Profile-Based Pretenuring

STEPHEN M BLACKBURN
Australian National University
MATTHEW HERTZ
University of Massachusetts, Amherst
KATHRYN S McKINLEY
University of Texas at Austin
J ELIOT B MOSS
University of Massachusetts, Amherst
TING YANG
University of Massachusetts, Amherst

Pretenuring can reduce copying costs in garbage collectors by allocating long-lived objects into regions that the garbage collector will rarely, if ever, collect. We extend previous work on pretenuring as follows. (1) We produce pretenuring advice that is neutral with respect to the garbage collector algorithm and configuration. We thus can and do combine advice from different applications. We find for our benchmarks that predictions using object lifetimes at each allocation site in Java programs are accurate, which simplifies the pretenuring implementation. (2) We gather and apply advice to both applications and Jikes RVM, a compiler and run-time system for Java written in Java. Our results demonstrate that building combined advice into Jikes RVM from different application executions improves performance regardless of the application Jikes RVM is compiling and executing. This *build-time* advice thus gives user applications some benefits of pretenuring without any application profiling. No previous work uses profile feedback to pretenure in the run-time system. (3) We find that application-only advice also consistently improves performance, but that the combination of build-time and application-specific advice is almost always noticeably better. (4) Our same advice improves the performance of generational, Older First, and Beltway collectors, illustrating that it is *collector neutral*. (5) We include an *immortal* allocation space in addition to a nursery and older generation, and show that pretenuring to immortal space has substantial benefit.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*memory management(garbage collection)*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Garbage collection, pretenuring, lifetime prediction, profiling

1. INTRODUCTION

Garbage collection (GC) is a technique for storage management that automatically reclaims unreachable program data. In addition to sparing the programmer the effort of explicit storage management, garbage collection removes two sources of programming errors: memory leaks due to missing or deferred reclamation; and memory corruption through dangling pointers

This material is based upon work supported by the National Science Foundation under grant numbers CCR-0085792, CCR-0311829, CCR-0311829, and EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

because of premature reclamation. The growing use and popularity of Java and C#, in which garbage collection is a required element, makes attaining good collector performance key to good overall performance. Here our goal is to improve collector performance by reducing GC costs for long-lived objects. We focus on *generational copying collection* [Appel 1989; Lieberman and Hewitt 1983; Ungar 1984] and demonstrate the generality of our approach using the *Older First* [Stefanović et al. 1999] and *Beltway* [Blackburn et al. 2002] collectors.

Generational copying GC partitions the heap into age-based generations of objects, where age is measured in the amount of allocation (the accepted practice in the GC literature). Newly allocated objects go into the youngest generation, the *nursery*. Collection consists of three phases: (1) identifying roots for collection; (2) identifying and copying into a new space any objects transitively reachable from those roots (called "live" objects); and (3) reclaiming the space vacated by the live objects. Rather than collecting the entire heap and incurring the cost of copying all live objects, generational collectors collect the nursery, place survivors in the next older generation, and collect successively older generations only if necessary. Because the rate of death among the young objects is typically high in object-oriented languages, generational collectors usually offer performance advantages over full heap collectors (this property is called the *weak generational hypothesis*).

Pretenuring allocates some objects directly into older generations. If pretenured objects are indeed long-lived, then the pretenuring avoids copying the objects from the nursery into the generation where they are allocated. An ideal pretenuring algorithm would inform the allocator of the exact lifespan of a new object, and then the allocator would select the ideal generation in which to place the object. The collector would thus consider an object only after it has sufficient time to die, avoiding ever copying it. If an object will die before the next nursery collection, then the allocator would place it in the nursery (the default), whereas if the object lives until the termination of the program, then the allocator would place it into a permanent region.

We develop pretenuring advice from application profiling, on a per allocation-site basis. For our suite of Java programs, we show that allocation-site advice results in accurate predictions, and that these predictions are robust over different input data. In contrast, languages such as C require calling context to produce accurate predictions [Barrett and Zorn 1993; Seidl and Zorn 1998]; Section 8 discusses these alternative prediction mechanisms.

We extend the approach of Cheng, Harper, and Lee (CHL) [Cheng et al. 1998], whose work inspired our research. Firstly, our advice generation process classifies each object as *immortal*—its time of death was close to the end of the program, *short lived*—its lifetime was less than a threshold value, or *long lived*—everything else. CHL instead classify objects (allocated at a particular allocation site) that usually survive a nursery collection in a generational collector as *long lived*, and those that do not as *short lived*. Secondly, CHL profile a given application and generational collector configuration (including a specific heap size) to generate pretenuring advice. We instead use precise object allocation traces, obtained using the Merlin precise trace generation tool [Hertz et al. 2002; 2005], to generate lifetime statistics from which we derive our advice, a more costly, but offline, process. Because these statistics are collector- and configuration-neutral, they are more general, which our experimental results confirm. Finally, we *normalize* our statistics according to the application's maximum volume of live objects and its total allocation, making our advice more scale-invariant.

The generality of our pretenuring advice results in two key advantages over previous work. (1) Since we normalize advice with respect to total allocation for a specific execution, we can and do combine advice from different applications that share allocation sites (e.g., classes in-

ternal to the JVM, and libraries). (2) We can and do use the advice to improve three distinct collectors that segregate objects based on their age: an Appel-style generational collector [Appel 1989], an Older First collector [Stefanović et al. 1999], and the Beltway collector [Blackburn et al. 2002], on ten benchmarks, eight from SPECjvm98.

In our experiments, we use Jikes RVM (formerly called Jalapeño) [Alpern et al. 1999; Alpern et al. 2000], a compiler and run-time system for Java written in Java, extended with the garbage collectors we investigate. We profile all our benchmarks, and then combine their pretenuring advice to improve the performance of Jikes RVM itself; we call this system *build-time pretenuring*. This advantage is unique to the Java in Java implementation, whereas C JVMs instead must manually manage their data structures. When measuring the effectiveness of our build-time pretenuring, we omit the application itself from the combined advice profile. Such advice is called *true* advice [Barrett and Zorn 1993].

We show that build-time pretenuring improves the performance of Jikes RVM running our benchmarks an average of 30% for tight heaps without any application-specific pretenuring. As the heap size grows, the impact of garbage collection time and pretenuring on total execution time decreases, but pretenuring still improves collector performance. Because CHL profile advice is specific to both the application and collector configuration, their system cannot readily combine advice for this purpose. Building pretenuring into the JVM before distribution means users will benefit from pretenuring without profiling their applications.

Using only our application-specific profile advice always improves performance, too: up to 10% on average for tight heaps. Our advice also yields on average significantly better performance than CHL advice, giving more than 10% improvement in tight heaps and 5% in large heaps. Combining our build-time and application-specific advice always yields the best performance: it decreases garbage collection time on average by 40% to 70% for most heap configurations. It improves total execution time on average by 36% for a tight heap.

We organize the remainder of the paper as follows. Section 2 offers some background on pretenuring and its expected benefits and costs. Section 3 discusses our approach to pretenuring and the collection and generation of pretenuring advice. Section 4 analyzes the lifetime behaviors of objects in our Java applications. We then describe our performance methodology and setting in Section 5. Section 6 presents execution time and related measurement results for pretenuring with generational collection for Jikes RVM at build-time, application-specific pretenuring with CHL and our advice, and the combination of application-specific and build-time advice. We further demonstrate the generality of our advice by showing the same advice improves an Older First collector and a Beltway collector. We consider issues of using pretenuring in practice (Section 7), compare related work with our approach (Section 8), and conclude (Section 9).

2. THE PRETENURING COLLECTOR, EXPECTED BENEFITS AND COSTS

For this work, we built an Appel-style generational collector [Appel 1989] that partitions the heap into a nursery and a second, older, generation. It also has a separate, permanent space (which we call *immortal*) that is never collected. The nursery size is *flexible*: it is the space not used by the older generation and the permanent space. We fix the total heap size to make fair comparisons. Some heap space is always reserved for copying (this space must be at least as large as sum of the nursery and the older generation in order to guarantee that collecting the nursery and then the older generation will not fail). When all but the reserved heap space is consumed, the collector collects the nursery, promotes surviving objects into the older generation, and makes the freed space the new nursery. After a nursery collection, if the old

generation's size is close to that of the reserved space, it triggers collection of the older generation. We call this collector *Appel* as a convenient shorthand and to emphasize its varying-size nursery, but one should keep in mind that it is just in the general style of Appel's original collector.

Expected benefit of immortal space. Long-lived objects allocated into immortal space avoid all copying, both the first copy from the nursery into the older generation, and the copy made each time we collect the older generation. There is also a space benefit. Because we never collect the immortal space, we need not reserve additional space into which to copy it, which frees space for use by the nursery and older generation.

Expected cost of immortal space. We never reclaim objects allocated in the immortal space, so if we pollute the space with objects that die quickly we effectively reduce the heap size (possibly running out of space entirely). However, we can tolerate some pollution because each object in immortal space commits half the space it would take if allocated elsewhere. A more subtle effect is that a short-lived object allocated in immortal space can cause retention of objects reachable from it. This effect is known as *nepotism* [Ungar and Jackson 1988]. It does not appear to occur very often, but suggests being conservative in pretenuring.

Expected benefit of old generation pretenuring. We save the work of copying the object from the nursery, if it survives nursery collection.

Expected cost of old generation pretenuring. If the object is shorter-lived and would have been reclaimed by a nursery collection, we pollute the older generation and cause an old generation collection sooner than we otherwise would. Nepotism may also occur.

It would appear that the space and time benefits of immortal space, when it is a good choice, are much larger (on a per-object basis) than those of old generation pretenuring. The overall benefit depends, of course, on the relative volume of short-, medium-, and long-lived objects, and whether their allocation occurs in patterns we can exploit.

Although we use the Appel-style generational collector here to motivate and describe pretenuring, our approach is general. We describe the application of pretenuring to two other collectors, Older First and Beltway. Similar benefits should accrue to parallel and concurrent collectors in terms of overall GC effort, perhaps reflected in higher throughput, fewer rounds of GC, better memory utilization, etc.

3. PRETENURING ADVICE METHODOLOGY

Two objectives are central to our approach: producing robust and general pretenuring advice, and understanding and testing the premise of per-site lifetime homogeneity on which the success of profile-driven pretenuring rests.

3.1 Gathering Information and Generating Pretenuring Advice

Any algorithm for generating pretenuring advice must consider the two major cost components: *relative copying costs* and *relative space consumption*. The copying cost includes scanning and copying an object when it survives a collection. Space cost has an indirect impact in that higher space overhead forces more frequent GCs. One way to conceptualize space cost is in terms of *space rental*: the space required by an object times the length of time it uses that space. On the two extremes, pretenuring advice that recommends pretenuring *all* objects into permanent space minimizes copying costs but increases space rental; and advice that recommends pretenuring *no* objects tends to minimize space rental at the expense of higher copying costs.

One of our goals is to generate advice that is neutral with respect to any particular collection algorithm or configuration. This goal precludes the use of the metric used by CHL [Cheng et al. 1998], which pretenures if the collector usually copies objects allocated at a particular site in the context of a specific generational collector configuration. Our approach is instead based on two fundamental object lifetime statistics: *age* and *time of death*. Object age indicates how long an object lives, and time of death indicates the point in the allocation history of the program at which the object becomes unreachable.

We normalize age with respect to *max live size*, following the garbage collection convention of equating time to bytes allocated. *Max live size* refers to the maximum volume (bytes) of live objects in a program execution, which indicates the *theoretical* minimum memory requirement of a program. This normalization will reduce differences between different runs of the same program where the size of the program's heap data structures is different. Object age is expressed as a fraction or multiple of the max live size. For example, an age of 0.25 means that during the lifetime of the object, $0.25 \times$ max live size bytes of allocation occurred.

We normalize time of death with respect to total allocation.[1] For example, consider an object allocated toward the end of the program that dies after the last allocation. It has a normalized time of death of 1.00. This normalization has the same intent as the one we apply to object age: to reduce differences in characterizing different runs of the same program, and thus to make our characterizations and our advice more independent of scale.

We illustrate the relationships between object age, time of death, max live size, and total allocation in Figure 1 for a Java version of health [Cahoon and McKinley 2001; Rogers et al. 1995] running a small input set, where we plot one point for each age and time of death combination that has a volume of objects exceeding a chosen threshold.[2] The bottom and right axes normalize "time" with respect to total bytes allocated for that program, while the top and left axes show time with respect to the program's max live size, which relates to a "heap full" of allocation. Note that the scales on opposite sides (e.g., top and bottom) are only showing normalization to different units. For the illustrated run, a point at (7,2) in terms of max live size is at about (0.77,0.22) in terms of total allocation. Such a point represents an object that died 77% of the way through the run (in terms of bytes allocated) and whose age was 22% of total allocation (and hence was allocated 55% of the way through the run).

This figure shows that a large number of objects have short lifetimes, and the horizontal "lines" of points indicate that throughout the execution of the program objects are most likely to die when they reach one of a small number of ages (for example about 0.2 and $0.45 \times$ max live size).[3] There are also times of death at which many objects die simultaneous, which appear as vertical "lines" in the figure.

The figure also illustrates how our object classification, discussed in detail in Section 3.1.2, puts objects into short-lived, long-lived, and immortal "bins".

3.1.1 *Object Lifetime Profiling.* We analyze age and lifetime statistics using an execution profile for each application. We obtain the profile by producing a precise object allo-

[1] The relationship between max live size and total allocation is a function of allocation behavior. In our Java programs, total allocation ranges from 9 to 91 times max live size.

[2] Plotting a point for every object obscures where the scatter plot is more and is less dense.

[3] This effect is particularly evident in health: it places objects in a queue and processes them and discards them in FIFO order (or requeues them). Thus these objects tend to have uniform lifetimes. For many other programs it would be more common to see clustering of time-of-death: data structures built over time and then discarded at a particular point in execution.

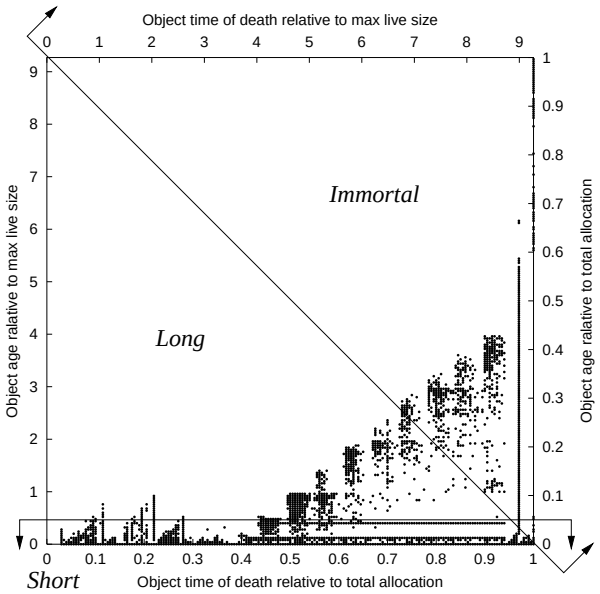


Fig. 1. Object Age and Death Distributions for health (6-128)

cation and death trace. We produce these traces using the Merlin tool [Hertz et al. 2002; 2005]. Merlin produces precise traces at much lower cost than previous approaches, making precise traces possible where in the past they were infeasible.[4] In an earlier version of this work [Blackburn et al. 2001], we instead did full-heap collections after every 64KB of allocation, over-estimating lifetimes by around 32KB on average. Although this approach leads to few classification errors, it requires us to adjust our pretenuring advice strategy a little because it distorts the space-rental calculations that indicate candidate sites for pretenuring. We ended up abandoning space-rental as our primary measure of site importance and now use allocation volume.

An object lifetime trace gives a sequence of object allocation and object death records, including the time of allocation, time of death, size of the object (particularly relevant for arrays, since the size may not be known until run time), and the allocation site. An allocation site corresponds to a particular new bytecode, i.e., Java class, method, and bytecode offset within the method. Since inlining can vary from run to run in an adaptive and dynamically compiled system, if inlining induces cloning of allocation sites, we group their statistics together (i.e., it is *as if* the method were not inlined). This combining improves advice across different applications, but may conflate distinct behaviors (though our results suggest that this issue is not significant for the programs we investigate).

From the trace we compute max live size, total allocation, and the normalized birth and death times for each object.

[4]The slowdown factor to produce perfectly accurate traces with Merlin is about 75–500x; for traces at a granularity of 4K bytes, which we believe sufficient for pretenuring judgments, the factor is 20-80x. One must also analyze the traces, which at present is slow because we have not invested in building fast analysis programs.

3.1.2 *Object Classification.* For each object allocated at a given site, we classify it into one of three bins: *short*, *long*, or *immortal*. We use the following algorithm:

(1) If an object dies later than halfway between its time of birth and the end of the program, we classify it as *immortal*.

(2) Otherwise, if an object's age is less than $T_a \times$ max live size bytes, then we classify it as *short*. We use $T_a = 0.45$ in our experiments below.[5]

(3) In all other cases, we classify an object as *long*.

Our *immortal* classification criterion is based on our previously noted observation that objects that will never be copied have a lower space requirement than objects that may be copied: the latter must have space reserved into which to copy them. Because in an Appel-style generational collector, the reserved space overhead is 100% (half the heap), it is reasonable to classify an object as immortal if *dead time* $\leq$ *lifetime* for that object, where dead time is the time from when the object dies to the end of the program.[6] Figure 1 illustrates this categorization. Note that one could use a different threshold value, but this threshold has a good intuitive motivation, and it also turns out that varying the threshold has little impact because few objects have values lying close to the threshold.[7]

Of course, our immortal category is heuristic. The following scenario is possible. We allocate object A near the start of the run and it dies a bit after the middle of the run, so it is classified immortal. Shortly before A dies, we make it point to some large recently allocated data structure B, which dies when A does (or shortly thereafter). Classifying A immortal causes B to be effectively immortal as well, an extreme case of nepotism. Such scenarios appear to be exceedingly rare. Section 7 considers ways to ameliorate this potential problem. If we increase the threshold for designating objects immortal, we will tend to reduce the magnitude of this problem (should it occur), but we also reduce the benefit of pretenuring.

3.1.3 *Allocation Site Classification.* Having classified each *object*, we then classify each *site*. Given an allocation site s that allocates a fraction S_s of short-lived objects, L_s of long-lived objects, and I_s of immortal objects, where S_s, L_s, and I_s are in terms of volume, i.e., bytes allocated (not number of objects),[8] we classify the site using homogeneity thresholds H_{if} and H_{lf}, as follows:

(1) If $I_s > S_s + L_s + H_{if}$, we classify the site as *immortal*.
(2) Otherwise, if $I_s + L_s > S_s + H_{lf}$, we classify the site *long*.
(3) In all other cases, we classify the site *short*.

[5]Previously [Blackburn et al. 2001] we used $T_a = 0.2$, however have we found that 0.45 works better since 0.2 corresponds to a very modest nursery, while 0.45 is more realistic. 0.2 is overly aggressive for pretenuring.

[6]The same principle applies to any collector with a non-copied immortal space and a need to reserve copying space for one or more younger generations. In particular it applies to the Older First and Beltway collectors with which we compare later.

[7]Previously we first separated out the *short* category and then discriminated *long* versus *immortal* [Blackburn et al. 2001]. We found that the new order works better because objects living until near the end of the program tend to be allocated at sites that allocate immortal objects. Classifying these objects *short* caused us to miss sites we should treat as *immortal*.

[8]This approach refines our previous work [Blackburn et al. 2001] and is more accurate for arrays whose sizes at the same allocation site can differ. We also tried space rental (size times lifetime) as a way to weight objects, but this over-emphasizes long-lived objects.

