Dynamic adaptive pre-tenuring

Timothy L Harris
Sun Microsystems Laboratories
One Network Drive
Burlington, MA 01803-0902
tim.harris@acm.org

ABSTRACT

In a generational garbage collector, a *pre-tenured* object is one that is allocated directly in the old generation. Pre-tenuring long-lived objects reduces the number of times that they are scanned or copied during garbage collection. Previous work has investigated pre-tenuring based on off-line analysis of execution traces. This paper builds on that work by presenting a *dynamic* technique in which the decision to pre-tenure a particular kind of object is taken at run-time. This allows decisions to depend on the inputs of a particular application run and also allows decisions to be changed as the application enters different phases. An implementation is presented for the ResearchVM Java Virtual Machine.

1. INTRODUCTION

Generational collection is a popular technique which aims to improve the performance of a garbage collected heap [3]. Objects within the heap are divided into two or more *generations* according to the elapsed time since their allocation[1]. In the simplest case there are only two generations termed *young* and *old*. The young generation is subject to more frequent garbage collection than the old generation. Objects are allocated into the young generation and subsequently *tenured* into the old generation if they survive longer than some threshold.

Such a scheme is effective because many objects are short-lived and so it is worthwhile concentrating on the young objects (which are likely to die) rather than on the older objects (which are likely to continue to survive). The fact that many objects die young makes it common to use a copying collector for the young generation: work is only performed copying the few objects that survive. Each generation is typically held in a separate part of the heap. This allows them to be managed by different garbage collectors and by different allocation policies. The physical separation means that an object must be copied when it is tenured. Consequently, long-lived objects are often copied several times before they come to rest in the old generation. Pre-tenured allocation avoids some of this copying.

There are other reasons why pre-tenuring may be desirable. Firstly, the decreased volume of allocation in the young generation may reduce the number of times that that region must be collected. Secondly, if non-copying collectors are used, then pre-tenuring may reduce fragmentation in the young generation. Thirdly, reducing the number of different locations occupied by long-lived objects may improve temporal locality of reference. Finally, in the current implementation, it is more efficient to allocate an object directly into the old generation rather than to copy it there. This is because the allocation function may be specialized according to the size of the object, allowing a useful optimization when manipulating the data structure for tracking inter-generational references.

This paper describes a technique which aims to improve the handling of long-lived objects in a two-generation system. In particular, it aims to predict when an object will be long-lived and to pre-tenure such objects by allocating them directly into the old generation. Unlike previous work, described in Section 2, we decide when to pre-tenure objects based on dynamic feedback using statistics gathered during the current run of an application. This avoids the need for a separate profile-gathering phase. It also allows pre-tenuring decisions to be changed during execution if the application moves between phases exhibiting different behavior. Section 3 illustrates the extent to which different statistics, available at object allocation time, may serve as predictors of the object becoming long-lived.

The implementation comprises two modules. The first is concerned with *object sampling*. It gathers lifetime statistics from a subset of the objects allocated. The second processes these statistics to select which kinds of object should be pre-tenured. It makes these pre-tenuring decisions by patching the code generated by a just-in-time compiler. These two modules are described in Sections 4 and 5 respectively.

Section 6 presents results from a number of widely-used benchmark applications. Finally, Section 7 describes future work and conclusions.

[1] As is customary, all references to *time*, *ages*, *older* or *younger* are in terms of an *allocation clock* (measuring the total volume of objects allocated since the program started) rather than a *wall clock* (measuring elapsed real time)

This paper makes two main contributions. Firstly, it uses dynamic feedback within a single run of an application. Doing so places a new emphasis on efficiently gathering statistics on object lifetimes. Secondly, it allows pre-tenuring decisions to be reversed and subsequently re-enabled as a program executes. As shown in Section 4, this uses a novel technique for identifying when an object allocated in the old generation is short lived.

2. RELATED WORK

Previous work has investigated feedback-based techniques for segregating long-lived and short-lived objects.

Barrett and Zorn attempt to predict *short-lived* objects in a number of allocation-intensive applications written in C [1]. They use profile-driven full-run feedback based on observed object lifetimes. Their motivation is to reduce the fragmentation caused by long-lived objects scattered throughout the heap. They are also able to reduce the cost of allocating short-lived objects by placing them contiguously and delaying deallocation until entire 4k batches become free.

They attempt to correlate short object lifetimes with the most recent n return addresses on the execution stack. They found that there is typically an abrupt step in the effectiveness of prediction when n reaches some critical value. These critical values varied between applications, but were usually not greater than 4.

The effect of using these predictions was evaluated through simulation using allocation traces. Each entry in the traces contained an identifier representing the object size and the complete call-chain to the allocation site. They estimate that the cost of computing a reasonable approximation to such an identifier may be between 9 and 94 RISC-style instructions for each memory allocation made. Such an overhead is, perhaps, reasonable for a free-list based allocator from the `libc` library. However, the fast-path of the existing allocator in the ResearchVM uses only 9 SPARC instructions and so even the best-case overhead of a further 9 instructions appears unsatisfactory.

Seidl and Zorn propose dividing the heap into a number of sections based on reference behavior and object lifetime [6]. They identify four kinds of object: *highly referenced* objects that are accessed frequently, *non-highly referenced* objects that are accessed infrequently, *short-lived* objects that are de-allocated soon after they are created and *other* objects which form the remainder of the heap. These divisions are designed to improve the program's usage of virtual memory pages. Their intuition is as follows: highly referenced objects should be densely packed together so that the pages they occupy will form part of the working set of the program, non-highly referenced objects should also be held with one another but segregated from other kinds of object, in the hope that the pages they occupy will not form part of the working set. Short-lived objects should also be held separately from the rest of the heap in order to avoid fragmentation in the remainder of the heap.

