods as hot. OOR is quite robust with respect to this threshold. One possible explanation for this insensitivity is that method heat tends to be bimodal—methods are either cold or very hot. Another explanation is that warm methods (those neither hot nor cold) tend not to impact locality through field traversal orders.

7. Different Platforms

This section examines the sensitivity of OOR to architecture variations, including processor speed and memory system. We run the same experiments as before on an three additional architectures.

933MHz PPC The Apple G4 has a 933MHz PowerPC 7450 processor, separate 32KB on-chip L1 data and instruction caches, a 256KB unified L2 cache, 512MB of memory, and runs Linux 2.4.25.

1.9GHz AMD The 1.9GHz AMD Athlon XP 2600+ has a 64 byte L1 and L2 cache line size. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, *exclusive* 512KB 16-way set associative L2 cache. The L2 holds only replacement victims from the L1, and does not contain copies of data cached in the L1. The Athlon has 1GB of main memory and runs Linux 2.6.0.

2.4GHz P4 The 2.4GHz Pentium 4 uses hyper-threading. It has a 64 byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, and a 512KB unified 8-way set associative L2 on-chip cache, 1 GB main memory, and runs Linux 2.6.0.

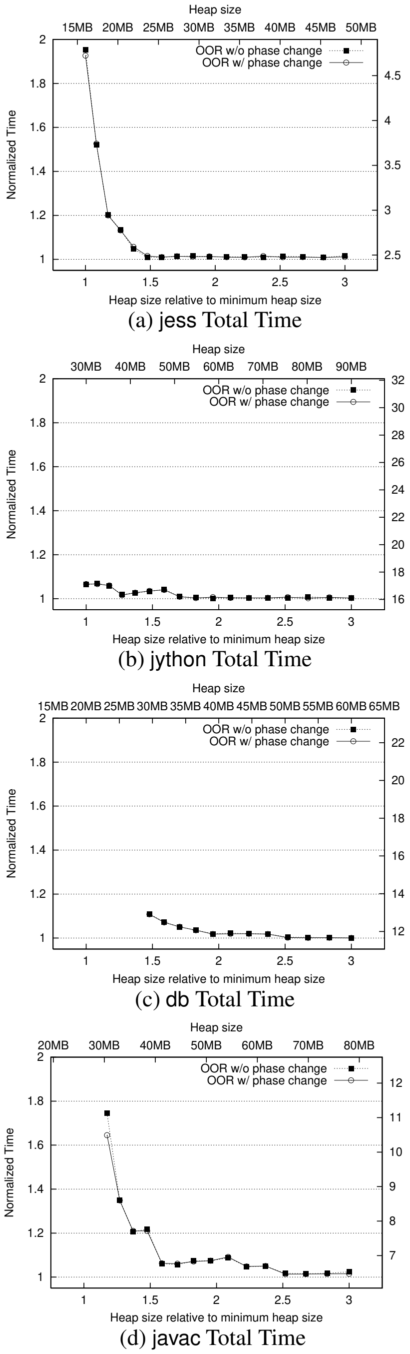
3.2GHz P4 The 3.2GHz Pentium 4 is configured identically to the 2.4GHz P4 except for the faster clock speed (see Section 5.1).

We present a representative benchmark with variations due to locality. Figure 11 shows jython on all four architectures. Not surprisingly, the two Intel Pentium 4 architecture graphs have very similar shapes and the 3.2GHz P4 is faster. Comparing between architectures shows that the memory architecture mainly dictates differences among traversal orders. The 1.9GHz AMD and 933MHz PPC are less sensitive to locality because they have larger and relatively faster caches compared to the P4s which have higher clock speeds. Interestingly, the slower AMD processor achieves the best, performance, possibly due to its large non-inclusive caches. However, on all four architectures, OOR consistently provides the best performance, across all benchmarks and architectures.