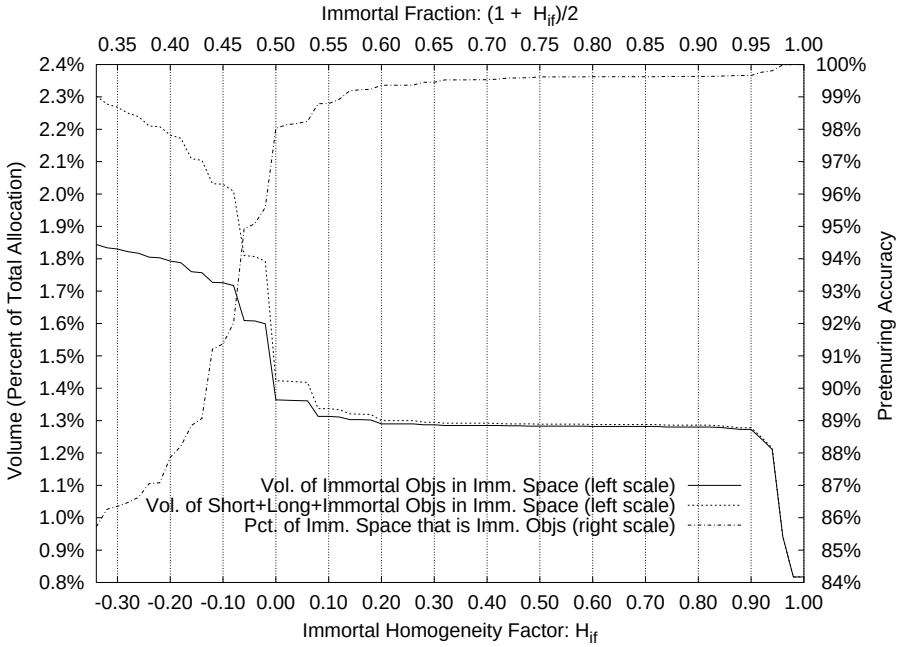


Fig. 2. Immortal Classification Accuracy by Volume—Geometric mean for all benchmarks

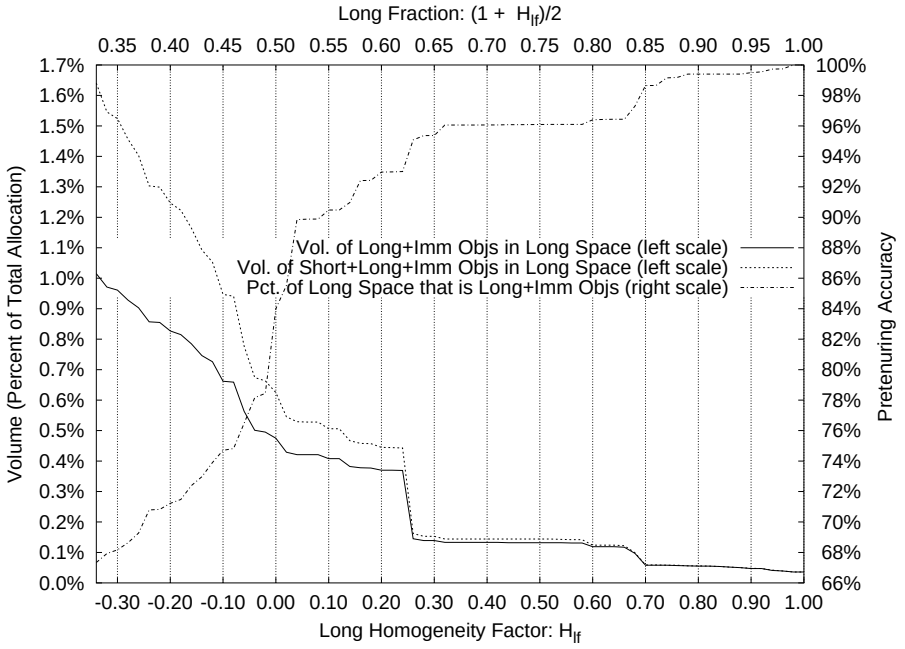


Fig. 3. Long Classification Accuracy by Volume—Geometric mean for all benchmarks ($H_{if} = 0.00$)

The homogeneity thresholds control the "aggressiveness" of the classification. For example, if $H_{if} = 0$, then we will classify a site *immortal* if the majority of the objects allocated there are *immortal*. If $H_{if} = 0.99$, then we require virtually all objects to be immortal ($I_s > 0.995$).

In our previous work, we used a single homogeneity factor, H_f, but (as we shall show) the cost/benefit factors are quite different for classifying *immortal* versus classifying *long*. If an object does live a very long time, we save significant CPU time when we classify it *immortal* because we avoid copying it not only in the initial nursery collection, but also in all later full-heap collections. Further, the space savings from having no copy reserve for the *immortal* region stave off future GCs (i.e., in effect it increases the heap size). On the other hand, if we classify a site as *long*, all we save is one copying of the object out of the nursery, and we actually *reduce* effective available space because we cannot reclaim the object except through a full-heap GC.

We now consider how to pick a good value for H_{if}. If H_{if} is too low, then too many sites will be classified immortal, causing too many non-immortal objects to be allocated in immortal space; if H_{if} is too high, then too few sites will be classified immortal, causing too little immortal allocation in immortal space to gain benefit. In Figure 2 we vary H_{if} from -0.33 to 1.00. (It is possible for I_s to be larger than S_s and L_s in the region $[-0.33, 0]$ even though $I_s < 0.50$. See the labels on the top x-axis.) The figure includes three curves. Consider first the curve labeled "Vol. of Immortal Objs in Imm. Space", whose scale is on the left y-axis. This curve gives the geometric mean (across benchmarks) of the ratio: volume of immortal objects that would be allocated into immortal space for a given value of H_{if} / total allocation. We see that this volume is very insensitive to H_{if} for values from 0.0 to 0.9. Going farther to the right will reduce the volume (and thus the potential benefit). Now consider the curve labeled "Vol. of Short+Long+Immortal Objs in Imm. Space", whose scale is also on the left y-axis. It shows the ratio: *total volume* of objects allocated into the immortal space for each value of H_{if} / total allocation. We see that it stays close to the first curve except for $H_{if} < 0$. The third curve shows the *accuracy* of the pretenuring, i.e., the ratio: total volume of immortal objects allocated in immortal space / total allocation in immortal space (the ratio of the two previous curves). Its scale is on the *right* y-axis. It shows quite clearly that accuracy drops off rapidly for $H_{if} < 0$.

One desires maximum benefit (greatest volume, hence smallest H_{if}) consistent with adequate accuracy (low accuracy "pollutes" immortal space and is risky, since we never reclaim the "polluting" objects). We use $H_{if} = 0.0$ from here on, and it seems to make this trade-off well, though values between 0.0 and 0.9 should all work about as well.

Now that we have fixed H_{if}, we consider the effect of H_{lf}. Figure 3 is similar to Figure 2 in its structure (but note the difference in vertical scales). However, it ignores sites already classified as immortal using $H_{if} = 0.0$, and considers only the short/long trade-off for the remainder of the sites (and objects). The first curve shows the fraction of long-lived objects (long+immortal) actually allocated into long space for each value of H_{lf}. (Again, we plot the geometric mean of this value across the benchmarks.) As with H_{if}, we find that there is a long flat region. In terms of accuracy, any value of H_{lf} greater than 0.25 should be all right, but since the benefit of long pretenuring is small, we demand high accuracy. We use $H_{lf} = 0.60$ in the remainder of the paper. We observe that, compared with immortal sites, the homogeneity of the (remaining) sites where long-lived objects dominate is not as good.

3.1.4 *Pruning Allocation Sites.* Finally, we drop sites whose total allocation is small, i.e., less than v times the total allocation of the program.[9] We used $v = 0.000002$. Our primary reason for doing this is that allocation advice for a site takes a certain amount of dynamically allocated table space in the JVM, effectively reducing the heap size, so we should drop sites whose pretenuring will have very little total effect. One can also claim that when the volume of a site is relatively low, we do not have adequate evidence to pretenure that site's objects.

3.1.5 *Combining Classifications from Different Program Executions.* We also combine data from different program executions to generate pretenuring advice. Our trace combining algorithm works as follows. For each site s, we generate new combined bins $S_{c,s}$, $L_{c,s}$, $I_{c,s}$. For each trace t, we first compute a weight w_t for each site: $w_t = v_s/v_t$, where v_s is the volume allocated at the site, and v_t is the total volume of allocation in the trace. We then compute the combined bins using weighted averages for all sites with trace information. Let $w_c = \sum_{t=1}^{n} w_t$. We use $S_s(t)$ to mean the value of S_s for trace t, etc. We show only the formula for $S_{c,s}$; $L_{c,s}$ and $I_{c,s}$ are computed analogously:

$$S_{c,s} \;=\; (\sum_{t=1}^{n} S_s(t) * w_t)/w_c$$

With these bins, we then use the same classification algorithm as above but with a different homogeneity factor. Unlike the case of H_{if} and H_{lf}, when combining information across traces (programs), we found it important to be conservative for both immortal and long advice. Therefore we use a single homogeneity factor, called H_{cf}, which we set to 0.9.

3.2 Jikes RVM Builds and Compilation Strategies

In our previous work [Blackburn et al. 2001], we used an optimization strategy in Jikes RVM that optimizes *every* method to the highest available optimization level. We call this "build" of the system *Opt*. Optimizing every method is not realistic for modern JVMs, because it performs much optimization of "cold" methods that does not pay back. In Jikes RVM (because it is written in Java) it also induces much additional heap allocation and increases GC load. Thus using Opt will tend to bias towards pretenuring for the compiler, which performs well but may miss opportunities in individual applications. We always optimize to the highest level the methods included in the system image, but treat application methods differently since they are compiled at run time in this methodology.

In contrast to Opt, the typical compilation strategy today is *adaptive*. For example, in Jikes RVM the *Adaptive Optimization System* (AOS) [Arnold et al. 2000] detects, as the program runs, which methods the application uses most frequently, and compiles those at progressively higher levels of optimization. It determines highly-used methods via sampling triggered by timers. Thus the AOS is non-deterministic (because it is timing dependent), making it somewhat problematic for experimentation where we wish to vary only one factor at a time.

Hence we developed a new *replay* approach.[10] Here we run an application a number of times (say 7) and determine, for each method, the highest optimization level to which the

[9] In our previous work, we ranked sites according to their total space rental, i.e., sum of (object size) × (object lifetime) across all objects allocated at the site [Blackburn et al. 2001]. This mechanism includes some low-volume (but high space rental) sites, particularly with perfect traces (because they dramatically reduce the reported lifetimes of most (short-lived) objects). Using volume is thus a better choice.

[10] Xianglong Huang and Narendran Sachindran jointly implemented the Replay compilation mechanism. This technique was previously termed "Pseudo-Adaptive" [Huang et al. 2004], but "Replay" is more suggestive of its function.

method is optimized in a majority of the runs. We put this information in an *advice file*. The Replay system reads the advice file, and when it first compiles a listed method, optimizes it directly to the advised level. (If there is no advice, it compiles using the simple, non-optimizing compiler.) It suppresses all adaptive recompilation. The effect is that the total compilation load is very similar to a typical Adaptive run, but the system is deterministic.

We present the bulk of our results using the Replay methodology, but also present the effects of pretenuring using the Adaptive and Opt compilation strategies in Section 6.8. As expected, Adaptive and Replay builds behave quite similarly, with and without pretenuring, but Opt builds allocate much more in the heap, and more often the compilation work, both in time and space, outweighs the application work.

By default, we profile an Opt build to produce the pretenuring advice, but Section 6.8 shows that using Replay builds for advice instead produces comparable accuracy.

4. PRETENURING ADVICE RESULTS

Profile-driven pretenuring is premised on homogeneous object lifetimes at each allocation site. Previous work shows that ML programs are amenable to a classification of sites as short and long, where long means "usually survives one nursery collection" (for a specific system configuration) [Cheng et al. 1998]. C programs are not homogeneous at each call site, but require the dynamic call chain to predict similar classes of lifetimes [Barrett and Zorn 1993; Seidl and Zorn 1998]. We show in this section that the allocation sites in our set of Java programs have adequately homogeneous lifetimes, with respect to our classification scheme, for pretenuring to work reliably.

4.1 Benchmark Programs

For evaluating both classification (here) and performance (Section 6), we use all eight programs from the SPEC JVM98 suite: _210_compress, _202_jess, _205_raytrace, _209_db, _213_javac, _222_mpegaudio, _227_mtrt, and _228_jack, plus pseudojbb[11] and health, the Olden C program that models a health care system [Cahoon and McKinley 2001; Rogers et al. 1995] rewritten in object-oriented Java. We run all benchmarks single-threaded.

Table I shows the total allocation in bytes, maximum live size in bytes, and the ratio between the two, for each benchmark, under the Opt and Replay configurations. The maximum live sizes are mostly similar, but the total allocation volume often differs a lot.

4.2 Homogeneity of Applications

The homogeneity of an allocation site can be defined using the information theoretic notion of *entropy*. Using bits as the unit, the entropy of a set of discrete probabilities P_j is:

$$\textit{entropy} \;=\; -(\sum_j P_j * \log_2 P_j), \qquad \text{where } \sum_j P_j = 1$$

Smaller entropy implies higher homogeneity, i.e., fewer bits needed to encode the labels (immortal, long, short) on a set of objects drawn in random order with these probabilities. If an allocation site is completely homogeneous, 100% with one label and 0% in others, its entropy is 0.00. If an allocation site is completely heterogeneous, 50% and 50% (in two categories),

[11] SPECjbb runs a fixed period of time and reports the number of iterations it executes, a throughput measure. We changed it to run a fixed number of transactions (70,000) and call the resulting program pseudojbb. It thus produces the same allocation load regardless of heap size, collector, etc. Its execution times are on the order of 10 seconds on our platform.

Benchmark	Opt runs			Replay runs		
	Max Live (bytes)	Alloc (bytes)	Alloc / Max Live	Max Live (bytes)	Alloc (bytes)	Alloc / Max Live
compress	8,826,084	199,944,756	22	8,819,296	116,641,428	13
jess	5,485,280	482,996,388	88	4,508,272	299,788,860	66
raytrace	6,839,684	233,821,460	34	6,863,452	124,286,536	18
db	10,709,640	178,830,988	16	10,732,380	86,687,156	8
javac	12,068,700	618,946,020	53	12,146,436	298,486,240	24
mpegaudio	4,410,732	134,921,104	30	3,599,032	27,684,656	7
mtrt	9,923,760	247,688,648	24	2,570,348	39,690,456	15
jack	5,810,536	533,734,388	91	3,947,152	346,126,536	87
pseudojbb	29,913,388	636,525,664	20	30,254,784	365,554,384	12
health (6-128)	4,349,588	40,283,616	9	4,163,776	29,013,560	6

Table I. Benchmark Characteristics: *Max Live* is the maximum live size and *Alloc* is total allocation.

its entropy is 1.00. Here is how we calculate site entropy when we consider short, long, and immortal labels:

$$\begin{aligned} short: &\ -(S_s * \log_2 S_s) - (L_s + I_s) * \log_2(L_s + I_s) \\ long: &\ -(L_s * \log_2 L_s) - (S_s + I_s) * \log_2(S_s + I_s) \\ immortal: &\ -(I_s * \log_2 I_s) - (L_s + S_s) * \log_2(L_s + S_s) \end{aligned}$$

Figure 4 shows the homogeneity curves of the geometric mean over all benchmarks before pretenuring, varying entropy from 0 to 1. We call those sites for which $I_s > L_s$ and $I_s > S_s$ *immortal_dom* sites (i.e., where *immortal* objects dominate the other two categories), and similarly we have *long_dom* and *short_dom* sites. For each entropy value, we calculate the total allocation volume of sites whose entropy is less than or equal to that value. There are three curves in the graph, one for immortal_dom sites, one for immortal_dom plus long_dom sites, and one for all sites. We normalize all volumes to the total allocation of the application. We use the right y-axis for the scale of the top curve (for all sites), and the left y-axis for the other two curves. The bottom x-axis is the value of entropy, and the top x-axis shows the corresponding fraction of the dominating category. The flatness of the immortal_dom curve in Figure 4 shows us that immortal_dom sites have extremely high homogeneity, most possessing entropy less than 0.2: immortal objects make up at least 96.9% of the volume of these sites. On the other hand, the steep increase at the right end of the immortal_dom plus long_dom curve tells us that a good portion of long_dom sites are *not* homogeneous. We should definitely not pretenure those sites. The homogeneity of all sites is pretty high, more than 90% have more than 85% of one kind, short, long, or immortal.

Figure 5 shows the homogeneity curves of the geometric mean over all benchmarks *after pretenuring*. Note that "before pretenuring" data have to do with sites classified by which lifetime dominates (immortal, long, or short), whereas "after pretenuring" data have to do with how we have *labeled* the sites, not which lifetime dominates by allocation volume. Our hope is that we choose only very homogeneous sites to pretenure. We plot this graph (Figure 5) according to our site classification using $H_{if} = 0.00$ and $H_{lf} = 0.60$. It also has three curves: immortal, immortal plus long, and all sites. This graph is almost the same as Figure 4, except that the jump in the right end of the immortal plus long curve nearly disappears, indicating that our pretenuring method is effective in filtering out the heterogeneous sites, and chooses to pretenure only sites with high homogeneity.

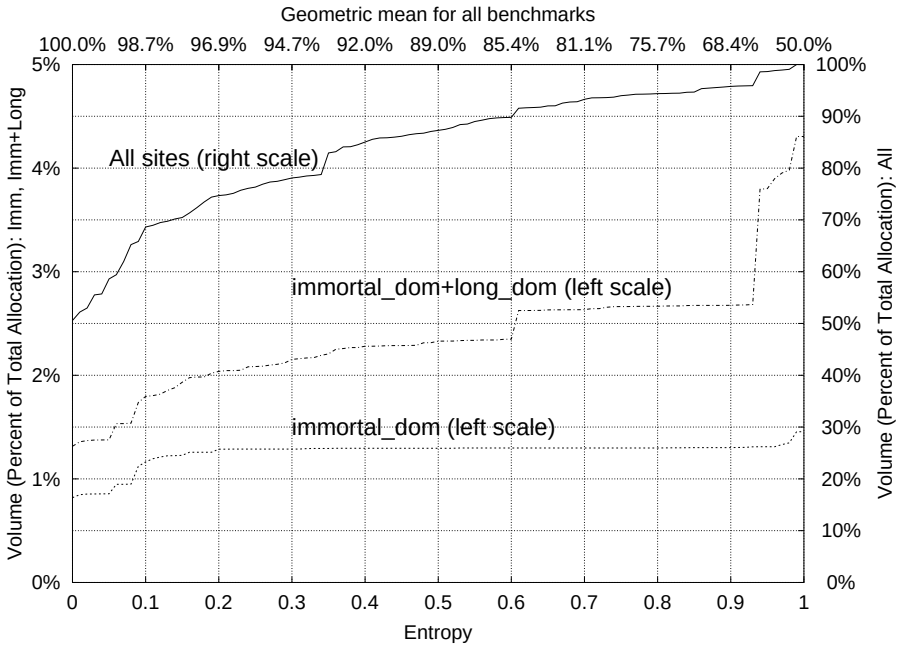


Fig. 4. Homogeneity before Pretenuring

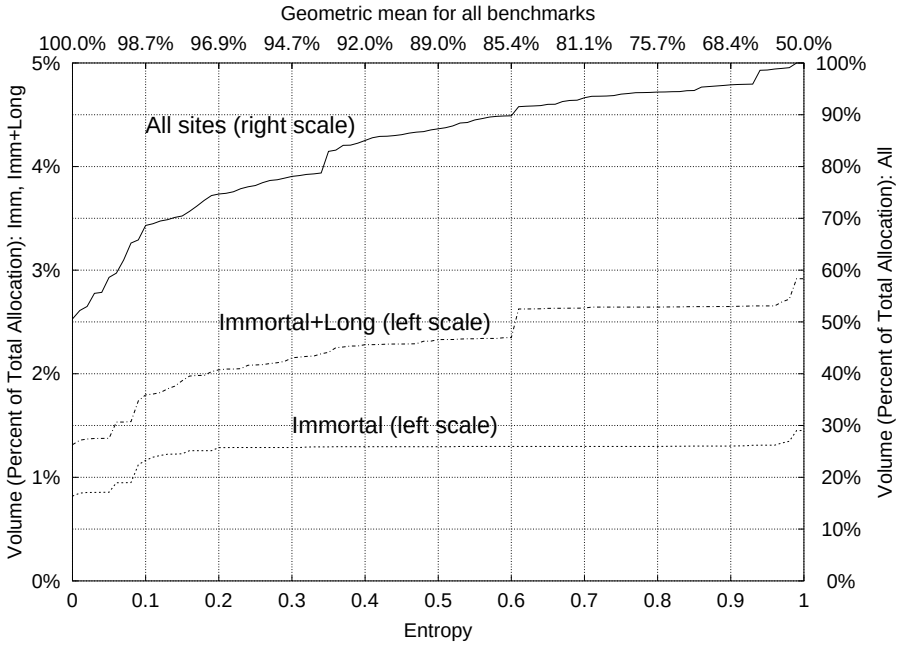


Fig. 5. Homogeneity after Pretenuring: $H_{if} = 0.00$ $H_{lf} = 0.60$

site #	objects	volume	vol %	bin %			classi-fication
				short	long	immortal	
javac							
1676	145492	60421512	9.761	99.59	0.04	0.37	s
1064	1496486	47887552	7.739	100.00	0.00	0.00	s
13	759989	32802440	5.300	92.84	0.04	2.98	s
1501	654754	20952128	3.385	100.00	0.00	0.00	s
692	602886	19588556	3.141	97.13	2.46	0.40	s
3269	145636	4077808	0.659	6.87	75.38	17.75	l
3278	49812	1793232	0.290	4.07	62.94	32.98	l
3296	40156	1766864	0.285	5.45	61.81	32.74	l
4126	45372	1633392	0.264	11.04	74.65	14.30	l
3326	96696	1547136	0.250	6.47	84.76	8.77	l
1747	5523	829228	0.133	0.83	2.34	96.84	i
551	157	590276	0.095	0.00	0.00	100.00	i
662	5	327740	0.055	0.00	0.00	100.00	i
529	1617	174636	0.028	0.00	0.00	100.00	i
1780	1	163852	0.027	0.00	0.00	100.00	i
combined							
1070	7044399	225420768	15.764	100.00	0.00	0.00	s
1513	3096079	99074528	6.928	100.00	0.00	0.00	s
693	2773149	89818452	6.281	92.08	6.86	1.06	s
848	2033521	65075872	4.551	89.25	9.41	1.34	s
747	956207	45897936	3.210	87.72	10.63	1.65	s
565	972	11664	0.001	4.83	1.77	93.40	l
25	10	380	0.000	2.52	2.52	94.95	l
454	10	360	0.000	2.63	2.63	94.74	l
324	20	320	0.000	2.94	2.94	94.12	l
353	20	320	0.000	2.94	2.94	94.12	l
1765	17118	4260740	0.298	1.05	0.53	98.43	i
664	52	3408496	0.238	0.00	0.00	100.00	i
555	638	2248540	0.157	0.00	0.00	100.00	i
726	10	1638520	0.115	0.00	0.00	100.00	i
727	10	1638520	0.115	0.00	0.00	100.00	i

Table II. Per-site Object Binning and Classification

We present two types of results in the remainder of this section. For the javac benchmark and for our combined advice, we illustrate our binning and classifications for a number of call sites in each. We then present aggregate advice summaries for each benchmark and the actual behavior of the sites to demonstrate the quality of our advice.