Seidl and Zorn's work used trace-driven full-run feedback to gather statistics about a number of large C applications, including AWK, PostScript and Perl interpreters. They identified two effective techniques for predicting, at allocation time, into which category an object should be placed:

- The *path point predictor* assumes that there is a high correlation between certain call sites in a program and the behavior of objects allocated in procedures 'below' these sites in the dynamic call graph. The intuition is that there are certain significant points at which the program changes between generating different kinds of object.

- The *stack contents predictor* uses a subset of the call chain at the time of allocation as a predictor of object behavior. For example it may consider the most recent n return addresses, for small values of n such as 3. In previous work the authors showed that this was effective for programs written in C++ because a few stack frames were sufficient to disambiguate allocations occurring in common functions, such as object constructors, or `malloc`-wrappers, that are invoked throughout the application [5].

Cheng, Harper and Lee describe profile-based pre-tenuring in the TIL compiler for Standard ML [2]. They identify allocation sites by their program counter – this is perhaps more effective in the context of ML rather than C because it is not customary for allocations to be made through layers of wrapper functions.

Cheng *et al* do not comment on whether the effectiveness of pre-tenuring is influenced by the usage patterns of heap-allocated data in functional languages (illustrated, for example, in Stefanović and Moss' analysis based on SML/NJ [7]). The major differences observed are a higher rate of allocating data records and a reduced rate of updates to existing data. However, it is unclear whether these language-level differences in the manipulation of data structures will be reflected in the native code generated by an optimizing compiler.

3. STATISTICS

The related work shows how information available at the time of an object's allocation can serve to predict its lifetime. However we believe that this paper is the first to consider profile-driven allocation in the context of an object-oriented language with automatic storage management. This section analyses the behavior of a number of popular benchmarks in order to confirm that typical programs executed over the Java Virtual Machine (JVM) [4] exhibit similar behavior to those studied in other systems. We will start by presenting a number of reasons why object allocation may exhibit different characteristics in this environment[2].

Suppose that `MethodA` in `ClassA` wishes to create a new instance of `ClassB`. This is typically achieved using the following bytecode sequence:

[2]Activation records are allocated automatically as part of each method invocation. They are not stored in the garbage collected heap and their management is not discussed further in this paper.

```
0: new <Class ClassB>
3: dup
4: invokespecial <Method ClassB.<init>()V>
```

There are two important steps to this process. The first, implemented by the `new` bytecode, is to allocate storage space in which to hold the new instance of `ClassB`. The second, implemented by the `invokespecial` bytecode, is to call a constructor of the desired class on the newly created space. In C-like terminology, the `new` instruction is the equivalent of a call to `malloc` while the `invokespecial` instruction corresponds to a call to some initialization function. Any particular `new` instruction allocates instances of only one particular class. Further bytecodes, `newarray`, `anewarray` and `multianewarray` are provided for instantiating arrays.

The separation of these two steps is significant to the present discussion because it means that the allocation occurs in the method creating the instance rather than in the implementation of the constructor. An observation in much of the related work is that multiple stack frames of allocation context may be required to reach the method, in this case `ClassA.MethodA`, that is 'really' making the allocation. This is because it is common, in C, for programmers to employ wrappers around the `malloc` function. This style of programming cannot be implemented cleanly over the JVM because it would require each use of the `new` bytecode to be changed into a method invocation. In practice, however, the common uses of `malloc`-wrappers are subsumed by the strong typing provided by the JVM and by the use of run-time exceptions for indicating out-of-memory conditions. These reasons suggest that fewer frames of allocation context may be required in the case of the JVM. Cheng *et al* similarly observed that a single program counter value was effective at separated allocation sites generating long-lived and short-lived objects in the TIL compiler for Standard ML [2].

Of course, there are programs which intuitively appear to benefit from more information than is provided at a single static allocation site. For example, suppose that each constructor of `ClassB` allocates an instance of `ClassC` and stores the only reference to this in a private field. If the lifetime of an instance of `ClassB` may be accurately predicted with a single stack frame of context then the lifetime of an instance of `ClassC` would be predictable using two stack frames of context.

As a concrete example, consider an implementation of `java.util.Hashtable`. Each instance of `Hashtable` contains a reference to an array of references to `java.util.Hashtable.Entry`. Each instance of `Entry` represents one of the key-value pairs that is stored in the hashtable. The array of references to entries is allocated in the constructor of `Hashtable`. The entries themselves are initially allocated when new key-value pairs are stored in the `Hashtable.put` method. If the number of entries reaches a threshold size then a second method, `Hashtable.rehash` is used to relocate the entries to a new array. The array of references to entries may therefore grow many times during the lifetime of the table. Three stack frames of context would be required to associate each re-allocation with the method that originally invoked `Hashtable.put`.

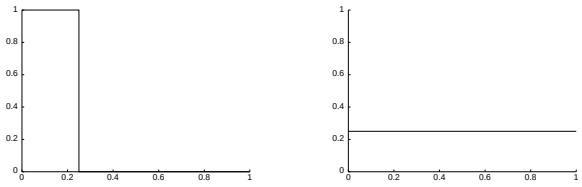

Figure 1: Best-case (left) and worst-case (right) mappings of objects to categories. In each case a total of 25% of objects were tenured, as shown by the fact that the area under each curve is 0.25. The best-case mapping separates objects into two kinds of categories: those containing only tenured objects and those containing only non-tenured objects. The worst-case creates a single kind of category, containing a mix of tenured and non-tenured objects.

