Finding Your Cronies:
Static Analysis for Dynamic Object Colocation[*]

Samuel Z. Guyer
The University of Texas at Austin
Austin, TX, 78712

sammy@cs.utexas.edu

Kathryn S. McKinley
The University of Texas at Austin
Austin, TX, 78712

mckinley@cs.utexas.edu

ABSTRACT

This paper introduces *dynamic object colocation*, an optimization to reduce copying costs in generational and other incremental garbage collectors by allocating connected objects together in the same space. Previous work indicates that connected objects belong together because they often have similar lifetimes. Generational collectors, however, allocate all new objects in a *nursery* space. If these objects are connected to data structures residing in the *mature* space, the collector must copy them. Our solution is a cooperative optimization that exploits compiler analysis to make runtime allocation decisions. The compiler analysis discovers potential object connectivity for newly allocated objects. It then replaces these allocations with calls to *coalloc*, which takes an extra parameter called the *colocator* object. At runtime, `coalloc` determines the location of the colocator and allocates the new object together with it in either the nursery or mature space. Unlike pretenuring, colocation makes precise per-object allocation decisions and does not require lifetime analysis or allocation site homogeneity. Experimental results for SPEC Java benchmarks using Jikes RVM show colocation can reduce garbage collection time by 50% to 75%, and total performance by up to 10%.

Categories and Subject Descriptors

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

General Terms

Languages, Performance, Experimentation, Algorithms

Keywords

Cooperative optimization, static analysis, compiler-assisted memory management

1. INTRODUCTION

This work introduces the *colocation* (or *crony*) optimization for garbage collectors that divide the heap into separately collected regions, such as generational collectors. A problem for these collectors is that they record and treat as live pointers into independently collected regions. Pointers in connected data structures that cross regions will cause the collector to retain their referents. For example, Figure 1 depicts a connected data structure that straddles two collection regions. Since collectors must assume that pointers between separately collected regions are live, they will retain this data structure, perhaps needlessly. Furthermore, previous work shows that connected objects usually die together [21, 22]. The goal of colocation is to allocate a new object directly into the same region as an object that will reference it. Grouping and collecting connected objects together will thus avoid the work of tracking and processing pointers from different regions, and more promptly reclaim objects when they die. Colocation is a *cooperative* optimization because it uses compiler analysis to selectively introduce dynamic allocation decisions.
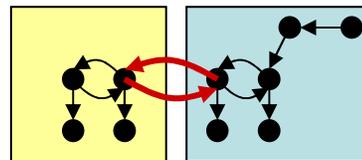

Figure 1: Connected structures belong in the same space.

Colocation uses the following static and runtime components. (1) A static compiler analysis finds old objects, called *colocators* that will reference newly allocated objects and that the program does not immediately overwrite. (2) A new allocation routine, *coalloc*, which takes a colocator object parameter. (3) At runtime, `coalloc` puts new objects in the same region as the colocator object. The analysis finds connections between newly allocated objects and existing objects, with special provisions to exclude volatile (quickly overwritten) connections. The interprocedural compiler analysis is flow-insensitive and exploits static-single-assignment (SSA) form. The compiler analysis need not be sound since the location of the object does not affect correctness, and we exploit this feature to make our analysis fast. We implement this analysis in the IBM Jikes RVM [2, 3] optimizing compiler.

At runtime, `coalloc` makes precise per-allocation decisions based on the current location of existing colocator objects. Unlike pretenuring [9, 12, 20] or prolific types [30], colocation does not require lifetime analysis or call site homogeneity. Homogeneity limits applicability, especially for library routines with many different client uses. Colocation and pretenuring are most likely

synergistic, but that study is beyond the scope of this paper. Colocation goes beyond age-based copying algorithms [8, 27, 33, 36] to exploit dynamic connectivity, but does not require strict analysis correctness as does connectivity-based collection [21]. We discuss related work in more detail in Section 2. Although we evaluate colocation in two generational collectors, it will work for any collector that divides the heap into independently collected regions.

Using MMTk [6, 7], a Memory Management Toolkit for Java in IBM Jikes RVM [2, 3], we evaluate the overhead of colocation, the potential reduction from connected objects in two generational collectors, and the performance impact. The current implementation produces its best results when all methods are compiled ahead of time – in this setting our analysis only increases compilation cost by about 10%. It is considerably less expensive than previous alias and escape analysis for Java [10, 15], and we believe we can further reduce this cost. We evaluate colocation using a copying nursery with both copying and mark-sweep mature spaces. It reduces garbage collection time for all the SPEC JVM benchmarks by up to a factor of 2. For a few benchmarks, the generational collectors augmented with colocation can execute in a smaller heap sizes than without it because colocation helps them to use heap space more efficiently. For most programs, the collection time improvements translate into total time improvements, of up to 10%, since garbage collection time is a fraction of total time.

The rest of this paper is organized as follows. In Section 2 we review related work. In Section 3 we describe the overall system for performing cooperative object colocation. In Sections 4 and 5 we describe our static analysis and run-time system. In Sections 6 and 7 we present our experiments and results.

2. RELATED WORK

This section overviews generational collectors and the opportunities they expose for colocation. We then compare our work to connectivity analysis, allocation for locality, and static and dynamic lifetime prediction and its use by pretenuring. Colocation is unique from previous work because it does not require call site homogeneity or lifetime profiling or prediction. We also compare our compiler analysis to other static heap analyses.

All generational collectors exploit the weak-generational hypothesis [27, 36], that young object die quickly. Copying generational collectors divide the heap, allocate the youngest objects into the nursery, and most frequently collect the nursery [5, 36]. To avoid scanning the mature space when collecting the nursery, the write barrier tests all heap pointer stores, and remembers the source of pointers that point from the mature space into the nursery at run-time. At collection time, it copies objects reachable from the stacks, registers, global variables, and these remembered pointers into the mature space. The collector re-examines remembered pointers first, since a later update may have overwritten the nursery target.

Recent work on connectivity-based garbage collection [21, 22] shows that connected objects often die together, which bolsters our hypothesis. However, their collector organization completely eliminates write barriers with a static analysis that allocates objects into static partitions that contain connected objects, forming a hierarchical directed acyclic graph of partitions. The collector must always collect ancestors together with descendents, and thus does not exploit dynamic connections. The colocation optimization exploits object connectivity in a much less restricted setting, and is thus able to couple itself with the high performance generational collectors, and could also improve other incremental collectors [8, 24, 29].

Research that uses copying collection to improve object locality [14, 23, 25, 37] can also exploit connectivity. This work locates objects that are frequently accessed together on the same cache line.

Object colocation may have a positive benefit on page locality since it will tend to allocate connected objects closer together, but it will not generally improve cache line locality. Locality and colocation optimizations are thus orthogonal, but could work well together.

The pretenuring optimization uses static profiling to classify objects as long lived, and then directly allocates them into the older generation [9, 12, 34, 35], or uses dynamic samples [1, 17, 20, 28] through weak pointers and write barriers. All of these techniques require call site lifetime homogeneity, which is restrictive. For example, the top allocation site in javac creates entry nodes for a hash table and the lifetimes are split 55-45 short-lived versus long-lived. Object colocation works without call site homogeneity because it asks on a per-instance basis: "Is the existing colocator in the nursery or mature space?"

Our compiler analysis is similar in spirit to work that finds connected heap objects [11, 19], but is much faster and less precise. In fact, colocation analysis need not be conservative, since the allocation of an object in the mature space or the nursery does not affect correctness, only performance. We use a flow insensitive, single pass analysis, and experiment with intraprocedural and interprocedural propagation. This approach makes the compiler analysis viable for a just-in-time compiler, where as escape [10, 15] and other pointer analyses are too costly in this context.

3. DYNAMIC OBJECT COLOCATION

The goal of dynamic object colocation is to allocate connected objects in the same garbage collection space. Since collectors use connectivity to determine survivors, the lifetimes of connected objects are correlated [22], and placing them in the same space can improve collector efficiency. Colocation produces this effect dynamically by determining in which space the source of a pointer resides, and then allocating the target of the pointer in the same space.

Similar to pretenuring, colocation tends to put short-lived connected objects in the nursery, and long-lived ones in the mature space. In contrast to pretenuring, however, a given allocation site can allocate to either the old or young space, depending on the objects involved. This flexibility is particularly important for allocation sites inside widely reused code, such as the Java container classes. For example, the **LinkedList** class contains an internal "link" class that makes up the backbone of the list, and the **add()** method allocates instances of this class in order to accommodate new elements. Pretenuring schemes that are triggered by types or allocation sites must decide, for all linked lists, in which space **add()** will place new instances of the link. This decision presents a difficult tradeoff in programs that create both short-lived and long-lived lists. In contrast, colocation avoids this tradeoff by making the decision dynamically: it allows **add()** to place new link elements into whichever space contains the existing elements of the list.

