Dynamic Object Sampling for Pretenuring

Maria Jump
Department of Computer Sciences
The University of Texas at Austin
Austin, TX, 78712, USA
mjump@cs.utexas.edu

Stephen M Blackburn
Department of Computer Science
Australia National University
Canberra, ACT, 0200, Australia
Steve.Blackburn@anu.edu.au

Kathryn S McKinley*
Department of Computer Sciences
The University of Texas at Austin
Austin, TX, 78712, USA
mckinley@cs.utexas.edu

ABSTRACT

Many state-of-the-art garbage collectors are generational, collecting the young *nursery* objects more frequently than old objects. These collectors perform well because young objects tend to die at a higher rate than old ones. However, these collectors do not examine object lifetimes with respect to any particular program or allocation site. This paper introduces low-cost object sampling to dynamically determine lifetimes. The sampler marks an object and records its allocation site every *n* bytes of allocation. The collector then computes per-site nursery survival rates. Sampling degrades total performance by only 3% on average for sample rates of 256 bytes in Jikes RVM, a rate at which overall lifetime accuracy compares well with sampling every object.

An adaptive collector can use this information to tune itself. For example, *pretenuring* decreases nursery collection work by allocating new, but long-lived, objects directly into the mature space. We introduce a dynamic pretenuring mechanism that detects long-lived allocation sites and pretenures them, given sufficient samples. To react to phase changes, it occasionally backsamples. As with previous online pretenuring, consistent performance improvements on SPECjvm98 benchmarks are difficult to attain since only two combine sufficient allocation load with high nursery survival. Our pretenuring system consistently improves one of these, _213_javac, by 2% to 9% of total time by decreasing collection time by over a factor of two. Sampling and pretenuring overheads slow down all the others. This paper thus provides an efficient sampling *mechanism* that accurately predicts lifetimes, but leaves open optimization *policies* that can exploit this information.

Categories and Subject Descriptors: D.3.3[Language Constructs and Features]:

General Terms: Languages

Keywords: Memory management, garbage collection, object sampling, dynamic pretenuring

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

1. Introduction

Many high performance garbage collectors today use *generational* organizations that separates young objects from old objects and then collects the younger objects more frequently. These collectors perform well because young objects usually die quickly and at a higher rate than older ones (the *weak generational hypothesis* [16, 25]). A problem for these collectors is that they usually do not accommodate or detect objects that do not follow this hypothesis.

This paper introduces a low-cost dynamic object sampling mechanism to determine object lifetimes in a generational collector. This sampling mechanism piggybacks on a contiguous *bump-pointer* allocator. It samples one object after every *n* bytes of allocation, where *n* is a power of two. Using an address alignment allows for efficient identification of samples. The sampler adds a word to the sampled object that identifies the object's allocation site, and increments a site allocation counter. During a collection, the collector notes any surviving sampled objects and computes survival rates for each allocation site. For sample rates of 256 and 512 bytes in Jikes RVM using MMTk, this mechanism accurately computes site lifetimes, adding on average a 1% to 3% time and 1% space overhead. Previous approaches sample objects with weak pointers that identify their allocation site [1]. Weak pointer sampling must trace both dead and live objects, incurring large overheads. Huang et al. [15] use type to predict object lifetimes, but type is not a good predictor. To our knowledge, our object sampling technique is the first to combine low overhead and accuracy.

Potential uses for dynamic lifetime information include adaptive garbage collection optimizations and algorithms. For example, *pretenuring* allocates long-lived objects directly into the mature space to reduce nursery copying costs in generational collectors. We introduce a dynamic pretenuring mechanism based on dynamic survivor rates. For a site with a high survival rate and sufficient samples, the collector modifies the allocation site to allocate subsequent objects directly into the mature space. To detect lifetime phase changes, *backsampling* occasionally allocates these sites into the nursery in order to reexamine their survival rate. We examine a range of heuristics for minimum number of samples, pretenuring thresholds, and backsample thresholds. Backsampling provides a robustness to mistakes as well as adaptiveness to phase changes.

The potential for pretenuring on the SPECjvm98 programs is low. Only two programs combine substantial collector load (i.e., lots of allocation) with nursery survival rates greater than 5%. Pretenuring speeds up one of these, _213_javac. It improves garbage collection time by a factor of two, and total time by 2% to 9% since collection time is generally a modest fraction of total time. For the other cases, the overheads and pretenuring errors slow programs down on average by 1% to 4%, and up to 16%. Prior dynamic pretenuring work [14, 15] achieves smaller improvements and similar worst case degradations, e.g., _205_raytrace slows down by 15% [14]. To prevent degradations when pretenuring is not applicable, the system could trigger sampling only when nursery survival statistics indicate some potential. For example, the collector could turn on sampling and dynamic pretenuring only when nursery survival rates rise above 15%. We did not explore this feature.

We thus introduce an efficient online sampling mechanism for determining lifetimes and a dynamic pretenuring mechanisms for exploiting object lifetimes. However, we leave open policies that use this information to consistently improve performance.

2. Related Work

This section compares our work to previous research on object lifetime prediction, dynamic object sampling, dynamic pretenuring, and static pretenuring.

Other than the weak generational hypothesis [16, 25], previous work using analytical modeling and experimental classification across programs has not yielded any additional general object lifetime hypotheses [9, 22]. However, many memory management techniques improve performance for a given program based on its individual characteristics.

To determine and exploit lifetimes dynamically, previous work uses write barriers and weak pointers. Domani et al. [11] and Qian and Hendren [17] use write barriers to trap and differentiate global and local heap pointers. They then collect the local heaps independently. Qian and Hendren further redirect sites that allocate global variables into the global heap. Both of these techniques can add significantly to the execution time of the program, whereas our mechanism adds negligible overhead.

Agesen and Garthwaite [1] sample objects by inserting weak pointers which identify the object allocation sites. Their approach is most similar in spirit to ours. After a collection, they must trace both the dead and surviving sampled objects through the weak pointers to gather statistics. They do not report overhead separately, but as part of dynamic pretenuring. Total performance improves and degrades on average by 1% to 2%, but _205_raytrace from SPECjvm98 degrades by 15%. We instead mark samples by their respective memory addresses. During collection, we need only track survivors. At the end of a collection, the allocation and survivor statistics completely specify lifetimes. These mechanisms reduce our space and collector time overheads compared to weak pointers. Both mechanisms require specialized allocation and collection support. Our object sampling is more general than weak pointers since it needs no language support.

Harris [14] uses Agesen and Garthwaite's sampling mechanism to make dynamic pretenuring decisions for Java programs in the context of a two generation collector. When the system detects a long-lived allocation site, it begins allocating into a mature generation. His system samples in the higher generation to determine whether or not to reverse a decision, but the infrequency of higher generation collections reduces the accuracy of these lifetime samples. We instead allocate the occasional pretenured site back into the nursery. Harris notes that these objects will always survive if they are connected to another pretenured object, and in this case our mechanism would not yield useful samples. We find that this case does not occur frequently, and thus we can react more quickly to phase changes. Harris uses separate thresholds for pretenuring and reversal. He uses backpatching to change the allocation site, rather than a load to determine the allocation region. Neither technique recompiles the method. Harris uniquely identifies the allocation site without any call chain information. Because Jikes RVM performs aggressive inlining, allocation sites in our system tend to have more context, which Harris suggests should be useful. His dynamic pretenuring results show both improvements and a few significant degradations but are limited to a single heap size. We find improvements in a wide range of heap sizes while using a faster collector, providing a more general mechanism, and incurring lower overhead.