3.1 Object lifetime in benchmark applications

In order to study the effectiveness of different levels of contextual information we instrumented the ResearchVM to record object allocation and lifetime. An object's lifetime was considered to extend until the space it occupies was reclaimed by the garbage collector. For each object allocated by the application under test, the VM logged the class of the object, the last 5 methods at the allocation site, each time the object was re-located during execution, the eventual time at which the object was de-allocated and if, at that time, the object had been promoted to the old generation.

These statistics were used to evaluate the effectiveness of various prediction techniques. Each candidate predictor is considered as a mapping from the allocation trace information onto a number of *categories*. An optimal predictor would divide objects into two categories: the first category containing only tenured objects and the second category containing only non-tenured objects. Conversely, the worst-case predictor would use only a single large category, containing all of the objects that have been allocated.

Figure 1 illustrates the format in which we shall present the results gathered from these traces. In these plots the categories are placed, from left to right, in decreasing order of the proportion of tenured objects that they contain. For example, the categories at the extreme left will contain only tenured objects while the categories at the extreme right will contain only non-tenured objects. The x-axis indicates the proportion of all objects which occur in categories up to that point. The y-axis indicates the proportion of tenured objects in each category. Note that the area under such a curve is constant for any application trace: it represents the total proportion of all objects that have been tenured.

Figures 2 and 3 show the results for the `javac` and `ellisgc` benchmarks.

The `javac` benchmark allocates 6.2×10^6 objects, of which 28% become tenured. Class alone seems to be a reasonably good predictor. 25% of the objects allocated during the experiment belong to classes of which at least 85% of the instances became tenured. Notable among these were the subclasses of `_213_javac.Node` used to represent the

AST in the compiler. A single frame of allocation context distinguishes most tenured and non-tenured instances of `java.util.Hashtable` and `java.util.Vector`. Two frames distinguish most of the arrays-of-object used to contain the elements in a `Vector` (allocated in the constructor) and three frames distinguish the expanded arrays allocated upon overflow. There is little additional benefit from considering more frames.

The `ellisgc` program is a synthetic benchmark that performs extensive memory allocation. It operates in three phases. The first phase allocates a large binary tree in order to 'stretch' the heap and avoid subsequent changes to its size. This structure becomes unreachable at the end of the first phase. The second phase allocates a further tree which remains reachable throughout the benchmark. The third phase allocates a number of smaller short-lived binary trees which become unreachable as soon as each is built.

Although this behavior seems quite different from that of a reasonable application, the results shown in Figure 3 do illustrate an important point. The simulated *optimal* predictor shows that around 65% of objects become tenured. However, none of the predictors using 1-5 frames of stack context is able to provide significantly useful results. The reason for this is that the binary trees are allocated recursively and so, aside from trivially small sizes, the top 5 frames are all recursive invocations of the binary-tree-node constructor.

Barrett and Zorn's *call-chain* predictor trims the run-time stack by removing cycles of recursive function invocations [1]. Such a predictor would be effective for `ellisgc` because 2 frames of context from such a trimmed stack would separate the method invocations used to trigger the three benchmark phases. One of the possible implementation techniques that they present is to update a global 'current allocation site' identifier by XORing it with a per-function identifier on each function call or return. These identifiers may be assigned by static call-graph analysis. As described in Section 2, even the best-case overhead of this technique appears to render it unsuitable for use in the ResearchVM. However, it is interesting to note that the XORing mechanism has the effect of generating only two different allocation site identifiers during deeply recursive invocations. The paper is not clear whether this is deliberate, or whether the primary motivation for using XOR is to simplify recovery of the previous allocation site identifier when a function returns.

Other benchmarks, such as `compress` from the SPEC JVM98 suite, exhibit low proportions of tenured objects – typically `String`s and arrays-of-char allocated from sites during initialization of the JVM and of the test harness.

3.2 Information to use at run-time

The results presented in the previous section illustrate that although an object's class is a reasonable predictor of its lifetime, the most recent few methods at its allocation site improve the accuracy of the predictions made. In our implementation, presented in Sections 4 and 5, we chose to use a single frame as contextual information. That is, we categorize objects on the basis of their class and of the program counter at the place at which the `new` bytecode instruction occurs.

This decision is motivated by implementation efficiency in general and by the desire to avoid any overhead on fast-path non-pre-tenured allocations. This clearly precludes updating any form of allocation site identifier on every method invocation. However, less clearly, it makes it impracticable to use even two frames of allocation context. This is because the method containing the `new` instruction would have to determine its caller at allocation time and then distinguish callers which should trigger pre-tenured allocation and callers which should not. In contrast, if decisions are based on a single frame of allocation context, then a pre-tenured allocation may be implemented simply by changing the way in which a particular occurrence of the `new` bytecode is compiled.

However, note that method inlining performed at run-time by the native-code generator may allow the single program counter value to reflect a number of 'virtual' stack frames. In particular the semantics of the `invokespecial` bytecode often enable nested constructors to be inlined.

4. SAMPLING OBJECTS

This section describes the technique used for sampling objects at run-time in order to estimate their lifetimes. The aim is to determine, for a hopefully representative subset of objects, which allocation sites produce objects that are predominantly tenured. We require some mechanism for associating an allocation site with an object while it remains alive and then detecting, when an object becomes unreachable, whether it was tenured or whether it remained in the young generation.

