McGill University
School of Computer Science
Sable Research Group

An Adaptive, Region-Based Allocator for Java

Sable Technical Report No. 2002-1

Feng Qian, Laurie Hendren
[fqian,hendren]@cs.mcgill.ca
Sable Research Group, School of Computer Science, McGill University

Abstract

This paper introduces an adaptive, region-based allocator for Java. The basic idea is to allocate non-escaping objects in local regions, which are allocated and freed in conjunction with their associated stack frames. By releasing memory associated with these stack frames, the burden on the garbage collector is reduced, possibly resulting in fewer collections.

The novelty of our approach is that it does not require static escape analysis, programmer annotations, or special type systems. The approach is transparent to the Java programmer and relatively simple to add to an existing JVM. The system starts by assuming that all allocated objects are local to their stack region, and then catches escaping objects via write barriers. When an object is caught escaping, its associated allocation site is marked as a non-local site, so that subsequent allocations will be put directly in the global region. Thus, as execution proceeds, only those allocation sites that are likely to produce non-escaping objects are allocated to their local stack region.

The paper presents the overall idea, and then provides details of a specific design and implementation. In particular, we present a paged region allocator and the necessary modifications of the Jikes RVM baseline JIT and a copying collector. Our experimental study evaluates the idea using the SPEC JVM98 benchmarks, plus one other large benchmark. We show that a region-based allocator is a reasonable choice, that overheads can be kept low, and that the adaptive system is successful at finding local regions that contain no escaping objects.

1 Introduction

The Java [2] programming language implicitly uses a garbage collector as its memory manager. Garbage collection has the advantage of freeing memory safely and often precisely. However, tracing and collecting objects adds overhead to the program, so reducing the frequency of garbage collection can be beneficial. The stack-allocation of objects is one promising approach to reduce the work of a garbage collector [8]. If an object does not escape a method, it can be created on the stack frame instead of in the heap. Objects on the stack can be reclaimed without intervention by the garbage collector.

There are several potential obstacles to performing object stack-allocation in a JVM. Existing stack-allocation techniques require a static *escape analysis* [5, 4, 15], which usually requires a whole program analysis and is not easily amenable to a JVM with dynamic loading and JIT compilation. Further, as discussed by Gay and Steensgaard [8], there are several restrictions on the kinds of objects that can be created on the stack. Objects with non-trivial *finalize* methods are not easily placed on the stack, arrays may be too large, and objects created in a loop with overlapping lifetimes are not stackable.

To address these issues in a JVM, we suggest an adaptive region-based allocator. In our approach we treat local regions as extensions of stack frames. Thus, stackable objects can be created in regions instead of stack frames without the restrictions above. If a region contains only non-escaping objects, then when its associated stack frame is popped, the region can be deallocated by returning its associated pages to the free list of the heap.

Since we wish to avoid potentially expensive static analyses, we instead take the approach of an **adaptive** strategy which dynamically categorizes allocation sites as *local* and *non-local*. Initially objects are assumed to be *local* and the VM relies on write barriers to detect escaping objects. When an object escapes, two actions are taken. First, the corresponding region is marked as dirty. Second, the allocation site of the escaping object is categorized as non-local, so that subsequent allocations from that site will be put directly in the global region. A dirty region is deallocated by appending the pages associated with that region to a global region waiting for the next garbage collection, while the pages of a clean region can be reclaimed immediately. As execution proceeds, more allocation sites creating escaping objects are marked as non-local, so that local regions are more likely to stay clean.

Our approach requires a region-based allocator as its basis. In general, paging heaps provide a flexible way to divide memory into regions for different purposes, for example, regions [14, 13, 6] or thread-specific heaps [12]. However, the maintenance of regions and unused space on pages can lead to extra costs and overheads when compared to a flat heap organization. Thus, we first outline the design of a region-based paging allocator and we show the behavior of Java programs on paging heaps. Our study shows that most allocations can be satisfied with inexpensive operations and that the correct choice of page size leads to little wasted space. This suggests that paging is a feasible heap organization for Java VMs.

We then show how to build the adaptive system for categorizing allocations into those which should be allocated in local regions associated with stack frames, and those that should be directly allocated in the global region. In the design we carefully consider the overheads, both in space and extra write barrier overhead, and we propose various approaches for keeping these overheads reasonably small.

This paper makes the following contributions:

1. We quantitatively studied the allocation behaviors of Java programs in a region-based allocator based on paging heaps. This study indicates that dividing heap into special regions is feasible for Java programs.

2. In contrast to previous work, our memory manager does not require expensive static analysis, programmer directives, or special type systems. It is completely transparent to the Java programmer, and requires relatively few modifications to the JIT compiler.

3. Our method focuses on the allocation of new objects and can work as an optimization in conjunction with other GC algorithms - we have implemented it in conjunction with an extant type-accurate copying collector. Our method also works on Java in its full generality, including multiple threads, dynamic class loading and the JNI.

4. We have carefully analyzed the overheads of our approach and we suggest several mechanisms for reducing this overhead. We use lazy region allocation to reduce the number of region allocations and deallocations. Further, even though our method requires some extra state in the object header, we show how to implement this in a standard two-word header with a small tradeoff in the performance of thin locks.

We have implemented the proposed approach in the Jikes RVM[1] [1] and have performed our initial study using its baseline compiler. This allows us to evaluate the approach's behavior and potential. We are currently working on extending the implementation to the optimizing compiler where we can evaluated further runtime speedups.

The paper is organized as follows. In Section 2, we introduce the overall structure of the system. The design and implementation of the allocator are presented in Section 3. Sections 4 and 5 describe the required modifications to the JVM and collector to utilize the new functionalities provided by the allocator. In section 6, we present experimental results that focus on the behavior of the approach. Finally, we discuss the related work in Section 7, conclusion and future improvements in Section 8.

2 Overview

The whole system consists of three parts: the allocator manages regions and allocates space for objects; the JIT compiler inserts instructions for acquiring and releasing a region in each compiled method; and the collector performs garbage collection when no more heap space is available.

In a region system, the heap space is divided into pages. The pages can be fixed-size or variable-size. In our system, we use fixed-size pages for fast computation of page numbers from addresses. The allocator is also a region manager. It manages a limited number of tokens. Each token is a small integer number identifying a region. Two regions, GLOBAL and FREE, exist throughout the execution of a program. Other, local regions, exist for shorter durations. They are assigned to and released by methods dynamically.

A high-level view of our memory organization is given in Figure 1. A more detailed description of the implementation is given in Section 3.

A region token points to a list of pages in the heap. The region space is expanded by inserting free pages at the head of the list. The GLOBAL region contains objects created by non-local allocation sites and pages

[1] formerly known as the Japapeño Virtual Machine.