Huang et al. [15] compute per-class rather than per-site allocation and survival statistics which is easy to implement, since each object header includes the type already. However, type is not a good predictor of lifetime. They use the Jikes RVM baseline compiler which does not produce high quality code and thus can hide any overhead. We use the adaptive optimizing compiler. Their approach degrades total execution time slightly for the two of three SPECjvm98 benchmarks they test, _202_jess and _228_jack, while improving _213_javac by 2% to 5%.

Another approach to lifetime classification for heap optimizations is static profiling [4, 7, 8, 13, 18, 19, 23, 24]. For instance, a static profiler finds allocation sites for long-lived objects in a generational collector and recompiles the program to allocate these directly to mature generations [7, 8, 23, 24]. A profile-driven approach is problematic for a just-in-time compiler. If programmers were willing to profile, they would compile ahead of time.

3. Object Sampling

This section describes and evaluates object sampling for lifetime prediction. The sampling mechanism requires a *bump-pointer* allocation with a copying collector. We focus here on lifetime sampling for newly allocated *nursery* objects, but this mechanism can sample other characteristics as well.

Bump-pointer allocators use monotonically increasing addresses within a contiguous region of memory by repeatedly incrementing (*bumping*) a pointer. This *fast path* of the allocation sequence uses only a few instructions including a test to check whether the allocation exceeds some boundary. Figure 1(a) illustrates this sequence. When the allocator exceeds the boundary, it calls the *slow path* which determines, for example, whether the allocator needs to request more memory or if it should trigger a collection. A sufficiently large allocation region makes the *fast path* the common case, and executes the more expensive *slow path* infrequently.

Figure 1(b) illustrates dynamic object sampling. Sampling adds no instructions to the most frequently executed *fast path* (compare lines 1-9 in Figure 1(a) & (b)). Object sampling however introduces an intermediate path whose test succeeds every `SAMPLE_PERIOD` bytes of allocation. For lifetime sampling, the allocator then records a one-word object tag which encodes the object allocation site and a magic number, which allows sampled survivors to be identified (`allocSample()`, line 23). Lifetime prediction also increments a per-site allocation counter. The compiler inline pragma, line 1 in Figure 1(a) and (b), directs the compiler to inline the *fast path* of the allocation sequence into the caller, leaving the *sample path* and cold *slow path* as method calls.

Instead of the addition of special actions at every allocation, the *sampling path* occurs only every `SAMPLE_PERIOD` bytes. We can tune the sampling rate by statically or dynamically adjusting it. Larger values, of course, trade lower overhead for fewer samples. We explore only statically specified sample rates here.

The garbage collector aggregates object statistics during collection. As the collector copies all reachable objects out of the nursery, it checks each surviving object. If the word before the object contains the magic number, the object is a sample, and the collector decodes the object's allocation site from the tag. This approach allows different sites to use different sample rates and frees the collector from knowing *a priori* whether an object had been sampled. Lifetime sampling computes transient and total object survival statistics

```
VM_Address alloc(int bytes)
  throws VM_PragmaInLine {
  VM_Address oldCursor = cursor;
  VM_Address newCursor = oldCursor.add(bytes);
  if (newCursor.GT(limit))         // need more memory?
    return allocSlow(bytes);
  cursor = newCursor;
  return oldCursor;
}
```

(a) Original bump pointer allocation

```
VM_Address alloc(int bytes, int siteID)
  throws VM_PragmaInline {
  VM_Address oldCursor = cursor;
  VM_Address newCursor = oldCursor.add(bytes);
  if (newCursor.GT(sampleLimit))   // need to sample?
    return sample(bytes, siteID);
  cursor = newCursor;
  return oldCursor;
}
VM_Address sample(int bytes, int siteID)
  throws VM_PragmaNoInline {
  VM_Address rtn;
  int required = bytes + SAMPLE_BYTES;
  VM_Address newCursor = cursor.add(required);
  if (newCursor.GT(limit)) {       // need more memory?
    rtn = allocSlow(required, siteID);
    if (rtn.isZero()) return rtn;  // we need to GC
  } else {
    rtn = cursor;
    cursor = newCursor;
    sampleLimit = roundUp(cursor, SAMPLE_PERIOD);
  }
  allocSample(rtn, bytes, siteID); // record sample
  return rtn.add(SAMPLE_BYTES);    // skip object tag
}
```

(b) Sampling bump pointer allocation

Figure 1: Changes to the Bump Pointer Allocation

in an array indexed by site. *Transient* statistics are for one collection phase (or several), while total statistics accumulate information over the entire program.

This mechanism samples larger objects more frequently and thus yields more accurate statistics for them. As Harris [14] points out, large objects are important – especially if they are prolific.

3.1 Overhead and Accuracy

To evaluates the lifetime sampling overhead and accuracy as a function of sample rate, we start by briefly describing our experimental setting, collector organizations, and Jikes RVM. We then demonstrate that sampling overheads are on average low, between 1% and 3%, and at worst 6%, for sample periods of 256 and 512 bytes. Even sampling every object adds an overhead of only on average 8% to 9%, but the worst case rises to 18%. We then compare the accuracy of sampling for collecting lifetime statistics for sample periods of between 32 and 4K bytes compared with sampling every object. We find that modest sample rates are sufficient to accurately predict nursery survival rates.

3.2 Methodology

We use the system and methodology described here for all the results in this paper.

3.2.1 Collector

We implement our technique in MMTk, a memory management toolkit in Jikes RVM version 2.3.0.1 (formerly known as Jalapeño). MMTk implements a number of collectors [5, 6]. We use a well performing [5] 4 MB *bounded* bump-pointer nursery and a mark-sweep mature generation (*GenMS*). However, our technique is compatible with other mature generation organizations. In MMTk, the *bounded* nursery takes a command line parameter as the initial nursery size, collects after the nursery is full, and resizes the nursery below the bound only when the mature space cannot accommodate a nursery of survivors. When the nursery size falls below a lower bound (we use 256KB), it triggers a mature space collection.

The mark-sweep mature space is organized as a segregated-fits free-list with lazy freeing. The allocator divides memory into blocks of same size chunks. The collector traces and marks live objects, and puts blocks with free objects on the appropriate free-block list. The allocator constructs the free object list for the block the first time it allocates from the block. (See Blackburn et al. [5] for additional MMTk details.)

3.2.2 Jikes RVM and Jikes Compilers

Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [2, 3]. We only use configurations that precompile as much as possible, including key libraries and the optimizing compiler (the *Fast* build-time configuration), and turn off assertion checking. We report two configurations: fully optimized compilation and pseudoadaptive compilation. In the fully optimized methodology, the optimizing compiler precompiles all methods. *Pseudoadaptive* compilation deterministically applies the optimizing compiler to frequently executed methods chosen by the adaptive compiler in previous (offline) runs. This methodology gives us a realistic mixture of optimized and unoptimized code, but does not expose the experiments to the natural variations in allocation and time due to timer-based adaptive compilation.