A two-level approach is taken in the ResearchVM to allocating normal objects the young generation copying collector. Most allocations are satisfied by sequentially placing objects within local allocation buffers (LABs)[3]. Each thread has a separate LAB, so an allocation is essentially implemented by incrementing a thread-local pointer. The complete code, including checks for LAB overflow, is only 9 SPARC assembly language instructions. Specialized allocation functions are used for small object sizes. If an allocation fails because it would cause the current LAB to overflow then a new LAB is obtained from the second level allocator. The implementation of the second-level allocator is similar, except that LABs are allocated from a shared stretch of memory using an atomic CAS operation. A young generation garbage collection occurs whenever the space used by the second-level allocator is exhausted.

4.1 Selecting objects on LAB overflow

Object sampling is implemented as part of the LAB overflow handler.

Although this avoids any change to the allocation fast-path it is not clear that it produces a representative selection of objects to sample. This is because, in general, larger objects are more likely to cause a LAB overflow than smaller objects. This skew is quite apparent when looking at bench-

[3]Objects created through reflection or through the `clone` method are allocated in the young generation using separate functions. Very large allocations occur directly in the old generation.

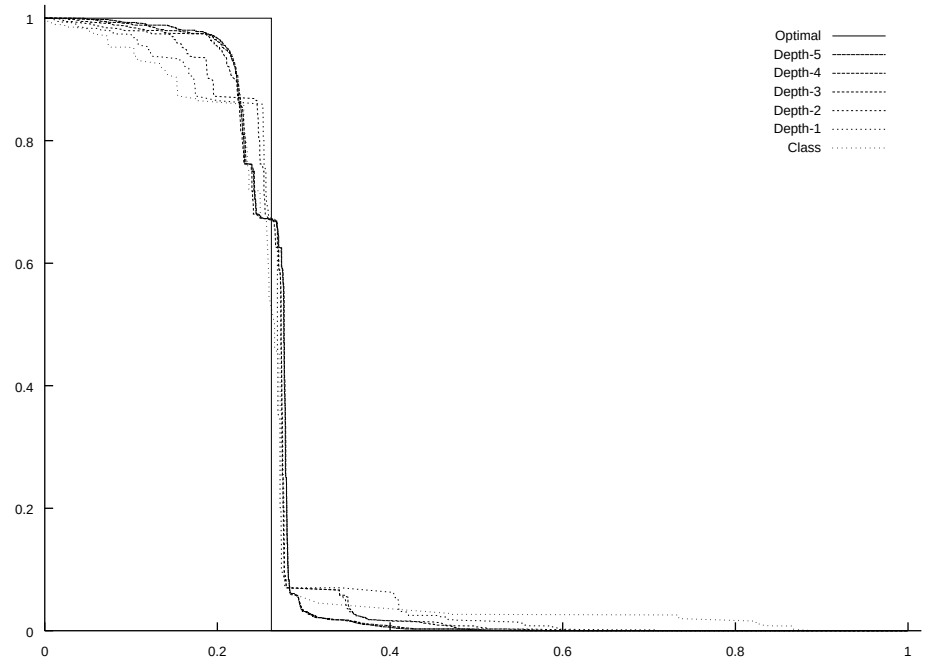

Figure 2: `javac` **source-to-bytecode compiler.**

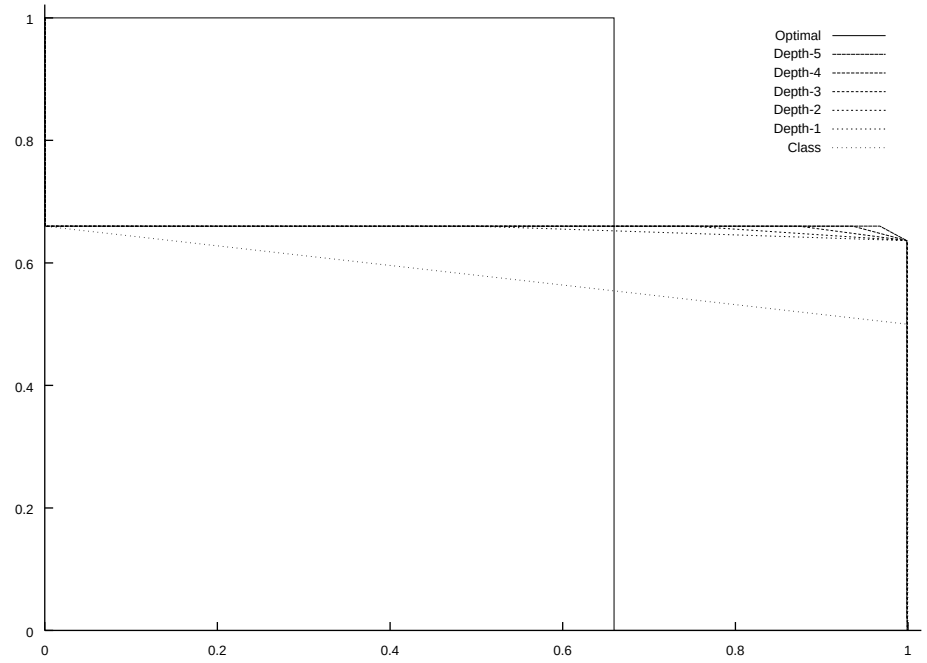

Figure 3: `ellisgc` **synthetic benchmark with a high allocation rate.**

mark statistics. For example in `javac` 26% of all objects allocated are arrays-of-char, but they account for 40% of the allocations selected by LAB overflows.