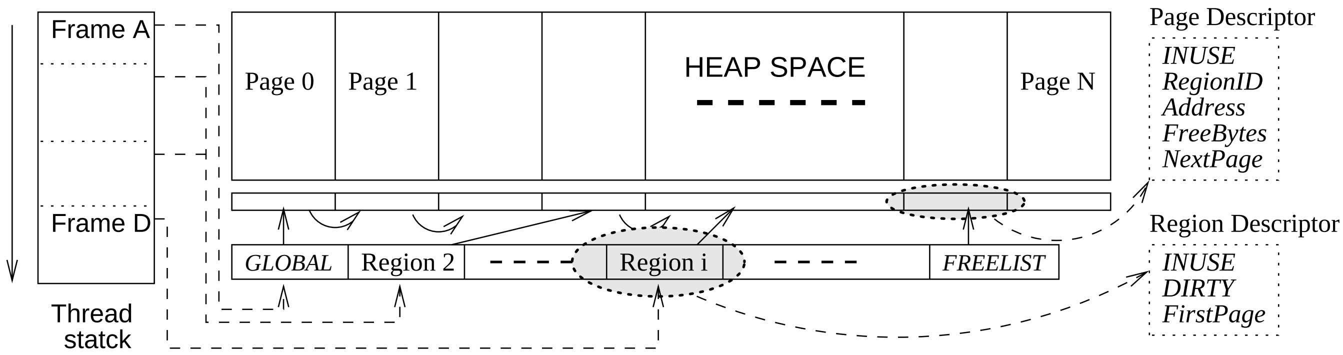


Figure 1: Memory organization of paged heaps with regions

containing objects that have escaped out of local regions. The GLOBAL region space can only be reclaimed by the collector. The system uses bit maps to keep track of free pages in the heap. The pages of a local region can be appended to the GLOBAL region or reclaimed by resetting their entries in the bit map.

A method activation obtains a region token by either acquiring an available token from the region manager or by inheriting its caller's token. The region identified by the token acts as an extension of the activation's stack frame. Before exiting, the activation releases the region if it was not inherited. It is clear that the lifetime of a local region is bounded by the lifetime of its host stack frame. There is a many-to-one mapping between stack frames and regions.

An object can be created in the region of the top stack frame or in the GLOBAL region. For the remainder of the discussion we need to define what we mean by an object *escaping* from a region, and a *non-local* allocation site.

Definition 1 *An object* escapes its allocation region *if and only if it becomes pointed to by an object in another region.*

Definition 2 *An allocation site becomes* non-local *when an object created by that site escapes.*

Given this definition of escape, there are only three Java bytecode instructions, `putstatic`, `putfield`, and `aastore`, that can lead to an object escaping. Therefore, it is sufficient to insert write barriers before these instructions to detect the escape of an object.

There is one additional situation that must be considered. When a method returns an object, the object may escape its allocation region via stack frames. However, this kind of escape can be prevented by either: (1) inserting write barriers before all `areturn` byte codes, or (2) requiring all methods returning objects to inherit their caller's region. In our implementation we have taken the second approach. It should be noted that objects passed to the callee as parameters are not a problem since the lifetime of the callee's stack frame is bounded by the caller's.

For an assignment such as $lhs.f = rhs$, the write barrier checks if the rhs is in the same region as the lhs object. When they are in different regions, the region containing the rhs object is marked as dirty. Since static fields are much like global variables, we assume that a *putstatic* always leads the rhs object escaping, and the region associated with this object is marked as dirty.

Each allocation site in a compiled method is uniquely indexed, and each object has a field in its header for recording the index of its allocation site (see Section 4 for a discussion of how this is accomplished without increasing the object header size). The allocator maintains a bit vector to record the states of the allocation sites. Besides marking the region dirty, the write barrier also marks an escaping object's allocation site as *non-local*. The allocator allocates objects in local regions only for local allocation sites. By not allocating objects for non-local sites in the local region, future activations of the method are very likely to have a local region containing only non-escaping objects.

The system is quite straightforward and we have implemented it in the Jikes RVM. The Jikes RVM is written in Java. It has a baseline compiler which generates machine code simulating a stack machine. It also has an optimizing compiler which uses many sophisticated optimizations. The prototype of the allocator is implemented with the baseline compiler only. When we present the JVM-related part in Section 4, the stack frame layout refers to the conventions of the baseline compiler.

3 Allocator

3.1 Heap organization

Various garbage collection techniques have different heap organizations. For example, mark-and-swap collectors use a single space, copying collectors divide space into two semi-spaces, and generational collectors divide the heap into several aging areas. In this report, the heap we are discussing refers to the space where new objects are allocated.

A region memory manager organizes a heap as pages. Without loss of generality, the heap in our system is organized as contiguous pages with a fixed size which is a power of 2. The starting address of the heap is aligned to the page size. Therefore, computing the page number for an address requires only subtraction and bit shifting. Some systems do not allocate the large objects from the paged heap, and do not allow an object to straddle two pages. In order to get a full picture of allocation behaviors, our system does not use a separate space for large objects and attempts to allocate objects on contiguous pages whenever it can.

Figure 1 gives a high-level overview of the memory organization that we use for regions implemented as paged heaps. Each page has a descriptor which contains the following information:

- `INUSE` indicates the page is in use.
- `REGIONID` records an integer token identifying the region this page belongs to.
- `ADDRESS` is the allocation point of this page.
- `FREEBYTES` counts how many bytes remain in the page.
- `NEXTPAGE` is the index of the next page in the region.

Heap pages are managed by a bit map (*INUSE* bit). Pages in a region are linked as lists by *NEXTPAGE* fields. The advantage of using linked lists is that pages in a region can be merged into another region by changing *REGIONID* and *NEXTPAGE* fields only. The bit map allows the allocator to search for contiguous free pages quickly, thus allowing objects to straddle pages.

Each region has a descriptor indexed by its *region ID* (token). The descriptor contains information for maintaining the status of the region and its linked list of pages. A region descriptor has the following fields:

- `INUSE` indicates the region is in use.
- `DIRTY` indicates whether any objects in this region have escaped.
- `FIRSTPAGE` is the index of the first page in this region; other pages are linked by page descriptors.

This organization provides sufficient information for region-based allocation. When allocating space in a region, the allocator first checks the free bytes of the first page. When there is not enough space left there, a free page is taken from the free list and inserted in the page list as the first page. Allocating space for large objects involves searching for contiguous free pages. We have measured the overhead for these allocations for our benchmarks, and as shown in Section 6, the frequency of expensive searches is quite low, indicating that this is a reasonable design.

3.2 Services

The internal heap organization is transparent to the JVM. The allocator provides a set of services to the JVM and collector. We describe these functions here.