8. The Copying Advantage

We now present evidence confirming the locality advantages of copying. We first examine mutator locality by comparing a standard copying collector with a non-copying mark-sweep collector. We then compare the mutator time of a non-copying mark-sweep collector with the total time of the copying collector to see whether the benefits of copying can ever outweigh the cost of garbage collection.

Figure 12(a) compares just the mutator performance of the bounded (4MB) nursery generational copying collector using OOR to a whole heap mark-sweep collector [5], labeled OOR and Mark-Sweep respectively. The figure shows mutator time as a function of heap size for javac, a representative program. OOR has a mutator-time advantage of around 8-10% over Mark-Sweep due to fewer L1 misses on javac (Figure 12(b)). The L2 and TLB misses follow the same trend, and this advantage holds across all of our benchmarks, ranging from a few percent on jython and compress, to 15% on pseudojbb and 45% on ps-fun. Our analysis confirms a prior result [4]: it is *locality* rather than the cost of the free-list *mechanism* that accounts for the performance gap. Note that this result is contrary to the oft-heard claim that non-moving collectors 'disturb the cache less' than do copying collectors.



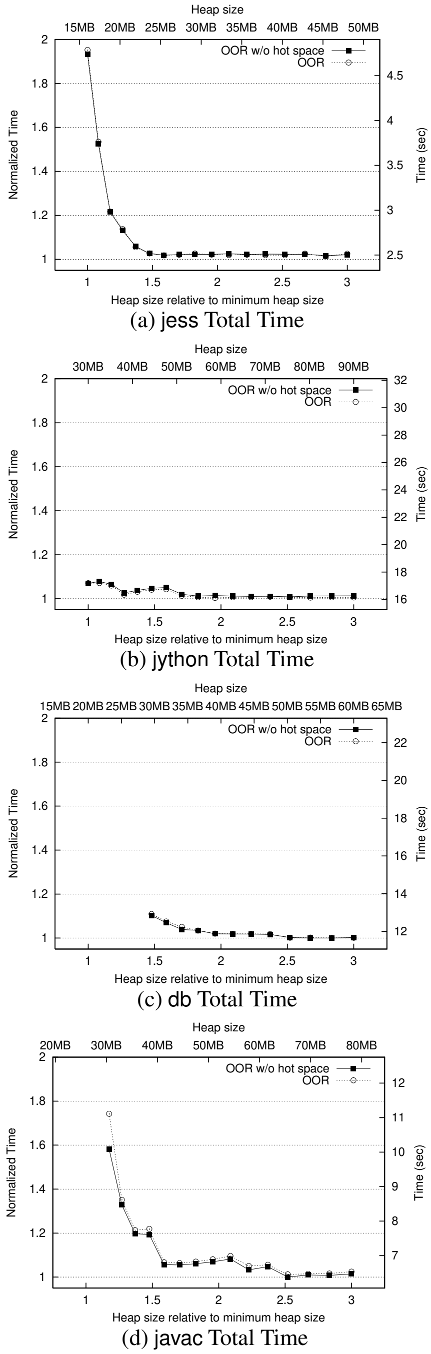
(a) jess Total Time

(b) jython Total Time

(c) db Total Time

(d) javac Total Time

Figure 7: Absence of Phasic Behavior in Standard Benchmarks



(a) jess Total Time

(b) jython Total Time

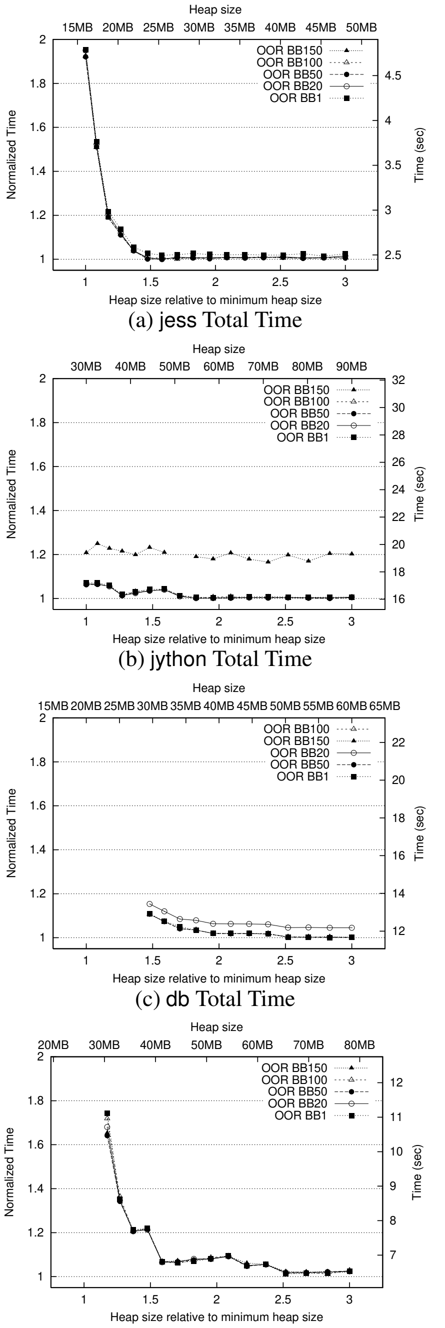