Our approach uses a new memory allocation routine, which we call *coalloc*, that takes an addition argument of type **Object**, which we call the *colocator*. Coalloc allocates the new object into the same space as the colocator. Unlike previous approaches to colocation for locality, we do not expose the allocation interface to the user [13]. Our system automatically identifies candidate allocation sites and computes appropriate colocators for them. Since finding a colocator requires knowledge of the future use of a new object (e.g., its incorporation into a list), our system performs this task at compile time using static analysis. Since the particular space in which the colocator resides is only known at run-time (and is collector-specific), our system makes allocation decisions at run-time using a dynamic test. Our system consists of two parts.

```
void Simple(A a){
    B newB = new B();
    a.f = newB;
}
```



```
void Simple(A a){
    B newB = coalloc B(a);
    a.f = newB;
}
```

Figure 2: Simple example: the newly allocated B object is stored in the A object parameter, therefore we convert the new into a call to coalloc, passing a as the colocator.

```
void BottomUp(A a){
    B newB = new B();
    C newC = new C();
    newC.f = newB;
    a.f    = newC;
}
```



```
void BottomUp(A a){
    B newB = coalloc B(a);
    C newC = coalloc C(a);
    newC.f = newB;
    a.f    = newC;
}
```
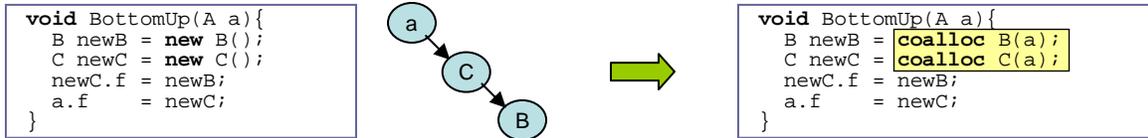
Figure 3: More complex example: we cannot use newC as the colocator for the new B because the new C is created later; however we can safely use a as the colocator for both.

Compile-time analysis identifies colocators and inserts calls to coalloc.

Run-time system provides the coalloc allocation routine.

Figure 2 shows a simple example of how colocation works. The code fragment on the left shows a method called **Simple** that creates a new object of type B and stores a reference to it in the object of type A pointed to by the variable **a**. The graph in the middle of the figure depicts the resulting data structure. Unless the program subsequently overwrites the reference, the newly allocated B object will live at least as long as A. Therefore, the variable **a** is a good choice for the colocator of the new B object. The code on the right shows the result of our compiler pass. Our analysis automatically identifies **a** as a suitable colocator and replaces the **new** construct with a call to coalloc, passing in the variable **a**. The coalloc routine makes an allocation decision based on **a**: if **a** refers to an object in the mature space, then coalloc puts the new B object in the mature space, otherwise coalloc puts it in the nursery. The key is that this decision depends on the run-time value of **a**, which can vary from one invocation of the method to another.

Unfortunately, it is not always possible to colocate a new object with the object that directly references it. The reason is that programs need not create objects in the same order that they connect them together. The method **BottomUp** in Figure 3 demonstrates this problem: **newC** is the logical choice of colocator for the allocation of **newB**, but **newC** does not yet exist at that point. One solution is to abandon colocation is such cases. Another solution is to attempt to transform the code so that the order of creation matches the order of connection. Both of these solutions require some form of dependence analysis and may still miss colocation opportunities.

Our solution is to use only the formal parameters of a method (including the receiver object) as potential colocators. The advantage of this solution is that the formal parameters are guaranteed to exist before any allocations take place in the method, so no dependence analysis is needed. This solution works because connectivity, and therefore survival, is transitive: since **newB** lives as long as **newC**, and **newC** lives as long as **a**, we can conclude that **newB** lives as long as **a**. Therefore, **a** is a suitable colocator for both allocation sites. The code fragment on the right in Figure 3 shows the application of this policy to the **BottomUp** method.

More formally, we select colocators as follows: given an allocation site A inside a method M, we choose a colocator C from among the formal parameters of M (including the receiver) such that during the execution of M the object or objects created at A become reachable, by some sequence of pointers, from C. If no parameter is suitable, then the allocation site has no colocator.

4. FINDING COLOCATORS

This section describes our static analysis for finding colocators. The goal of this analysis is to find a suitable colocator for each allocation site. We start by presenting our basic analysis algorithm, followed by two interprocedural enhancements. This algorithm can discover most of the potential connectivity between objects in a program. This information, however, is often overly aggressive for colocation because some of the connections are volatile and short-lived at runtime. Examples of volatile connections include references that are created conditionally and containers that are cleared. Therefore we add to our algorithm a set of heuristics that prune out potentially volatile connections.

Our colocation analysis resembles existing algorithms for pointer analysis, but differs in several unique ways. Central to these differences is that our analysis is unsound: its results do not necessarily represent all the connections that might occur at runtime. Therefore it cannot be used for traditional optimizations, which require sound analysis in order to preserve program correctness. For colocation, however, our analysis need not be sound because changing the allocation space is always safe, even if it's not always profitable.

We exploit our exemption from soundness in several ways. First, the volatility heuristics mentioned above intentionally ignore certain connections between objects. We present these heuristics in detail in the last part of this section. Second, we significantly simplify our algorithm. For example, since colocation is only concerned with overall reachability of one object from another (as opposed to pointer aliasing), we do not need to accurately model the number of pointer hops between objects. This flexibility allows us to employ a simple and compact representation for the method summaries used in our interprocedural analysis. Finally, by giving up soundness we avoid the problems presented by certain Java language features, such as dynamic class loading, reflection, and native methods. These features present significant challenges for sound pointer analysis algorithms [?]. Our analysis can safely ignore these features even in programs that use them, obviating the need for the so-called "closed-world assumption".

In the remainder of this section we present our analysis in detail and point out its unique features, particularly those that make it unsound. We implement this analysis using the Jikes RVM optimizing compiler, which includes an internal representation based on static-single assignment form (SSA) [16]. Each method consists of a list of simple operations applied to temporary variables (virtual

registers). Our analysis algorithm is flow insensitive in that it does not associate analysis information with particular program points. However, since only one definition of a variable reaches each use, SSA form provides some flow sensitivity.

4.1 Basic colocation analysis

The basic colocation analysis builds a graph that represents a conservative approximation of the connectivity between objects in a method. This algorithm resembles Andersen-style pointer analysis [4, 18]: it is flow-insensitive and inclusion-based. The graph it creates captures connectivity among the objects allocated by the method and connectivity from local variables to those objects. The compiler then searches this graph to identify potential colocators. The analysis starts by identifying the relevant components of each method:

S	Set of statements in the method (in compiler IR)
V	Set of variables in the method
$p_i \in V$	Formal parameters – indexed by parameter number i
$a_s \in S$	Allocation sites – indexed by statement s

Each node in the connectivity graph represents a heap-allocated object: either an "old" object (pointed to by a parameter) or a new object (generated by an allocation site). For example, Figures 2 and 3 depict the connection graph for the two example code fragments where **a** represents an old node, and **B** and **C** represent new nodes. The graph nodes are identified as follows:

$o_i \in N_{old}$	A node for each parameter p_i – "old" objects
$n_s \in N_{new}$	A node for each alloc site a_s – "new" objects
$N = N_{old} \cup N_{new}$	Set of all graph nodes

Edges in the graph are directed and represent possible points-to relationships. An edge between two nodes represents a pointer between two heap-allocated objects. Our analysis does not distinguish between the different fields of an object. There are also edges from elements of V to nodes in the graph, which represent pointers from the method's local variables into the heap. We initialize the points-to graph with an edge from each of the formal parameters to its corresponding old object. This initial structure implies that parameters do not alias each other, which is not generally a safe assumption.

points-to: $(N \cup V) \to 2^N$	Graph edges – a mapping from a variable or node to its possible targets
$\forall i$:*points-to*$(p_i) = o_i$	Initialize parameter variables to point to "old" objects

The analyzer takes one pass over the statements in a method, adding edges to the points-to graph according to the analysis rules shown below. The rules for allocation, assignment, and SSA ϕ functions are straight-forward: they just transfer the points-to sets from the right-hand side expression to the left-hand side variable.

Op	Statement	Effect
`new`	`s:v = new O();`	*points-to*$(v) = n_s$
`assign`	`v = y;`	*points-to*$(v) \cup =$*points-to*(y)
`phi`	`v= ϕ(v0,...);`	*points-to*$(v) \cup = \forall i,$*points-to*(v_i)
`getfield`	`v = y.f;`	*points-to*$(v) \cup =$ *points-to*(y)
`aload`	`v = y[i];`	
`putfield`	`v.f = y;`	$\forall m \in$ *points-to*(v) :
`astore`	`v[i] = y;`	*points-to*$(m) \cup =$ *points-to*(y)

Unlike other pointer analysis algorithms, our rule for `getfield` (and `aload`) does not dereference the right-hand side variable. Instead, it ignores the field altogether and just treats the statement as an assignment. Skipping the dereference operation further simplifies analysis and it does not affect overall reachability. For example, we can treat `v=y.f` as `v=y` because anything reachable from `y.f` is also reachable from `y`.