There are two services for region operations as shown in Figure 2. Internally, free region tokens are managed by a stack. The `newregion` method pops a region token from the stack, and pre-allocates one free page for it (pre-allocation is only used with lazy region allocation, to be explained in section 4). If no token is currently available, the GLOBAL one is returned. The `freeregion` operation has to call the internal method `_yesRegionDirty` to check the DIRTY field in the region descriptor. Only when the region is clean, can pages be reclaimed by `_addPagesInFree`. Otherwise, `_addPagesInGlobal` will append pages to the page list of the GLOBAL region.

```
int newregion() {
  if (_rid_stack.empty()) {
    return GLOBAL;
  } else {
    int rid = _rid_stack.pop();
    _addOneFreePage(rid);
    return rid;
  }
}

void freeregion(int rid) {
  if (rid == GLOBAL) return;

  if (_yesRegionDirty(rid)) {
    _addPagesInGlobal(rid);
  } else {
    _addPagesInFree(rid);
  }

  _rid_stack.push(rid);
}
```

Figure 2: Services for regions

As outlined in Figure 3, the allocator provides two services for write barriers. The `checkwrite` method is called before *putfield* and *aastore* byte codes. The addresses lhs and rhs point to the left hand side and right hand side objects. The `_getPageNo` method computes the page number for the address by subtracting the starting address of the heap and bit shifting. The region ID is retrieved from the page descriptor by `_getRegionId`.

The write barrier for *putstatic* calls `markescaped` directly. As we explained in section 2, the allocator uses a bit vector to record the states of allocation sites. The `markescaped` service not only marks the region as dirty, but also sets the state of the allocation site to *non-local*. Internal methods `_getNewsiteNo`, `_markNonlocal`, and `_markDirty` facilitate this task. In the object header, a bit in the status word is used to mark an object as escaped. The method `_yesEscaped` filters out further operations on an escaped object.

The main function of the allocator is to allocate space for an object. With regions, the allocation of space is somewhat complicated. The allocation process is illustrated in figure 4. Here, we present only a high-level abstract of two internal methods `_getHeapSpace` and `_expandRegionSize`. Appendix A provides detailed pseudo-code for these functions.

The allocation method first checks the state of the allocation site. Only local sites are eligible for allocation from local regions. The internal method *_getHeapSpace* allocates space in the first page if the free space is larger than the required size. If the first attempt fails, *_expandRegionSize* is called to get free pages from the free list. The *_expandRegionSize* method first looks at pages following the current page. If the request can't

```
boolean checkwrite(ADDRESS lhs,
                   ADDRESS rhs) {
  if (rhs == VM_NULL)
    return true;

  if (_yesEscaped(rhs))
    return false;

  int pl = _getPageNo(lhs);
  int pr = _getPageNo(rhs);

  if (pl == pr) return true;

  int rl = _getRegionId(pl);
  int rr = _getRegionId(pr);

  if (rl == rr) return true;

  _markEscaped(rhs);
  return false;
}

void markescaped(ADDRESS rhs) {
  if (rhs == VM_NULL)
    return;

  if (_yesEscaped(rhs))
    return;

  _markEscaped(rhs);
}

private void _markEscaped(ADDRESS rhs) {
  int ns = _getNewsiteNo(rhs);
  _markNonlocal(ns);

  int pr = _getPageNo(rhs);
  int rr = _getRegionId(pr);

  _markDirty(rr);
}
```

Figure 3: Services for barriers

be satisfied from these pages, it then looks for contiguous pages by scanning the bit maps. This is the most expensive operation in a region-based allocator.

These services also provide the facilities required by the garbage collector to perform collections. We discuss the collection process in Section 5.

4 Adaptive VM

To utilize regions, a JVM needs the following modifications:

1. Each allocation site is assigned a unique index at compilation time.
2. The object header has a field for recording the index of its allocation site.

```
ADDRESS alloc(int rid, int size) {

  ADDRESS addr = _getHeapSpace(rid, size);
  if (addr == OUT_OF_MEMORY) {
    collect();
    addr = _getHeapSpace(rid, size);
    if (addr == OUT_OF_MEMORY) {
      _outOfMemory();
    }
  }

  return addr;
}

ADDRESS _getHeapSpace(int rid, int size) {
/* 1. allocate space from the first page
 * 2. if failed, call _expandRegionSize
 */
}

ADDRESS _expandRegionSize(int rid, int size) {
/* 1. check if enough pages following
 *    the first page are available.
 * 2. if not, search contiguous pages
 *    in the free list.
 * 3. if both attempts fail,
 *    return OUT_OF_MEMORY.
 * 4. expand the region size by adding
 *    free pages to the linked list.
 */
}
```

Figure 4: Allocating spaces

3. The stack frame has a slot for the region ID at a fixed offset from the frame pointer.

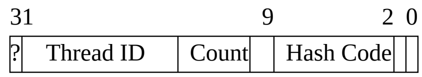
4. The method prologue and epilogue have additional instructions to initialize the region ID slot.

5. Write barriers are inserted before *putstatic*, *putfield*, and *aastore* byte codes.

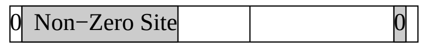
The allocation method has two more parameters than before: the index of an allocation site is a runtime constant, and the region ID is fetched from the stack frame.

When deciding whether or not a method is eligible for a new local region, our implementation uses following criteria:

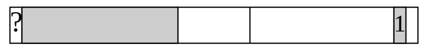
- A native call is assigned the GLOBAL region id.
- The *<clinit>* method always uses the GLOBAL region since we know that it initializes static fields.
- The *<init>* method inherits the caller's region because it initializes the instance fields.
- A method returning an object is not eligible for a new region. This rule eliminates the need for a write barrier for the *areturn* byte code. More importantly, as pointed out by [8], there are many methods just allocating objects for the caller.
- A one-pass scan of the byte codes counts the allocation sites of each method. If the number is lower than a threshold, no local region is needed. We currently use a threshold of 1.



(a) Structure of the status word



(b) Status of a non-escaping object



(c) Status of an escaped object

Figure 5: Sharing bits with thin locks

- The first executed method of the booting thread is assigned the GLOBAL region ID.

In our initial development it became clear that making *newregion* and *freeregion* calls on each activation is too expensive for the run time system since many activations may have empty regions, either because their allocation sites have become non-local or because no object is allocated. To eliminate these empty regions, we use lazy region allocation. An eligible method first saves a special region ID, e.g. `0`, in the region ID slot, indicating the stack frame needs a dynamic region, but it has not yet been allocated. The code for allocation first checks the ID, and then calls *newregion* only when necessary. The *freeregion* method is called only when the region ID is a valid one. If a method inherits a region from its caller, it must write back its current region ID to the caller's stack frame.

Another implementation issue is how to encode the allocation site index in the object header. A two-word object header is a quite popular design on most JVMs. One word of the header is used as a status word. Our implementation avoids growing the object header by storing the allocation site index in space already used by the thin lock [3].