4.3 Detailed Classification Results

Table II shows some of our per-site object classifications for javac and for our combined advice for the Jikes RVM build-time system. We include the top 5 sites classified as immortal, top 5 long, and top 5 short. We rank these by their allocation volume.

We include the number and volume of objects the site allocates, and show the percentage of objects that are binned as short, long, or immortal. Using $T_a = 0.45$, $H_{if} = 0.00$, $H_{lf} = 0.60$, and $H_{cf} = 0.90$, we show our resulting classification. Notice that many allocation sites are

Program	Immortal Space %				Long Space %				Overall accuracy
	vol%	$\frac{i_o \wedge i_s}{i_s}$	$\frac{l_o \wedge i_s}{i_s}$	$\frac{s_o \wedge i_s}{i_s}$	vol%	$\frac{i_o \wedge l_s}{l_s}$	$\frac{l_o \wedge l_s}{l_s}$	$\frac{s_o \wedge l_s}{l_s}$	
compress	0.97	96.06	0.07	3.87	0.034	1.76	78.48	19.76	95.52
jess	0.68	98.23	0.40	1.37	0.001	3.58	96.42	0.00	98.23
raytrace	2.11	99.66	0.05	0.29	0.002	38.66	59.09	2.25	99.65
db	5.49	93.61	0.97	5.42	0.656	0.11	99.89	0.00	94.29
javac	0.65	98.41	0.49	1.10	4.468	24.31	69.75	5.94	94.61
mpegaudio	2.31	96.05	0.08	3.87	0.051	1.73	84.02	14.24	94.82
mtrt	0.95	99.29	0.09	0.62	2.074	47.88	52.12	0.00	99.78
jack	0.46	99.33	0.11	0.56	1.948	5.63	81.12	13.25	89.16
health	12.84	79.67	3.53	16.80	0.002	40.64	59.36	0.00	79.67
pseudojbb	0.56	96.96	2.30	2.74	3.417	49.76	50.00	0.24	99.37
Geo Mean		97.82	0.24	1.94		9.05	85.22	5.73	97.39

Table III. Per-program Pretenuring Decision Accuracy (percent, weighted by volume)

homogeneous: the majority of objects at a site are in a single bin. For some sites, especially in the combined trace, objects are well distributed among bins. For javac, we classify many sites as long (l), and in the combined trace, several sites as immortal (i). Thus, we find sites to pretenure into the long lived and immortal space.

Program	Immortal Objects %			Long Objects %				Overall %		
	vol%	$\frac{i_o \wedge (i_s \vee l_s)}{i_o}$	$\frac{i_o \wedge s_s}{i_o}$	vol%	$\frac{l_o \wedge l_s}{l_o}$	$\frac{l_o \wedge s_s}{l_o}$	$\frac{l_o \wedge i_s}{l_o}$	good	neut	bad
compress	1.26	74.13	25.87	18.02	0.15	99.85	0.00	4.99	95.01	0.00
jess	0.99	66.89	33.11	1.75	0.04	99.80	0.16	24.23	75.67	0.10
raytrace	2.71	77.66	22.34	2.24	0.06	99.90	0.04	42.51	57.47	0.02
db	5.62	91.42	8.58	1.75	37.42	59.54	3.04	78.59	20.69	0.72
javac	2.31	74.73	24.27	5.22	59.69	40.25	0.06	64.30	35.66	0.04
mpegaudio	2.75	80.66	19.34	5.77	0.75	99.22	0.03	26.55	73.43	0.02
mtrt	2.52	76.85	23.15	2.60	41.65	58.32	0.03	58.99	40.99	0.02
jack	0.82	79.70	30.30	4.85	32.59	67.40	0.01	27.94	62.05	0.01
health	10.89	93.94	6.06	1.08	0.10	57.96	41.94	85.48	10.74	3.78
pseudojbb	4.26	52.65	47.35	3.62	47.23	52.72	0.05	50.16	49.82	0.02
Geo Mean		78.18	21.82		3.18	96.68	0.14	46.16	53.78	0.06

Table IV. Per-program Pretenuring Decision Coverage (percent, weighted by volume)

To consider the issue of binning in summary form across all sites of a benchmark, we consider the fraction of short, long, and immortal objects that end up being allocated in short, long, and immortal space (as determined by our labels for the allocation sites). Where x and y range over s, l, and i (for short, long, and immortal, respectively), we define $x_o \wedge y_s$ to be the volume of x category objects allocated in y space. Thus $i_o \wedge s_s$ is the volume of immortal objects allocated in short space. Similarly, we define $\frac{x_o \wedge y_s}{y_s}$ to be the ratio of the volume of objects of category x allocated into space y to the total volume of objects allocated into space y. For example, $\frac{i_o \wedge i_s}{i_s}$ means the volume of immortal space allocation used for immortal objects (an accurate classification), while $\frac{s_o \wedge i_s}{i_s}$ means the volume of immortal space allocation used for short objects (an inaccurate classification). We change the denominator when we wish to

indicate disposition according to the labelling of objects rather than by space; thus $\frac{i_o \wedge s_s}{i_o}$ is the fraction of immortal object volume that is in short space, distinct from $\frac{i_o \wedge s_s}{s_s}$, which is the portion of short space consumed by immortal objects.

The nine decision pairs fall into three categories, *neutral*, *bad*, and *good*, with respect to the non-pretenured status quo. Neutral pretenuring advice allocates objects into the nursery ($s_o \wedge s_s$, $l_o \wedge s_s$, and $i_o \wedge s_s$). Bad pretenuring advice allocates objects into a longer lived region than appropriate ($s_o \wedge l_s$, $l_o \wedge i_s$, and $s_o \wedge i_s$). Following bad advice tends to waste space and induce more frequent collection. Good pretenuring advice allocates objects into longer lived regions, but not too long lived ($i_o \wedge i_s$, $l_o \wedge l_s$, and $i_o \wedge l_s$). Following good advice reduces copying without wasting space.

Table III gives these summary statistics for each benchmark and overall, stated as percentages. It also indicates the percent of total allocation volume that goes to the immortal and long spaces, and the percentage (by volume) of pretenured objects coming from the "good" categories. Put another way, of the volume of objects pretenured, it tells how much is "correct". We see that, with the exception of health, where we allocate a significant volume of short objects into immortal space, our accuracy is quite high.

A converse question is this: of the total volume of immortal (long) objects, i.e., allocated across *all* sites, what percentage do we pretenure? This we call the *coverage*, and we show it in Table IV. Put another way, this indicates how much of the volume of immortal and long objects ended up appropriately pretenured. The table uses the notation $\frac{i_o \wedge (i_s \vee l_s)}{i_o}$, which means the fraction of the immortal objects that are pretenured, into either immortal or long space (expressed as a percentage). While there is noticeable variation across benchmarks, on average we pretenure the bulk of immortal objects, around 80%. Because we are much more conservative about classifying sites as long, we do not pretenure a large fraction of long objects, only a few percent. Overall, we give mostly good and neutral advice and very little bad advice (even for health less than 4%).

Figure 6 shows how H_{if} affects the accuracy and the coverage, by fixing H_{lf} at 0.60 and varying H_{if} from -0.33 to 1.00. We use the left y-axis for the accuracy curve and the right y-axis for the coverage curve. This graph shows the geometric mean of all benchmarks. We see that the accuracy increases quickly to 98% as H_{if} goes up to 0.00, and then grows much more slowly. Although H_{if} has little impact on the coverage, we reach a maximum around $H_{if} = 0.00$. This is because we classify immortal sites before long sites: when H_{if} is small, we pull more long objects into immortal space.

Figure 7 shows the impact of H_{lf} by fixing H_{if} at 0.00 and varying H_{lf} from -0.33 to 1.00. Clearly, H_{lf} has much larger impact on the coverage than H_{if} does. If H_{lf} is too large (meaning we are very conservative), the coverage drops rapidly, and if H_{lf} is too small, the accuracy drops to an unacceptable level, although we have much better coverage. Both graphs confirm that we have chosen good H_{if} and H_{lf} values for our experiments.

5. PERFORMANCE EVALUATION METHODOLOGY

We first describe how we modify memory allocation to use pretenuring advice, then overview additional relevant aspects of Jikes RVM and GCTk (the garbage collection toolkit we built to work with Jikes RVM), and finally discuss how we measure and configure our system.

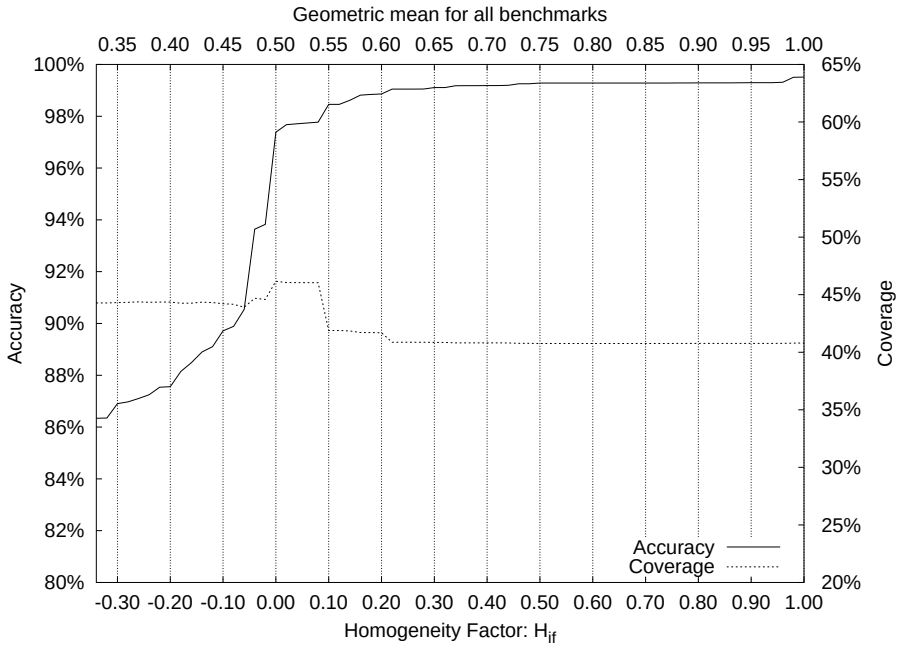


Fig. 6. Impact of H_{lf} on Accuracy and Coverage ($H_{lf} = 0.60$)

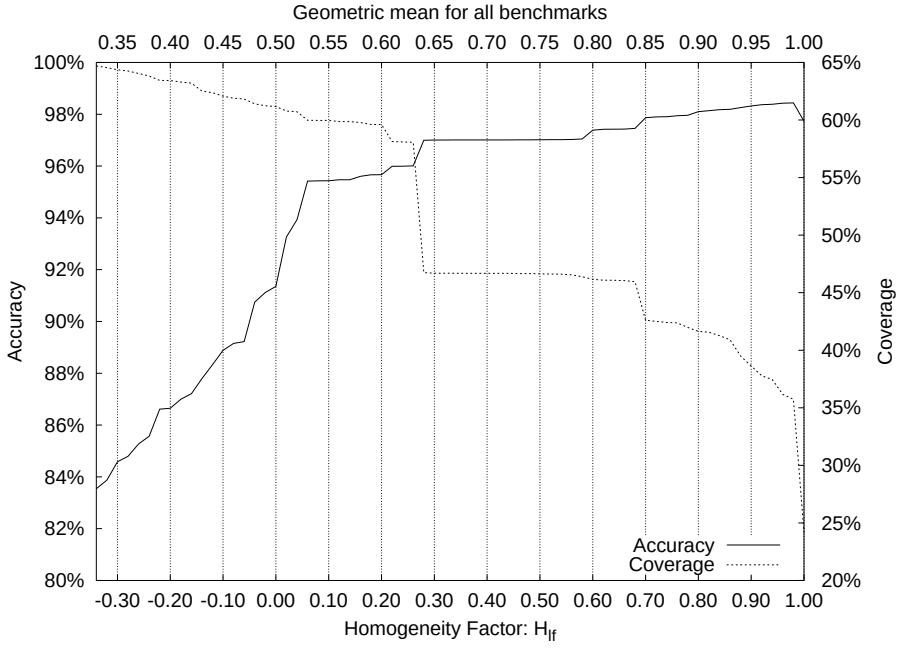


Fig. 7. Impact of H_{lf} on Accuracy and Coverage ($H_{lf} = 0.00$)

5.1 Using Pretenuring Advice

The generational, Older First, and Beltway collectors have three object insertion points: a primary allocation point (the nursery), a primary copy point (the second generation, copy zone, and the second belt respectively), and an allocation point in permanent (immortal) object space. Our advice classifications map allocations to these insertion points in the obvious way.

We modified the Jikes RVM compilers to generate an appropriate allocation sequence when compiling each `new` bytecode if the compiler has pretenuring advice for that bytecode. We provide advice to the compiler as a file of $\langle \text{site string}, \text{advice} \rangle$ pairs, where the site string identifies a particular `new` bytecode within a class. By providing advice to the compiler at *build time* (when building the Jikes RVM boot image [Alpern et al. 2000]), allocation sites compiled into the boot image, including the Jikes RVM run-time system and key Java libraries, can pretenure. If advice is provided to the compiler at *run time*, allocation sites compiled at run time, including those in the application, can pretenure.

The *advice* part of a pair indicates which of the three insertion points to use. Since the nursery is the default, we provide advice only for long-lived and immortal sites.

In application-specific pretenuring, we use *self* advice [Barrett and Zorn 1993], i.e., the benchmark executions use the same input when generating and using advice. In build-time pretenuring, we use combined advice, omitting information from the application to be measured, which is called *true* advice.

Using an advice file is not the only way one might communicate pretenuring advice to a JVM; bytecode rewriting is another possibility when one does not have access to the JVM internals. BIT is a bytecode modification tool that facilitates annotation of arbitrary bytecodes [Lee and Zorn 1997]. Similarly, IBM's Jikes Bytecode Toolkit[12] allows bytecode manipulation. Since our pretenuring advice is implemented inside Jikes RVM, we manipulate the intermediate representation directly. Also, for build-time pretenuring, we avoid modifying a large number of Jikes RVM class files by using just one simple text file for all pretenuring advice.

5.2 Jikes RVM and GCTk

Jikes RVM is a high performance JVM written in Java; its performance is comparable to commercial JVMs on the same (PowerPC) platform [Alpern et al. 2000]. Because Jikes RVM uses its own compiler to build itself, a simple change to the compiler gave us pretenuring capability with respect to both the JVM run-time and user applications. The clean design of Jikes RVM means that the addition of pretenuring to Jikes RVM (beyond the garbage collectors and allocators themselves) is limited to writing a simple advice file parser and making the above minor change to the compiler. These changes totaled only a few hundred lines of code.

We developed GCTk, a new garbage collection toolkit for Jikes RVM and the precursor to its current toolkit, MMTk [Blackburn et al. 2004b; 2004a]. GCTk is an efficient and flexible platform for GC experimentation, that exploits the object-orientation of Java and the JVM-in-Java property of Jikes RVM. GCTk implements a number of copying GC algorithms, and their performance is similar to prior monolithic Jikes RVM GC implementations. Our Appel-style generational collector, which we call *Appel* simply as a convenient shorthand name, is well tuned and uses a fast address-order write barrier [Stefanović et al. 1999] to detect and remember references from the old generation to the nursery.[13] When performing a full heap

[12] Available at http://www.alphaworks.ibm.com/tech/jikesbt.

[13] It records the exact address of the older-to-younger pointer and thus is fast for both the mutator and the collector.

collection, it traces through boot image objects, which are never themselves collected, rather than apply a write barrier to that region.[14] We extend the algorithm in a straightforward way to include an uncollected region for immortal objects. Since the immortal region is generally small, we scan it for references to younger objects rather than apply a write barrier and maintain a remembered set. We implemented the Older First GC algorithm [Stefanović et al. 1999; Stefanović et al. 2002] and Beltway [Blackburn et al. 2002] using the GCTk, and added an immortal region to them as well.

5.3 Experimental Setting and GC Configuration

We performed our experimental timing runs on a Macintosh Power Mac 4e. It has one 733 MHz PowerPC 7450 processor, 32KB on-chip L1 data and instruction caches, 256KB unified L2 cache, and 512MB of memory, and runs PPC Linux 2.4.

As indicated in Section 3.1, a time-space trade-off is at the heart of each pretenuring decision. In order to understand better how that trade-off is played out and to make fair comparisons, we conduct all of our experiments with a range of fixed heap sizes. We express heap size as a function of the minimum heap size for the benchmark in question. We define the *minimum heap size* for a benchmark to be the smallest heap in which the benchmark can run when using an Appel-style generational collector without pretenuring. This amount is at least twice the max live size. We determine it experimentally, and show this size for each benchmark in Table V.

Benchmark program	Minimum heap size (MB)
_201_compress	18
_202_jess	10
_205_raytrace	14
_209_db	21
_213_javac	24
_222_mpegaudio	8
_227_mtrt	20
_228_jack	9
health_6_128	9
pseudojbb	56

Table V. Minimum Heap Size at which Programs Run (Non-Pretenuring)

For the generational algorithm, we collect when the sum of the space consumed by the three allocation regions (nursery, older generation, and permanent object space) and the reserved region reaches the heap size. We collect the older generation, as per the Appel algorithm, when it approaches the size of the reserved region. In this scheme, the nursery varies from being as large as (half of) the heap down to a small minimum size.

5.4 Second Iteration Experimental Methodology

An ordinary run of a benchmark program performs one iteration of the program, inducing loading and compilation of application classes as needed. As soon as a method is needed, we use the Replay system to optimize it to the same optimization level it acquired in a majority of

[14] This is a tradeoff between a more complex write barrier, incurring overhead on every pointer store, and a faster write barrier with more GC time overhead.

Adaptive runs. Thus, the run of the program includes compilation time. Further, Jikes RVM's compilers, being written in Java, allocate their data structures into the application heap. Hence the run includes compiler allocation as well, placing additional load on the collector. We measure this first iteration including compilation time. Eeckhout et al.'s [Eeckhout et al. 2003] results show that Jikes RVM's behavior can still dominate the application in this measurement. We therefore run the application for *two* iterations, and use the Replay technique. In the first iteration, because of Replay we attain high code quality for frequently executed methods. We then perform a full heap collection, disable further optimization, and iterate the application. This second-iteration measurement includes no compile time, only application and collection time, and does not include the full heap collection that we inserted between the iterations.

This methodology is also closer to the many JVMs that use a compiler written in C, and which allocate "on the side", not in the Java heap. The second iteration measurements also approximate applying our profile-directed pretenuring only to the Java libraries in other systems.

6. PERFORMANCE EVALUATION RESULTS

We now present execution time and other results using generational collection for build-time pretenuring, application-specific pretenuring with our advice and CHL advice (as used by [Cheng et al. 1998]), and the combination of build-time and application-specific pretenuring. We present single-iteration results and second-iteration results.

We demonstrate that our advice is collector-neutral by showing that it improves very different collectors as well, the Older First collector and the Beltway collector. In all of the experiments, we use the pretenuring advice parameters $T_a = 0.45$, $H_{lf} = 0.60$, $H_{if} = 0.00$, and $H_{cf} = 0.90$ as described in Section 3.1.1.

We generally report times normalized with respect to the non-pretenured case. We report measurements for a range of heap sizes, normalized with respect to the minimum size at which the program will run in the non-pretenuring collectors (as shown in Table V). The range we used was from that minimum size to three times that size. We stopped there because most curves have reached or nearly reached their asymptotes by that point. We also report cache and translation look-aside buffer (TLB) misses using performance counters.

We begin with some basic measurements of the benchmarks, and also examine the non-pretenuring case to see what room for improvement there may be.

6.1 Non-pretenuring Measurements

To help interpret the magnitude of improvement we obtain, we present in Figures 8 and 9 the percent of total execution time spent in GC for the non-pretenuring case. Assuming there is minimal impact on mutator execution time, these results give an upper bound on the improvement we can obtain by speeding up GC. The x-axis of the graphs gives the normalized heap size. The y-axis is percent of total execution time spent in GC. Since later we will be presenting results from both first and second iterations of each benchmark programs, the graphs include two curves, one for the first iteration of the benchmark and one for the second. Note that the percentages are computed from the ratio of the GC time to the execution time of *each iteration separately*.

We observe that the percent of time spent collecting tends to be higher for second iterations. This difference is partly because the optimizing compiler (invoked only in the first iteration) is computation intensive compared to the volume it allocates. The compiler also allocates a significant amount, which explains why the first-iteration curves are high for small heaps.