After it builds the points-to graph, the analysis simply computes which, if any, allocation nodes are reachable from each parameter in the graph. We test reachability by computing the closure over the points-to function for each parameter. A parameter is a potential colocator for an allocation site if that node is in the closure.

reach(p)	$=$	$\{ m \|$	$m \in$*points-to*$(p) \vee$
			$m \in$*reach*$($*points-to*$(p))\}$
coloc(a_s)	$=$	$\{ p_i \|$	$n_s \in$*reach*$(p_i)\}$

Notice that the analysis may find multiple suitable colocators for a single allocation site. In early experiments we compared the effects of choosing different colocation policies: (1) taking the first colocator, in parameter order, (2) only using `coalloc` if there is a single colocator, (3) combining multiple colocators at run-time by taking the conjunction or disjunction of the colocation decisions. We found, however, no significant difference in the run-time effect of the different policies. Most of the time, the colocators agree on the colocation decision. Therefore, all the results shown in Section 7 use policy (1).

4.2 Interprocedural algorithm

The analysis algorithm described so far is intraprocedural: it only considers allocations and connections that occur within a single method. It is common, however, for programs to create objects in one method and assemble them in a different method. To handle this case, we compute a simple summary for each method and apply the summary wherever the method is called. Since our analysis does not require soundness, we can safely ignore method calls when no summary is available. In practice, though, we find the summaries are critical for effective colocation and we quantify these benefits in Section 7. Method summaries cover two programming constructs found frequently in object-oriented programs: factories and containers.

4.2.1 Factory methods

Factory methods are a common design pattern in object-oriented programming: a factory method just creates and returns objects on behalf of other methods, and thus behaves as an allocation routine. Our solution is to detect these methods, and then treat them as allocation sites in their callers. We describe the modifications to our analysis below, and Section 5 shows the run-time instrumentation for colocation in a factory method.

To detect factory methods we add the following analysis rule to collect the set of variables R that might be returned from a method:

Op	Statement	Effect
`return`	`return v;`	$R \cup =$*points-to*(v)

At the end of the analysis, the analyzer checks to see if any allocation nodes are reachable from returned variables. If so, it marks the method as a factory and records the allocation sites that generate the returned objects. Section 5 describes how we provide colocators for these sites.

isfactory(m)	$=$	true	if $\exists a_s : a_s \in$*reach*(R_m)
		false	otherwise

We also need a rule to handle the factory call sites. This rule mirrors the existing rule for regular allocation sites:

Op	Statement	Effect
`call`	`s: v = obj.m();`	if $isfactory(m)$: Create node n_s as alloc site, $points\text{-}to(v) = n_s$

4.2.2 Connector methods

Another common programming practice is the use of container classes, such as the standard Java library. Container classes present a problem for our intraprocedural analysis because they encapsulate the code that connects new objects to their containers. For example, at a call to `Vector.addElement()` our intraprocedural analysis cannot determine that this method creates a connection between the vector and the input argument. Our solution is to provide this additional information in the form of method summaries.

We compute a connection summary for a method while computing colocators. In addition to detecting allocation nodes that are reachable from the parameters, we compute reachability between parameters. For each parameter p_i, if some other parameter p_j is reachable from p_i then we record the parameter numbers as a pair:

$$Summary(m) \quad = \quad \{\, (i,j) \mid p_j \in reach(p_i) \,\}$$

For example, our analysis generates a summary consisting of $(0,1)$ for the `Vector.addElement()` method because the new element (parameter 1) is attached to the receiver Vector (parameter 0). During analysis, we use the method summary at a call site by applying the **putfield** rule to each of the integer pairs (i,j). Note that the edges created by this rule might represent many edges in the callee method, but collapsing those edges into a single edge does not affect overall reachability.

Op	Statement	Effect
`call`	`obj.m(v0,...);`	$\forall (i,j) \in Summary(m)$: $apply$ `vi.f = vj`

4.3 Volatility heuristics

This section describes our volatility heuristics, which help prevent overly aggressive colocation. The colocation analysis described above detects almost all potential connectivity between the objects in a program. However, this analysis is too aggressive because programs often introduce volatile or unstable connectivity. For example, programs sometimes quickly overwrite a connection, or only install connections under special conditions. Excessive colocation can force objects with dramatically different lifetimes into the same space, hurting the efficiency of collection. In our experiments using generational collectors this effect manifests itself as an excess of short-lived objects colocated in the mature space, requiring costly full-heap collections to recover.

These heuristics are conservative in the sense that if a reference appears to be volatile then we exclude it from colocation. For some programs these additions are overly conservative, but in several cases they prevent pathological behavior.

The code fragment in Figure 4 shows two examples of volatile connections. The first example creates a new string, but only adds it to the container if it is not already there. At compile time, we do not know how frequently that condition might be true, so we act conservatively and avoid colocating the string with the container. In the second example, the loop fills the container with new objects, but then immediately clears it. Again, to conservatively prevent excessive colocation, the analysis prohibits colocation. To capture

```
1  void Volatile(Container c, Value v) {
2    // -- The newly created string is not always
3    //     stored in the container:
4    String value_name = v.toString();
5    if ( ! c.contains(value_name))
6      c.add(value_name);
7
8    // -- Objects don't remain in the container
9    //     for long...
10   for (...) {
11     c.add(new String(...));
12   }
13   c.clear();
14 }
```

Figure 4: Examples of volatile connectivity.

this notion of volatility, we place two additional conditions on the analysis rules and modify the reachability computation.

We place two restrictions on the `putfield` rule in order to avoid volatile or uncertain connections. First, we prohibit colocation when the `putfield` that connects a new object to an old one is guarded by a condition, but the creation of the new object is not. We test for this case using post-dominance: We only apply the `putfield` rule when the `putfield` post-dominates the creation of the stored object.

Second, we prohibit colocation when the program stores the results of a getfield operation. Our reasoning is that the object produced by a getfield is already connected to some other data structure, so the additional connectivity is unlikely to help colocation. We can imagine cases where this condition would help colocation. For example, if an object is stored in a temporary object before being connected to another data structure. We prefer to act conservatively, and find this opportunity is rare.

Our third heuristic is designed to detect cleared data structures. During the analysis we compute the set of objects C into which the program explicitly stores null. We add the following analysis rules:

Op	Statement	Effect
`putfield null`	`v.f = null;`	$C \cup = points\text{-}to(v)$
`astore null`	`v[i] = null;`	

During the closure computation we do not follow the outgoing edges from cleared objects:

$$reach(p) \quad = \quad \{\, m \mid \; m \notin C \;\wedge \\ (\, m \in points\text{-}to(p) \;\vee \\ m \in reach(points\text{-}to(p)) \,) \,\}$$

5. RUNTIME SYSTEM

This section describes the run-time components of dynamic object colocation: (1) The `coalloc` routine, which replaces the regular memory allocation routine and performs the run-time colocation test, (2) the mechanism for passing colocators down through factory methods to the underlying allocation sites, and (3) an extension to `coalloc` that speculative colocates objects based on their relative ages. This last feature is more aggressive than the standard colocation system, but can improve colocation in collectors that use an unbounded nursery.

5.1 Coalloc

The compiler replaces calls to the regular memory allocation routine with a `coalloc` call *only* when the analysis finds a suitable colocator. Figure 5 shows an abstraction of the original code and its replacement. Since we use two-generational collectors for all of our experiments, `coalloc` tests the colocator to decide whether to allocate the new object in the nursery or in the mature space.

Our VM assigns specific address ranges to each of these spaces, so we can determine which space the colocator occupies by a simple address comparison. In our collectors the nursery space resides at a higher range than the mature space, so the less-than test in Figure 5 returns true if the colocator is not in the nursery. Since allocation time typically represents less than 1% of total time [23], and since these values are usually in registers, this overhead is negligible.

```
1  public VM_Address alloc(int bytes) {
2    return nursery.alloc(bytes);
3  }
```
(a) Original allocation.

```
1  public VM_Address coalloc(int bytes,
2                            VM_Address colocator) {
3    if (! colocator.isZero() &&
4        colocator.LT(NURSERY_START))
5      return matureAlloc(bytes);
6    else
7      return nursery.alloc(bytes);
8  }
```
(b) Coalloc.

Figure 5: The colocator argument selects the allocation space in `coalloc`.