A number of alternative schemes were considered to reduce this skew. The LAB overflow handler could cause the sampling of the next object to be allocated. This could be implemented by leaving the allocation pointer 'broken' so as to cause the LAB overflow handler to be re-entered on the next allocation. However, in the case of `javac`, this changes the skew rather than avoids it. This is because the arrays-of-char were typically allocated while manipulating text and the subsequent allocation is frequently an instance of `java.lang.String`. More generally, the first-level allocator could select the request n after the genuine overflow. This qualitatively reduces the skew, but it is less clear whether that benefit is balanced by the need to make $n + 1$ non-fast-path allocations.

However, after comparing each of these candidate schemes using full-run allocation traces, it appears that the skew introduced is not an important problem. Although it leads to certain kinds of object being sampled more frequently, it does not distort the proportion of tenured objects which occur in each category. The simple scheme, sampling each object allocated from the LAB overflow handler, was therefore implemented.

4.2 Sampling objects in the young generation

As mentioned in the introduction to this section, some mechanism is needed to allow:

- a sampled object's allocation site to be associated with that object during its lifetime,
- this information to be accessed when the object becomes tenured or when it is reclaimed by the garbage collector.

The approach taken allows sampled objects to be held in the same run-time format as ordinary objects and avoids *any* overhead when tenuring or reclaiming non-sampled objects. The technique, termed *object fingerprinting*, is more generally applicable – it is deployed here to store allocation-time information for particular objects.

An auxiliary data structure, separate from the main garbage-collected heap, contains per-instance information for each object being sampled. This information consists of two fields – the memory address of the object's allocation site and a *weak root* that refers to the object. The garbage collector makes a series of callbacks after each collection cycle, at which point the referents of the weak roots may be examined in order to determine whether the sampled objects remain in the young generation, whether they have been tenured or whether they have become unreachable.

As with the weak reference objects (`java.lang.ref.WeakReference`) available from the Java programming language, the existence of a weak root referring to a particular object is not considered by the garbage collector when establishing which objects are reachable. Weak roots for object sampling are conceptually *weaker* than any of the kinds of reference that a user may create within their program, or that that ResearchVM employs internally for other reasons.

Figure 4 shows how the data structure is organized. The per-sampled-object information is held in *chunks*. Each chunk contains an *age* field, an *occupancy* field and a number of *slots*. Each slot is either empty or contains information about a sampled object. Each thread is associated with a *current chunk* and, when allocating a sampled object, it records the allocation site and new object reference in the *next slot*. The chunks are held in a linked list.

The *occupancy* field logically contains a count of the number of non-empty slots. However, in order to avoid updating it whenever a sampled object is allocated, the value actually recorded in a *current chunk* is offset by the number of empty slots beyond the *next slot*. The *age* field contains the time at which the chunk became a *current chunk*. The use of this value will be described below in Section 4.3.

The data structure is traversed as part of each garbage collection. Each sampled object is examined. If the object has been retained by the copying young generation collector then the weak root is updated to refer to the new location of the object. If the object has been marked unreachable by the collector then the weak root is cleared, the *occupancy* of the chunk is decremented and the count of non-tenured objects for that category is incremented. If the object has been promoted by the young generation collector then the weak root is also cleared, the *occupancy* of the chunk is decremented and the count of tenured objects for that category is incremented. A chunk is removed from the list of active chunks when its occupancy falls below a threshold level.

Per-category information is held in a hash table indexed by a combination of the object's class and the allocation site program counter. Section 5 will describe how this information is used to reach pre-tenuring decisions.

4.3 Sampling objects in the old generation

The scheme described in the previous section gathers statistics about the proportion of tenured objects within the young generation. This information is insufficient for implementing an effective dynamic pre-tenuring system.

For example, the first phase of the `ellisgc` benchmark allocates a large binary tree in order to 'stretch' the heap to an appropriate size for the remainder of the benchmark. Many of the objects allocated during that phase will become tenured, particularly if the size of the young generation is initially small. The same allocation sites are used during the final phase when allocating short-lived data structures. It would therefore be unfortunate if an irreversible decision to pre-tenure was taken on the basis of the first phase.

This problem occurs to a lesser extent in systems based on full-run feedback because they would consider allocation from all three phases together. The benchmark allocates a substantial volume of objects during the third phase and these would 'dilute' the tenured objects allocated during the first two phases. Allowing pre-tenuring decisions to be reversed may turn this problem into an advantage: objects are

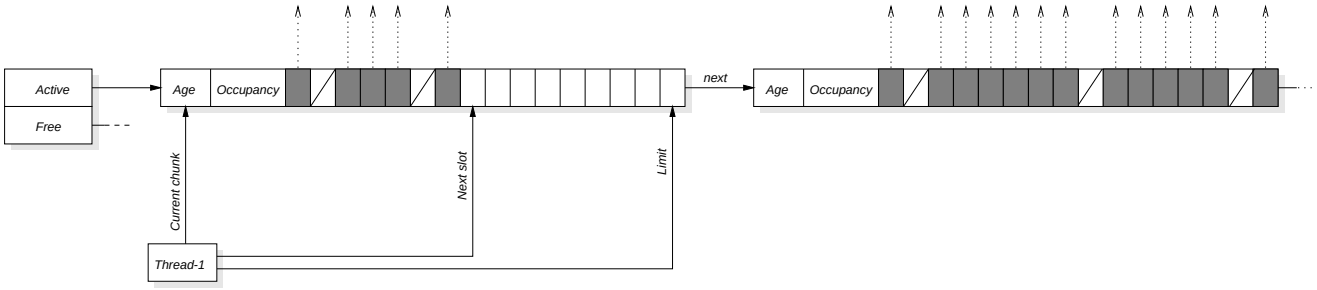

Figure 4: Run-time data structures used for object sampling. The *Active* and *Free* fields, on the far left, are global variables which identify the heads of linked lists.

implicitly categorized according to allocation time as well as allocation site.