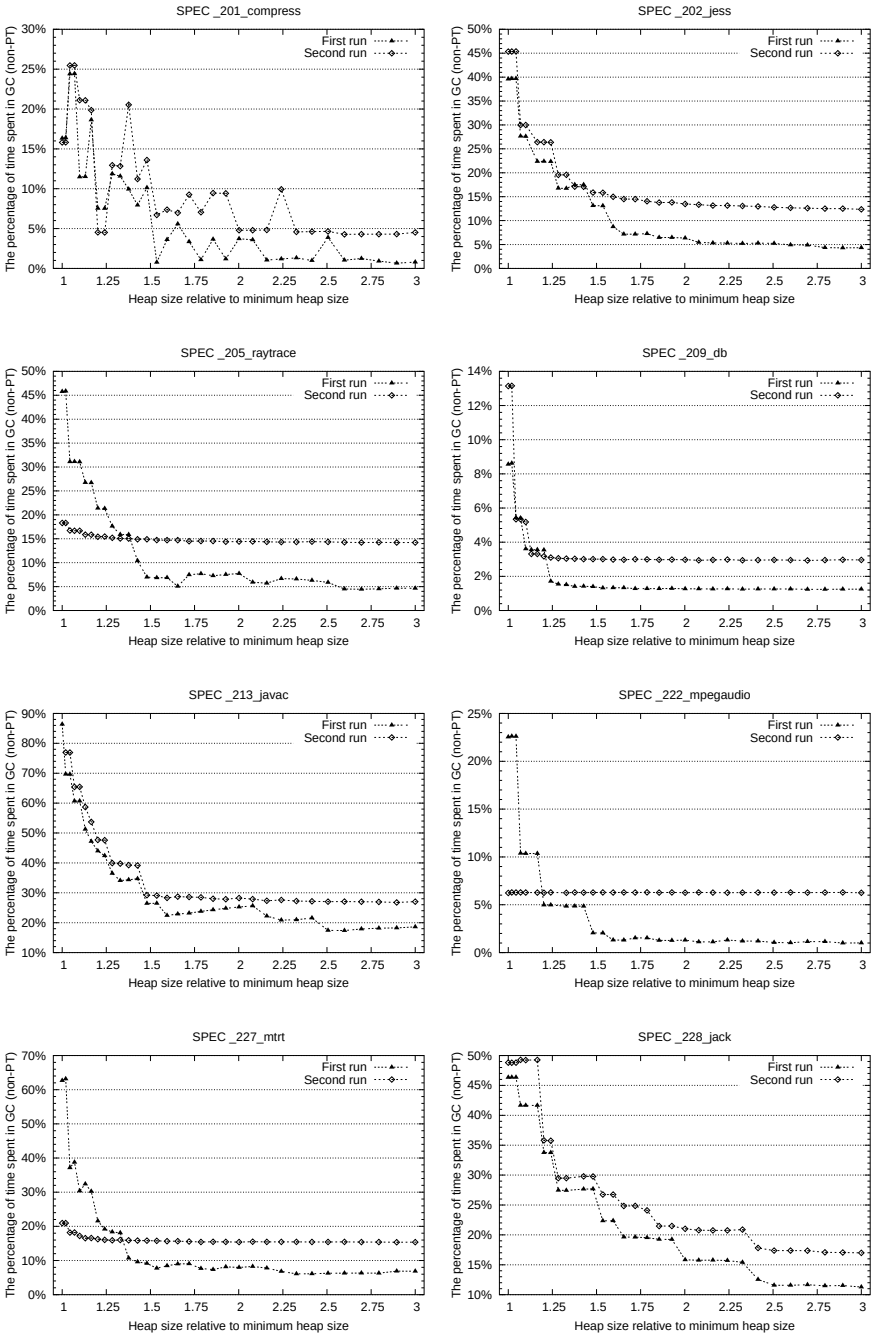


Fig. 8. SPEC Benchmarks: Percent of Time Spent Collecting (No-Pretenuring)

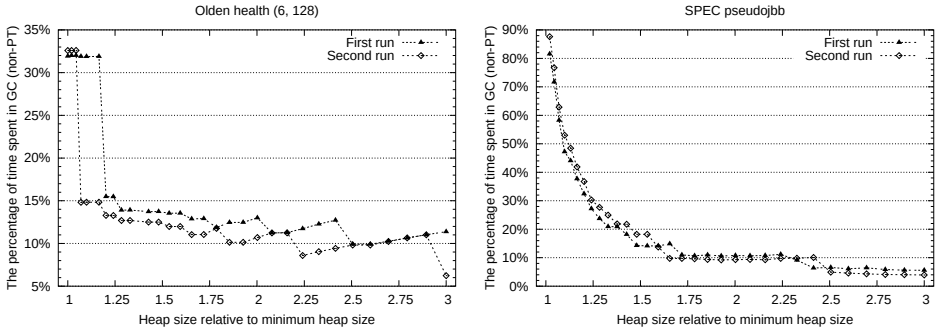


Fig. 9. Olden Health (6,128) and pseudojbb: Number of GCs; Percent of Time Collecting

Another factor that contributes to the difference is that, in obtaining the second-iteration measurements, we enable explicit GC, which is attempted by the SPEC harness and also by javac. We enabled explicit GC so that we could cause a full-heap GC between the first and second iteration, and thus give the second iteration a clean slate. The first-iteration numbers are from different runs using our default settings, which disable explicit GC.

Another effect is that many of the second-iteration curves are nearly flat. This is partly because of explicit GC invocations, which result in essentially fixed GC time independent of heap size. (The graphs of second-iteration number of GCs in Figures 24 and 26 bear this result out.) Another reason for the flat second-iteration behavior is that many programs produce mostly garbage after initially allocating some longer lived objects. At a heap size large enough to contain first-iteration compilation data structures, the second-iteration data fit quite comfortably and the heap size quickly reaches a value requiring a minimal number of full-heap GCs, and it is these GCs that are responsible for most of the GC time.

6.1.1 *Basic GC Speed.* We also measured the non-pretenuring collector's "raw" speed, using the FixedLive test program, part of the Jikes RVM distribution. It first creates a chosen volume of live objects arranged in a binary tree structure. Then it creates objects that it immediately discards, so as to force the live objects to be collected repeatedly. We created 100 MB of live objects, each 24 bytes in size (8-byte header plus two 4-byte int fields plus two 4-byte reference fields). This experiment gave a tracing (live object copying) rate of 17.2 MB/s and an allocation rate of 131 MB/s. If we keep the same volume but increase the object size to 192 bytes, we obtain a tracing rate of 50 MB/s and allocation rate of 308 MB/s. Clearly there are significant per-object overheads. A simple regression fit on four points suggests that per-object tracing cost is about 950-1000 ns and per-byte copying cost is about 15-20 ns.

The rates we obtained for FixedLive under gcj 4.0.1 (Boehm collector) on the same machine are 52 MB/s tracing and 92 MB/s allocation. We would expect our copying collector to be slower than a mark-sweep collector on this kind of benchmark (little fragmentation), because of its copying work. Also, the Boehm collector does deferred sweeping, which would affect the allocation rate, not the tracing rate, as they are reported by this benchmark. While we must conclude that our collector's speed could be improved, it is fast enough that there would still be useful savings from pretenuring for a somewhat faster collector, especially in relatively small heaps (where GC time is a substantial fraction of total time).

6.2 Build-Time Pretenuring

Build-time advice is true advice; in these experiments, we combine advice (Section 3.1.1) from each of the *other* benchmarks. Because pretenuring will occur only at sites pre-compiled into the Jikes RVM boot image, build-time advice does not result in pretenuring of allocation sites within an application. However, because considerable allocation occurs from those sites compiled into the boot image (quite notably from the Jikes RVM optimizing compiler and key Java libraries), build-time advice has the distinct advantage of delivering pretenuring benefits without requiring the user to profile the application.

Figure 10 shows the total performance improvement for each benchmark, using build-time pretenuring normalized with respect to the generational collector without pretenuring. The x-axis is the heap size, in multiples of the minimum heap size, for 33 points from 1 to 3; the y-axis is execution time relative to not pretenuring. All our results use the same x-axis. (Figures 17, 18, 19, 20, and 21 show individual program results for total performance, garbage collection time, number of collections, and copying work. We discuss them in Section 6.4.)

Notice that there is a lot of *jitter* for each benchmark in these graphs. This jitter is present in our raw performance results for each specific allocator as well as in the normalized improvement graphs we show. The jitter is mostly due to variations in the number of collections at a given heap size. Small changes in the heap size can trigger collections either right before or after significant object death, which affects both the effectiveness of a given collection and the number of collections. This effect illustrates that GC evaluation should, as we do, use many heap configurations, not just two or three. Pretenuring neither dampens nor exaggerates this behavior, but is subject to it.

In some cases, build-time pretenuring degrades total performance by a few percent, but for most configurations, programs improve, sometimes significantly. Improvements tend to decline as the heap size gets larger because the contribution of garbage collection time to total time declines as the heap gets bigger, simply because there are fewer collections. Pretenuring thus has fewer opportunities to improve performance, but pretenuring still achieves an improvement on average of around 3% even for large heaps. All programs improve on average, and for javac, mtrt, and pseudojbb, in a number of configurations the improvement is more than 50%. These improvements result from reducing copying and saving copy reserve in the garbage collector, and the significant decrease in GC time improves overall execution time.

6.3 Application-Specific Pretenuring

This section compares our classification scheme to the CHL scheme [Cheng et al. 1998] using application-specific (self) advice. Given an application running with a generational collector with a fixed nursery size, CHL advice generation first measures the proportion of object instances that survive at least one minor collection on a per-allocation site basis. CHL classifies as long-lived those allocation sites for which a high proportion survive (we implemented their approach with the same 80% threshold they used). CHL then pretenures (allocates) objects created at these sites into the older generation, and allocates objects from all the other allocation sites into the nursery in the usual way. Because of allocation-site homogeneity in ML (which we also observed in Section 4 for our Java programs), their approach is fairly robust to the threshold.

The key differences between the two classification schemes are (a) that our advice is neutral with respect to the garbage collector algorithm and configuration, and (b) that we include an immortal category and our collector puts immortal objects into a region that it never collects.

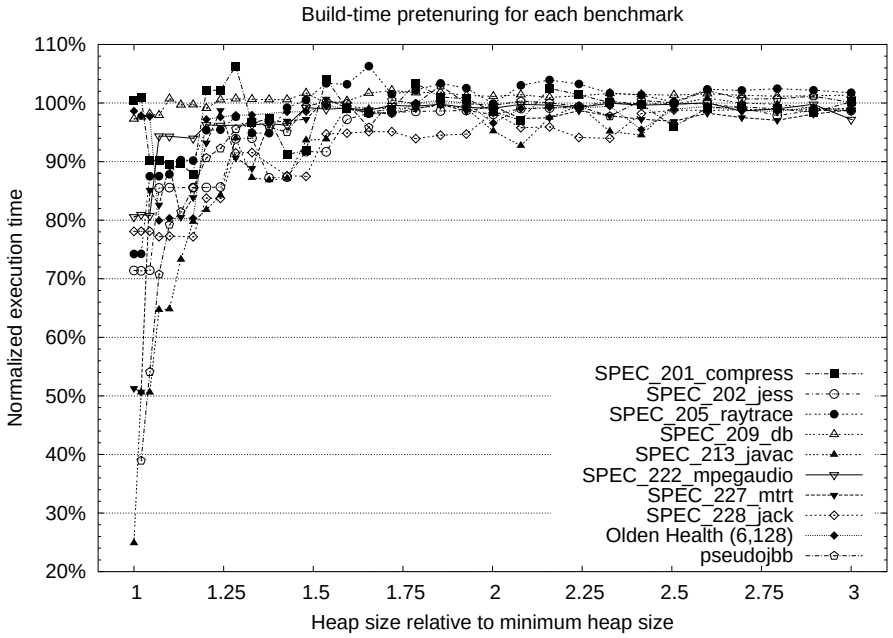


Fig. 10. Relative Execution Time for Build-Time Pretenuring

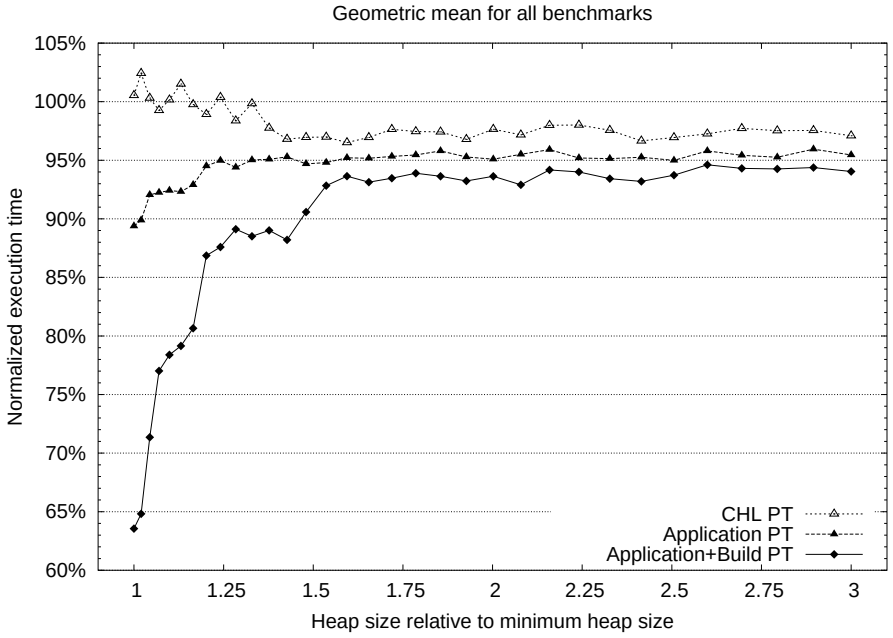


Fig. 11. Relative Execution Time for Application-Specific Pretenuring

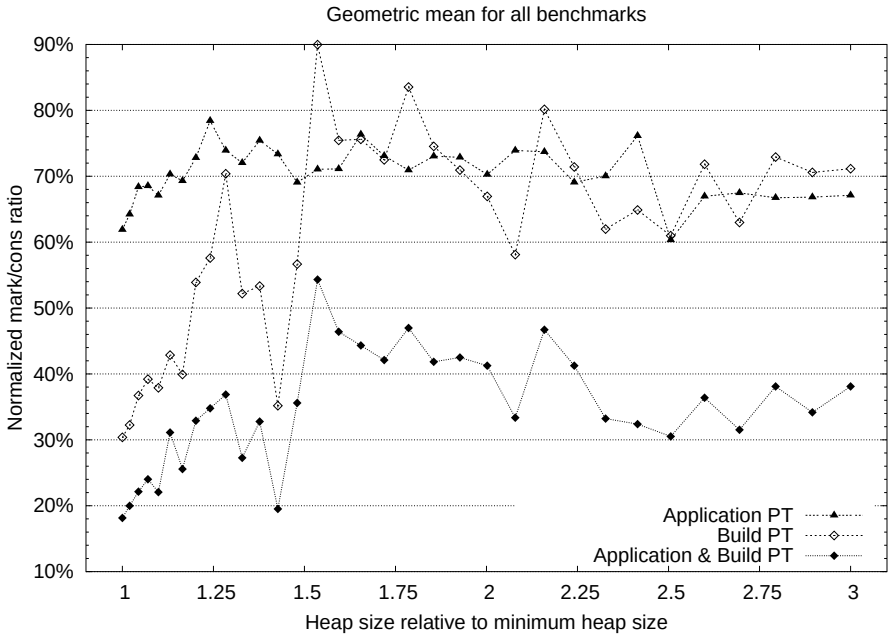


Fig. 12. Comparing Application-Specific, Build-Time, and Combined Pretenuring: Relative Mark/Cons Ratios

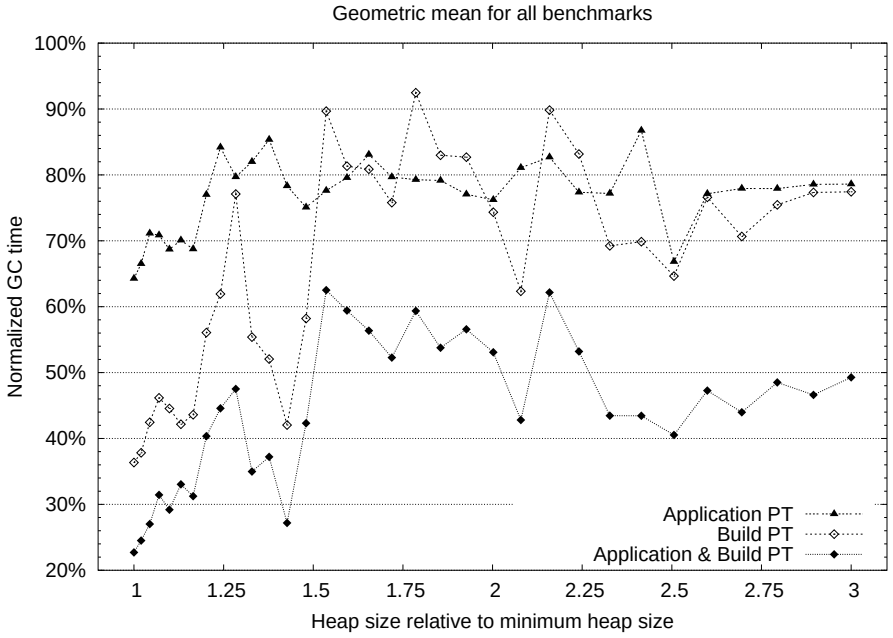


Fig. 13. Comparing Application-Specific, Build-Time, and Combined Pretenuring: Relative Garbage Collection Time

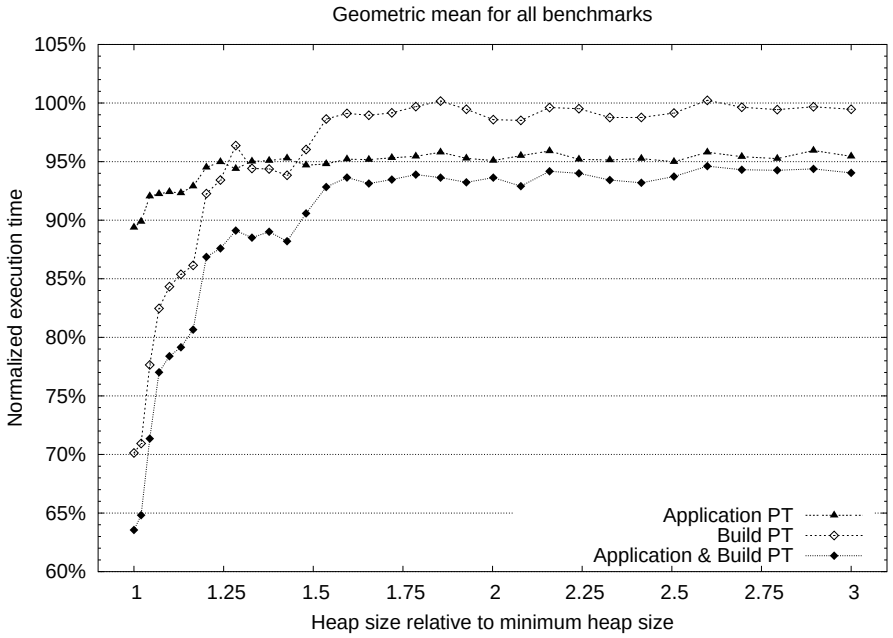


Fig. 14. Comparing Application-Specific, Build-Time, and Combined Pretenuring: Relative Execution Time

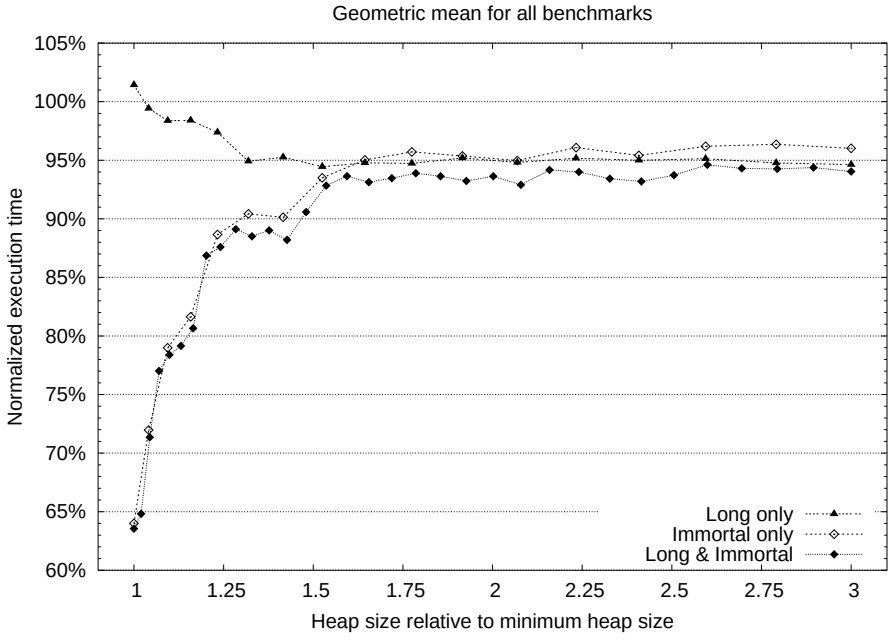


Fig. 15. Comparing Long-only, Immortal-Only, and Combined Pretenuring: Relative Execution Time

The first of these makes our approach more general and the second improves performance. Our pretenuring allocates on average 4% of objects into the immortal space (see Table III), and these decisions are overwhelmingly correct (because our decisions to pretenure to immortal space are so conservative). Since both schemes get the same total heap size in our experiments, allocation into the immortal region (because it requires no copy reserve) increases the space available to the generational collector (see Figure 16). While 4% may not sound like much, in tight heaps it can result in a large proportional increase in nursery size, and can thus lower GC time significantly.