(c) db Total Time

(d) javac Total Time

Figure 8: OOR without Hot Space

We now examine the overall impact of garbage collection when the locality advantage and collection overhead are combined. Figure 13 compares the *total* execution time of the copying collector (labeled OOR) with the *mutator* time of mark-sweep (Mark-Sweep), which we regard as an approximation to the performance of explicit memory management. We use a standard free-list allocator [14, 4] and subtract the cost of garbage collection. The approximation is imperfect. On one hand, the application does not pay the cost of `free()`. On the other hand, it does not reclaim memory as promptly as explicit memory management does. In both graphs, the performance of the copying collector is normalized against the mutator time for Mark-Sweep. A result less than 1 indicates that the total time for the copying collector is less than the Mark-Sweep mutator time.

Three patterns emerge in our results. Figure 13(a) shows three representative benchmarks: pseudojbb, ps-fun, and ipsixql. ipsixql is the only outlier where the Mark-Sweep mutator actually has consistently better performance than the copying collector. Seven benchmarks are like pseudojbb. In modest to large heaps, the locality advantage of copying garbage collection compensates for its collection costs, to the point where the total time of OOR is the same as the mutator time of Mark-Sweep. ps-fun represents five benchmarks, where the locality advantage is so significant that OOR improves over the mutator time