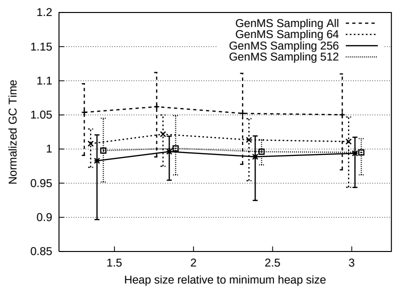
Eeckhout et al. [12] show that including adaptive compilation in performance measurements obscures application behavior. We thus report only application performance by running two iterations of each benchmark. The first run uses one of the compiler configurations from above, and then turns off compilation. Before the second iteration, a whole heap collection flushes compiler objects from the heap.

Jikes RVM compiler aggressively inlines methods. This policy is a doubled-edged sword. On the positive side, it provides extra context to differentiate call sites. If context is unnecessary, i.e., all calls to this allocation have very similar lifetime statistics, it takes longer to determine lifetimes for each individual inlined site than it will if one site does all the allocation. Previous work on C and ML suggest this context is useful [4, 8], whereas work on Java found it may not always be necessary [7].

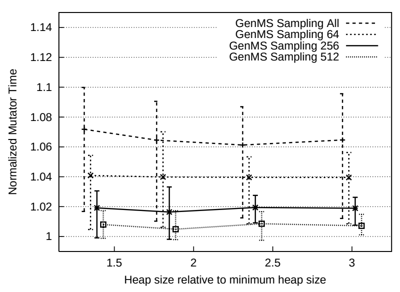
3.2.3 Benchmarks, Architecture, and Measurements

We evaluate our techniques using the SPEC JVM benchmarks and pseudojbb, a variant of SPEC JBB2000 [20, 21] that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. We perform all of our experiments on a 3.2 GHz Intel Pentium 4 with hyper-threading enabled, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 1GB of main memory, running Linux 2.6.0.

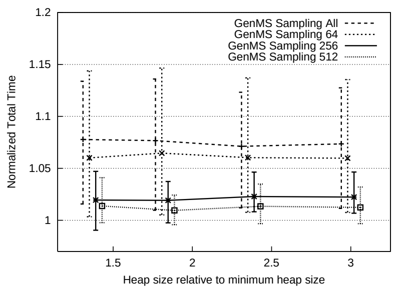
We explore the time-space trade-off by executing each program on five heap sizes, ranging from the smallest one possible for the execution of the program to three times that size. We execute timing runs five times in each configuration and choose the best execution time (i.e., the one least disturbed by other effects in the system). We perform separate statistics gathering runs that accumulate overall and individual collection statistics. We compute and report statistics such as the number of collections, the number of samples, the number of surviving samples, bytes allocated between collections, and bytes copied per collection.



(a) GC Time

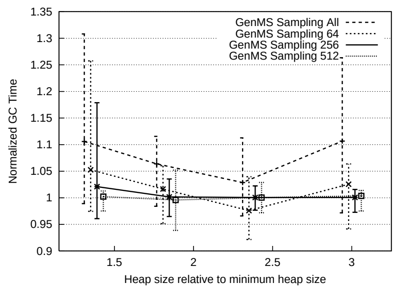


(b) Mutator Time

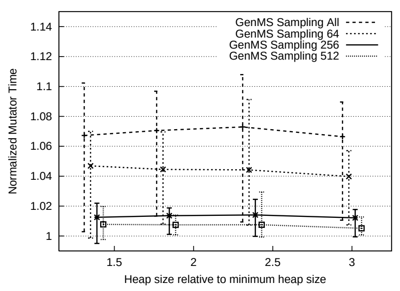


(c) Total Time

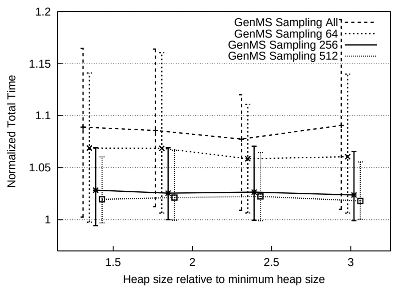
Figure 2: Sampling Overhead for Pseudoadaptive Compilation



(a) GC Time



(b) Mutator Time



(c) Total Time

Figure 3: Sampling Overhead for Optimizing Compilation

3.3 Lifetime Sampling Overhead

This section presents sampling time and space overheads, shown in Figure 2 with the pseudoadaptive methodology and in Figure 3 with fully optimized application code. The figures normalize sampling against no sampling for a range of heap sizes using the geometric mean of our benchmarks. Error bars show show the variations for sampling every 512, 256, and 64 bytes, as well as all objects. The direct overhead of sampling has two components, the *spatial* overhead of a four byte site identifier and the *computational* overhead of periodically executing `sample()` (Figure 1(b), lines 10-25).

Since lifetime sampling adds a four byte site identifier to each sampled object, it increases space requirements by at most 0.8% for a 512 byte sample rate, and 1.6% for 256. The impact of this spatial overhead on garbage collection time is subtle, as shown in Figures 2(a) and 3(a). One would assume that the dominant cost would always be the additional work associated with collecting the nursery more frequently, but more subtle effects of perturbing collection trigger points can dominate. Changing when a collection occurs can have cascading positive and negative effects on promotion results and locality. For example, if the program is about to allocate some medium lifetime objects, an earlier collection avoids copying them. Any change in the amount of allocation results in these effects [7]. Only when every allocation is sampled is the average collection time overhead significant (5% to 10%). For 256 and 512 byte sample rates the average collection overhead is negligible and is dominated by perturbations, which can produce up to 15% degradation and 10% improvement.

The mutator time overheads are very reasonable. They average between 0.5% and 2% of mutator time for sample rates of 256 and 512. Figures 2(b) and 3(b) show mutator time overhead and 2(c) and 3(c) show the total time overhead. The error bars show the worst case overheads and a few tiny improvements. For sampling at 256 and 512 bytes these range from -0.5% to 3.5%, and are slightly lower with the optimizing compiler. This mutator overhead includes the additional instructions required to sample, but also includes the effects of data and instruction locality, which presumably account for the tiny performance improvements. The overhead is still low on average (4% to 5%) for 64 bytes.

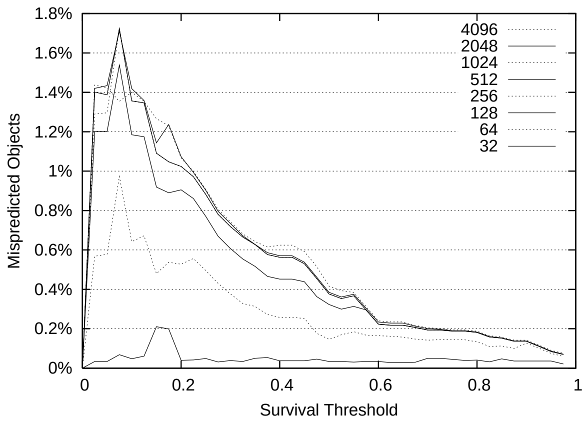
For sample rates of 256 and 512, total time overhead ranges from 6% to less than -1%. The average rate is between 1% and 3%, and is slightly higher using the optimizing compiler. The average object size is 32 bytes which includes Jikes RVM's 8 byte header [10]. Sampling every 64 bytes thus samples approximately every two objects. When sampling every 64 bytes or all objects, overheads grow substantially, up to 18% worst case, but 7% to 9% on average.