Figure 11 compares CHL and our application-specific pretenuring, using the generational collector, which has a flexible nursery size. The figure shows the average relative execution time using a geometric mean of our benchmark programs. On average our advice performs at least 2% better than CHL advice, except in a tight heap where the impact of immortal objects is highest and our advice performs significantly better.

Because CHL advice generation is specific to program, collector, and collector configuration, it cannot be combined for build-time pretenuring without significant change to the algorithm. We make no further comparisons with CHL because of this drawback and because, as we have just illustrated, our three-way classification offers better performance than the CHL two-way scheme on average, and much better performance than CHL for tight heaps.

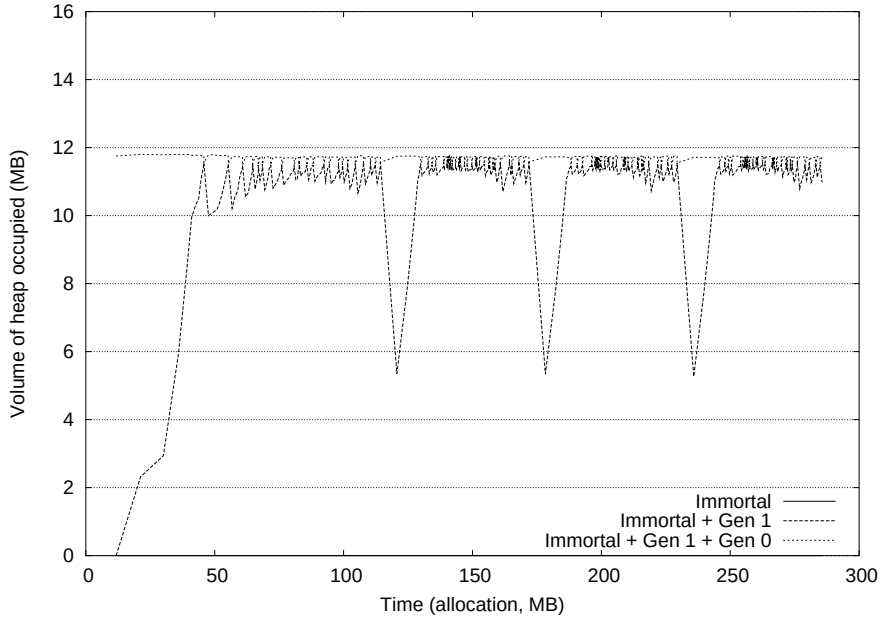
6.4 Combining Build-Time and Application-Specific Pretenuring

In this section we show that combining build-time and application-specific pretenuring results in better performance than either one alone. For these three pretenuring schemes, we present results using the geometric mean of the benchmarks for relative mark/cons ratio in Figure 12, the geometric mean of the relative garbage collection time in Figure 13, and the geometric mean of the relative execution time in Figure 14.

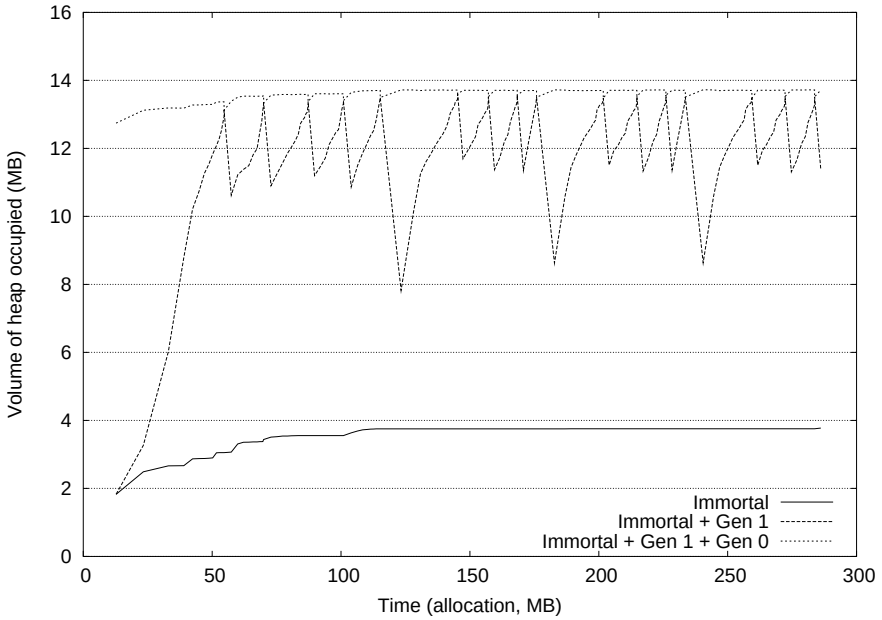
Figure 12 shows the mark/cons ratio for each pretenuring scheme, relative to not pretenuring. The mark/cons ratio is the ratio of bytes copied ("marked") to bytes allocated ("cons'ed"). The figure explains *why* pretenuring works: it reduces copying. In all cases, pretenuring reduces the volume of objects the collector copies. Reductions range from 10% to 81%, which is quite significant when minimum heap sizes can be as large 60MB (pseudojbb).

Figure 16 offers additional insights. Figure 16(a) shows heap usage over time for a run of the javac benchmark without pretenuring, and Figure 16(b) shows it with pretenuring. Both runs use a heap size of 24MB. The top line in each graph shows the total heap consumption immediately before each GC. The second line shows the space consumed by the older generation immediately before each GC (both nursery and full heap collections). Finally, the bottom line shows the immortal space consumption, which is always zero in Figure 16(a).

Note that in pretenuring, allocation to immortal space effectively increases the size of the heap because it does not need to reserve space to copy immortals. (Of course the total space available is the same in both cases.) Thus the pretenuring graph's total *occupied* heap size is larger. Because the copy reserve is smaller, the nursery is larger (by half the occupancy of the immortal space). This larger nursery delays the growth of the older generation and defers older generation collections, in addition to reducing the frequency of nursery collections. The lowest points in space consumption of the older generation (the second line) are very similar in both graphs, which shows that pretenuring does not allocate many immortal objects inappropriately (if it did, the second line would be higher for pretenuring). Also note that the shapes of the four troughs in the second lines towards the right side of the figures. When not pretenuring, the bottoms of the troughs are flat, showing that there is no direct allocation to the older generation.



(a) _213_javac: Heap Profile Without Pretenuring



(b) _213_javac: Heap Profile With Build-Time and Application-Specific Pretenuring

Fig. 16. Comparison of Heap Usage Over Time Without and With Pretenuring

With pretenuring, they show an upward slope to the right, indicating direct allocation to the

older generation.

In summary, pretenuring performs better because it does less copying. It reduces copying in two ways: direct allocation into the older spaces avoids copying to promote longer lived objects; and the immortal space effectively increases the size of the heap, thus reducing the number of GCs and the amount of copying.

Figure 13 shows that the reduction in copying cost significantly and consistently reduces GC time, especially considering the advice is true rather than self advice for build-time pretenuring. In particular, combined (application and build-time) pretenuring improves collector performance between 40 and 80% for most heap sizes. Combined pretenuring is on average the most effective of the three. In large heaps, application-specific pretenuring is on average nearly as good, but build-time pretenuring offers significantly higher advantage than application-specific in small heaps, because it includes a higher volume of immortal allocation.

These results carry over to execution time (Figure 14). We see that all the pretenuring schemes improve performance. Average improvements are usually between 1% and 6% in larger heaps and 11% to 36% in very tight heaps, but as shown in Figures 17 and 21, some individual programs improve more.

It may strike one as surprising that pretenuring consistently gives benefits even at the larger heap sizes, which have fewer collections. As we will see in Section 6.9, part of the benefit, about 2–3% on average, is from improved locality (fewer cache and TLB misses). The other 2–3% average improvement in execution time is from reduced GC time. Recall that pretenuring reduces the cost of nursery collection, and the percentage improvement in GC time will be higher when GC is invoked *less* often (same amount of copying saved, but less total copying). Also, the heap sizes we use are not large enough for GC improvements to disappear relative to total execution time. One would need rather larger heaps to obtain that effect.

6.5 Immortal-only and Long-only Pretenuring

We investigated the relative importance of immortal and long pretenuring by refining our advice as follows:

(1) **Immortal-only:** Take the advice previously generated and discard any *long* advice (those sites will be treated as *short*). We retain the *immortal* advice.

(2) **Long-only:** Take all *immortal* advice and treat it as *long*. All *short* advice remains the same.

(3) **Both:** Keep both the *immortal* and *long* classifications as before.

Figure 15 shows results using these three sets of advice. The figure reveals that long-only gives a robust average improvement of about 5% at larger heap sizes, but its cost increases noticeably at smaller heap sizes (because it forces more frequent collections) overcoming its benefits. Immortal-only is always beneficial, enormously so in tight heaps, because it increases the effective heap size (as shown in Figure 16 for javac). At larger sizes it does not give quite as much benefit as long-only. Doing both kinds of pretenuring robustly obtains both benefits.

6.6 Comments on Specific Benchmarks

We now analyze noteworthy features of the individual benchmark results. Figures 17, 18, 19, 20, and 21 show individual program results for total performance, garbage collection time, number of collections, and mark/cons ratio.

jess, jack, *and* mpegaudio. Application-specific pretenuring does not help much, and occasionally degrades performance slightly because of nepotism. Both jess and jack have very small nursery survival rates (less than 1% for jess, and 3% for jack). Since the application-specific objects are mostly short-lived, application-specific pretenuring puts only a very small volume of objects into higher spaces. A similar pattern exists for mpegaudio, which has a higher nursery survival rate but does very little allocation and so places little stress on the garbage collector. Pretenuring shows benefit with build-time advice since the survival rates for Jikes RVM objects are higher. For example, in jess, application-specific pretenuring allocates 210KB of immortal objects and 450KB of long objects, while build-time pretenuring produces 2700KB of immortal objects.

mtrt *and* raytrace. Both build-time and application-specific pretenuring offer substantial performance improvement, up to 10% and 50% in tight heaps, because of the heap space saved by pretenuring immortal objects. For larger heaps, build-time pretenuring gives only slight improvement, but application-specific pretenuring improves performance by 4–6%, and the combination gives an additional 1–2% improvement.

javac. This program has a substantial number of long and immortal sites, and thus build-time pretenuring is relatively less important. In tight heaps, most of the benefit comes from build-time pretenuring at immortal sites (by saving copy reserve space), and application-specific pretenuring has much less benefit, or even degrades total execution time. The reason is that javac suffers from nepotism, which we also observe in pseudojbb. Larger heaps reduce the effects of nepotism, and the benefit of less copying shows up. Here we observe that application-specific pretenuring gives about 5–6% performance improvement.

health *and* db. These applications have some large data structures that are used throughout the execution. Thus application-specific pretenuring can bring benefits by saving copying cost. For health, application-specific pretenuring improves performance by up to 27% for tight heaps, and 8–13% for larger heaps. Although build-time pretenuring alone does not have much benefit, the combination yields an additional 5% improvement.

pseudojbb. Application-specific pretenuring suffers from nepotism in tight heaps, and gives only slight improvement (about 1%) in larger heaps. Build-time pretenuring gives huge improvement, up to 60%, by saving copy reserve. For larger heaps, pseudojbb spends most of its time in the mutator, usually spending less than 10% of its time in GC. Hence, although we are able to reduce GC time by more than 10% in most cases, the improvement in total execution time is limited to around 1%.

compress. Application-specific pretenuring has little effect, but build-time pretenuring produces large variation in performance across heap sizes. This variation comes from large variation in the volume of objects copied, as can be seen in the mark/cons ratio graph for compress. This variation is not so much because of pretenuring itself but because the pretenuring causes the moments when GC is triggered to move a bit, and compress uses a number of large short lived objects. If one collects at a "bad" moment, one ends up copying these large objects and then throwing them away at the next full-heap collection (which will come sooner than in a "good" run, because we promoted a large object that will die soon). The effect is more pronounced at smaller heap sizes because more frequent GCs make it more likely that we promote a large short-lived object, but this behavior, though repeatable (deterministic), is chaotic with respect to heap size. (It is one way in which *jitter* arises.)