Suppose however that the scheme described in Section 4.2 were to be applied to pre-tenured objects. When an object has already been pre-tenured, there is no clear analogue for promotion *from* the old generation. Objects could be classified as short-lived or long-lived depending on whether they have survived a single old generation collection.

However, that is unsatisfactory because old generation collection cycles are infrequent – even an object that dies before the end of its first old generation collection may have survived for comparable young-generation-allocated objects to have been tenured. Furthermore, the long intervals between old generation collections could mean that a large number of chunks are required to track all of the sampled objects.

One tempting approach is to allocate sampled pre-tenured objects into the young generation. This would make them subject to frequent garbage collection and would allow them to be handled in the same way as regular sampled objects. However, consider again the binary trees allocated during the `ellisgc` benchmark. The tree structure means that all nodes other than the root are referenced from a parent in the tree. Whenever a pre-tenured node is sampled then although that node would be allocated in the young generation, its parent would generally be allocated in the old generation. This old-to-young reference would exist until the parent is reclaimed. As before, the infrequency of old generation collections would preserve the sampled objects.

Instead, the implemented approach samples objects directly in the old generation but exercises caution over which samples are considered representative. The system maintains a current *tenuring age*, that is, an estimate of the time between the allocation of an object and its subsequent tenuring. Sampled pre-tenured objects are held in a separate list of chunks to regular sampled objects. The pre-tenured chunks are examined after every collection. Any dead objects in chunks younger than the tenuring age are deemed to have been short-lived. Any live objects in chunks older than the tenuring age are deemed to have been long-lived.

The tenuring age is updated after every young generation collection. It is calculated as a rolling average of the ages of chunks containing objects that have just been tenured.

5. PRE-TENURING

The previous section described how we gather statistics about the proportion of short-lived and long-lived objects within each category. This section describes how these are used to decide when to pre-tenure a particular category of object and how these decisions are implemented.

In common with the work of Cheng *et al* we use a simple threshold-based scheme to trigger pre-tenured allocation. This is controlled by two parameters. The first specifies a minimum number of sampled objects within a category and the second specifies a proportion of these objects which must have been long lived.

The minimum object count is used to avoid basing a decision to pre-tenure object allocation on atypical behavior during initialization or before many sampled objects have been observed. The per-category counts of long-lived and short-lived objects are halved whenever more than twice the minimum number of sampled objects have been observed. This avoids special handling of arithmetic overflow and aids the detection of application phase changes.

Separate thresholds are used to decide when to reverse a decision to pre-tenure a category of object. By default these are equal to the parameters to trigger pre-tenuring. However, it may be the case that some applications benefit from setting different thresholds – for example to prevent unwanted oscillation between regular and pre-tenured allocation.

In general, binary patching is used to implement decisions to start or to stop pre-tenuring a particular category of object. Each category is identified by the memory address of the invocation of an allocation function. The style of allocation used may be modified by changing the target of this invocation. Such updates occur when mutator threads are already suspended for garbage collection (although, on many systems, an update could be performed by a single atomic write instruction).

We do not attempt to pre-tenure objects allocated from interpreted code. This is because the ResearchVM executes most code using a fast non-optimizing compiler and any significant allocation site will be compiled. Pre-tenuring decisions are recorded in per-method structures for use if a method is re-compiled.

		Before	After
Young generation GC	Total time/ms	3486	3177
	Count	255	229
	Mean duration/ms	13.7	13.9
Old generation GC	Total time/ms	2072	2058
	Count	10	10
	Mean duration/ms	207	206
Total GC	Total time/ms	5560	5236
	Count	265	239
	Mean duration/ms	21.0	21.9
Heap size	Total/Mb	17	17
	Free at last GC	73%	76%
Run-time	Total/s	30.39	30.25

Figure 5: The performance of the `javac` benchmark with dynamic pre-tenuring disabled (left) and enabled (right).

6. RESULTS

It is difficult to provide a fair comparison between the performance of the VM when dynamic pre-tenuring is enabled and the performance when it is disabled.

As described in the introduction, pre-tenured allocation may reduce the number of times that objects are relocated. However, this goal can be trivially achieved by increasing the size of the heap and thereby reducing the frequency of garbage collections. Other metrics may also provide a false impression of the effect of the technique. For example, although effective pre-tenured allocation may be expected to reduce the number of young generation collections, naïve overly-enthusiastic pre-tenuring would have a similar effect. It is therefore necessary to examine a set of metrics to determine the overall impact on application performance.

Figure 5 illustrates the effect that dynamic pre-tenuring has on the performance of the `javac` benchmark. The benchmark ran using a 1Mb young generation and 16Mb old generation. The size of both generations was fixed so that the total memory footprint of the VM was the same in each case. The young generation was managed by a copying collector using separate semispaces. The old generation was managed using a mark-compact collector.