5.2 Factory methods

Factory methods colocate objects based on the use of objects in the calling method. Therefore, we provide a mechanism for the caller to pass a special *factory colocator* down into the factory method. Ideally, we might alter the factory method interface to accept an additional object argument. However, this strategy requires us to make sure that any potential callers and any factory subclasses are properly modified to reflect the new interface. Therefore, the current system instruments the caller to store the factory colocator inside the VM, and the callee retrieves the value and holds it in a local variable. This strategy is easy to implement and is correct even if the caller does not recognize the callee as a factory method. Figure 6 shows an example of this instrumentation. We avoid confusion and contention across threads by allowing the VM to save factory colocators on a per-thread basis.

```
1  void someMethod(Container c) {
2
3    VM_save_factory_colocator(c);
4    Element e = Factory.makeElement();
5    c.add(e);
6  }
```
(a) Caller.

```
1  class Factory {
2    Element makeElement() {
3      Object factory_colocator =
4          VM_get_factory_colocator();
5      return coalloc(..., factory_colocator);
6    }
7  }
```
(b) Callee.

Figure 6: The caller passes colocators to Factory methods.

5.3 Speculative age-based colocation

In generational collectors, dynamic object colocation primarily serves to allocate new objects into the mature space *only* when the colocator is in the mature space. However, we can also look at the relative age of a colocator, even if it currently resides in the nursery. Figure 7 shows a diagram of the nursery space with a colocator close to the older end of the nursery and the current bump pointer at the young end of the nursery. The colocator is almost certainly live at

this point, and thus likely to survive the next collection, especially for large nurseries. Therefore we can speculatively place the new object in the mature space when the colocator is old, but still in the nursery.
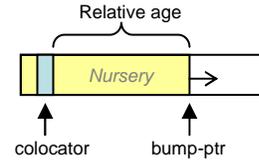
Figure 7: Age-based colocation: even if the colocator is not in the mature space, if it is "old enough" we can allocate the new object in the mature space.

```
1  public VM_Address coalloc(int bytes,
2                            VM_Address colocator) {
3    int age = nursery.cursor.diff(colocator).toInt();
4    if (! colocator.isZero() &&
5        (colocator.LT(NURSERY_START) ||
6         (age > AGE_THRESHOLD)))
7      return matureAlloc(bytes);
8    else
9      return nursery.alloc(bytes);
10 }
```

Figure 8: Coalloc routine with age-base speculative promotion.

The implementation of this feature involves adding an address-relative test to the `coalloc` routine. Figure 8 shows the modified `coalloc` routine with an age-relative colocation test. In Section 7 we show the effect of this policy under an unbounded "Appel-style" nursery using a 4 MB age threshold. This nursery configuration delays collection of the nursery as long as possible, resulting in relatively old objects residing in the nursery instead of in the mature space. Age-based colocation places new objects in the mature space when their colocators are old enough, but not yet in the mature space.

6. METHODOLOGY

This section briefly describes our experimental methodology, including our generational collectors, MMTk, Jikes RVM, compile-time strategy, and platform.

6.1 Generational Collectors

We perform our experiments in an efficient, composable Java memory management toolkit that implements a wide variety of high performance collectors that reuse shared components [7]. MMTk manages large objects (8K or bigger) separately in a non-copy space, and puts the compiler and a few other system pieces in the boot image, an immortal space. We apply colocation to two generational collectors with different mature space policies, and two different nursery configurations.

The first collector is a generational copying collector (GenCopy) that divides the heap into two parts, a copying nursery for newly allocated objects, and a mature space that is managed using semi-space collection [5, 36]. A *write barrier* remembers pointers from the mature space to the young space. For every pointer store, the compiler inserts write-barrier code. At execution time, it conditionally records pointers depending on the collector policy. Gen-Copy collects the nursery when it is full (see nursery policy discussion below). It finds all reachable objects by tracing from the roots (stacks, registers, statics, and remembered set) and promoting survivors into the mature space. We use a variant of depth-first

copying order that attains good mutator locality [23]. Since the mature space is a semi-space, it must reserve half of its space for copying.

The second collector is a generational collector with a mark-sweep mature space (GenMS). The mark-sweep space uses a segregated free-list modeled after Lea's allocator [26]. The system collects this space by tracing and marking the live objects using bit maps, and lazily finds free slots during allocation. Tracing is thus proportional to the number of live objects, and reclamation is incremental and proportional to allocation. MMTk uses 51 size classes that attain a worst case internal fragmentation of 1/8. Collection of the nursery proceeds in the same manner as GenCopy. Since GenMS need not reserve half the heap for copying, it is more space efficient than GenCopy. However, our results confirm recent work showing that copying collectors produce better mutator locality, which outweighs space efficiency in some cases [23]. See Blackburn et al. for additional MMTk details [6, 7].

We test both GenCopy and GenMS under two nursery configurations: a *bounded* 4 MB nursery and an unbounded *Appel* [5] nursery. In MMTk, the *bounded* nursery takes a command line parameter as the initial nursery size, collects when the nursery is full, and resizes the nursery below the bound only when the mature space cannot accommodate a nursery of survivors. When the nursery size falls below a lower bound (we use 256KB), it triggers a mature space collection. An *Appel* nursery uses the same discipline, but with the heap size as the upper bound. Previous work finds that these two have similar performance, but the Appel configuration is sometimes slightly faster and the bounded 4 MB nursery has lower average pause times [6].

Colocation is sensitive to the nursery configuration because it determines which objects end up in the mature space and when. For example, with an unbounded nursery the first collection only occurs after the whole heap has been exhausted, which delays the initiation of colocation. With a 4 MB nursery, collection occurs earlier, allowing colocation to start working earlier. For this reason, we focus on the 4 MB bounded nursery.

6.2 IBM Jikes RVM and compiler

Jikes RVM (v 2.3.0.1) is a high-performance VM written in Java with an aggressive adaptive just-in-time optimizing compiler [2, 3]. We use configurations that pre-compile as much as possible, including key libraries and the optimizing compiler itself (the *Fast* build-time configuration), and turn off assertion checking.

Our experiments direct the compiler to optimize all methods in the application before executing the program and measuring performance. While this strategy is not strictly necessary, it significantly improves the effectiveness of colocation. We added our colocation analysis and instrumentation phase to the sequence of high-level optimizations that take place in SSA form. The compiler analysis goes bottom up on call graph to obtain interprocedural summaries for all methods (see Section 4). The overhead of optimizing the entire application is quite high, but the fraction of this overhead added by the colocation analysis is only 5% to 10%. By comparison, other pointer analysis and escape analysis, appear to be significantly more costly [10, 15]. In addition to the analysis, the more complex allocation routine places a heavier load on the optimizing compiler. For now we view colocation as an ahead-of-time optimization, which might be suitable for a Java-to-bytecode compiler.

6.3 Experimental Platform

We perform all of our experiments on a 3.2 GHz Intel Pentium 4 with hyper-threading enabled, with an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB uni-

fied 8-way set associative L2 on-chip cache, 1GB main memory, and runs Linux 2.6.0.

7. RESULTS

This section presents our findings from applying dynamic colocation to the SPEC JVM98 benchmarks, under the four generational garbage collector configurations (see Section 6.1). In this setting, the primary benefit of colocation is to reduce the cost of nursery collections by allocating some objects directly in the mature space. We start by describing the potential for colocation: the amount of memory copied from the nursery by the unmodified collectors. We quantify runtime overhead of `coalloc`, which is on average less than 1%, using a configuration that includes all colocation instrumentation, but always allocates objects in the nursery. We examine the tradeoff between accuracy and efficacy of colocation, finding that our analysis finds much of the potential while making some, but not many errors.

We then present the central result of the paper. We examine the impact of colocation on performance by measuring garbage collection time, mutator time, and overall execution time. Colocation substantially reduces copying from the nursery without overburdening the mature space. The resulting collection time improvements translate into total execution time improvements. Colocation is particularly effective on the three benchmarks with high nursery survival rates. Finally, we explore the design space of the colocation analysis by showing the impact of turning off various features, including the volatility heuristics and the interprocedural summaries.

7.1 Potential of colocation

Table 2 presents the allocation characteristics of our benchmarks: the total allocation in MB (*Total*) and the amount the collector promotes from the nursery to the mature space (*Copy*) in MB and as a percentage. We order programs by their nursery survival rate. These base statistics show that colocation in this generational setting has the potential to improve pseudojbb, javac, and db since a significant fraction of their nursery objects survive. We omit compress because it allocates only 3 MB into the nursery, and thus it never triggers a nursery collection.

Table 1 shows the compile-time properties of the benchmarks: the numbers of methods and allocation sites, as well as the specific colocation decisions the compiler generates. The second column shows the number of methods identified as factory methods. The third column shows the total number of allocation sites. The last three columns show the number of allocation sites converted to `coalloc`. In general, the compiler finds many opportunities for colocation, but these opportunities comprise less than half of all the allocation sites. For factories, we separate out the uses of `coalloc` inside the factory method itself from uses of the factory method in the caller.