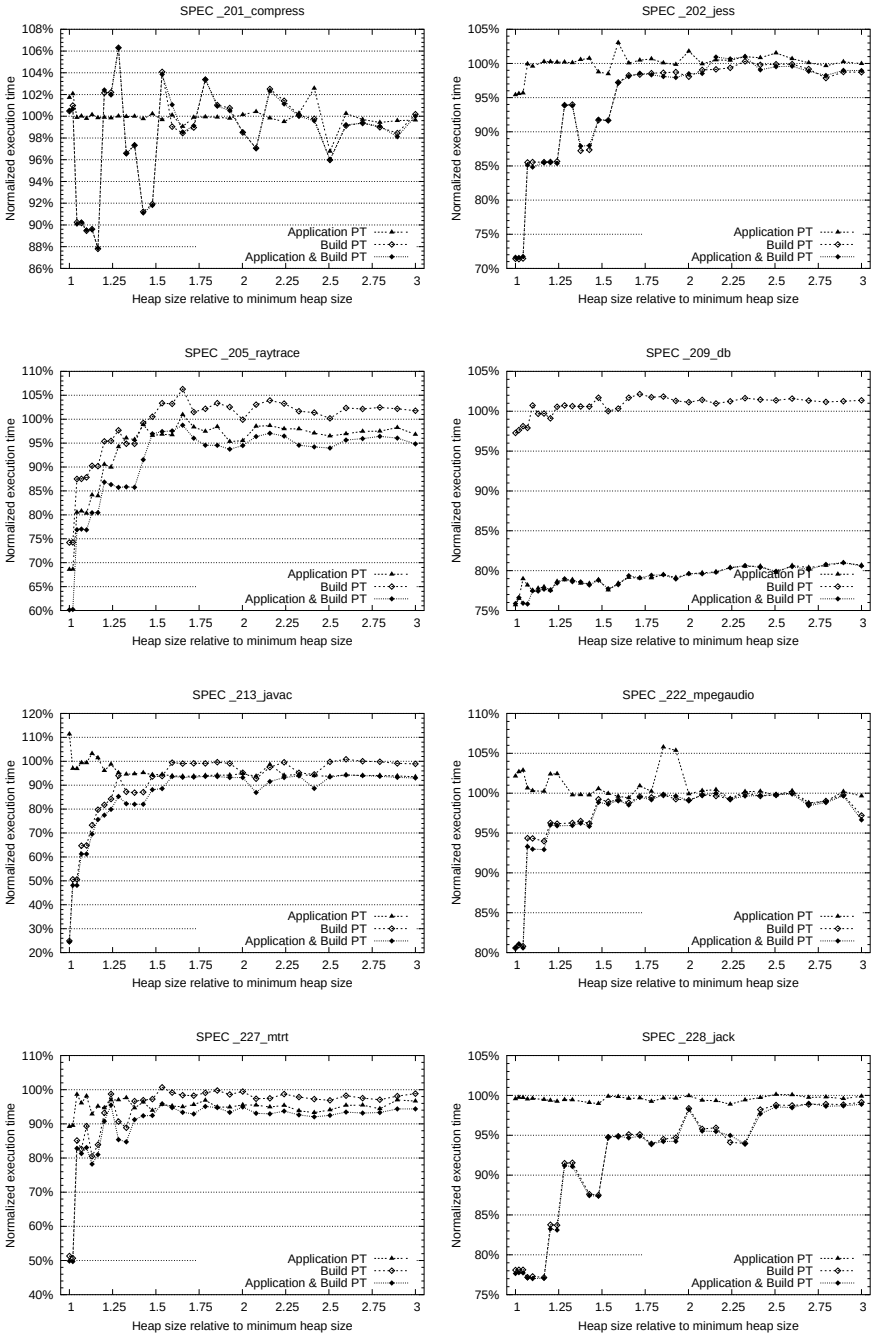


Fig. 17. SPEC Benchmarks: Execution Time Relative to Non-Pretenuring

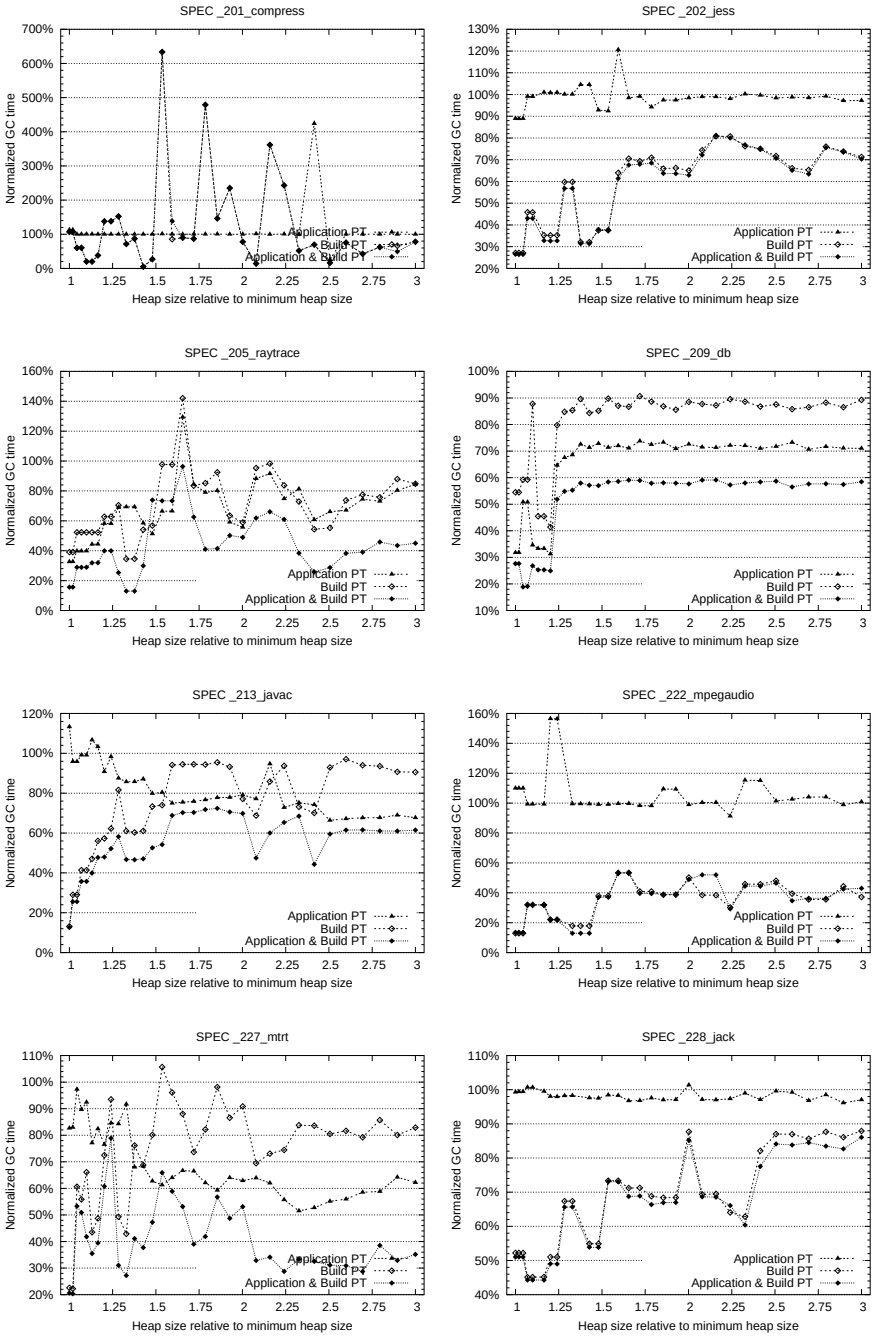


Fig. 18. SPEC Benchmarks: GC Time Relative to Non-Pretenuring

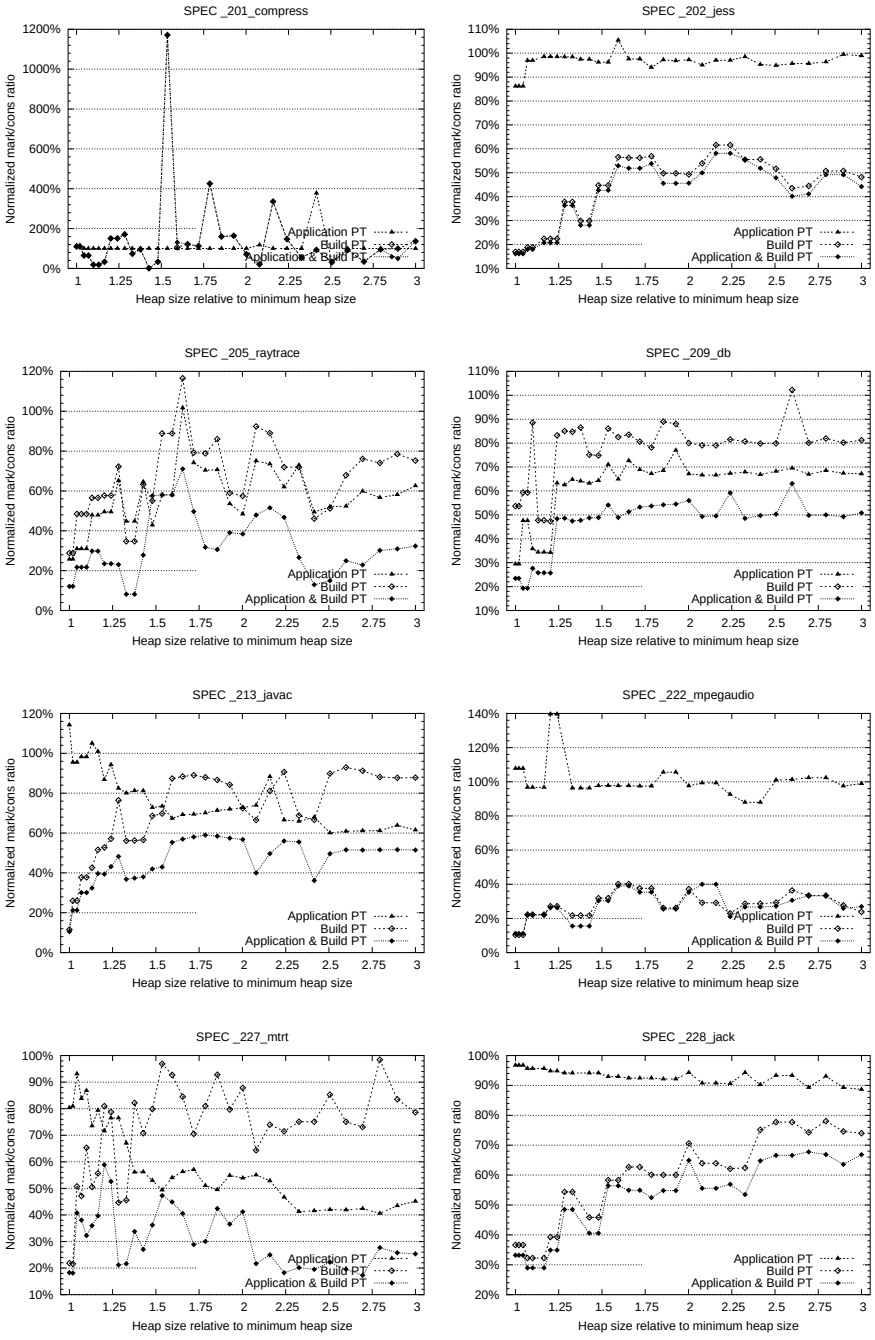


Fig. 19. SPEC Benchmarks: Mark/Cons Ratio Relative to Non-Pretenuring

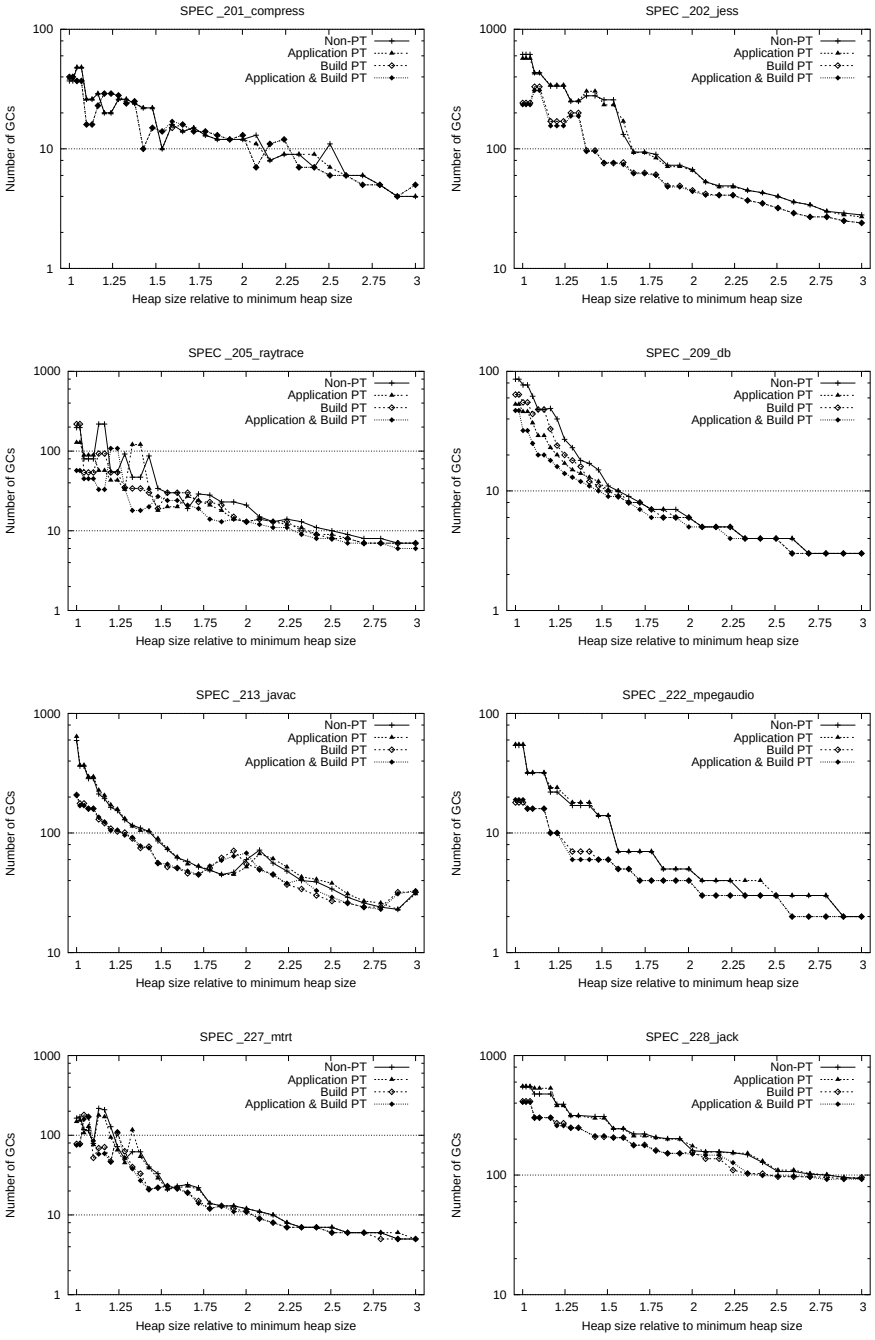


Fig. 20. SPEC Benchmarks: Number of Garbage Collections

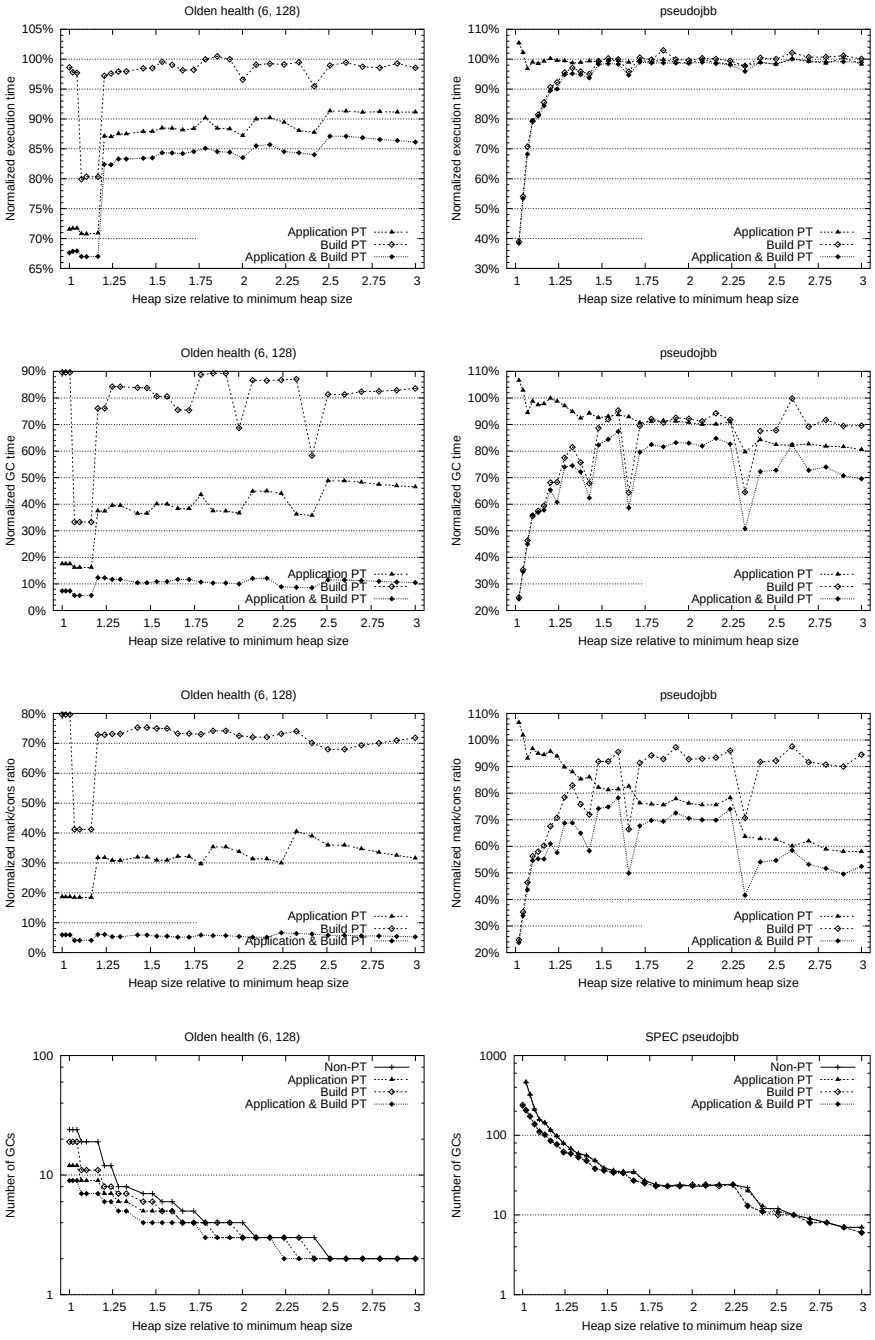


Fig. 21. Olden Health (6,128) and pseudojbb: Execution Time, GC Time, Mark/Cons Ratio, and Number of GCs

6.7 Second Iteration Results

We previously described the first- and second-iteration experimental methodology. We now present these results for second-iterations of the SPEC benchmarks: execution time (relative to non-pretenuring second-iteration time) in Figure 22, GC time (relative to non-pretenuring second-iteration GC time) in Figure 23, and number of GCs in Figure 24. The corresponding results for health and pseudojbb appear in Figures 25 and 26, along with graphs showing the geometric mean of all the benchmarks.

While there are some individual variations, as to be expected, second-iteration relative performance is quite comparable to first-iteration, which shows at least these two things: (a) our scheme is improving application performance overall, not just the Jikes RVM compilers; and (b) our approach is likely to give useful benefits to JVMs with run-time systems that do not allocate into the application heap (i.e., ones written in C).

6.8 Effects of Compilation Strategy and Trace Generation

Section 3.2 describes various compilation strategies for Jikes RVM, namely Opt, Adaptive, and Replay. Figures 27(a), (b), and (c) show results using these three compilation strategies, all with the same advice (developed from traces generated from Opt builds). These graphs show two important things for our purposes. First, the similarity between Figures 27(b) and (c) demonstrates that Adaptive and Replay behave virtually the same with respect to pretenuring. Second, while Figures 27(a) and (b) are a little less similar, they retain the same trends. The primary difference is that the Opt runs do much more optimizing compilation, which results in more allocation at build-time pretenured sites. Hence build-time pretenuring is relatively more important for Opt runs.

By default, we profile Opt runs to produce advice. Figures 27(c) and (d) compare generating advice from Replay runs versus Opt runs. We see there is essentially no difference.

6.9 Locality Effects

Since pretenuring results in possibly rather different placement of objects in the heap, one might wonder how it impacts memory reference locality. In particular, does it increase or decrease cache and translation look-aside buffer (TLB) misses? We performed runs that collected hardware performance monitor statistics on Level 1 (L1) and Level 2 (L2) cache misses and TLB misses, presented in Figure 28(a, b, c). As usual, the x-axis is relative heap size. The y-axis is *relative* miss *rate*. More specifically, for each run where we measured L1 (L2, TLB) misses, we computed the miss rate as the number of misses divided by the number of cycles the run took. The graphs show these rates for pretenuring relative to the rates without pretenuring. We show this for build-time, application-specific, and combined pretenuring, each a separate curve in each graph. These are all for the Appel Replay collector, and we present the geometric mean across all benchmarks.

To interpret the results and see why we developed them this way, consider a point at which overall execution time improves with pretenuring. If the improvement is because there are fewer total cycles and proportionately fewer misses, we would obtain a miss rate ratio of 1.0, meaning that there is no locality difference and the improvement has to do with number of instructions executed rather than cache performance. If the ratio is less than 1.0, then at least some of the improvement is coming from improved locality (lower miss *rate*), and if the ratio is greater than 1.0, we are seeing overall improvement in the face of a higher miss rate (unlikely but theoretically possible).

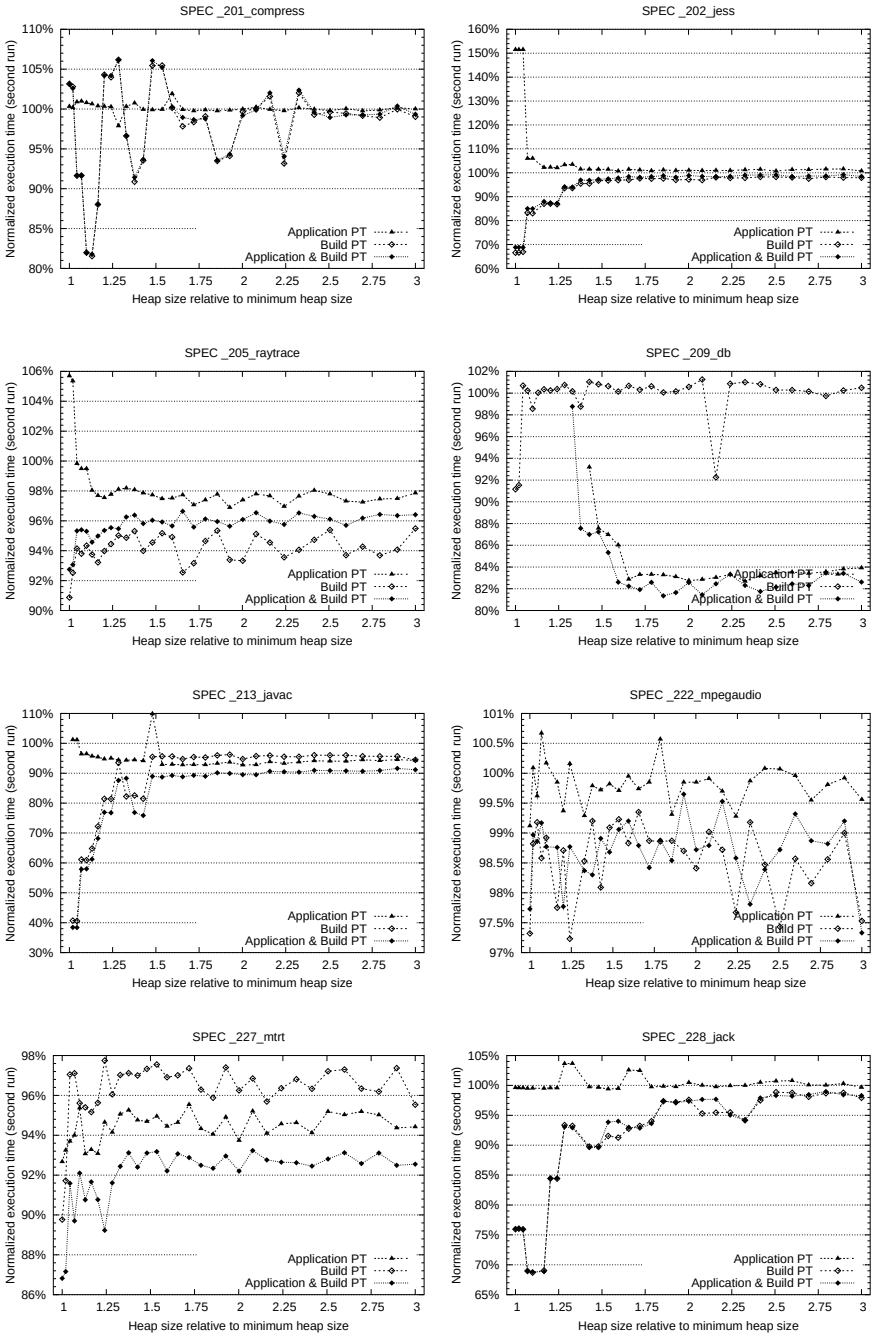


Fig. 22. SPEC Benchmarks: Second Iteration Execution Time Relative to Non-Pretenuring