3.4 Lifetime Sampling Accuracy

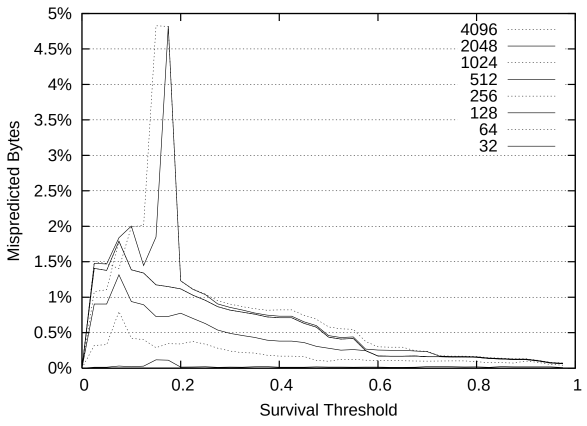
This section evaluates the error introduced by sampling when computing nursery survival rates. For each allocation site in a program, we establish the actual survival rate (the number of surviving objects over the total number of objects allocated), and the survival rate predicted with sampling. Depending on the particular demographics of the subset of objects at that site which were sampled, the survival rate can be overestimated or underestimated.

We use a *survival threshold* to classify sites as either short-lived or long-lived. For example, one might classify all sites with a survival rate higher than 80% as long-lived, and all others as short-lived. We then measure sampling accuracy in terms of *site misprediction*. At each site, we determine for each sample rate whether the *sampled survival rate* would cause the site to be classified differently from the site's *actual survival rate*. We then quantify this misprediction by summing for a given survival threshold the objects and bytes allocated at mispredicted sites. We repeat this process for a range of survival thresholds from 0 to 1.

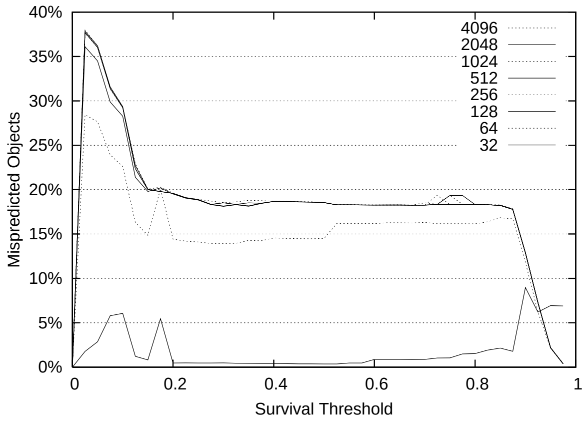
Figure 4 plots site mispredictions for _202_jess and _213_javac, which are representative and diverse. The x-axis varies the survival threshold and the y-axis plots the level of misprediction for all sites



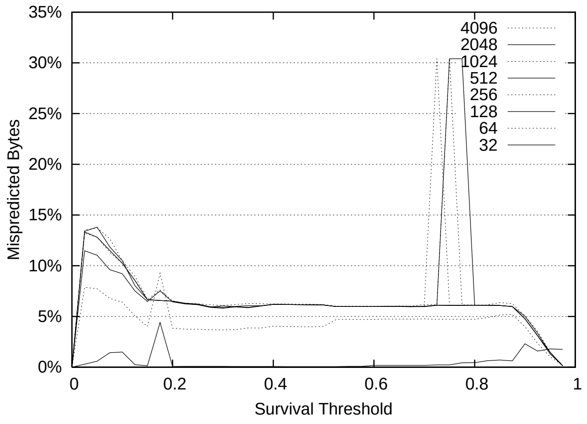
(a) _202_jess Site Misprediction (Objects)



(b) _202_jess Site Misprediction (Bytes)



(c) _213_javac Site Misprediction (Objects)



(d) _213_javac Site Misprediction (Bytes)

Figure 4: Site Mispredictions as a Function of Survival Threshold and Sample Rate

that are mispredicted at a given survival threshold. Consider a site with an *actual* survival rate of 20% and a *predicted* survival rate of 80%. For survival thresholds between 0% and 20% the site would be *correctly* classified as 'long lived' (both predicted and actual survival rates are greater than the threshold). However, for thresholds between 20% and 80% it would be *incorrectly* classified as 'long lived' (the predicted survival rate is greater than the threshold, but the actual survival rate is not). For thresholds above 80% it would be *correctly* classified as 'short lived' (both predicted and actual survival rates are lower than the threshold). All objects and bytes allocated at this site would be included in all points on the site misprediction curve between the 20% and 80% survival thresholds. A site that was perfectly predicted would never contribute to the site misprediction curve. The figures plot site mispredictions with respect to objects (Figure 4(a) & (c)), and bytes (Figure 4(b) & (d)) of allocation associated with each mispredicted site. For example, sampling is less accurate for sites that have short-lived objects (survival rates between 5% and 15%) on both programs, compared with predicting objects with long lifetimes (rates above 60%).

The figures include one line for each sample rate from 32 bytes through to 4KB. Figure 4(a) shows that the level of misprediction in _202_jess is very low ($< 2\%$), even at coarse sample rates. As the threshold grows, mispredictions become even less common. This trend reflects that _202_jess has mostly low survival sites. Figure 4(b) measures site mispredictions in bytes and has almost the same curve as (a), but with a considerable spike just before the 20% survival threshold. Note that only the two coarsest sample rates experience this spike. This spike reflects a site that allocates large objects with a survival rate a little under 20%.

Figure 4(c) shows site misprediction for _213_javac accounts for nearly ten times as many objects (around 20% of all allocations). The level of misprediction in _213_javac is almost the same for all sample rates greater than 32, suggesting that the mispredicted sites are predominately allocating objects < 64 bytes in size. The flat curve illustrates a 20% misprediction rate between survival thresholds of 20% to 90%. Figure 4(d) shows that as a percentage of bytes, mispredictions are much lower, only around 6%. Mispredicted sites thus tend to allocate very small objects. _213_javac has a sensitive spot at around 75% where the very coarsest sample rates see substantial degradation. This result suggests a site or sites allocating large objects with around 75% survival rate.

Overall, sample rates between 128 and 1024 bytes have high accuracy, with typical site misprediction rates in bytes ranging from nearly zero to around 6%. For these benchmarks with few long-lived objects, their behavior follows _202_jess; i.e., mispredictions are highest at the lowest thresholds.

4. Dynamic Pretenuring

This section describes an example use of object sampling: dynamic pretenuring. Pretenuring seeks to reduce the load on the nursery collector by allocating long-lived objects directly in to the mature space. Our dynamic pretenuring system consists of the following steps: (1) determining which allocation sites produce long-lived objects; (2) redirecting the allocation site directly into the old space; and (3) backsampling to detect allocation site phase changes in a timely manner. The policy components use the following thresholds.

Minimum Samples : The minimum number of samples required from an allocation site during an allocation phase for the site to be considered for pretenuring.

Pretenuring Threshold : The survival rate above which a site is pretenured.

Backsampling Policy : The backsampling functions include constant (*cbs*), linear (*lbs*), and exponential (*ebs*).

Backsampling Shift : The backsampling trigger as a function of the number of objects used to make the pretenuring decision.

Decay Shift : The amount by which mature statistics should be decayed.

We now describe these components in more detail.

4.1 Pretenuring Statistics and Policies

We compute aggregate and transient lifetime statistics for pretenuring. As Section 3 describes, the sampler increments a counter for the allocation site of every sampled object. During a nursery collection, the collector increments a site counter for any surviving sampled objects allocated from each site. At the end of a nursery collection, we compute survival rates for this collection (*transient*) and aggregate statistics. We only use one collection phase for transient in our experiments. This separation focuses on those sites that changed in the last allocation phase (those with non-zero transient entries). For a site with sufficient samples and survival rate, dynamic pretenuring starts to allocate from the site into the old space after the first garbage collection.