Benchmark	Methods		Allocation sites			
					Factory	
	total	Factory	Total	coalloc	inside	caller
pseudojbb	598	89	1404	120	77	155
javac	919	121	1953	512	274	431
db	192	18	683	83	25	36
mtrt	322	18	747	138	24	35
jack	430	22	1386	198	35	90
raytrace	324	19	751	138	25	35
jess	605	49	1266	255	159	88

Table 1: Compile-time colocation decisions

7.2   Colocation overhead

Colocation incurs a small runtime overhead that results from the additional test on the colocator object in each call to `coalloc`. We measure this overhead using a version of `coalloc` that includes the test, but still allocates all objects in the nursery. This configuration separates the direct overhead of `coalloc` from secondary effects of colocation, such as changes in locality. Figure 9 shows that that the overhead of the additional test is on average less than 1%, and thus a negligible consideration.

7.3   Accuracy and efficacy of colocation

In our current implementation, colocation reduces copying from the nursery by allocating some new objects in the mature space, bypassing the nursery. This effect, however, only yields a benefit under two conditions: first, colocation must select the right objects to place in the mature space, and second, it must do so often enough to significantly lower nursery survival rate. These two requirements, which we refer to as *accuracy* and *efficacy* respectively, are competing forces. For example, we could increase efficacy arbitrarily by allocating most or all new objects in the mature space, but since many of these objects do not belong there the resulting inaccuracy would force many expensive full-heap collections. Similarly, we could improve accuracy by using a more conservative colocation analysis, but the low efficacy would yield little improvement in performance. The following measurements show that our formulation effectively balances these two requirements.

Ideally colocation would always select exactly those objects that would have survived nursery collection, so that no objects are copied – perfect accuracy and efficacy. For a number of reasons, however, colocation cannot attain this goal. First, some nursery survivors

are not connected to objects in the mature space but are instead reachable from the stacks and global variables. Second, colocation can only start to place new objects directly in the mature space once some initial set of colocators is already there. Therefore colocation requires a 'warm up' of at least one nursery collection to produce these initial colocators. Third, some allocation sites produce objects whose lifetimes are not accurately predicted by their connectivity. In these cases, our volatility heuristics conservatively place these objects in the nursery to avoid triggering excess full-heap collections. Finally, even with these heuristics, colocation can mistakenly place objects in the mature space.

Figure 10 shows the effects of colocation on the allocation and copying of each benchmark as compared to the unmodified collectors. To focus on accuracy, we measure these values using specially instrumented collectors configured with a 4 MB nursery with an infinite mature space. (Table 2 also presents these raw numbers and adds percentages.) Each bar shows the amount of memory that ends up in the mature space, broken down into two parts: the dark part represents memory copied from the nursery and the light part represents memory allocated directly in the mature space. The bar labeled "Base" shows the behavior of the unmodified collections, which allocate all objects in the nursery. We normalize the graph to this value because it represents the potential for colocation. The bar labeled "Coloc" shows the result of colocation. Our goal is to push down the dark bar (reduce copying) without allowing the total size of the bar (copying plus mature space allocation) to significantly exceed the base value.

For all but one benchmark, colocation reduces copying by 50% to 75%. Colocation is usually accurate as well, increasing mature space allocation by 1 to 6% on four programs, but is not accurate on pseudojbb. We discuss the impact of pseudojbb's behavior on performance below. jess has the fewest nursery survivors, and the smallest reduction. Furthermore, colocation increases mature space promotion usage by 28%, but in absolute terms the amount of memory (0.6 MB) is so low that it has little impact on performance. The most significant reductions are for javac, pseudojbb, and db which are non-trivial applications that allocate large amounts of memory and have high nursery survival rates.
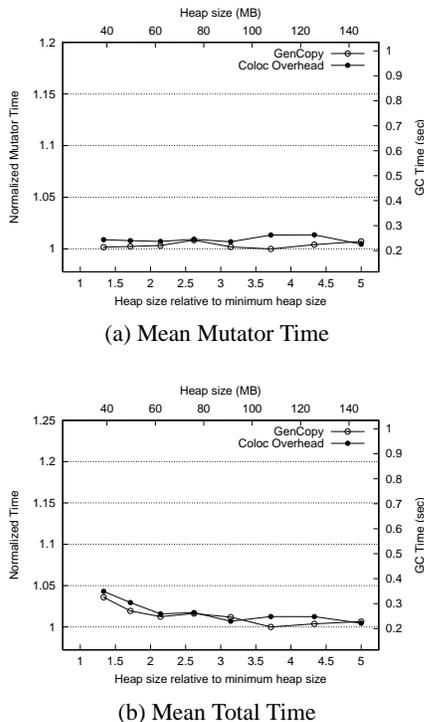
7.4   Write barrier

Colocation also has the potential to reduce intergenerational pointers, and therefore reduce the number of write barriers. The last two columns of Table 2 show the percent of all writes that the write barrier records in the remembered set (remset). We observe this secondary reduction for pseudojbb, javac, and db, which is not surprising since these are the benchmarks for which colocation is most effective. For jess and jack colocation slightly increases the number of remset entries. For raytrace and mtrt (which are closely related programs), however, the number of remset entries grows considerably. As an absolute percentage, the number is still low. These could be errors, but Figure 11 shows how colocation can increase remset entries, while still improving overall performance. If only part of a data structure is colocated in the mature space then a broad slice of it may span the boundary between spaces.

7.5   Performance

We present the geometric mean for collection, mutator, and total time (Figure 12) using a 4 MB bounded nursery, and the individual program results (Figures 13, 14, and 15) with and without colocation.

Figure 12 shows that colocation consistently reduces collector work in a bounded 4 MB nursery, reducing collection times from 40% to 60% lower in large heaps. In overall time, colocation pro-

(a) Mean Mutator Time

(b) Mean Total Time

Figure 9: Colocation overhead (mean over all benchmarks): colocation instrumentation has a low overhead.

| | Allocation (MB) | | | | | | Write barriers % taken | |
| | | Base | | Colocation | | | | |
	Total	Copy	% Surv %	Copy	% Surv %	Mature space	Base	Colocation
pseudojbb	216	59.8	27.7	23.1	10.7	63.6	5.72	3.51
javac	185	47.7	25.8	13.8	7.5	34.9	2.70	0.99
db	82	7.7	9.4	4.1	5.0	3.7	1.22	0.17
mtrt	142	6.4	4.5	3.3	2.3	3.2	0.07	0.36
jack	231	6.7	2.9	3.6	1.6	4.2	8.19	8.57
raytrace	135	3.2	2.3	0.9	0.7	2.4	0.01	0.33
jess	261	2.1	0.8	2.0	0.8	0.7	0.09	0.17

Table 2: Benchmark Characteristics. Copying and colocation are measured using a 4 MB nursery and infinite mature space.
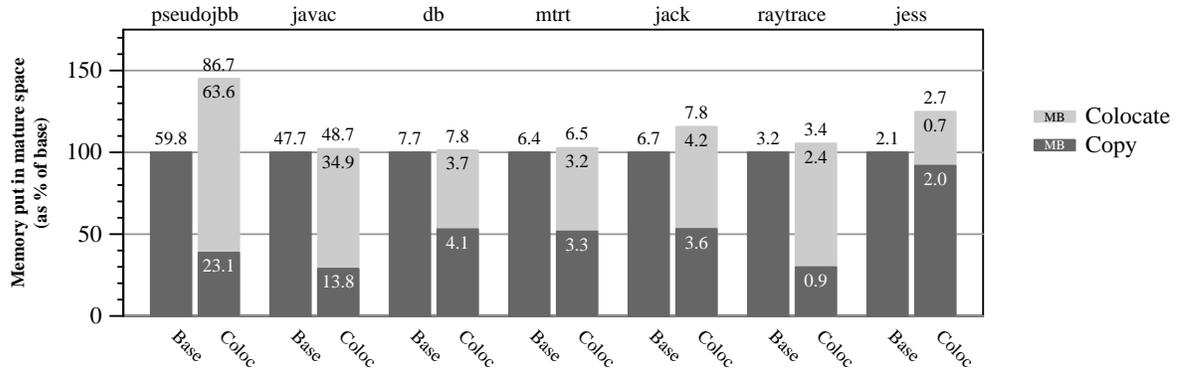
Figure 10: Colocation reduces copying without significantly increasing mature-space allocation in all but two cases.