In the Jikes RVM, the thin lock uses 13 bits for recording the ID of the locking thread, and 6 bits for counting locks. Figure 5(a) shows the structure of the status word. The bit 31 is called the *monitor shape bit* which is 0 if the lock is thin and 1 if it is fat. The thin lock mechanism attempts to check the monitor shape bit and the thread ID field in the status word first. Without an allocation site index, the common case will be that the monitor shape bit and thread ID field are zero. With such an index, however, this test will fail on a non-escaping object of which status word a non-zero allocation site index is stored in the thread ID field as shown in Figure 5(b). But the attempt will still succeed on an escaped object whose status word is shown in Figure 5(c). In Figure 5(b) and (c), bit 1 in the object status word marks the object as escaped or not[2]. When the thin lock fails on a non-escaping object, the allocation site field is cleared and the escaping bit is set, then the lock is attempted again. To ensure correctness, an escaped object never becomes non-escaping, and whenever an object is marked as escaped, the region is marked as dirty.

The system only adds a check on the uncommon path of the thin lock and may need one check on the common path in very few cases. The mechanism allows us to encode the site number up to $2^{13} - 1$. For large applications, it would be possible to use both the thread ID and lock count to store a 19-bit allocation site index if their positions were reversed (to ensure that small allocation site index still produce a non-zero thread ID field).

There are some other issues related to Java semantics [10]. An exception may transfer control to the caller without going through the method epilogue. In this case, the exception mechanism must release the region before unwinding the stack frame. This is done by the following pseudo-code:

[2] The bit 1 is used for write barrier purpose in other types of GC. In our prototype implementation in a copying collector, this bit is used as the escaping bit.

```
void unwindStackFrame( ... ) {
  ......
  int fp  = _getCurrentFramePointer();
  int rid = _getRegionID(fp);
  if (rid != GLOBAL) {
    int callerfp = _getCallerFP(fp);
    int callerrid = _getRegionID(callerfp);

    if (rid != callerrid) {
      freeregion(rid);
    }
  }
  ......
}
```

If an object has a non-trivial finalize method, the JVM has to run the finalizer before the space is reused. The region-based allocator organizes the list of objects with non-trivial finalizers by region ID. When the pages of a region are about to be reclaimed, the finalize methods of objects in the region get called.

5 Collector

Changes to the collector are optional. The collector must ensure that if an object escapes its original region, that region is marked as dirty. One way of ensuring this would be to introduce write barriers during collections. However, this may sacrifice the efficiency of the collector. There is a trade-off between precision and performance. If all live objects are copied to dirty regions, no barriers are needed. So, the second option is to copy all live objects to the GLOBAL region of another space. This does not violate the above rule since the global region never gets released. This strategy has the same efficiency as a normal copying collector. However, copying all objects to the GLOBAL region may cause some objects created in the next epoch to be treated as escaped, and their associated allocation sites marked as non-local, unnecessarily.

Our current implementation keeps objects in their original region as much as possible, and marks all live regions as dirty after the collection. Now objects in the root set are divided into subsets by their regions, with each live region corresponding to a subset of the root set. The collector starts with collecting all reachable objects from the subset belonging to the GLOBAL region. In the next step, the collector collects objects reachable from the subset corresponding to each local region. All objects copied to the GLOBAL region are marked as escaped to allow fast checks in barriers (the states of the allocation sites are not changed). Although this strategy makes some stackable objects in current live regions unstackable, it does not require write barriers and will not make allocation sites non-local unnecessarily. Currently, we do not have experimental evidence to show which option is better in reality.

6 Results

We implemented a region-based paging heap and a prototype allocator in the Jikes RVM with the baseline compiler. Porting of the system to the optimizing compiler is in progress. The region-based allocator is implemented in a semi-space copying collector using Cheney's tracing algorithm. The current implementation uses a uniprocessor configuration. However, it can be implemented in existing parallel collectors with little effort.

To understand the program behavior, we did detailed profiling of the allocation behaviors, and report the experimental results of the following aspects:

- the allocation behavior of the region-based paging allocator;
- the percentage of space reclaimed by local regions, and the reduction in collections due to local region allocation;

- the behavior of write barriers;
- the impact on thin locks; and
- the effect of adaptivity.

6.1 Benchmarks

We report experimental results on the SPEC JVM98 benchmark suite and *soot-c*. The *soot-c* benchmark is a Java byte code transformation framework that is quite object-oriented, and which has several phases with potentially different allocation behaviors.

We first provide some measurements to give some idea of the allocations performed by each benchmark. Table 1 shows the profiles of allocation sites. The column labeled *Compiled* gives the total number of allocation sites in compiled methods. It includes the allocation sites in the JVM, libraries and benchmark code. The column labeled *Used* lists the number and percentage of allocation sites which created at least one object. On average 26% of the allocation sites create at least one object. The columns labeled *Non-local* and *Local* show the fraction of used allocation sites which are categorized as *non-local* and *local*. An allocation site is categorized as *local* if it is never marked as *non-local* by the adaptive algorithm. The last column, labeled *Max RID*, gives the maximum number of regions used by the benchmark at the same time. This gives us some idea of the number of region tokens required. Note that a program with deep recursion may use a large number of regions.

Benchmark	Compiled	Used	Non-local	Local	Max RID
compress	2108	346(16%)	115(33%)	231(67%)	11
db	2117	358(16%)	124(34%)	234(66%)	11
jack	2396	614(25%)	204(33%)	410(67%)	16
javac	2871	895(31%)	437(48%)	458(52%)	56
jess	2407	577(23%)	276(47%)	301(53%)	9
mpegaudio	3266	1502(45%)	157(10%)	1345(90%)	12
mtrt	2228	497(22%)	196(39%)	301(61%)	19
soot-c	3030	1158(34%)	551(52%)	507(48%)	14

Table 1: Allocation sites

In all of our experiments the total heap size is set to 50M, from which the JVM uses about 1.5M as the boot area. The JVM itself shares the same heap with applications. We do not distinguish the objects created by the system or the benchmarks. The heap is divided into two semi-spaces. A 25M heap is quite small for most of our benchmarks, which forces the garbage collector to work.

6.2 Choice of Page Size

The choice of page size may affect utilization of heap space. A larger page size will allow more allocations to be satisfied on the first page. On the other hand, smaller page size will reduce the amount of *froth* (unused pieces of the heap due to the allocation of chunks/pages of memory[3]). Table 2 shows the the effect of different page sizes on the number of garbage collections needed and the froth rates. The column labeled *Base collections* gives the number of collections needed for the base semi-space copying garbage collector, without the regions. The three columns labeled *clc* give the collections required for the region-based allocator, assuming page sizes of 256 bytes, 1K bytes and 4K bytes.[4] Similarly the columns labeled *froth* give the wasted space for the three different page sizes (computed by unused_bytes/allocated_bytes). Note that a page size of 4K leads to a large froth rate for several benchmarks, most notably `javac` (130%),

[3] This term was introduced by Steensgaard [12].
[4] We disabled System.gc() calls for both collectors.

`jack` (27.5%) and `soot-c` (23.5%). The very high froth rate for `javac` also seems to increase the number of garbage collections, which is more than double that of the base collector.