Pretenuring polices can use either transient or aggregate statistics. To react quickly to phase changes, we use transient statistics to begin pretenuring any site with a survival rate exceeding a threshold t_s. This policy is very aggressive and introduces some errors but quickly captures newly allocating sites producing long-lived objects.

4.2 Dynamic Allocation Targets

In order to act on pretenuring decisions, we add a dynamic test to the allocation sequence (see Figure 5). It uses only a two instructions, an array lookup (line 12)[1] and a conditional branch (line 3). This implementation is simple and easily generalizes. As our results section shows, when the optimizing compiler inlines the entire allocation sequence and then optimizes it in context, the overhead of this additional test is on average 1% to 2%.

Another approach would be to backpatch the allocation instructions, which completely removes allocation time overhead. We originally implemented this approach, but later concluded that it was a premature optimization. The backpatcher was complex and extremely brittle as it had to parse and manipulate instruction sequences generated by an aggressive optimizing compiler, and the nature of those sequences was different between platforms and subject to change as the code in the allocation sequence and the compiler evolved. We ultimately choose this simpler approach since it is very robust and although the overhead is not zero, it is very low.

[1]Our implementation uses a special instruction that avoids the array bounds check.

```
1 ...
2   case NURSERY_SPACE:
3     region = nursery.alloc(isScalar, bytes);
4     break;
5 ...
```

(a) Original allocation

```
1 ...
2   case NURSERY_SPACE:
3     if (DynamicPretenure.nurseryAlloc(site))
4        region = nursery.alloc(isScalar, bytes, site);
5     else
6        region = matureAlloc(isScalar, bytes, site);
7     break;
8 ...
9
10 public final static boolean nurseryAlloc(int site)
11   throws VM_PragmaInline {
12   return pretenureTable[site] >= 0;
13 }
```

(b) Allocation with Dynamic Test

Figure 5: Dynamic Test Added to the Allocation Sequence

4.3 Backsampling

Once the system decides to pretenure a site, allocating its objects into the mature space, the sampler can no longer compute that site's nursery survival rate. If the decision were wrong or the application behavior changed, the system would never know. To avoid this situation, we use *backsampling*. Backsampling periodically allocates pretenured sites back in the nursery for one allocation phase, thereby providing an opportunity to reassess the site's survival rate.

We experiment with different policies that vary the frequency of backsampling, based on the backsampled transient and total survival rates. We implement backsampling by initializing the site's *mature counter* to the negative of the *backsampling target*, which is the total number of allocations used in making the pretenuring decision. Each time an object is allocated into the mature space, the mature counter is incremented, and when it reaches zero, the site allocates into the nursery for one allocation phase. If at the next collection the survival rate is no longer high enough, the system reverses the pretenuring decision. Otherwise, the system reinforces the pretenuring decision by changing the backsampling target according to one of the following heuristics:

- The constant heuristic (cbs) leaves the backsampling target as the number of allocations used to make the original decision (n).
- The linear heuristic (lbs) makes it harder to backsample the site by initializing the backsampling interval to a multiple ($f \geq 1$) of n where f grows linearly with each consecutive agreeing decision.
- The exponential (ebs) heuristic, f grows exponentially as a power of 2.

The constant heuristic backsamples the most frequently, the linear less than constant, and the exponential backsamples the least frequently. Backsampling is a conservative mechanism. It reduces the effectiveness of good choices, but protects the system from sampling errors and changes in allocation lifetime phase behavior.

5. Dynamic Pretenuring Results

This section evaluates dynamic pretenuring. We first discuss its potential on our benchmarks and find two programs that might benefit from pretenuring. We then present the overheads due to the change in the allocation sequence (see Figure 5), and the combination of this together with sampling overheads. We show that the total overhead is on average 0 to 4%.

Next, we evaluate the accuracy and coverage of the pretenuring decisions, some decisions are accurate, but coverage is poor. We miss pretenuring opportunities due to the warm-up and backsampling periods or lack of allocation site lifetime homogeneity. We explored the parameter space for pretenuring to find good configurations and report the ones with the best performance for _213_javac. Table 1 shows these configurations. One thing that we did not vary is the use of transient or aggregate statistics, these results always use transient statistics. Aggregate are more conservative, and would probably reduce some errors.

We report total time, garbage collection time, and mutator time results for dynamic pretenuring for these configurations. We show that dynamic pretenuring improves one program by up to 9%, and degrades all the others. Reasonable configurations can degrade performance by up to 16%, and poor ones by even more.

5.1 Potential of Pretenuring

Table 2 shows key characteristics of our benchmarks using pseudoadaptive compilation and an infinite heap with a 4MB nursery. The *alloc* column in Table 2 indicates the total number of megabytes allocated. The second column lists the ratio of total allocation to the minimum heap size (the smallest heap that the benchmark can run in) for the GenMS collector in MMTk and thus quantifies garbage collection load. We order the table by the % *nrs srv* ratio, which indicates the percentage of objects that survive a nursery collection. This percentage indicates the potential for dynamic pretenuring to eliminate unnecessary copying.

Only three of these programs are likely to benefit from pretenuring, pseudojbb, _213_javac, and _209_db. In particular with a 4MB nursery, pseudojbb and _213_javac perform 50 and 53 nursery collections (respectively), whereas _209_db performs 20. However, closer examination of _209_db and pseudojbb show pretenuring is unlikely to improve them. In pseudojbb, only a few allocation sites produce the majority of long-lived objects, but these same sites produce many short lived objects as well. Thus, these sites never produce survival rates high enough to benefit from pretenuring without more calling context than we examine here. In _209_db, all the long-lived objects are allocated in the first 8MB of allocation. Dynamic pretenuring misses these opportunities while it is warming up.

5.2 Pretenuring Overheads

Figure 6 reports the overhead for the dynamic test that is added to the allocation sequence. Since we are measuring overhead, this test is set to dynamically resolve to false. It shows that the runtime overhead of the dynamic test in Figure 5 is around 1% in the fully optimized case and in the noise in the adaptive case. In a non-optimized setting the overhead of the extra memory load and conditional will be swamped by other inefficiencies. The mix of optimized and non-optimized code in the adaptive case hides the small overhead in the optimized code.

Figures 7 and 8 reports the overhead for sampling and the dynamic allocation test. We measure this by running dynamic pretenuring with a configuration which will not actually pretenure by setting the pretenuring threshold above 100%. Figure 7 uses the pseudoadaptive methodology, and Figure 8 the fully optimized application code. Again, all the results exclude the compiler itself. Using the optimizing compiler on the adaptive pretenuring allocation sequence lowers its average overhead by 2% to 3%, but increases the variation from a range of 8% to -10%, to a range of 12% to -15%. The compilation differences again reflect that a fully optimized setting exposes any overhead more.