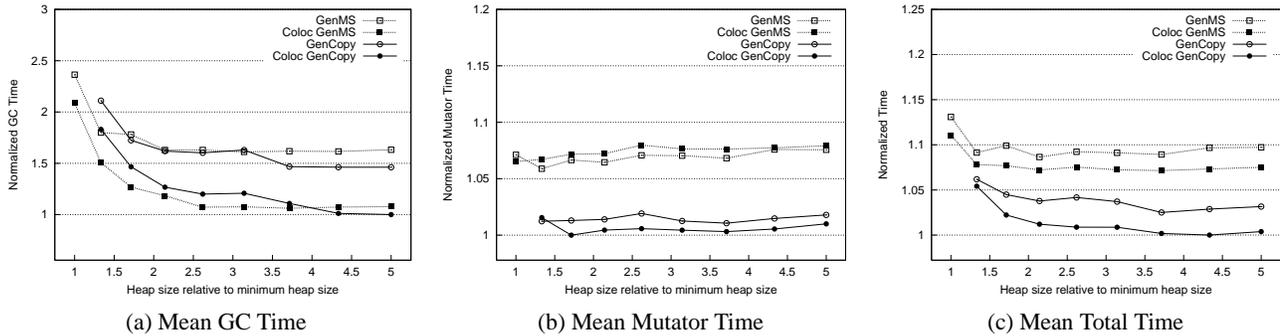(a) Mean GC Time (b) Mean Mutator Time (c) Mean Total Time

Figure 12: Colocation results with 4MB nursery: (a) Colocation reduces work for the garbage collector, (b) improves locality for copying mature space, (c) these benefits are reflected in overall execution time.

duces reasonable improvements for the GenCopy collector, but not for the GenMS collector. This result is explained by the mutator time graph. In a copying mature space, colocation improves locality without significant allocation overhead. In a mark-sweep mature space, however, locality is poor and allocation is more expensive – objects placed directly in the mature space cannot even benefit from fleeting nursery locality [23]. This effect is likely to hurt the mutator time in any scheme that allocates objects directly in the mature space.

Figure 13 reports the reduction in collection time for the individual benchmark programs. These results show four kinds of behavior under colocation. First, for javac, raytrace, jack, and mtrt colocation chooses the right objects to allocate in the mature space, reducing collection time for both collectors and across a range of heap sizes. Second, for jess and jack, our analysis detects the potentially high mutation rate in the mature space (using the volatility heuristics described in Section 4) and prevents incorrect colocation. Without this special case, colocation can cause collection time in jess to grow by a factor of three or four.

Third, for db colocation chooses the right objects to allocate in the mature space, but the performance improvement is not due to the reduction in garbage collection time, but due to the reduction in mutator time. We only see this improvement in GenCopy, which suggests that it is a result of locality: colocation places critical data structures together in the mature space in allocation order.

Finally, for pseudojbb colocation places many objects in the mature space that would not have survived a nursery collection. At larger heap sizes the cost of these incorrect decisions is hidden – pseudojbb even shows a measurable improvement. In small heaps excess allocation in the mature space triggers whole-heap collections more frequently, and degrades performance.
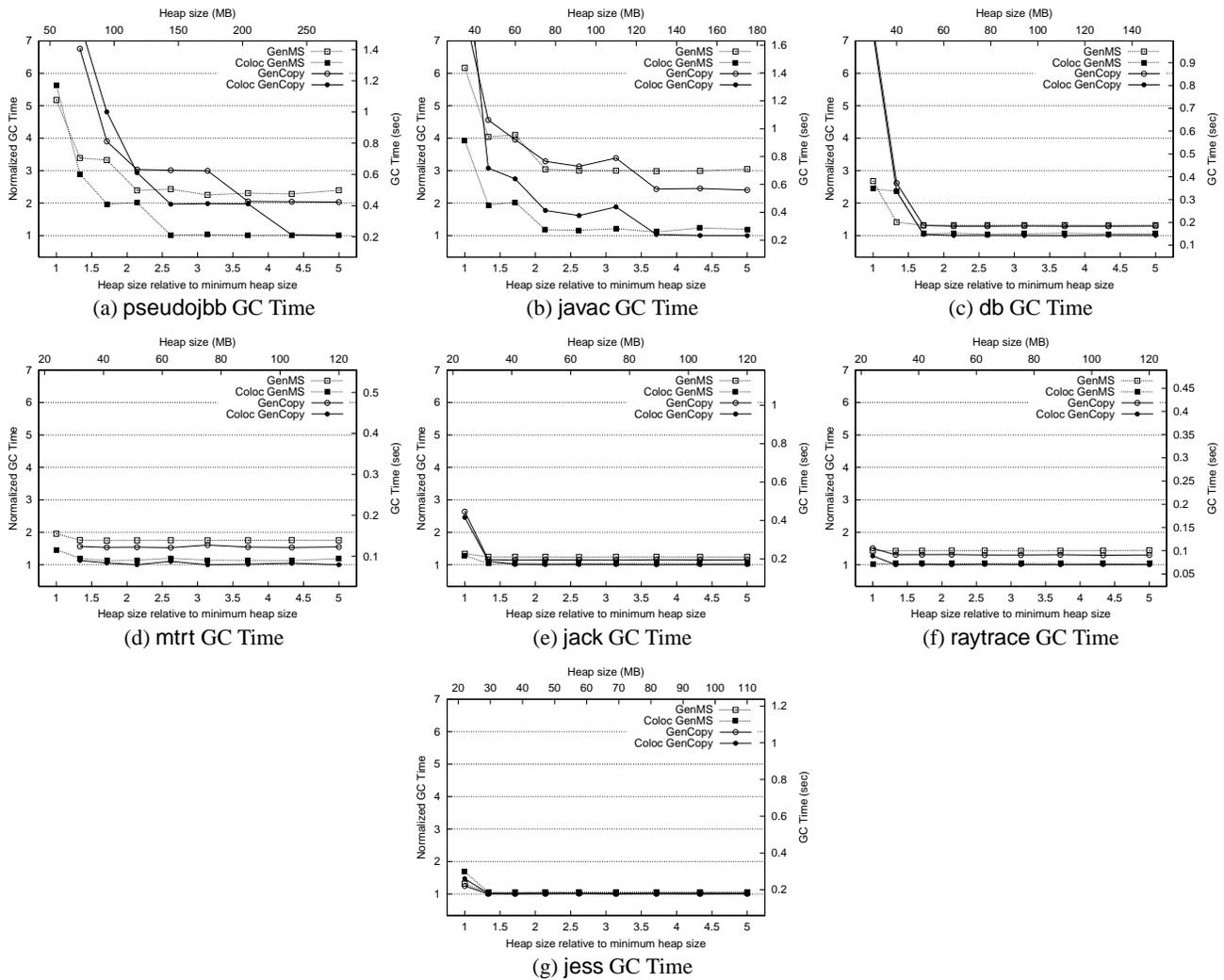
(a) pseudojbb GC Time (b) javac GC Time (c) db GC Time

(d) mtrt GC Time (e) jack GC Time (f) raytrace GC Time

(g) jess GC Time

Figure 13: GC Time with and without Colocation, 4 MB bounded nursery.

7.6 Design space analysis

We explore the analysis design space by turning off components of the interprocedural analysis and volatility heuristics. These experiments use an unbounded mature space to isolate nursery behavior. The results, shown in Figures 16, 17, and 18, use the same axes as Figure 10 described at the beginning of this section. We also present performance numbers using the unbounded nursery.

Figure 16 shows the effects of removing the interprocedural components of the analysis. The left bar shows the full colocation algorithm (same as Figure 10.) The middle bar shows the results without any interprocedural summaries. The effect is most detrimental to the larger benchmarks such as javac, jack, and pseudojbb, which rely on complex data structures to store their data. The right bar shows the results of excluding factory colocation: this configuration effectively disables colocation for programs that rely heavily on factories, such as pseudojbb. In fact, pseudojbb allocates almost all important data structures through a single factory.

Figure 17 shows the effects of turning off the analysis heuristics that prune volatile references out of the connectivity graph. For some benchmarks eliminating the putfield and cleared-object tests actually improves performance – these programs contain references that appear volatile, but are in fact stable. However, the heuristics are critical for many of the programs, which would otherwise rapidly fill the mature space with garbage. jess and jack, in particular, use container classes to hold ephemeral objects.

Figure 18 shows the effects of using speculative age-based colocation (see Section 5). This feature primarily benefits smaller benchmarks, such as db and mtrt, which allocate many long-lived objects in the first nursery collection. Speculative colocation allows these programs to put objects in the mature space before the first nursery collection.