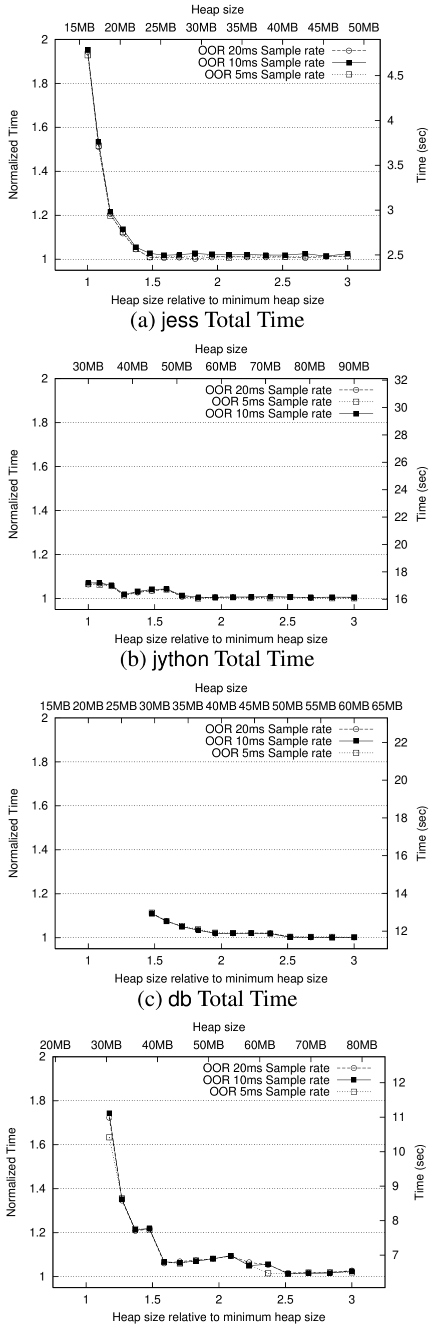
(a) jess Total Time

(b) jython Total Time

(c) db Total Time

(d) javac Total Time

Figure 9: Using Different Policies to Determine Cold Fields



(a) jess Total Time

(b) jython Total Time

(c) db Total Time

(d) javac Total Time

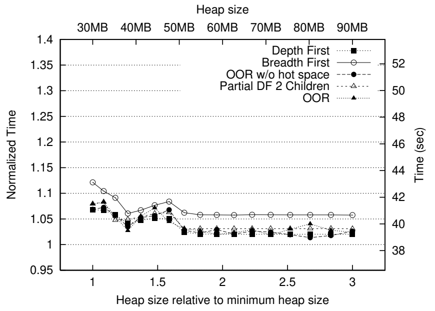
Figure 10: Using Different Policies to Determine Hot Methods

of Mark-Sweep, even in small heap sizes. Figure 13(b) is remarkable because it shows that for one of the largest and most realistic benchmarks in our suite, garbage collection produces a net performance *win*. These results stand against the conventional wisdom that garbage collection always comes at a performance price.

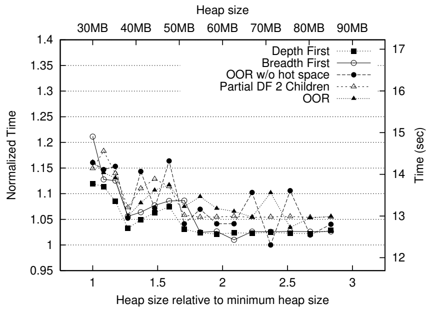
9. Conclusions

We show that the performance of class-oblivious traversal orders can be unpredictable and expose programmers to variations outside of their control. We show that our online object reordering system eliminates copying order *gambling*. It has a negligible overhead, is amenable to the virtual machine context, and adaptively matches or improves over the best static, class-oblivious order for a given program.

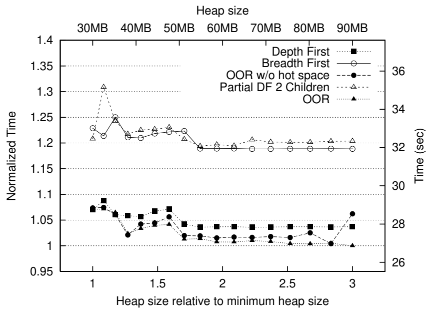
Common wisdom holds that the software engineering benefits of garbage collection come with a performance penalty. We show that copying collectors have a locality advantage over the free-list organizations of explicitly managed memory. Copying collectors achieve good locality by placing contemporaneously allocated objects together in memory and copying connected objects together in the mature space. OOR further adapts copying order to program access patterns. Since future processors will demand locality to achieve high performance, we can look forward to a future where garbage collection combines software engineering and performance benefits.