Benchmark	Base # collections	256 Bytes		1K Bytes		4K Bytes	
		clc	froth	clc	froth	clc	froth
compress	7	7	0.03%	7	0.11%	7	0.47%
db	4	4	0.05%	4	0.23%	4	1.05%
jack	9	7	1.29%	8	5.97%	9	27.52%
javac	12	12	4.96%	15	29.41%	25	130.42%
jess	12	11	0.13%	11	0.53%	11	2.19%
mpegaudio	0	0	0.62%	0	2.10%	0	9.05%
mtrt	7	1	0.03%	1	0.09%	1	0.38%
soot-c	15	13	1.09%	13	4.89%	15	23.49%

Table 2: Effect of page size on # of collections and froth

From the perspective of number of collections and froth, the smaller pages seem better. However, this is not the complete story. One must also consider the overhead for allocations. The cheapest form of allocation is when the newly allocated object fits on the current page, the second cheapest is when the allocation can be allocated on the next page, and the most expensive is when one must search the freelist for enough contiguous pages to meet the allocation request. These overheads are summarized in Table 3. Considering the three page size configurations: 256-byte, 1K, and 4K, the allocations are categorized into three types, which reflect three possibilities in *_getHeapSpace* and *_expandRegionSize* in Figure 3.

1. `firstpage` The space is available in region's first page.
2. `nextpage` The region is expanded with immediately contiguous pages.
3. `searching` Search for contiguous pages in the free list.

A large size (4K) allows most of allocations to be satisfied with cheap costs. But, as we demonstrated in Table 2, the froth rate may run out of control. From Table 3 we see that there is not a large difference in the behavior of allocations when comparing page sizes of 1K and 4K. However, even though a smaller page size (256 bytes) reduces froth rates, the allocation distribution changes dramatically, with many more allocations requiring expensive operations. From these results we conclude that the trade-off between page size and froth rate is worth considering when using a paging heap. For our remaining experiments we chose a page size of 1K, which gives us both reasonable froth rate and reasonable allocation overhead.

6.3 Region-reclaimed space

The next important measurement is to find out the percentage of space that is reclaimed from local regions. That is, how much space can be reclaimed when using the region-based approach. Recall that the region can be reclaimed when a stack activation is popped only when the dirty bit has not been set (i.e. the region is clean). If any object in the region has escaped, then the dirty bit will be set, and this region must be added to the GLOBAL region which will be collected by the garbage collector.

The table given in Figure 6(a) gives the bytes reclaimed from clean regions and the percentage they represent of total allocated bytes, when the page size is 1K. Different page sizes give very similar numbers. The percentage of region-reclaimed bytes varies between benchmarks. In the best case, *mtrt* has 80 percent of total allocated memory reclaimed by regions, with the number of collections reduced from 7 to 1. In the worst case, *db* has less than 1% region-reclaimed space, with no impact on the number of collections.

Another way to look at the behavior of the regions is to examine the number of bytes allocated from local regions over the duration of the execution. Figure 6(b) shows the fraction of bytes allocated that are allocated from local regions. The x-axis is an abstraction of time, with each unit corresponding to 1Mbytes

Benchmark	page size	firstpage	nextpage	searching
compress	256	82.73%	16.23%	1.04%
	1K	94.96%	4.74%	0.30%
	4K	98.43%	1.44%	0.13%
db	256	92.24%	7.69%	0.07%
	1K	98.04%	1.92%	0.03%
	4K	99.49%	0.48%	0.03%
jack	256	91.64%	6.46%	1.91%
	1K	97.79%	1.59%	0.63%
	4K	99.48%	0.33%	0.19%
javac	256	89.56%	8.58%	1.85%
	1K	97.43%	2.00%	0.57%
	4K	99.41%	0.50%	0.09%
jess	256	86.75%	12.89%	0.36%
	1K	96.71%	3.27%	0.02%
	4K	99.16%	0.83%	0.01%
mpegaudio	256	84.97%	12.09%	2.94%
	1K	95.54%	3.52%	0.94%
	4K	98.59%	0.98%	0.43%
mtrt	256	96.04%	2.58%	1.39%
	1K	99.51%	0.38%	0.11%
	4K	99.88%	0.10%	0.02%
soot-c	256	88.28%	9.85%	1.88%
	1K	96.85%	2.66%	0.49%
	4K	99.21%	0.68%	0.11%

Table 3: Behaviors of allocations

of allocations. The y-axis shows the fraction of those 1Mbytes that were allocated from a local region. For example, the graph labeled `mtrt` indicates that after an initial startup, about 90% of all allocated bytes are allocated from local regions. In contrast, the graph labeled `jess` shows that this benchmark quickly declines to less than 10% of all allocations from local regions. The graph labeled `soot-c` shows a widely fluctuating rate as the program progresses. This is likely because `soot-c` is quite a complex benchmark with many different phases. It is interesting to note that if all the bytes allocated to local regions are also region-released, then the area under the curves of Figure 6(b) should be equal to the region-reclaimed number shown in Figure 6(a). This appears to be the case, as confirmed by our measurements given in Section 6.6, showing that almost all local regions are clean when released.