This feature also helps colocation work more effectively in an unbounded Appel nursery [5]. Figure 19 shows the geometric means of collector time, mutator time, and overall time using an Appel nursery. Colocation is less effective in this nursery configuration, but still yields 15% to 25% improvement in collection time for GenMS. These improvements occur for the smaller heap sizes, where performance improvements are harder to obtain. Unfortunately, colocation degrades locality in GenMS, which overwhelms this benefit and results in a net slowdown for the overall runtimes.
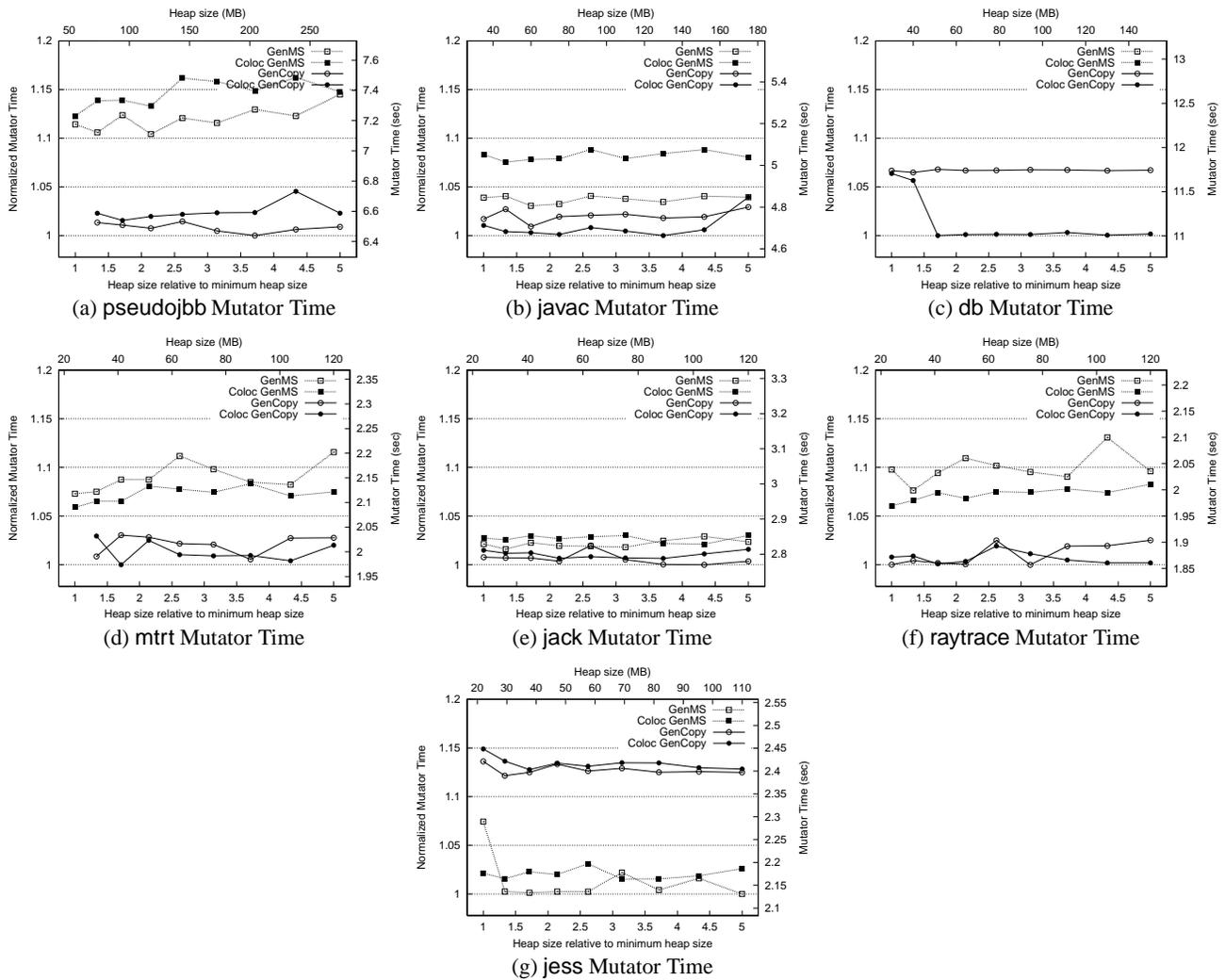
Figure 14: Mutator time with and without Colocation, 4 MB bounded nursery.

8. CONCLUSION

This paper introduces dynamic object colocation, a new cooperative compiler and runtime optimization to improve the performance of generational and other incremental garbage collectors. We demonstrate a practical compiler analysis that computes object connectivity information and passes it to the garbage collector so that connected data structures can be colocated in the same garbage collection space. Our analysis finds many opportunities for profitable colocation and reduces garbage collection time, sometimes dramatically, on our benchmarks for two generational collectors. These improvements translate to improvements in total execution time as well. Colocation makes a unique use of static and dynamic information, and should play well with other optimizations to further improve performance. Previous work suggests heap organizations that segregate objects by connectivity, but with the restriction that the objects must never install cross region pointers [21, 22]. The success of colocation instead suggests collector organizations that group connected objects into separately collected regions where a write barrier handles cross region pointers.

9. REFERENCES

[1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *ACM International Symposium on Memory Management*, pages 121–127, Minneapolis, MN, Oct. 2000.

[2] B. Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, Nov. 1999.

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

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

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

[6] 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, New York, NY,
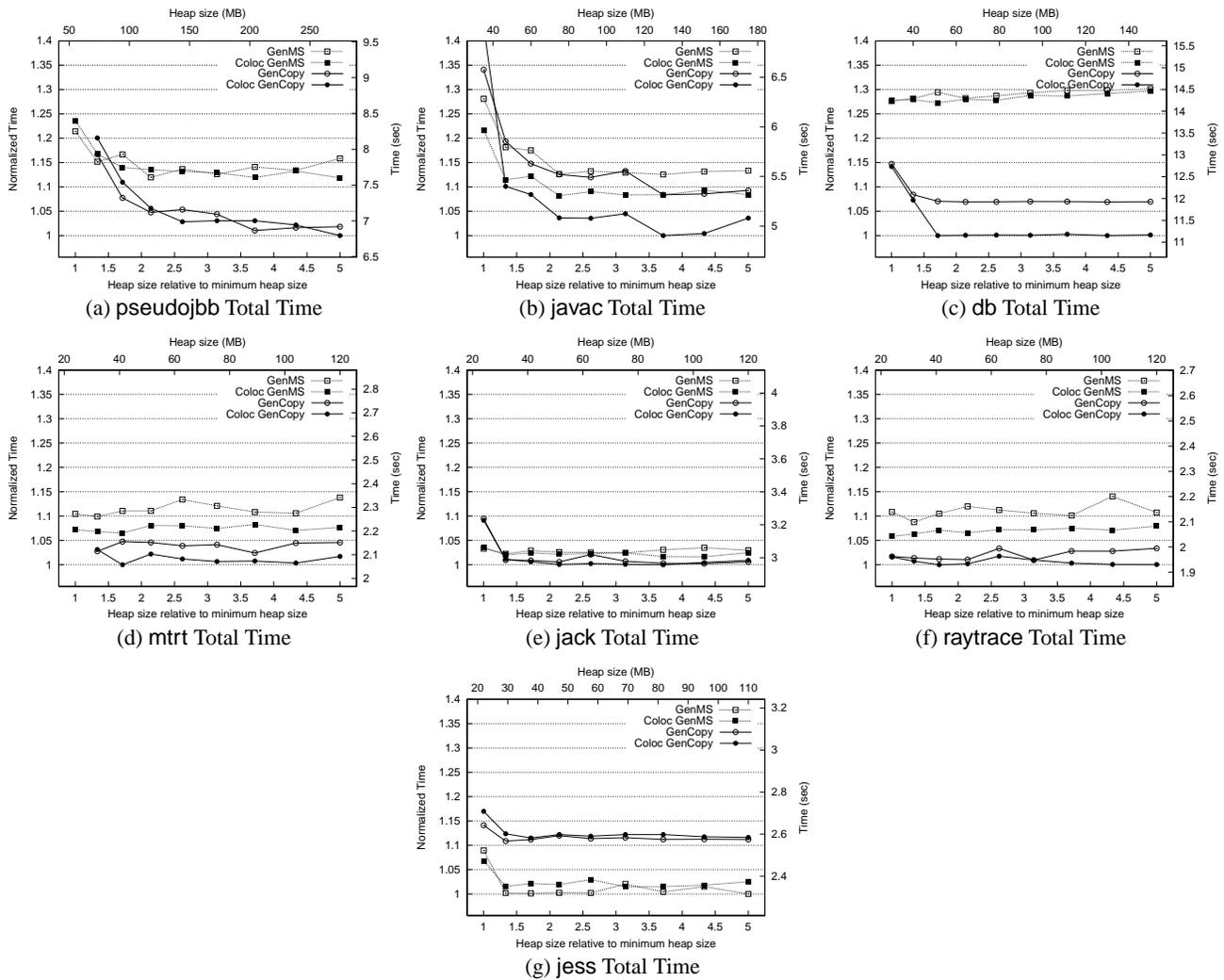
(a) pseudojbb Total Time

(b) javac Total Time

(c) db Total Time

(d) mtrt Total Time

(e) jack Total Time

(f) raytrace Total Time

(g) jess Total Time

Figure 15: Total Time with and without Colocation, 4 MB bounded nursery.

June 2004.

[7] 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.

[8] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *ACM Conference on Programming Languages Design and Implementation*, pages 153–164, Berlin, Germany, June 2002.

[9] S. M. Blackburn, S. Singhai, M. Hertz, , K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 342–352, Tampa, FL, Oct. 2001. ACM.

[10] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.

[11] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Languages Design and Implementation*, pages

296–310, White Plains, NY, June 1990.