Pre-tenured allocation was enabled for categories containing at least 95% long-lived objects and at least 100 sampled objects. The same threshold was set for reverting to young generation allocation, although no such decisions were made during the `javac` benchmark.

Dynamic pre-tenuring has a marked effect on young generation garbage collection. The number of collections is reduced by 11%. The total time spent in young generation collection is reduced by 9%. There is a slight improvement but, more importantly, no degradation in the number of old generation collections or the total time that they take. There is a 6% reduction in the time spent in garbage collection. Garbage collection as a whole accounts for 17% of the benchmark execution time and so the reduction translates to a slight overall improvement of 0.5%.

Figure 6 shows a summary of the results for a number of the benchmarks. The list omits benchmarks during which fewer than 10 garbage collections occurred. As before the JVM was configured to use a 1Mb young generation and a 16Mb old generation. The size of the young generation was fixed. The size of the old generation was allowed to expand up to 64Mb. However, the heuristics used to control heap expansion did not choose to enlarge the heap.

The four charts illustrate, clockwise from top-left, how dynamic pre-tenuring affects the reported result of the benchmark, the garbage collection time, old generation collection time and young generation collection time. Three bars are plotted for each benchmark on each chart. These correspond, from left to right, to the original performance, the performance *without* application phase detection and the performance *with* application phase detection. The bars extend between the minimum and maximum values obtained from five invocations of each test. The point marked is the arithmetic mean of the five values. These have been normalized so that the original performance is 1. In each case higher values indicate worse performance.

The `compress`, `jess`, `cst`, `db`, `anagram` and `javac` benchmarks all exhibit a clear reduction in overall execution time when using dynamic pre-tenuring. The performance of the `raytrace` and `gcbench` tests are clearly degraded.

The behavior of `gcbench` is particularly interesting. It is a shorter-running variant of the `ellisgc` benchmark and exhibits the same three-phase operation described in Section 3.1. Without phase detection, the total time spent in young generation collection is reduced to 7% of the original value. This is because the behavior observed during the first phase of the benchmark causes *all* subsequent allocations to be made into the old generation – a decision which causes the total time spent in old generation collection to increase by a factor of over 100 and the total execution time by a factor of 2.9. Phase detection ameliorates this in the `gcbench` test by reverting to young generation allocation: resulting in an overall degradation by a factor of between 1.18 and 1.25.

During the `javac` benchmark, $6.2 * 10^6$ objects were allocated. 7.7×10^4 were sampled. During the total run-time of 30.4s, 57ms was spent examining the chunks containing sampled objects and 3ms was spent examining the per-category information to select when to change object allocation strategy. Twenty five allocation sites were selected for pre-tenured allocation. These were within the subclasses of `ConstantPoolData`, subclasses of `Node`, or arrays of such classes. The pre-tenured allocation sites occurred within the methods of Parser and `ConstantPool`. These 25 allocation sites generated 1.0×10^6 pre-tenured objects, comprising 16% of those allocated overall. A total of 41 chunk structures were allocated, occupying 22k of memory at run time. The per-category information occupied a further 40k.

7. CONCLUSION

The results shown in this paper indicate that, for many benchmarks, it is practicable and worthwhile to employ dynamic pre-tenuring. We showed in Section 3 that it is possible to predict the lifetime of many objects at the point