	OLPT parameter values		
parameter	80 LBS	80 EBS	85 LBS
minimum samples	8	4	10
pretenuring threshold	80%	80%	85%
backsampling policy	linear	exponential	linear
backsampling shift	1	4	1
decay shift	0	0	1

Table 1: Configuration Settings for Base Results

Benchmark	alloc (MB)	alloc: min	% nrs srv
pseudojbb	210	4:1	41
_213_javac	172	6:1	28
_209_db	74	4:1	11
_227_mtrt	117	6:1	6
_228_jack	225	18:1	3
_205_raytrace	110	6:1	3
_202_jess	261	18:1	1
_201_compress	105	7:1	0

Table 2: Benchmark Characteristics

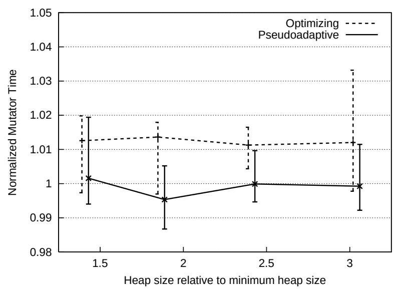


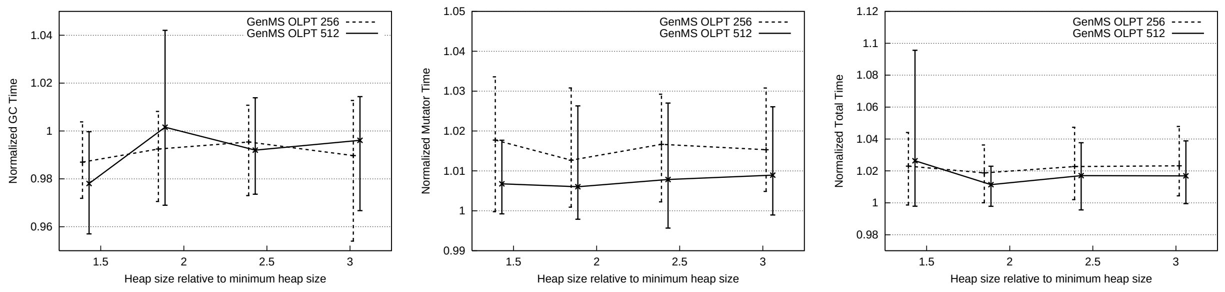
Figure 6: Dynamic Allocation Test Overhead with the Optimizing Compiler: Geometric Mean of Total Execution Time

5.3 Accuracy and Coverage

We now quantify the *accuracy* and *coverage* of pretenuring decisions in bytes of allocation. Accuracy measures how many of the objects chosen for pretenuring were actually long-lived. Coverage measures how many of the long-lived objects were actually chosen for pretenuring. Accuracy can be high while missing opportunities (low coverage).

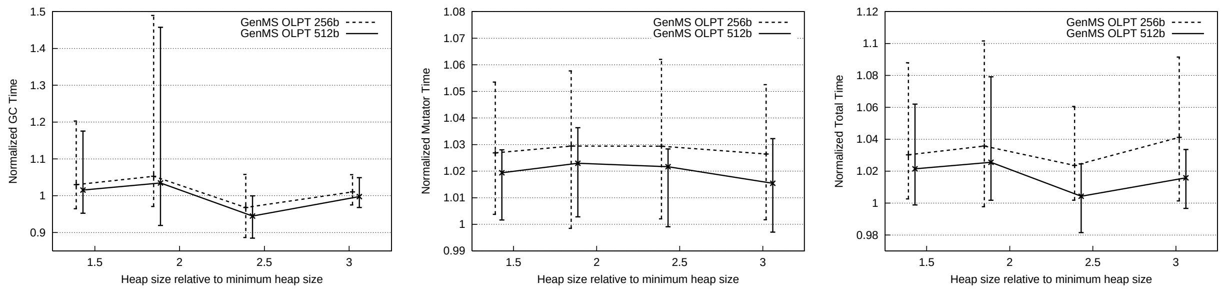
Figure 9(a) shows accuracy, and 9(b) shows coverage for each benchmark and a range of pretenuring thresholds. We assume an infinite mature space, thus the nursery is always 4MB. This configuration therefore examines accuracy without cascading the penalty of mistakes. The other parameters values are the same as 80 LBS from Table 1.

In Figure 9(a) the height of the bars represents the total volume of pretenured objects. The solid portion shows long-lived objects (i.e. *correctly* pretenured), and the striped portion indicates short lived objects (i.e. *incorrectly* pretenured). Pretenuring accuracy is 80% or better for _213_javac, which is the only benchmark we speed up. The error rate for _209_db is 34%, and even worse for pseudojbb at just under 50%. We expect errors to grow with a lower threshold because while decisions are per-allocation site, here we measure individual objects. For example, if a site has an 80% survival rate and is pretenured, the 20% of objects which are short lived at that



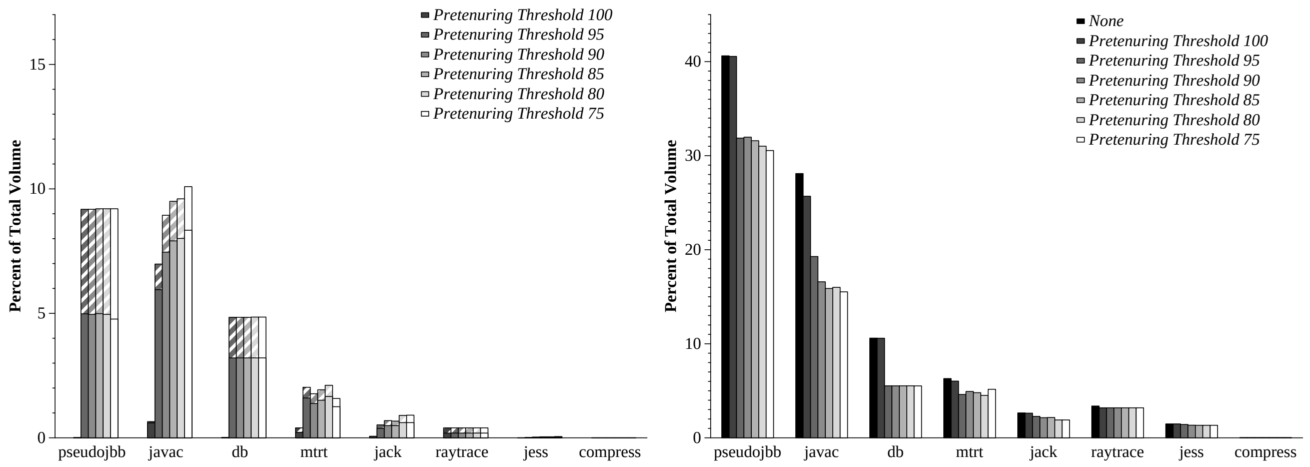
(a) GC Time (b) Mutator Time (c) Total Time

Figure 7: Sampling and Pretenuring Overhead for Pseudoadaptive Compilation



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

Figure 8: Sampling and Pretenuring Overhead for Optimizing Compilation

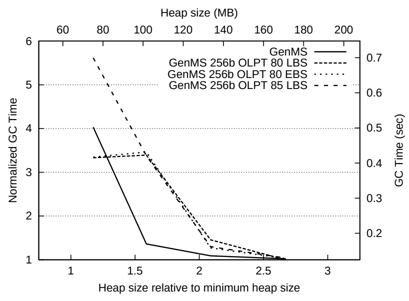


(a) Pretenured Objects, Correct (solid) and Incorrect (striped) (b) Long Lived Objects *Not* Pretenured

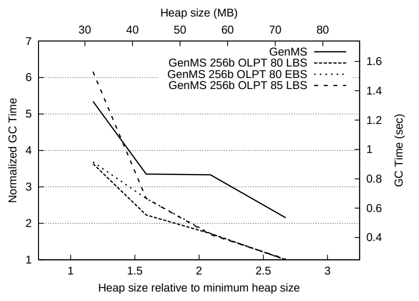
Figure 9: Accuracy (a) and Coverage (b) of Pretenuring in Percent of Total Volume

site will be incorrectly pretenured. These results barely show this trend, thus the errors are most likely due to sampling errors (recall that we used a 256 byte sample rate), and from heterogeneous allocation lifetime phases. Error rates are higher for _227_mtrt, _228_jack, _205_raytrace, and _202_jess, but the pretenuring volume is extremely low.