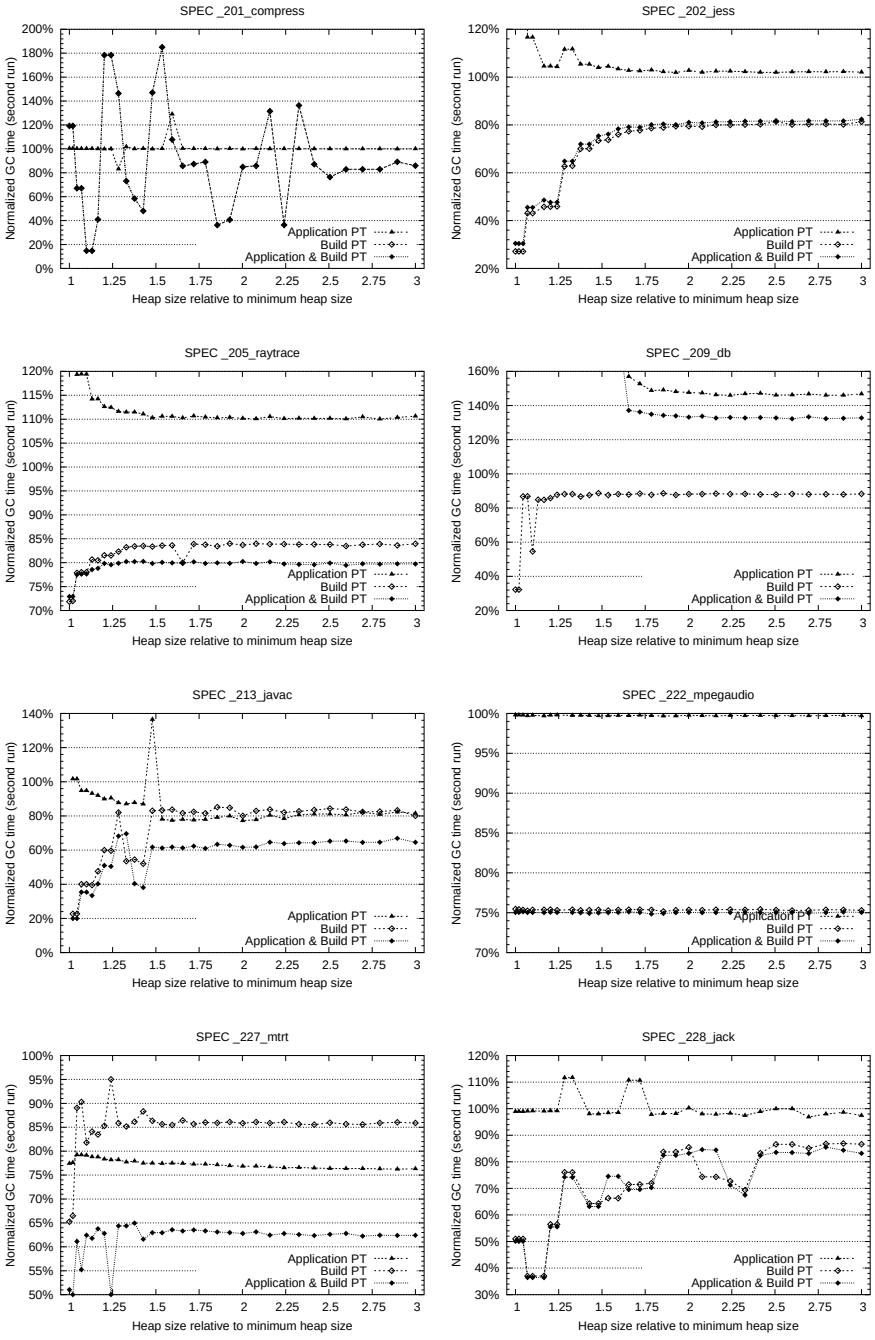


Fig. 23. SPEC Benchmarks: Second Iteration GC Time Relative to Non-Pretenuring

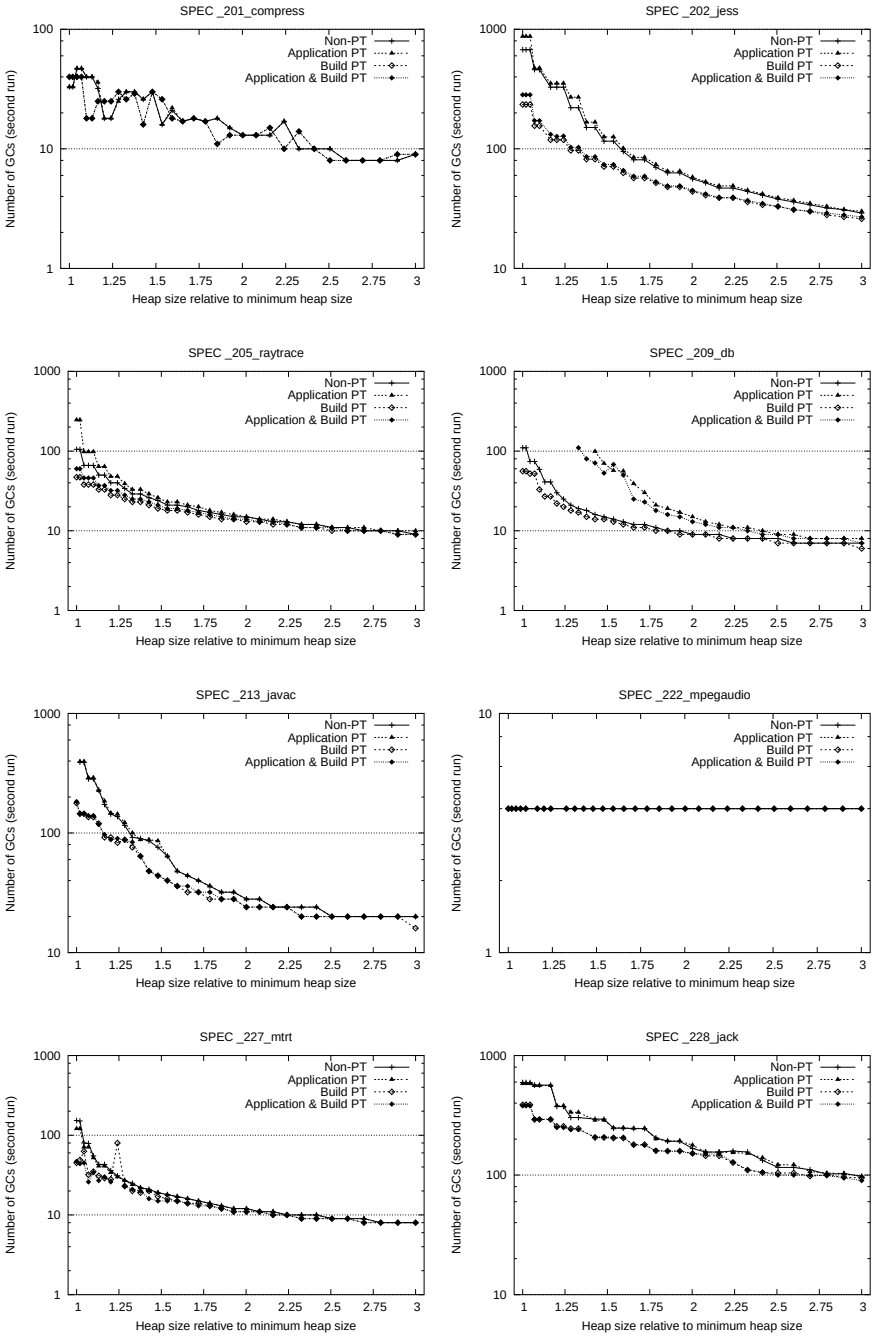
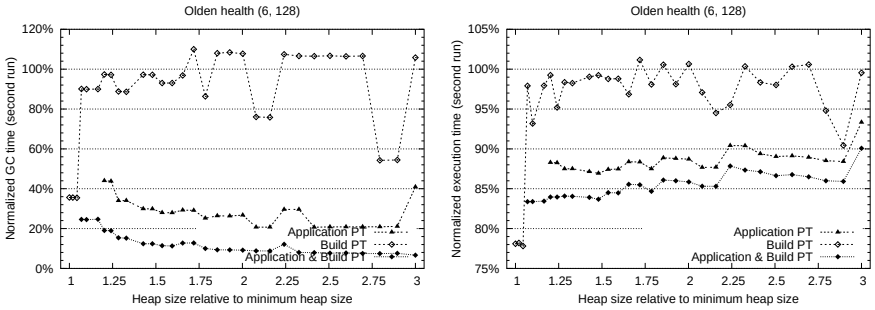
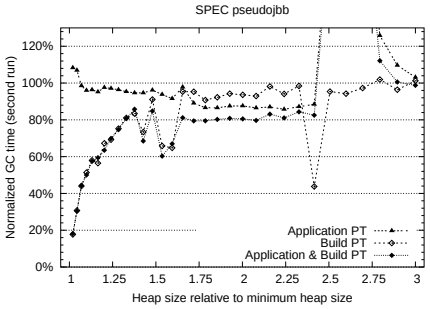


Fig. 24. SPEC Benchmarks: Second Iteration Number of GCs

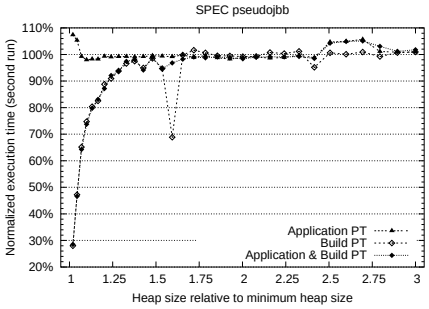


(a) Health GC 2

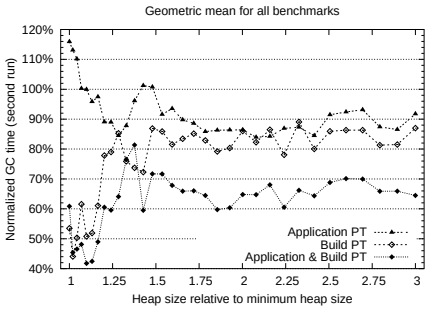
(b) Health Exec 2



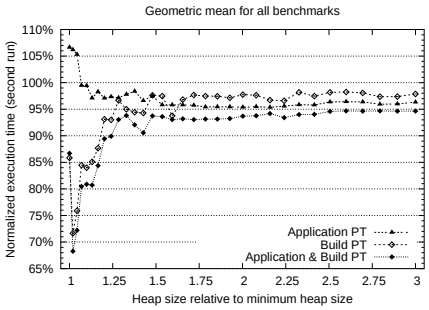
(c) PseudoJBB GC 2



(d) PseudoJBB Exec 2



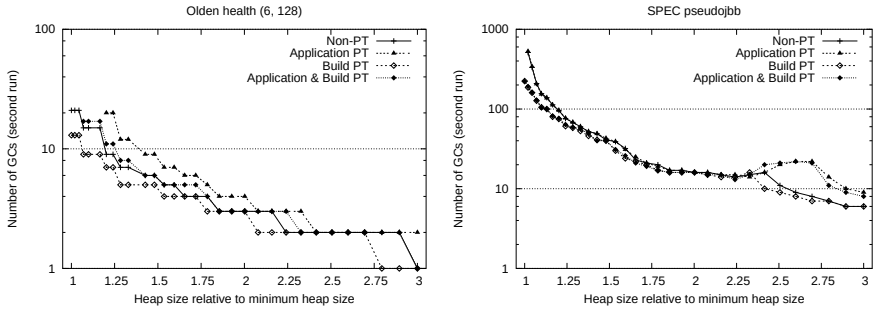
(e) Geometric Mean GC 2



(f) Geometric Mean Exec 2

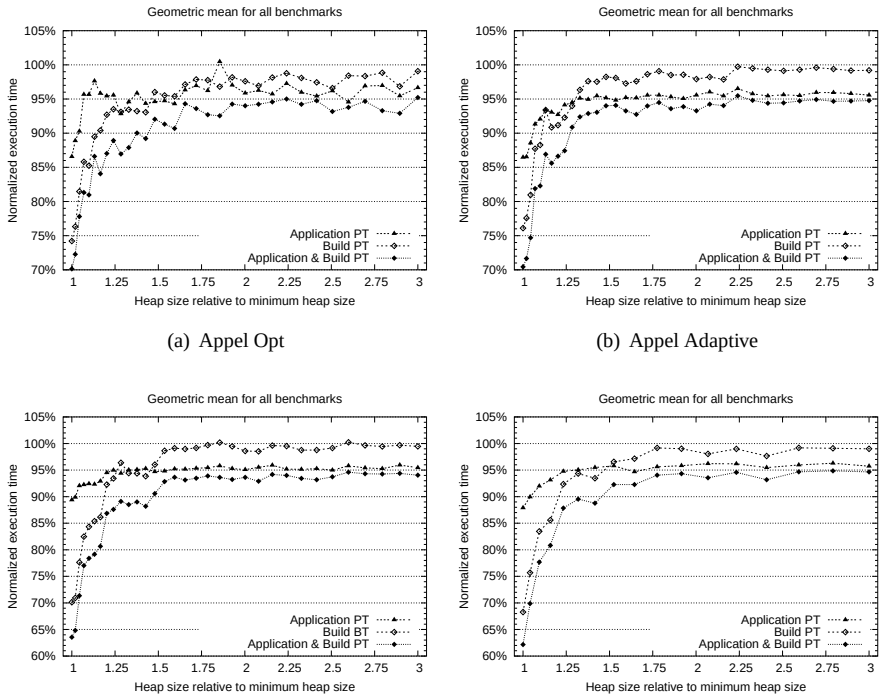
Fig. 25. Olden Health (6,128) and pseudojbb results, and Geometric Means: Second Iteration GC and Execution Time Relative to Non-Pretenuring

Generally, we see improvements in locality rather than degradations. The L1 miss rate ratio curves are similar to our performance curves, the L2 miss rate ratios indicate general improvement, and the TLB miss rate ratios show that build-time and combined pretenuring usually reduce TLB miss rates but application-specific pretenuring sometimes gives reductions and sometimes improvements. Therefore, pretenuring is not overly disturbing the good locality of nursery allocation in a contiguous region [Blackburn et al. 2004a], nor is it degrading GC



(a) Health GCs 2 (b) PseudoJBB GCs 2

Fig. 26. Olden Health (6,128) and pseudojbb: Second Iteration Number of GCs



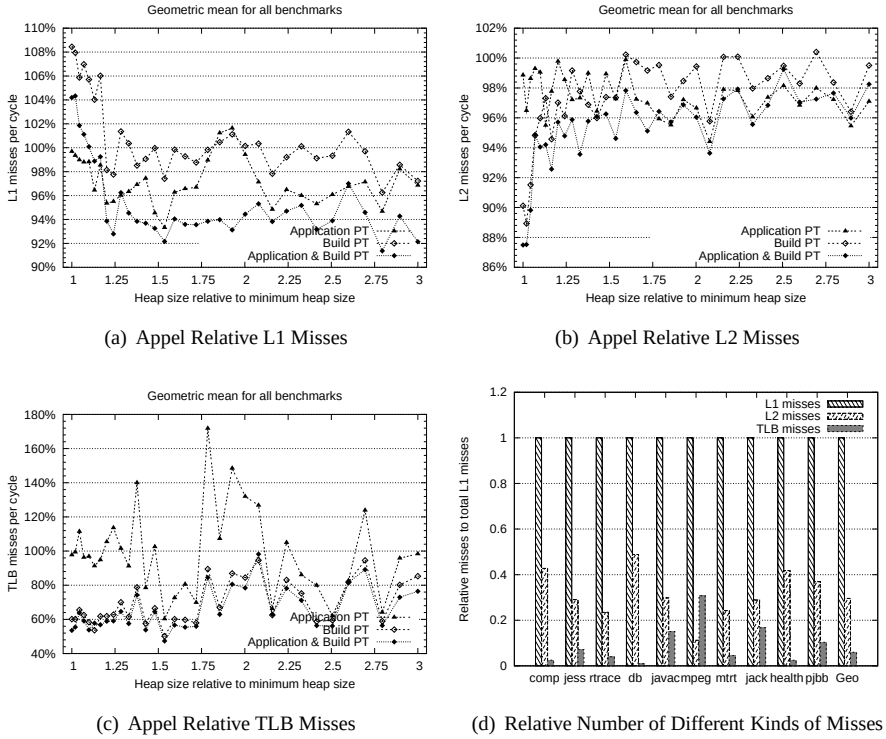
(a) Appel Opt (b) Appel Adaptive

(c) Appel Replay: Advice from Opt traces (d) Appel Replay: Advice from Replay traces

Fig. 27. Comparing Pretenuring Under Different Compilation Options

locality.

Finally, we ask the question: How does pretenuring affect *mutator* execution time (as opposed to the collector)? This indirectly indicates locality benefits, since mutator instruction execution should be quite comparable with and without pretenuring. Figure 29 shows just mutator time under build-time, application-specific, and combined pretenuring, each relative



(a) Appel Relative L1 Misses

(b) Appel Relative L2 Misses

(c) Appel Relative TLB Misses

(d) Relative Number of Different Kinds of Misses

Fig. 28. Comparing Cache and TLB Locality

to no pretenuring. There is a set of bars for each benchmark program and for the geometric mean. We see there is one case of more than minimal slow down: 4% for build-time pretenuring on raytrace. In most cases there is little effect, but in several cases there is considerable reduction in mutator time under pretenuring, most notable db and to a lesser extent health. We observe that db is known to be highly sensitive to exact layout of heap data, since it repeatedly traverses long singly-linked lists, which cautions reading too much from the db results. Still, we find on average a slight reduction in mutator time when pretenuring.

6.10 Application-Specific Advice with Other Inputs

Space precludes thorough consideration of how well application-specific advice collected from one program run (trace) affects execution of the same program with different inputs. However, since the SPEC benchmarks come with different inputs "sizes", we performed some simple comparisons. The "sizes" available are 1 (intended only for testing that a program runs), 10, and 100. We use size 100 runs to develop our traces and in all the other evaluations presented here. Figure 30 shows the geometric mean of size 10 performance relative to non-pretenuring. This averages the eight SPEC benchmarks, plus health run at with parameters (5,128).[15] As the figure shows, we still see improvement, though not as great a fraction, probably because

[15]It seemed pointless to run pseudojbb just for a shorter time, since the behavior would be so self-similar. The same may be true of some SPEC benchmarks.

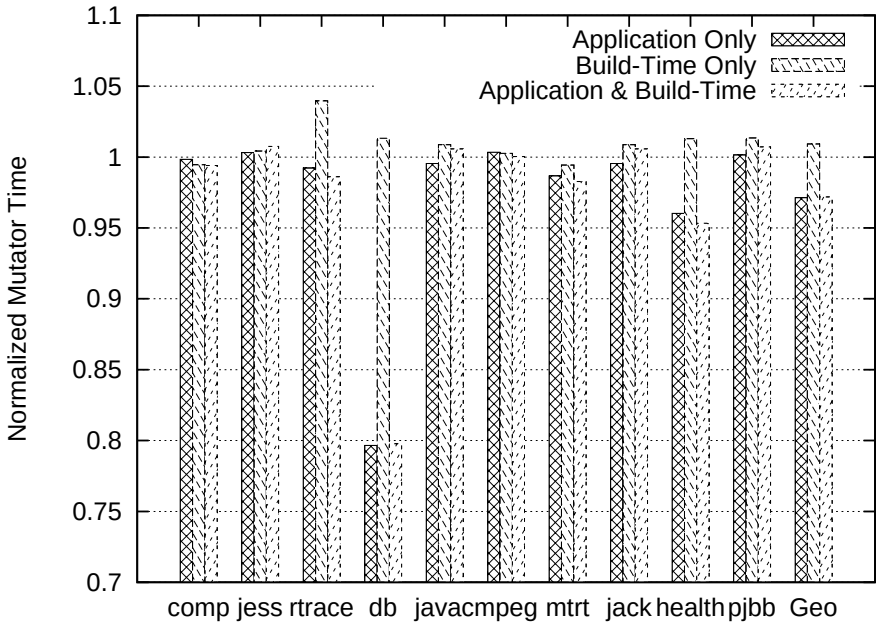


Fig. 29. Comparing Mutator Time to Estimate Mutator Locality Effects

these runs are so short and thus do relatively less allocation and collection. Still, we see that the advice is never harmful, that build-time pretenuring remains useful, and that application-specific and build-time pretenuring provide cumulative benefit.

We also explored developing advice from size 10 traces, but found that they did not run long enough to produce very useful application-specific advice: the runs were dominated relatively more by compilation. Also, in short runs there is an increased risk of labeling as immortal objects (and sites) that should not be, just because the run was not long enough for our criterion to weed them out.

6.11 Pretenuring with Other Collectors

We now consider the question of how well our pretenuring advice works with other age-based collectors. Specifically, we consider the Beltway and the Older First (OF) collectors. It is important to emphasize that we use *exactly* the same collector-neutral pretenuring advice for all three collectors.

6.11.1 *Pretenuring with the Beltway Collector.* For the Beltway collector, we use the configuration 25.25.100, which is reported to perform well [Blackburn et al. 2002]. This configuration has three belts. The first belt is the nursery and its size is 25% of the usable space (12.5% of the total heap size, which includes the copy reserve). When the nursery belt is full, Beltway promotes survivors of nursery collections to the second belt, which consists of four increments, each sized up to 25% of the usable space. This belt can grow, provided the heap is not full. When the second belt is full, Beltway collects the oldest window of the belt, and promotes survivors to the third belt. The third belt has only one window, which can be as large as 100% of the usable space (50% of the total heap size). Collections on the third belt guarantee

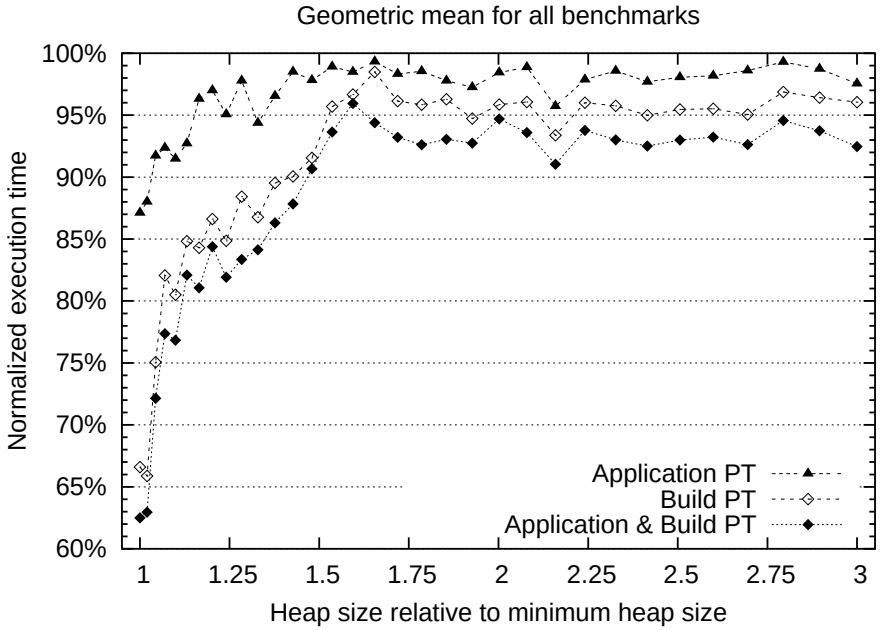


Fig. 30. Relative Execution Time of Size=10 Runs with Size=100 Advice

completeness of the collector (i.e., that it eventually collects any garbage object). When the heap is full and the other belts are empty, Beltway collects the third belt. These collections are rare.

We added to Beltway an *immortal* belt that is never collected. With pretenuring advice, Beltway directly allocates immortal objects on the immortal belt. It puts long-lived objects into the youngest window of the second belt, so that they can stay on the second belt for the longest possible time before Beltway collects them.

Figure 31 shows the geometric mean of the relative performance for all our benchmarks with the modified Beltway collector, normalized with respect to the Beltway collector without pretenuring. We show build-time, application-specific, and combined pretenuring results. Application-specific Beltway pretenuring always improves performance by about 4–6%, except only for the tightest heap sizes, where javac and pseudojbb suffer from nepotism and experience degradation of 30% and 8%, respectively. All other benchmarks have substantial improvement with application-specific pretenuring at tight heap sizes, so we observe only a 3% degradation in the geometric mean. Build-time Beltway pretenuring improves performance by up to 12% in tight heaps, and by about 2% for larger heaps. The improvements for combined Beltway pretenuring are about 15% in tight heaps, and 7% in larger heaps. Note that we achieve less benefit from pretenuring in tight heaps than we do for the Appel-style generational collector. Beltway's performance advantages over generational collection without pretenuring come partly because Beltway uses a dynamic copy reserve and thus uses heap space more efficiently. Hence, pretenuring gives relatively less benefit to Beltway.