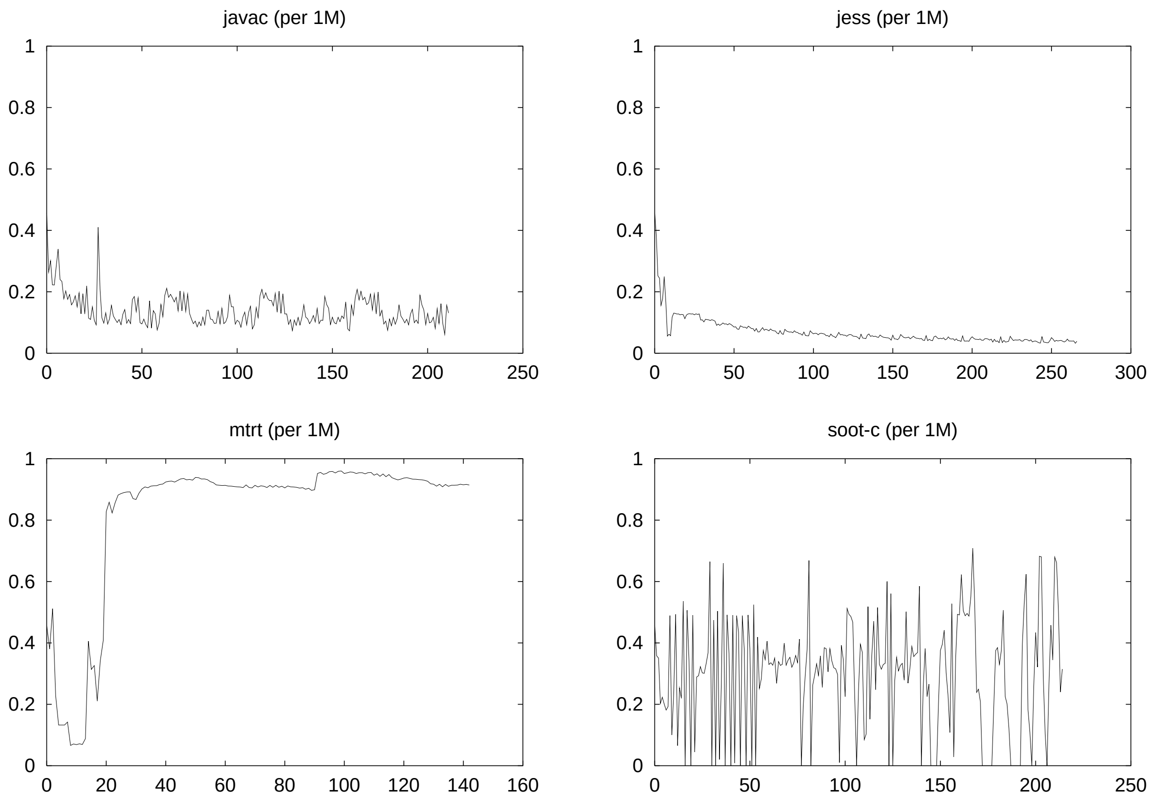
6.4 Write Barrier Behavior

Another important aspect of the collector we measured is the behavior of write barriers, as summarized in Table 4. Overall, a write barrier has bounded constant time as shown by the pseudo-code in Figure 3. However, it is still a burden to the system. To get better idea of how to optimize the barriers, we categorize the *checkwrite* for *putfield* and *aastore* into five types.

1. the right hand side is a null pointer,
2. the right hand side object is already marked as escaped,
3. both objects are in the same page,
4. both objects are in the same region, and

Benchmark	base collections	total allocated	region-reclaimed	clc	froth
compress	7	116M	15.39M(13.27%)	7	0.11%
db	4	77M	0.57M(0.74%)	4	0.23%
jack	9	223M	51.00M(22.87%)	8	5.97%
javac	12	212M	18.65M(8.80%)	15	29.41%
jess	12	267M	17.36M(6.50%)	11	0.53%
mpegaudio	0	7M	1.96M(28.00%)	0	2.10%
mtrt	7	143M	115.15M(80.52%)	1	0.09%
soot-c	15	219M	40.72M(18.59%)	13	4.89%

(a) Region-reclaimed space



(b) Bytes allocated in local regions

Figure 6: Behavior of Local Regions

5. the *LHS* and *RHS* objects are not in the same region.

The first case checks if the right hand side reference is a null pointer, and the second case checks the escaping bit, which requires a load and a compare instruction. As shown in Table 4, the majority of checks are filtered out by these two cases, which indicates it is beneficial to separate these two cases as a common path and inline them. The remaining three cases can be processed by a method call. Similarly the write barriers for *putstatic* are also categorized into three types, with the first two cases benefiting from inlining.

BENCHMARK	checkwrite for `putfield` and `aastore`					markescaped for `putstatic`		
	null	quick	samepage	sameregion	escaped*	null	quick	escaped
compress	15.07%	83.64%	1.05%	0.17%	0.07%	0.00%	91.45%	8.55%
db	0.12%	99.88%	0.00%	0.00%	0.00%	0.00%	90.48%	9.52%
jack	11.79%	87.58%	0.62%	0.01%	0.00%	0.00%	91.00%	9.00%
javac	10.81%	88.84%	0.32%	0.03%	0.00%	2.90%	89.63%	7.47%
jess	0.22%	99.75%	0.02%	0.00%	0.00%	1.88%	85.63%	12.50%
mpegaudio	0.42%	98.12%	0.08%	1.38%	0.00%	3.20%	83.56%	13.24%
mtrt	13.82%	84.49%	1.66%	0.02%	0.00%	0.00%	86.44%	13.56%
soot-c	5.20%	91.88%	2.71%	0.21%	0.00%	0.00%	93.48%	6.52%

*In this table, zero only means the rate is lower than 0.005%.

Table 4: Behaviors of write barriers

6.5 Impact on Thin Locks

We also profiled the impact of sharing bits with thin locks. Table 5 shows the rate of failed locks because of sharing bits. *Compress* and *mpegaudio* have only a few thousand locks in a full run, so their results cannot represent the real effect of sharing bits. On other benchmarks, the rate of spoiled locks is no more than 5%.

Benchmark	thin locks	spoiled locks
compress	1.6K	172(9.58%)
db	45.2M	2915(0.01%)
jack	9.4M	497016(5.00%)
javac	14.7M	341585(2.26%)
jess	4.8M	4881(0.10%)
mpegaudio	5.6K	497(8.15%)
mtrt	1.3M	25816(1.92%)
soot-c	5.6M	73204(1.30%)

Table 5: Impact on thin locks

6.6 Effect of Adapting

The last behavior that we studied was the effect of the adaptive part of our algorithm. The basic idea of our approach was to mark allocation sites as non-local as soon as they are found escaping the first time. The justification of this decision was that this would prevent this allocation site from spoiling clean regions in the future, and we expected that this would lead to most local regions being clean at release time. Figure 7(a) shows the number of local regions that are clean at release time over the duration of the execution of the program. The x-axis is an abstraction of time, with each point representing the release of 1000 local regions (100 for `jess`). The y-axis shows the number of those local regions that are clean at release time. Accompanying the title of each graph is the number of clean regions and total allocated regions. It is very clear that after a short startup time, the system quickly adapts so that almost 100% of the local regions are clean at release time. There is occasionally a small dip, but then the system adjusts and it goes back to almost 100%. So, it does appear that the system adapts well.

In order to see what would happen without adapting, we removed the part of the algorithm that marks an allocation site as non-local, so that **all**[5] allocations are placed in the local regions. Figure 7(b) shows the result in this case. First, note that many more regions are created, and the scale on these graphs are now per 10000 local regions (1000 for `jess`). However, we can also see some interesting trends. The benchmark `mtrt`

[5]except native calls, < *clinit* >, and the first executed method of the boot thread, see Section 4.

appears to create mostly non-escaping objects, so for this benchmark it is not such a bad idea to just put all objects in local regions. For benchmarks `javac` and `soot-c`, we see that removing the adaption leads to many more regions, and many of those regions are not clean. In these cases the adaption works to cull those dirty regions. For `jess` we see a very interesting behavior, in that in the last two thirds of the execution, a lot of objects appear to be non-escaping, and the number of clean regions stays quite high. With the adaption, we get a higher percentage of clean regions, but we don't find nearly as many. In this case we suspect that there are some allocation sites which sometimes produce escaping objects, and sometimes not. A more complicated prediction scheme appears to be necessary for this kind of benchmark.