[12] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *ACM Conference on Programming Languages Design and Implementation*, pages 162–173, Montreal, Canada, May 1998.

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

[14] 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.

[15] J. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, Nov. 2003.

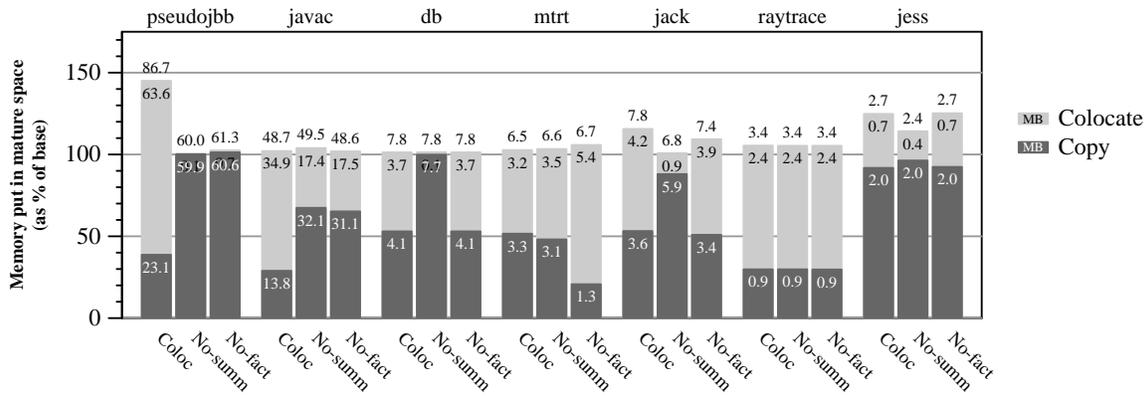[16] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment

Figure 16: Interprocedural analysis is important, particularly for the larger benchmarks such as javac, jack, and pseudojbb.
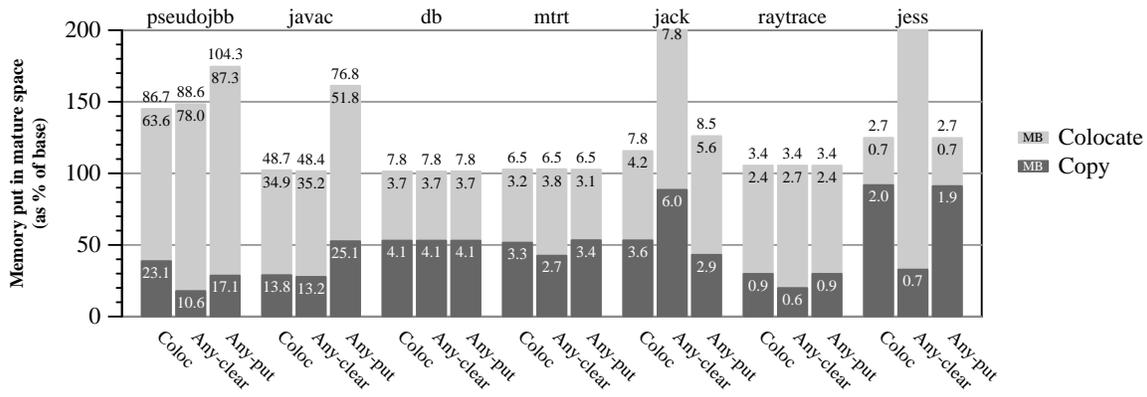

Figure 17: Volatility heuristics effectively prevent excessive colocation, particular in jess which otherwise puts 150 MB in the mature space.
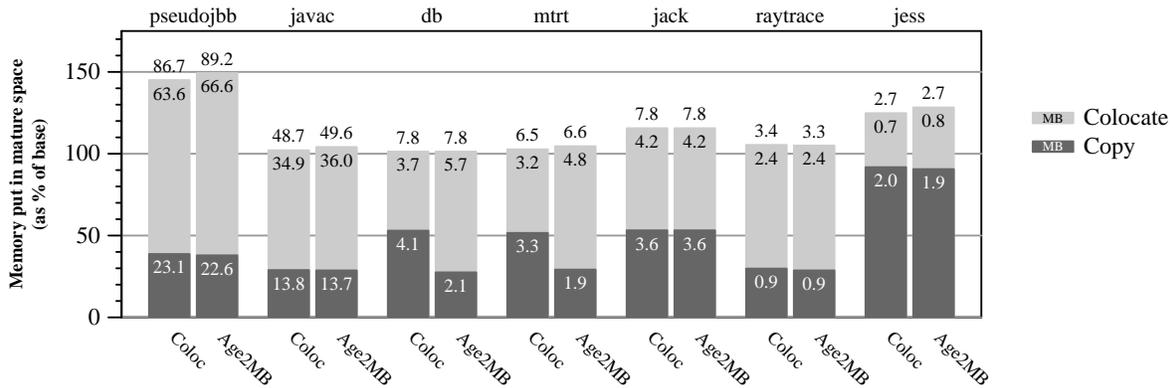

Figure 18: Speculative age-based-colocation helps the smaller benchmarks such as db and mtrt which need to start colocation before the first nursery collection.

form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[17] T. Domani, G. Goldshtein, E. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ACM International Symposium on Memory Management*, pages 76–87, Berlin, Germany, June 2002.

[18] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. In *ACM Conference on Programming Languages Design and Implementation*, pages 242–256, June 1994.

[19] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[20] T. L. Harris. Dynamic adaptive pre-tenuring. In *ACM*

(a) Mean GC Time   (b) Mean Mutator Time   (c) Mean Total Time
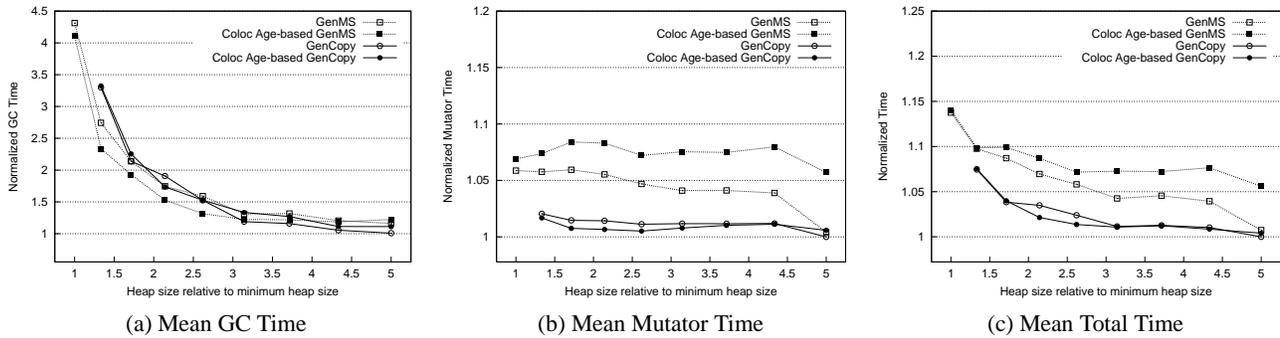
Figure 19: Colocation results with unbounded nursery: Colocation yields less of a benefit for collection time, and incurs a higher mutator time overhead.

(a) No colocation

(b) With colocation
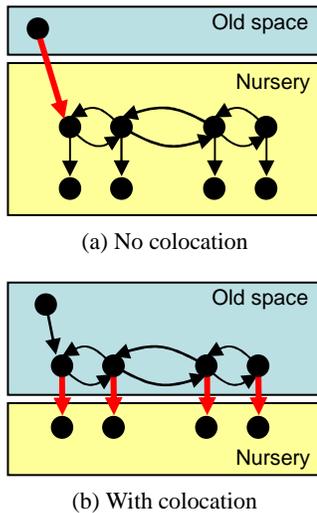
Figure 11: Colocation can increase remset sizes even when it is working correctly.

International Symposium on Memory Management, pages 127–136, Minneapolis, MN, Oct. 2000.

[21] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 359–373, Anaheim, CA, Oct. 2003.

[22] M. Hirzel, J. Hinkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ACM International Symposium on Memory Management*, pages 36–49, Berlin, Germany, June 2002.

[23] X. Huang, Z. Wang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, and P. Cheng. The garbage collection advantage: Improving mutator locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, 2004. To appear.

[24] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management, St. Malo, France*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[25] 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.

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

[27] 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.

[28] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *ACM International Symposium on Memory Management*, Berlin, Germany, June 2002.

[29] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–343, Anahiem, CA, Oct. 2003.

[30] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimzations. In *ACM Symposium on the Principles of Programming Languages*, pages 295–306, Portland, OR, Jan. 2002.

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

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

[33] D. Stefanović, K. McKinley, and J. Moss. Age-based garbage collection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 370–381, Denver, CO, Nov. 1999.

[34] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–17, San Diego, California, Nov. 1988.

[35] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

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

[37] 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.