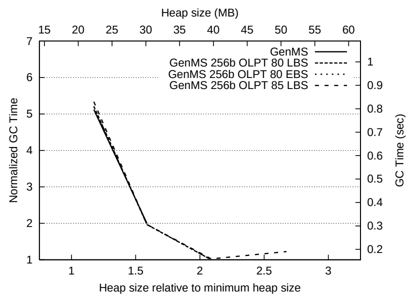
Figure 9(b) shows coverage; each bar represents the volume of long-lived objects that are *not* pretenured under different pretenuring regimes. The first bar, 'None', shows the volume when no pretenuring is performed, and therefore reflects the total volume of long-lived objects. The remaining bars vary the pretenuring threshold, although only pseudojbb, _213_javac, _209_db, and _227_mtrt show any sensitivity to the threshold. The proximity of these bars to the 'None' bar shows dramatic under-pretenuring. At best we see 43% coverage (in _213_javac with 75% pretenuring threshold), but in most cases the coverage is much lower. Reasons for under-pretenuring include objects missed during 'warm up' of the sampling mechanism, objects missed due to the sample rate, and lack of homogeneity at allocation sites (long-lived objects allocated from predominantly short-lived sites). Notice less than 5% of allocation is long lived for the remaining programs.



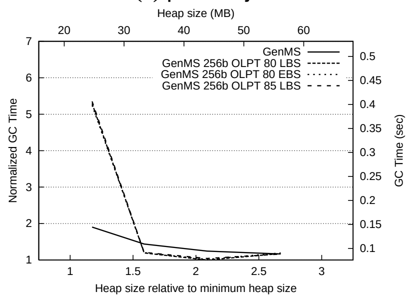
(a) pseudojbb



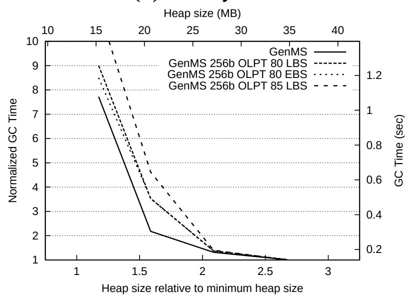
(b) _213_javac



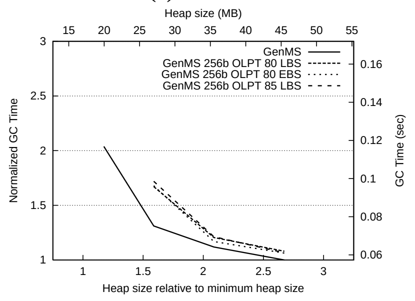
(c) _209_db



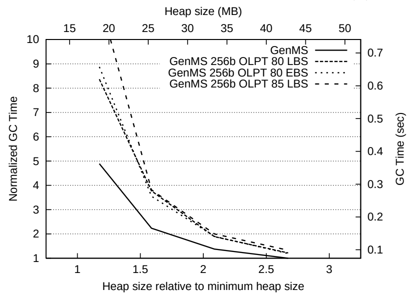
(d) _227_mtrt



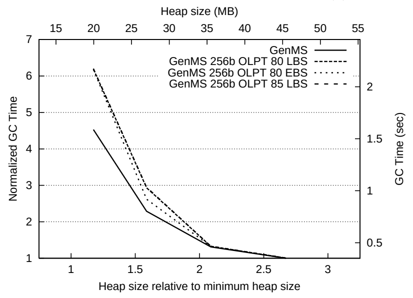
(e) _228_jack



(f) _205_raytrace



(g) _202_jess



(h) _201_compress

Figure 10: Garbage Collection Time

5.4 Pretenuring Results

We examined a range of these thresholds and policies. For the remaining experiments, we use the values in Table 1 which generally, but not uniformly, give the best results in our experiments.

The goal of pretenuring is to reduce garbage collection load. Figure 10 shows that we are only able to systematically reduce garbage collection load in _213_javac, where improvements are as much as a factor of 2.5. We see some modest improvements in _227_mtrt. _209_db and _228_jack are not significantly changed by pretenuring, but the remaining benchmarks all see degradations in garbage collector performance. In a small heap, erroneously pretenured objects needlessly occupy the mature space, reducing the bounded nursery size and triggering expensive full-heap collections. The pretenuring configurations also show sensitivity to heap size; 85 LBS is particularly bad in a small heap, but matches the best performance in large heaps.

The improvements in GC time for _227_mtrt and _213_javac translate to total time in Figure 12. _213_javac improves by around 3% on average and by as much as 9% in a tight heap. _227_mtrt improves by around 2% but degrades significantly in a tight heap. All other benchmarks show degradations in total time. Interestingly, _213_javac shows a greater degradation in mutator time, presumably due locality degradations caused by the disruption of allocation order that follows from a relatively high pretenuring rate.

6. Conclusion

This paper introduces a low-overhead object sampling technique. We show that sampling can accurately predict allocation site survival rates. To use these predictions, we introduce a dynamic pretenuring scheme. Since few of our benchmark programs can benefit from dynamic pretenuring, attaining performance improvements on even these is very challenging. Although we are the first to show significant performance improvements on any of the SPECjvm98 benchmarks using dynamic pretenuring [14, 15], we also show significant degradations. The question therefore remains as to whether there is a pretenuring *policy* or other optimization policies that can benefit from lifetime sampling.