We also collected our overall measurements for the two cases, with adaption and without adaption. These are summarized in Table 6. Note that in some cases, most notably `javac`, the froth rate (589% instead of 29%) and number of garbage collections (96 instead of 15) grows quite drastically when the adaption is switched off. However, as we might have predicted from the graph in Figure 7(b), the performance for `jess` is much better for the case of no adaption. This is a clear sign that we must look at other forms of adaption that are more robust when the objects created from a site sometimes escape, and sometimes do not. Overall, it seems that the adaptive algorithm gives better performance and controls excessive froth.

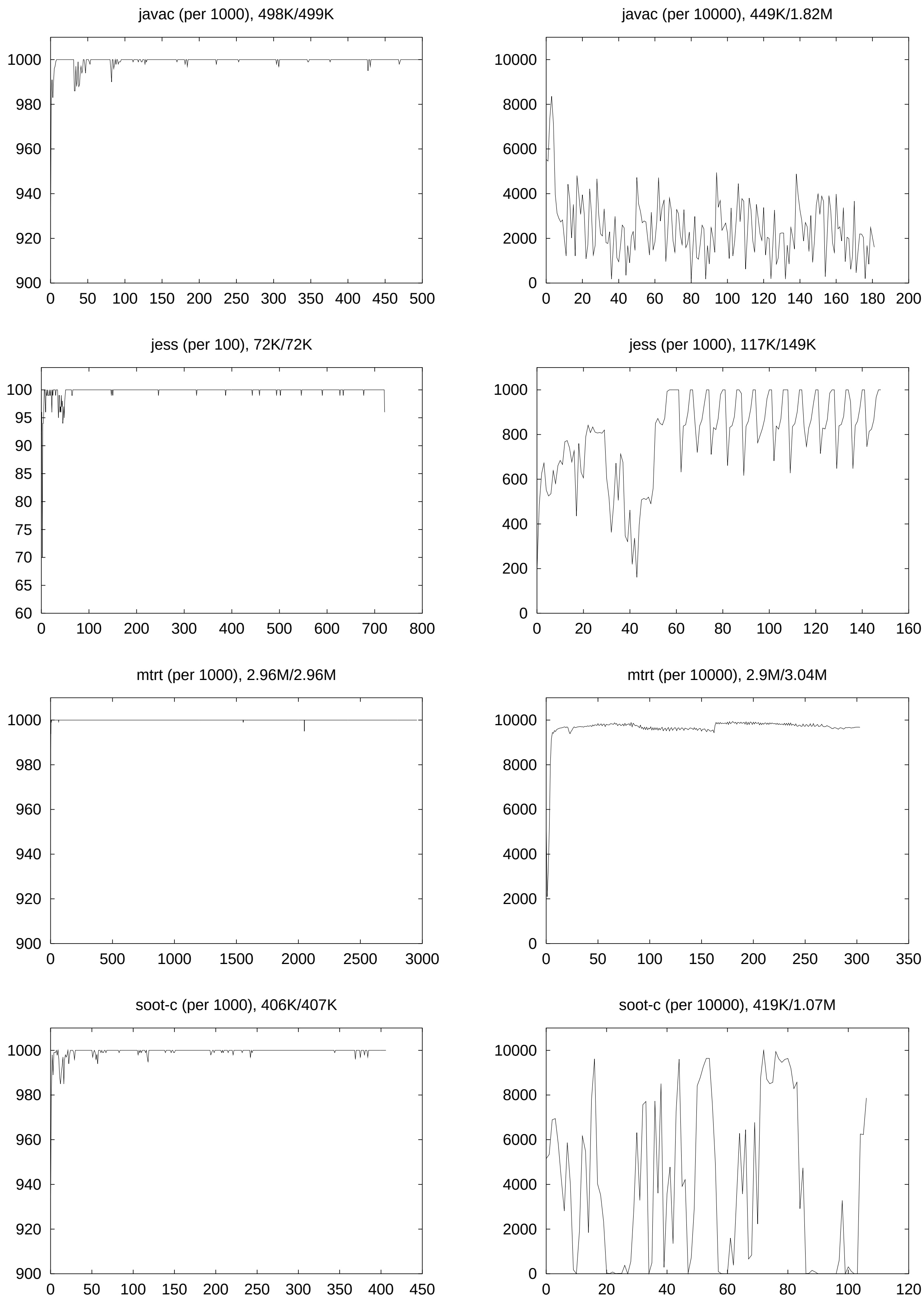
			With Adaption			No Adaption		
Benchmark	base collections	total allocated	region-reclaimed	clc	froth	region-reclaimed	clc	froth
compress	7	116M	15.39M(13.27%)	7	0.11%	14.75M(12.72%)	7	3.61%
db	4	77M	0.57M(0.74%)	4	0.23%	0.53M(0.68%)	4	5.77%
jack	9	223M	51.00M(22.87%)	8	5.97%	94.66M(42.45%)	7	16.26%
javac	12	212M	18.65M(8.80%)	15	29.41%	24.41M(11.51%)	96	589.09%
jess	12	267M	17.36M(6.50%)	11	0.53%	224.92M(84.24%)	2	9.04%
mpegaudio	0	7M	1.96M(28.00%)	0	2.10%	1.37M(19.57%)	0	128.43%
mtrt	7	143M	115.15M(80.52%)	1	0.09%	112.56M(78.71%)	6	62.42%
soot-c	15	219M	40.72M(18.59%)	13	4.89%	7.82M(3.57%)	57	276.54%

Table 6: Effect of Adaption

6.7 Summary

Our current implementation, using the baseline compiler, was aimed at producing a prototype that could be used to measure the behavior of the system, as we have presented in this section. Our numbers show that: 1) page size is important, but with the appropriate page size, the overhead for froth and the frequency of expensive searches for free pages is quite low; 2) for many benchmarks a significant percentage of allocated memory can be placed in local regions which are still clean at release time; 3) appropriate choices for the barrier operations can put the common cases on a low-cost path; 4) the overhead for sharing space with thin locks seems acceptable; 5) the adaptive part of the algorithm is important for focusing the system on the allocation sites that are likely not to escape; and 6) for many, but not all, benchmarks the adaptive system finds more local regions than a non-adaptive system.

We did try measuring runtime improvement using this prototype, but it turned out that the overheads in our current implementation are still too high, and this can lead to an overall slow down. This is due to several factors, including: 1) the cost of region management, 2) the cost of write barriers, and 3) small helper methods used by region and barrier implementation. Since many of these functions are currently written in Java, using small methods, we expect that the optimizing compiler will inline aggressively and optimize away much of the overhead. We are currently working on this implementation in order to determine if the overheads can be reduced to an acceptable level.



(a) With adaption

(b) No adaption

Figure 7: Clean regions released

7 Related work

We have described a region-based allocator using paging heaps. Although we use the terminology `region` here, the technique does not involve any region inference algorithm [14]. The technique provides an alternative way to allocate objects on stack in a JVM. There is much literature on garbage collection, region-based memory management and object stack-allocation, thus we focus on those systems most closely related to our work.