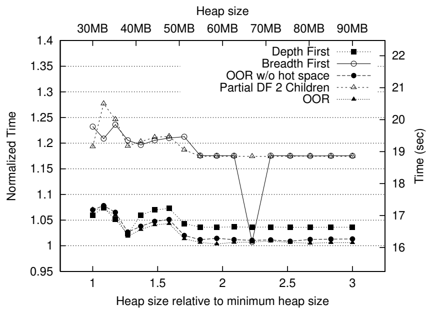
(a) jython 933MHz PowerPC Total Time



(b) jython 1.9GHz AMD Total Time



(c) jython 2.4GHz P4 Total Time



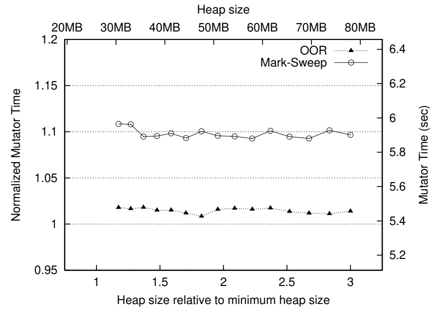
(d) jython 3.2GHz P4 Total Time

Figure 11: Performance on Different Architectures

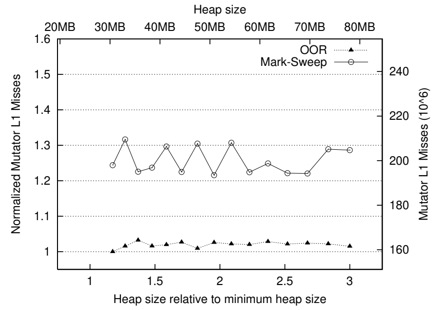
10. REFERENCES

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.



(a) Mutator Time



(b) Mutator L1 Cache Misses

Figure 12: Mutator Performance for Copying and Mark-Sweep Collectors on javac

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, NY, NY, June 2004.

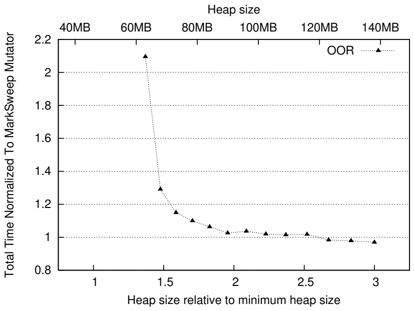
[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *Proceedings of the International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004.

[6] S. M. Blackburn, K. S. McKinley, J. E. B. Moss, S. Augart, E. D. Berger, P. Cheng, A. Diwan, S. Guyer, M. Hirzel, C. Hoffman, A. Hosking, X. Huang, A. Khan, P. McGachey, D. Stefanovic, and B. Wiedermann. The DaCapo benchmarks. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/Benchmarks.

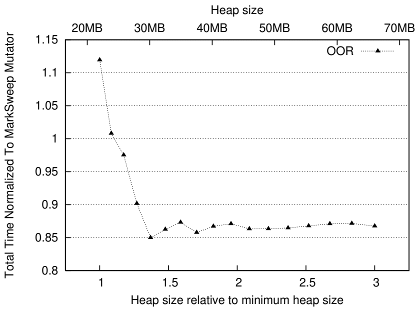
[7] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

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

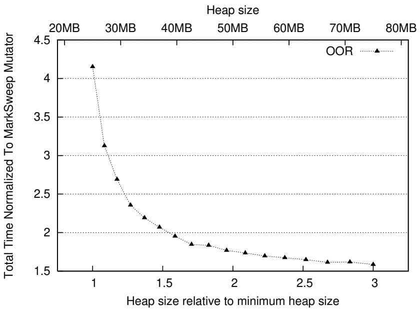
[9] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ACM International Symposium on Memory Management*, pages 37–48, Vancouver, BC, Oct. 1998.



(a) Total Time, pseudojbb



(b) Total Time, ps-fun



(c) Total Time, ipsixql

Figure 13: Garbage Collection vs. Idealized Mark-Sweep

[10] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 92–115, June 1999.

[11] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, Oct. 2003.

[12] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

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

[14] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[15] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[16] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanovic, and C. Weems. The DaCapo project. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/.

[17] M. Pettersson. Linux Intel/x86 performance counters, 2003. http://user.it.uu.se/ mikpe/linux/perfctr/.

[18] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM Symposium on the Principles of Programming Languages*, pages 140–153, Portland, OR, 2002.

[19] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[20] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[21] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[22] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.