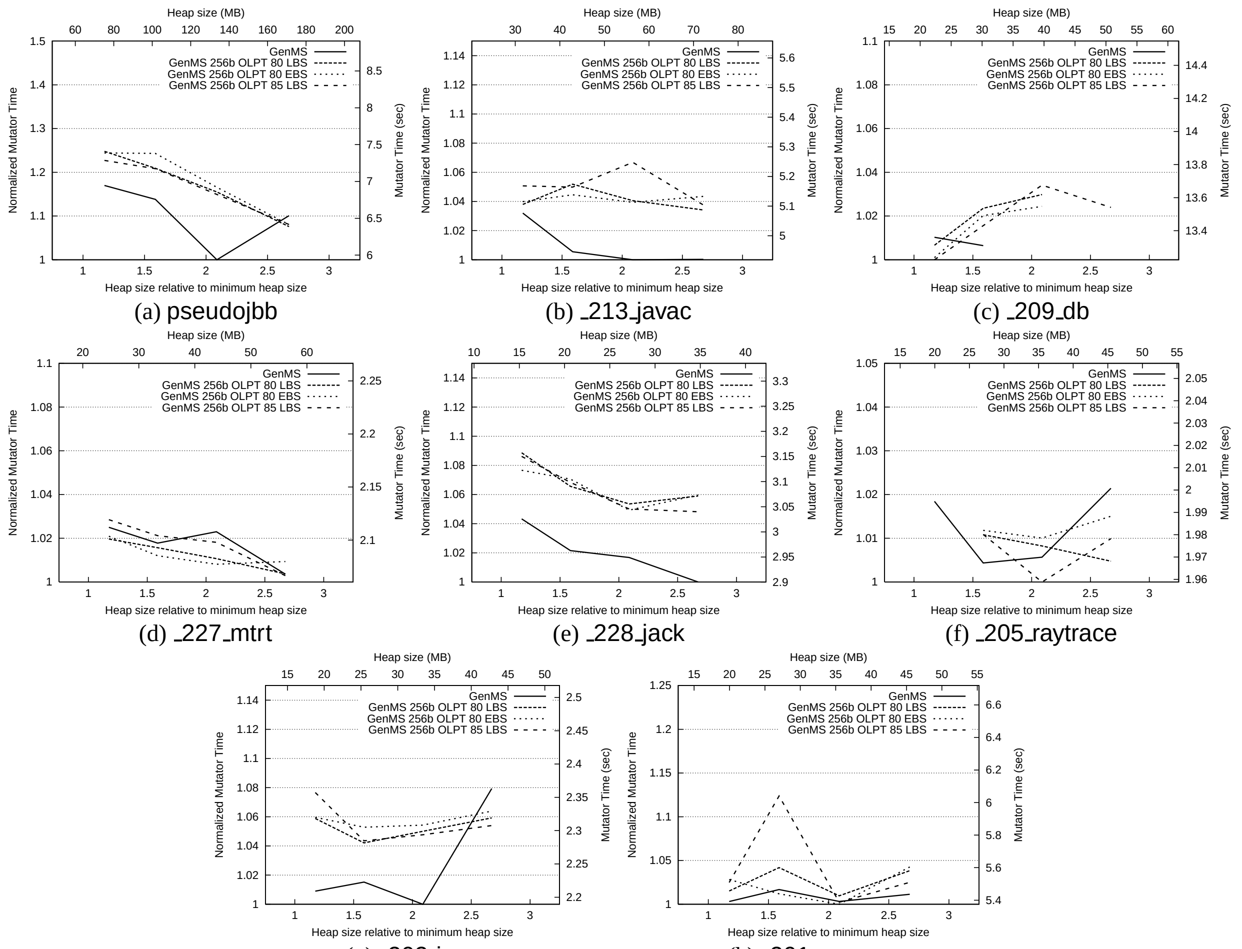
7. Acknowledgements

We thank Steve Dropsho, Chip Weems, and Eliot Moss for their input and discussions on the preliminary versions of this work.

8. REFERENCES

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

[2] B. Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems,*



(a) pseudojbb (b) _213_javac (c) _209_db

(d) _227_mtrt (e) _228_jack (f) _205_raytrace

(g) _202_jess (h) _201_compress

Figure 11: Mutator Time

Languages, and Applications, pages 314–324, Denver, CO, November 1999.

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

[4] D. A. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 187–196, Albuquerque, New Mexico, June 1993.

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

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

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

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

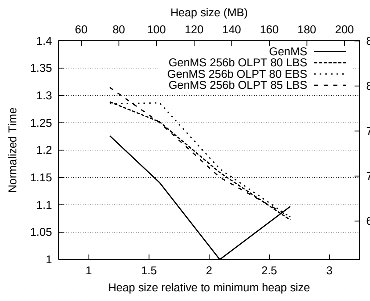
[9] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 73–85, Las Vegas, NV, June 1997.

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

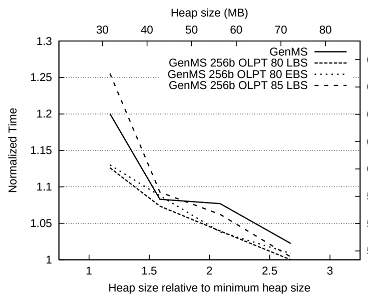
[11] T. Domani, G. Goldshtein, E. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ACM International Symposium on Memory Management*, pages 76–87, Berlin, Germany, June 2002.

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

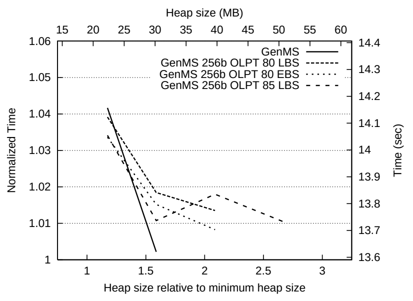
[13] D. R. Hanson. Fast allocation and deallocation of memory



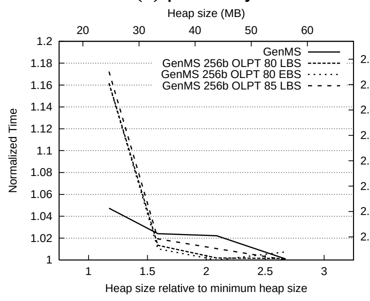
(a) pseudojbb



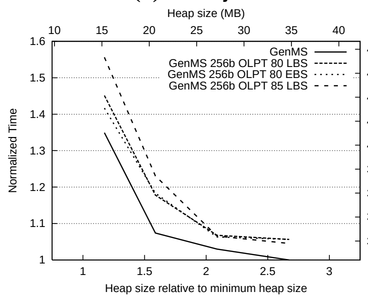
(b) _213_javac



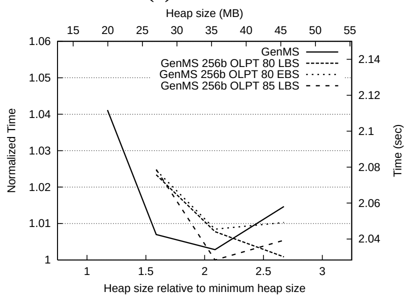
(c) _209_db



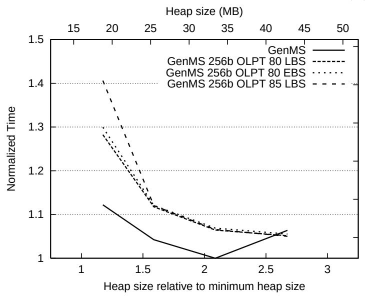
(d) _227_mtrt



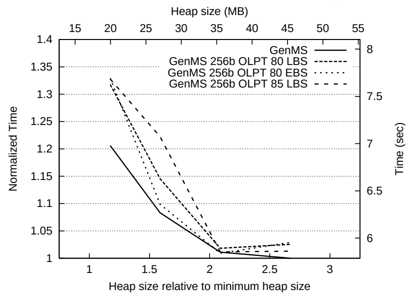
(e) _228_jack



(f) _205_raytrace



(g) _202_jess



(h) _201_compress

Figure 12: Total Execution Time

based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[14] T. L. Harris. Dynamic adaptive pre-tenuring. In *ACM International Symposium on Memory Management*, pages 127–136, Minneapolis, MN, October 2000.

[15] W. Huang, . Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *Proceedings for the 2004 IEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 133–140, Austin, TX, March 2004.

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

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

[18] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, San Jose, CA, November 1998.

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

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

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

[22] D. Stefanović, K. S. McKinley, and J. E. B. Moss. On models for object liftime distributions. In *ACM International Symposium on Memory Management*, pages 137–142, Minneapolis, MN, October 2000.

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

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

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