Tofte's region inference system [14, 13] automatically infers regions for objects. It achieves automatic memory management by compiler analyses. Gay and Aiken's C@ [6] and RC [7] provide language support for regions. C@ does not require an inference algorithm. It uses reference counting and stack scans to determine the safety of reclaiming a region. The main point of our work was to develop a system that works for an existing language, Java, and that is transparent to the Java programmer. Steensgaard [12] proposed thread-specific heaps for multi-threaded programs. Both systems require the heap to be organized as pages/chunks. We studied the allocation behaviours of Java programs on paging heaps. The primary results suggest that Java programs are sensitive to the page size.

Escape analyses [5, 4, 15, 8] for Java determine whether the objects created by an allocation site may escape certain scopes. Mainly the analysis results can be used in two optimizations. Thread escape analysis results can be used to remove unnecessary synchronizations, and escape analysis to find method-bounded allocation sites can be used to create objects on the local stack frames. However, the cost of the analyses prevents them from being used at run time, and Java semantics may pose restrictions on stackable objects. Our region-based allocator aims to reduce the work of garbage collector by allocating objects on temporary regions. The technique needs no analyses and may be suitable for a run time system like a Java Virtual Machine.

McDowell [11] reported the number of potentially stackable objects in a set of Java benchmarks. Like other escape analyses, McDowell also made the assumption that a compile time algorithm must make a decision for all objects created by an allocation site, although he was using dynamic profiling information to conclude the results. Our system does not require this limitation. An allocation site may create objects on local regions before any of them is found to be escaped. Extensions of our adaption algorithm may also allow allocation sites to become local again, even after being marked as non-local.

Hallenberg [9] introduced garbage collection into individual regions in Tofte's region inference system for the ML Kit. Although our collector shares close name with his system, the structures are quite different. In his system, the region inference algorithm creates regions, and inside a region, a copying collector collects live objects. The backbone of our system is a garbage collector, and the region is a natural way to extend stack frames. The region organization serves as the basis of adaptive allocation. Interested readers can find the design of Hallenberg's system in [9], chapter 11.

8 Conclusions

We have presented an adaptive, region-based allocator for Java Virtual Machines and studied the allocation behavior of Java programs with paging heaps. The main idea is to detect on-the-fly these allocation sites that do not escape their region, and then manage these allocations in local regions that can be released when the associate stack frame is popped.

We implemented the system using the Jikes RVM baseline compiler and associated garbage collector, and we used this prototype to study the behavior of a collection of Java benchmarks, including the SPEC JVM98 benchmarks. This study showed that the design of the system is crucial, including an appropriate choice of page size, and techniques for minimizing space overhead and region allocation/deallocation overhead. We also studied the adaptive mechanism of our system, and found that it quickly found regions from which no object escaped.

Given our encouraging results so far, we are currently pushing this work forward in several directions. First, we are porting the system to the optimizing compiler, where we hope that the overhead of region-management and write barriers can be reduced due to aggressive inlining and other compiler optimizations.

This will allow us to provide meaningful run-time speedups. Our quantitative measurements of the behavior of the regions and barriers indicate that this should be possible.

Our second major area of investigation is to look at a wider range of adaptive mechanisms. One could think of our 1-bit dirty mark on regions as a very coarse, and relatively naive predictor. When we turn off the adaptation, this corresponds to a predictor that always predicts that no object will escape. Our experiments show that this second method works well for those programs where, in fact, a large portion of the objects do not escape. We would now like to examine other more complex predictors. For example, we could use more than 1 bit, and only mark an allocation site as non-local after some number of objects escape. We could also increase the granularity of the predictor by associating dirty bits with pages within regions, rather than having 1 bit per region. Another possibility is to reset allocation sites to being local at intervals, for example at collection time. This may help in the case where the same allocation site is sometimes local and sometimes non-local. As part of this study we would like to measure the upper bound (i.e. what would happen with a perfect predictor), and also the effect of the regions on memory locality.

Our final area of research is to examine the effect of our system when coupled with different garbage collectors. As we pointed out in section 5, it should be relatively straightforward to incorporate our ideas in a variety of collectors.

9 Acknowledgement

We would like to thank the Jikes RVM group at IBM T.J. Watson for providing the Jikes RVM which makes our experiment possible. We particularly appreciated Stephen Fink's helpful comments. The GCTk toolkit from University of Massachusetts facilitated our implementation. We also appreciated John Jorgensen's proofreading of this report.

References

[1] B. Alpern, P. Attanasio, A. Choi, D. Fink, M. Hind, D. Hummel, V. Litvinov, T. Mergen, V. Russell, H. Serrano, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language (Third Edittion)*. Addison-Wesley, 2000.

[3] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[4] B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34, Nov 1999.

[5] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 1–19, Nov 1999.

[6] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, Montreal, Canada, June 1998.

[7] D. Gay and A. Aiken. Language Support for Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[8] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-based Programs. In *Proceedings of the International Conference on Compiler Construction (CC 2000)*, pages 82–93, 2000.

[9] N. Hallenberg. Combining Garbage Collection and Region Inference in The ML Kit, 1999. Master's thesis. Department of Computer Science, University of Copenhagen, Denmark.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[11] C. McDowell. Reducing garbage in Java. http://www.cse.ucsc.edu/research/embedded/pubs/gc/.

[12] B. Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. *ACM SIGPLAN Notices*, 36(1):18–24, January 2000.

[13] M. Tofte. A Brief Introduction to Regions. *ACM SIGPLAN Notices*, 34(3):186–195, 1999.

[14] M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[15] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 187–206, Nov 1999.

A Pseudo-code for allocating space

```
ADDRESS _getHeapSpace(int rid, int size) {
  int crtpage = _getFirstPage(rid);
  int freebytes = _getFreeBytes(crtpage);

  // 1st attempt: allocate on the first page
  if (freebytes >= size)
  {
    // fast path
    ADDRESS addr = _getAddress(crtpage);
    _setAddress(crtpage, addr+size);
    _setFreeBytes(crtpage, freebytes-size);
    return addr;
  }

  // slow path
  return _expandRegionSize(rid, size);
}

ADDRESS _expandRegionSize(int rid, int size) {

  // 2nd attempt: check following pages
  int firstpage = _getFirstPage(rid);
  int leftover = _getFreeBytes(firstpage);
  int pages = _computePages(size-leftover);
  if (checkFreePagesFrom(firstpage+1, pages))
  {
    ADDRESS addr = _getAddress(firstpage);
    /* adjust page descriptors and
     * region descriptors.
     */
    ......
    return addr;
  }

  // 3rd attempt: search contiguous free pages
  firstfree = _getFirstPage(FREELIST);
  pages = _computePages(size);
  if (searchFreePagesFrom(firstfree, pages))
  {
    ADDRESS addr = _getAddress(firstfree);
    /* adjust page descriptors and
     * region descriptors.
     */
    ......
    return addr;
  }

  return OUT_OF_MEMORY;
}
```