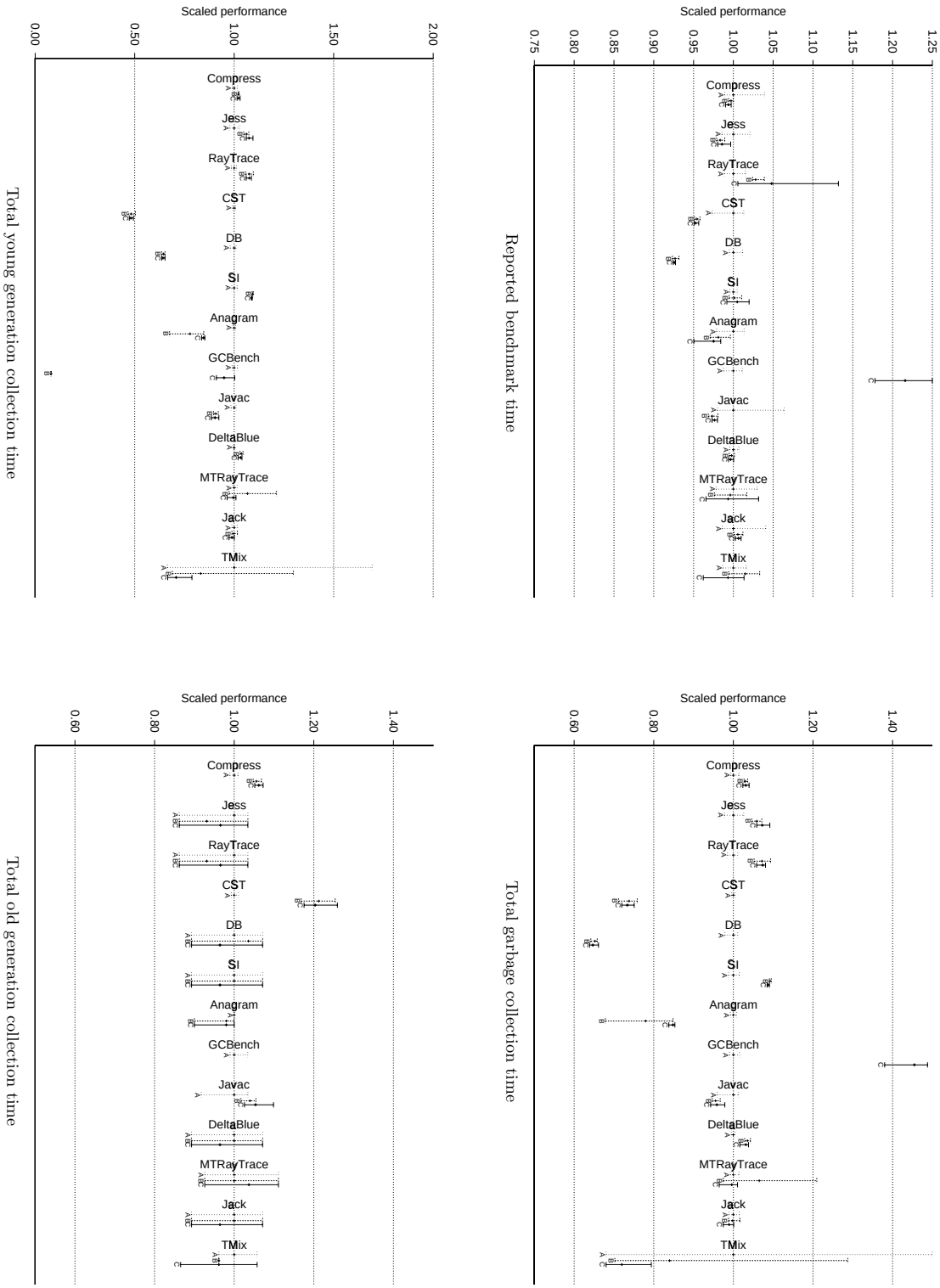

Figure 6: Threshold=95% (100 sampled objects). Bars correspond to normalized original performance *(A)* , dynamic pre-tenuring without phase detection *(B)* and dynamic pre-tenuring with phase detection *(C)* . As explained in Section 6, the results without phase detection are not plotted for the GCBench test.

of their allocation. The techniques presented in Sections 4 and 5 allow such predictions to be made with low overhead at run-time, in terms of both time and space. The results of the previous section show that an overall reduction in execution time can be achieved, even when deploying the techniques in a garbage collection system that has already been subject to extensive optimization. Using application phase detection reduces the worst-case performance for benchmarks where dynamic pre-tenuring is not effective. It is additionally worth noting that the total run times of many of the benchmarks are less than 30 seconds: these short run times provide a particularly harsh environment in which to obtain overall improvements using dynamic feedback.

Widely reported server-side Java benchmarks, such as the *Volano* test application, are designed to stress I/O and multithreading performance within the JVM rather than storage management. It will be interesting to re-evaluate dynamic pre-tenuring when more suitable benchmarks are available and, in particular, to quantify the extent to which phase detection enables opportunities for optimized allocation that cannot be realized through off-line analysis.

There are a number of additional possibilities for future work. A first option is to investigate a system with more than two generations. This would require a richer set of statistics to be gathered because objects could no long be classified as simply tenured or non-tenured. It may be possible to select an allocation generation based on both the expected lifetime and on the variance between the sampled objects. A second promising possibility is to base pre-tenuring decisions on *coarser* information than the object class and allocation site. Observe, from the `javac` benchmark, that the classes which become pre-tenured are closely related in the inheritance hierarchy. It may be possible to identify a common supertype (possibly an abstract class or an interface) and to share sampling information between all of its subtypes.

A further alternative would be to base allocation-time decisions on a combination of the class being instantiated, a single frame of allocation context and also on the allocation site of the `this` reference at its point of allocation. The intuition is that considering `this` would allow properties to depend on objects in addition to allocation sites. For example the `put` method on `hashtable` could allocate buckets differently when invoked on a predicted-long-lived hashtable from when invoked on a predicted-short-lived table. A straightforward implementation of this behavior may be possible by providing multiple virtual method tables per class and, for a particular instance, selecting between these at allocation time based on the predicted behavior of that instance.

However, it is less clear whether the information needed to trigger such decisions could be gathered effectively in a system based on dynamic feedback. This is because, in optimized native code generated from Java bytecode, the `this` reference may not be in a well-known location – in some methods it may even have been removed as a dead value. It would, perhaps, be more appropriate to implement such a scheme using full-run feedback in which the profile-gathering phase could operate with relevant optimizations disabled.

8. ACKNOWLEDGMENTS

The work described in this paper was carried out during an internship with the Java Technology Research Group at Sun Labs. The idea of tracking objects through weak roots was suggested by Ole Agesen and Alex Garthwaite. In addition to Ole and Alex I am indebted to Dave Detlefs, Christine Flood, Steve Heller, Nir Shavit and Guy Steele for making my time at Sun so enjoyable and worthwhile. For their help in the preparation of this paper I'd also like to thank Steven Hand, Richard Mortier and Ian Pratt from the University of Cambridge Computer Laboratory and, of course, the anonymous reviewers for their insightful comments.

9. REFERENCES

[1] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *ACM SIGPLAN Notices*, 28(6):187–196, June 1993.

[2] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. *ACM SIGPLAN Notices*, 33(5):162–173, May 1998.

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

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[5] M. L. Seidl and B. G. Zorn. Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, Jan. 1997.

[6] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. *ACM SIGPLAN Notices*, 33(11):12–23, Nov. 1998.

[7] D. Stefanović and J. E. B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming*, pages 43–54. ACM Press, June 1994.