6.11.2 *Pretenuring with the OF Collector.* We found that the same advice can improve an Older First (OF) collector [Stefanović et al. 1999]. The OF collector organizes the heap

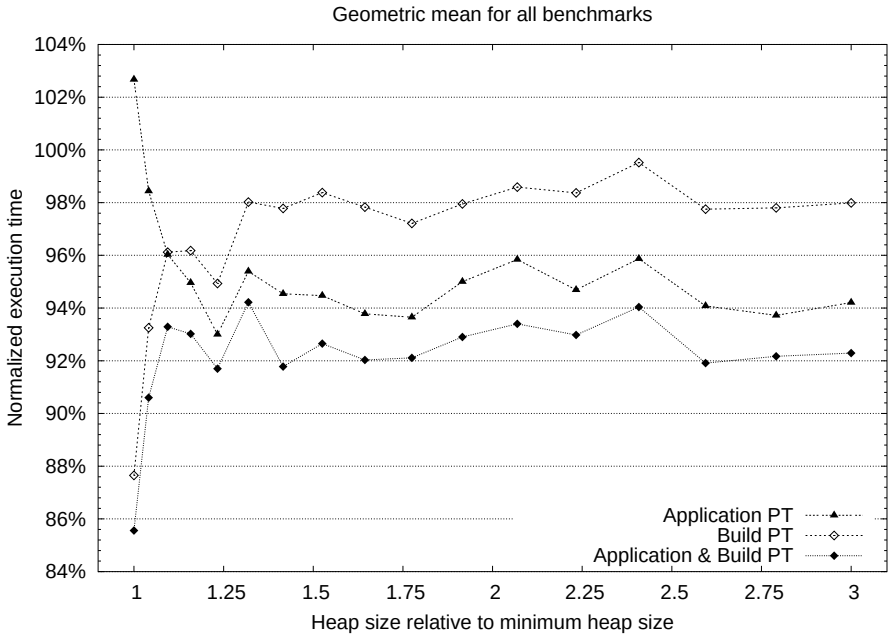


Fig. 31. Relative Execution Time for Pretenuring with the Beltway Collector

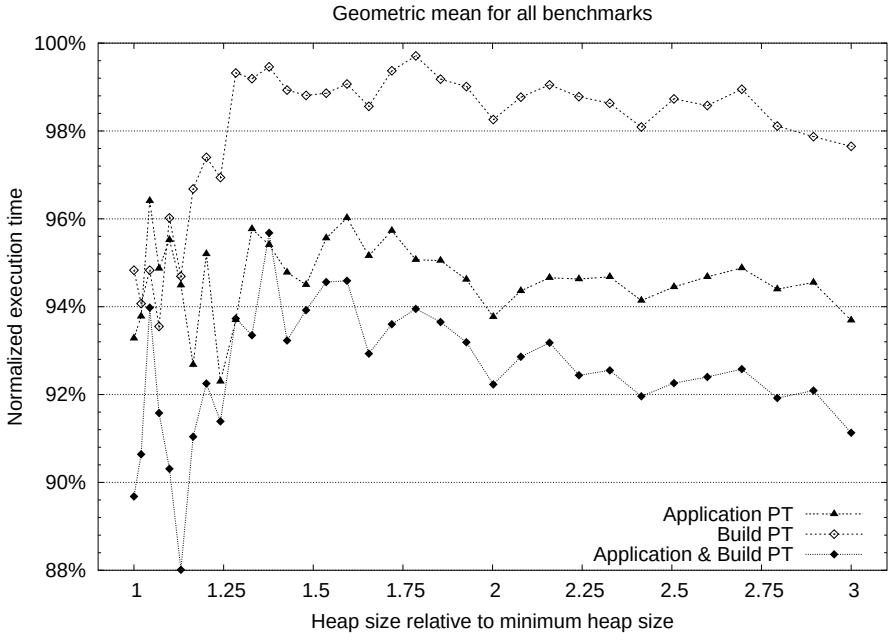


Fig. 32. Relative Execution Time for Pretenuring with the OF Collector

in allocation order. View the heap as a queue; the oldest objects are at the tail and the OF allocator inserts newly allocated objects at the head of the queue. OF begins by positioning the window of collection at the end of the queue, which contains the oldest objects. During a collection, it copies and compacts the survivors in place, returns free blocks to the head of the queue, and then positions the window closer to the front of the queue, just past the survivors of the current collection. When it bumps into the allocation point for the youngest objects, it resets the window to the oldest objects. See Stefanović et al. for more details [Stefanović et al. 1999].

With pretenuring advice, OF puts immortal objects in a reserved space that is never collected. OF allocates long-lived objects at the copy point for the previous collection, which gives them the longest possible time before OF will consider them for collection. OF continues to put short-lived objects at the head of the queue. As with the generational collector, we use a fixed sized heap, reduced by the space allocated to immortal objects. We set the collection window size to $0.3\times$ heap size.

Figure 32 shows the geometric mean of the relative performance for all our benchmarks, normalized with respect to the OF collector without pretenuring, for build-time, application-specific, and combined pretenuring. Application-specific OF pretenuring improves performance in all cases, ranging from 3% to 8% Again, build-time pretenuring improves performance, and additional improvements from combined pretenuring are consistent and significant, ranging from 4% to 12%.

Since the OF collector visits older objects more regularly than the generational collector, there is potential for better improvements, and it is realized in these results. However, our implementation of the OF collector is currently not well tuned, and does not include key details such as an address order write barrier [Stefanović et al. 1999]. These drawbacks prevent direct comparisons between the performance of the OF and generational collectors with or without pretenuring. Indeed, these comparisons are not pertinent to the subject of this work. The key point of this section is that we can use the same advice in these vastly different collectors and it improves performance as well.

7. USING PRETENURING IN PRACTICE

Here we used GC configurations suited to clarity of experimental methodology. In practice one would probably adjust some of the policies to produce a system more convenient for production use:

—We used a fixed total size for the heap to ease comparisons, but in practice adaptive heap growth and shrinkage is more appropriate. If a program runs a moderate length of time and accumulates some amount of garbage in the immortal space (and nepotism in the long space), growing may be an easy way to handle the problem if the space "leak" is not very great. (One may also adjust heap size in response to available real memory, as explored by Yang et al. [Yang et al. 2004].)

—While our technique appears reliable, it does not *guarantee* to bound growth of the immortal space [Boehm 2002] or of objects in other spaces retained because of nepotism. Thus one might apply a "back up" collector from time to time, e.g., global marking, either separately or as part of an older generation collection. This can determine the volume of dead objects in the immortal space, and of those objects retained in other spaces because of them. If the dead immortal volume is relatively large, one could apply sliding compaction (say) to the immortal space. (Current versions of GCTk's successor package, MMTk [Blackburn et al.

2004b; 2004a], make this relatively easy to build and configure.) If the volume is relatively small, one could zero out the bodies of the dead immortal objects, preventing long-term nepotism.

—An additional concern is programs that have popular allocation sites with poor lifetime homogeneity, or that profiling mispredicts so that we generate inappropriate immortal or long advice for them. One possible direction is to apply pretenuring *adaptively* [Harris 2000; Jump et al. 2004]. These designs would need extensions to deal with an immortal space. It may be reasonable to use a simpler mechanism, where we mark all immortals objects with their allocation site (or perhaps just a sampled fraction of them). If our backup immortal space collection mechanism detects particular offending allocation sites, we can patch the site to allocate to a shorter lived space.

—A particular concern about quality of advice is that certain coding practices may increase the lifetime heterogeneity of allocation sites. In particular, *factory methods*, i.e., methods whose purpose is to allocate an object on behalf of their caller, may tend to contain allocation sites with more heterogeneous lifetimes. This effect is mitigated if the factory methods are inlined. Increased lifetime heterogeneity will usually just reduce the potential benefit of pretenuring by disqualifying more allocation sites.

—The concept of pretenuring applies to generational systems, and is not particular to copying GCs. Would our pretenuring scheme be effective for, say, a system uses linear allocation into a nursery, but manages older objects with mark-sweep or occasional sliding compaction? One expects some shifting of design points, but we argue that the same general approach is likely to produce useful benefits, though perhaps not as great as seen with copying. In our scheme the immortal space is particularly helpful because it needs no copy reserve and this effectively frees space for allocation in younger generations. That is, it effectively increases the heap size. In non-copying systems or ones that copy out of a nursery to non-copying spaces, there would be no (additional) space benefit. However, one still obtains a processing time benefit, assuming that the strategy for most collections does not need to examine most of immortal space. (Thus one would prefer remembered sets that usually remember particular referring *slots* rather than remembering referring *objects*.)

7.1 Online Pretenuring?

Our approach is based on feeding back profile information from previous, instrumented runs of programs. Could it be applied online? While we compile into generated machine code the region into which each allocation site is to allocate, it is possible to change the allocation site on the fly by patching, or regenerating the code, or inserting a test. However, it is conceptually problematic to apply online anything like our definition of immortal objects, since it depends on knowing when the program will end. As we discuss in more detail below, neither of the online pretenuring schemes [Harris 2000; Jump et al. 2004] obtain much improvement.

7.2 Whither Profile Feedback?

Doing profile feedback is tedious for users, so the most obvious way to exploit our approach is to use build-time pretenuring, which has significant benefit and may be more reliable than application-specific pretenuring decisions. Also, even though Merlin is much faster than previous techniques, the slowdown for even a granulated trace is 20–80x, further suggesting build-time pretenuring as being more reasonable in most instances.

However, one can imagine collecting, at modest overhead, somewhat coarse-grained object lifetime statistics from many runs and integrating them into a database. One could run an analyzer and advice generator on this database periodically, and future runs could use the advice. This is a way to make the feedback automatic and non-intrusive, an interesting idea for future work.

8. RELATED WORK

We first compare our work to previous research on generational garbage collectors, object lifetime prediction, and pretenuring. We then relate it to work on prediction and object segregation for C programs with explicit allocation and freeing.

Ungar pioneered the use of generational copying garbage collection to effect quick reclamation of the many short-lived objects in Smalltalk programs [Ungar 1984]. Performance studies with a variety of languages demonstrate that well-tuned generational collector performance generally ranges from 10% to 40% of the total execution time [Ungar and Jackson 1988; Zorn 1989; Ungar and Jackson 1992; Barrett and Zorn 1995; Tarditi and Diwan 1996; Cheng et al. 1998; Blackburn et al. 2004a].

Ungar and Jackson use online profiling to identify longer-lived objects in a two generation collector for Smalltalk [Ungar and Jackson 1988; 1992]. Their tenured object space corresponds roughly to our immortal space, in that they never collect it. However, they do not pretenure (allocate any objects directly into tenured space). Rather, they copy into tenured space objects that survive a given number of nursery collections. They adjust this number, the *tenuring threshold*, by keeping track of the volume of nursery objects that has survived one collection, two collections, etc. Thus, their system keeps long-lived objects in the nursery, repeatedly copying them to keep from tenuring them, in order to avoid tenured garbage. They use the object demographics that they obtain from a given nursery collection to set the tenuring threshold for the next collection. The goal is to tenure as few bytes as possible while keeping the nursery space from growing too large and thus exhibiting unacceptable pause times when it is collected. They further outline a multi-generational approach that would copy the long-lived objects fewer times. They notice immortal objects, but since those were insignificant in their system, they take no special action. We allocate immortal objects directly into a permanent space. We thus never copy immortal objects. We also have the potential never to copy long-lived objects, but we may.

Cheng et al. (CHL) evaluate pretenuring and lifetime prediction for ML programs in the context of a generational collector [Cheng et al. 1998]. Similar to Ungar and Jackson, they divide the heap into two regions: a fixed size nursery and an older generation. They collect the nursery on every collection, and both spaces when the entire heap fills up. They generate pretenuring advice based on profiles of this collector, and classify call sites as short-lived or long-lived. Most objects are short-lived, and allocation sites are bimodal: either almost all objects are short-lived, or all are long lived. Their advice is dependent on their collection algorithm and the specific configuration, whereas our pretenuring advice is based on two collector-neutral statistics: age and time of death. We therefore can and do use it with different configurations of a generational collector, and with altogether different collectors, Older First and Beltway.

CHL statically modify those allocation sites where 80% or more of objects are long-lived to allocate directly into the older generation, which is collected less frequently than the nursery. We allocate instead into three areas: the nursery, the older generation, or the permanent space. We never collect our permanent space. At collection time, their system *must scan* all pre-

tenured objects because they believed that the write barrier cost for storing pointers from the pretenured objects into the nursery would be prohibitive. We instead perform the write barrier as needed; this cost is very small in our case. The cost of scanning is significant [Cheng et al. 1998; Stefanović et al. 1999], and as they point out, it reduces the effectiveness of pretenuring in their system. We never collect immortal objects, and only collect long-lived objects later when they have had time to die. In summary, our pretenuring classification is more general, and our collectors more fully realize the potential of pretenuring. Most importantly, the more general mechanism we use to gather statistics and generate advice enables our system to combine advice from different executions and perform build-time pretenuring, which is not possible in their framework.

Harris, and Jump et al., present dynamic pretenuring schemes. Harris [Harris 2000] samples using Agesen and Garthwaite's [Agesen and Garthwaite 2000] approach, which inserts weak pointers and after a collection computes object lifetime statistics. Harris then pretenures into the older generation of a two generation collector, and samples older objects to stop pretenuring, and thus can react to phase changes. He does not report accuracy or overhead, but does not improve performance. The dynamic pretenuring approach of Jump et al. [Jump et al. 2004] improves only one program, javac. However, they develop an inexpensive and accurate mechanism for tracking object lifetimes based on frequent samples (one object out of every 256 bytes allocated).

For many benchmarks, dynamic pretenuring will always suffer because programs often allocate a high proportion of immortal and long-lived objects at the very beginning of the program [Jump et al. 2004], before any dynamic scheme has time to train itself. Since static pretenuring relies on prior runs, it is not subject to this drawback. Furthermore, it is accurate and improves performance. However, it does require a profiling run and does not respond to phase changes.

For explicit allocation and deallocation in C programs, Hanson performs object segregation of short-lived and all other objects on a per allocation site basis with user specified object lifetimes [Hanson 1990]. Barrett and Zorn extend Hanson's algorithm by using profile data to predict short-lived objects automatically [Barrett and Zorn 1993]. To achieve accurate results, their predictor uses the dynamic call chain and object size, whereas we show that in Java prediction does well with only the allocation site. Subsequent work by Seidl and Zorn predicts short-lived objects with only the call chain [Seidl and Zorn 1998]. In these three studies, a majority of objects are short-lived, and the goal is to group them together to improve locality (and thus performance) by reusing the same memory quickly. Barrett and Zorn's allocator dynamically chooses between a special area for the short-lived objects, and the default heap. Because we attain accurate prediction for an allocation site, we indicate statically where to place each object in the heap, which is cheaper than dynamically examining and hashing on the call chain at each allocation. Since in their context "long-lived" is the conservative assumption, Barrett and Zorn predict "short-lived" only for call chains where 100% of the allocations profile to short lived. In a garbage collected system, our conservative prediction is instead "short-lived". We also differentiate between long-lived and immortal objects, which they do not.

Demers et al. [Demers et al. 1990] looked at other ways of identifying allocation sites with context, in particular using stack pointer values as a cheap approximation to detailed calling context. On contrast to this run-time technique, We need a *static* prediction, since we compile in the choice of allocation area. Of course it might be possible to apply our static prediction to highly homogeneous sites and a more contextual one to more heterogeneous sites, but we obtained good results with the static predictor.

The work we present here adds several dimensions over our prior work [Blackburn et al. 2001]. We now use exact life time information to generate advice, while our previous advice used frequent collections (e.g., every 64K bytes of allocation), a technique that over-estimates object lifetimes. This change prompted a revised advice classification scheme whose sensitivities we explore experimentally. Our new technique improves the quality of our advice. Consequently, this advice significantly improves application pretenuring. Now application pretenuring improves performance consistently, whereas in our previous work it did not. We further use a more modern Java compilation strategy (Adaptive), modified to produce deterministic, yet realistic results (Replay), and see that it affects the relative impact of application-specific versus build-time pretenuring—because there is less allocation by the optimizing compiler, build-time pretenuring has relatively less impact, though it is usually still useful. We also add a richer set of benchmarks, more in-depth analysis, and the Beltway collector to our results. We also include here second-iteration results, cache and TLB miss performance, results from using long-run advice for short runs, and more statistics for the non-pretenuring collector that is the standard against which we compare. These additions further demonstrate the applicability and generality of our approach.

A technique somewhat complementary to pretenuring is a *large object space* (LOS) [Caudill and Wirfs-Brock 1986; Ungar and Jackson 1992; Hicks et al. 1998]. One allocates large objects (ones exceeding a chosen size threshold) directly into a non-copying space, effectively applying mark-sweep techniques to them. This technique avoids ever copying these objects, and can noticeably improve performance. GCTk does not support LOS, so we do not compare here the relative benefits of LOS and pretenuring. Some JVMs allocate large objects directly into older spaces; i.e., they use size as a criterion for pretenuring. (These older spaces may also be mark-sweep, so they are effectively implementing pretenuring *and* LOS.) While pretenuring large objects may be generally helpful in a two-way classification system (a point that requires further analysis), it could be disastrous to pretenure into our immortal space using size as the sole criterion. The compress benchmark is an example of this: it allocates and discards large arrays.

9. CONCLUSIONS

This paper makes several unique contributions. It offers a new mechanism for collecting and combining pretenuring advice, and a novel and generalizable classification scheme. We show application-specific pretenuring using profiling works well for Java. Our per-site classification scheme for Java finds many opportunities to pretenure objects; to reduce copying, garbage collection time; and to reduce total time, sometimes significantly. We show that the combination of build-time and application-specific pretenuring offers the best improvements. We are the first to demonstrate the effectiveness of build-time pretenuring, and we do so using *true* advice. Because Jikes RVM is written in Java for Java, we profile it and any libraries we choose to include, combine the advice, then build the JVM and libraries with that advice, and ship. User applications thus can benefit from pretenuring without any profiling. These results thus demonstrate an advantage of the Java-in-Java approach.

Acknowledgments

We thank Sara Smolensky who did the first studies that inspired this work. We thank Sharad Singhai for his substantial contributions to an earlier version of this work. We also thank John Cavazos, Asjad Khan, and Narendran Sachindran for their contributions to various incarnations of this work. We thank our associate editor, Ben Zorn, and the anonymous reviewers for their

numerous helpful comments. Finally, we thank the members of the Jikes RVM (formerly Jalapeño) team at IBM T.J. Watson Research Center who helped facilitate this research, IBM Research for making Jikes RVM widely available, and the broader Jikes RVM community for their contributions to the research platform.

REFERENCES

AGESEN, O. AND GARTHWAITE, A. 2000. Efficient object sampling via weak references. In *Proceedings of the International Symposium on Memory Management*. ACM, Minneapolis, MN, 121–127.

ALPERN, B., ATTANASIO, D., BARTON, J. J., BURKE, M. G., P.CHENG, CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J., SMITH, S., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM System Journal 39*, 1 (Feb.), 211–238.

ALPERN, B., ATTANASIO, D., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., MERGEN, M., NGO, T., SHEPHERD, J., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *Proceedings of the ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. ACM Press, Denver, CO, 314–324.

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software–Practice and Experience 19*, 2, 171–183.

ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. ACM Press, Minneapolis, MN, 47–65.

BARRETT, D. A. AND ZORN, B. 1993. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, Albuquerque, NM, 187–196.

BARRETT, D. A. AND ZORN, B. 1995. Garbage collection using a dynamic threatening boundary. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, La Jolla, CA, 301–314.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004a. Myths and realities: The performance impact of garbage collection. In *ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*. ACM, NY, NY, 25–36.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004b. Oil and water? High performance garbage collection in Java with JMTk. In *Proceedings of the International Conference on Software Engineering*. ACM, Scotland, UK, 137–146.

BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND B.MOSS, J. E. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*. SIGPLAN, ACM Press, Berlin, Germany, 153–164.

BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINLEY, K. S., AND MOSS, J. E. B. 2001. Pretenuring for Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. SIGPLAN, ACM Press, Tampa, FL, 342–352.

BOEHM, H.-J. 2002. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM Press, New York, NY, USA, 93–100.

CAHOON, B. AND MCKINLEY, K. S. 2001. Data flow analysis for software prefetching linked data structures in Java. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society Press, Barcelona, Spain, 280–291.

CAUDILL, P. J. AND WIRFS-BROCK, A. 1986. A third-generation Smalltalk-80 implementation. In *OOPSLA'86 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, Portland, OR, 119–130.

CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, Montreal, Canada, 162–173.

DEMERS, A., WEISER, M., HAYES, B., BOEHM, H., BOBROW, D., AND SHENKER, S. 1990. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM Press, New York, NY, USA, 261–269.

EECKHOUT, L., GEORGES, A., AND BOSSCHERE, K. D. 2003. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. ACM, Anaheim, CA, 169–186.

HANSON, D. R. 1990. Fast allocation and deallocation of memory based on object liftimes. *Software—Practice and Experience 20*, 1 (Jan.), 5–12.

HARRIS, T. L. 2000. Dynamic adaptive pre-tenuring. In *Proceedings of the International Symposium On Memory Management (ISMM)*. ACM Press, Minneapolis, MN, 127–136.

HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., MCKINLEY, K. S., AND STEFANOVIĆ, D. 2002. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*. SIGMETRICS, ACM Press, Marina Del Rey, CA, 140–151.

HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., MCKINLEY, K. S., AND STEFANOVIĆ, D. 2005. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*. To appear.

HICKS, M., HORNOF, L., MOORE, J. T., AND NETTLES, S. 1998. A study of Large Object Spaces. In *ISMM'98 Proceedings of the First International Symposium on Memory Management*. ACM Press, Vancouver, BC, 138–145.

HUANG, X., WANG, Z., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., AND CHENG, P. 2004. The garbage collection advantage: Improving mutator locality. In *Proceedings of the ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*. ACM, Vancouver, BC, 69–80.

JUMP, M., BLACKBURN, S. M., AND MCKINLEY, K. S. 2004. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM Press, New York, NY, USA, 152–162.

LEE, H. B. AND ZORN, B. G. 1997. BIT: A tool for instrumenting Java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*. USENIX Association, Monterey, CA, 73–82.

LIEBERMAN, H. AND HEWITT, C. E. 1983. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6, 419–429.

ROGERS, A., CARLISLE, M. C., REPPY, J. H., AND HENDREN, L. J. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 2 (Mar.), 233–263.

SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, San Jose, CA, 12–23.

STEFANOVIĆ, D., HERTZ, M., BLACKBURN, S. M., MCKINLEY, K., AND MOSS, J. 2002. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Memory System Performance*. ACM Press, Berlin, Germany.

STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. 1999. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*. ACM Press, Denver, CO, 379–381.

TARDITI, D. AND DIWAN, A. 1996. Measuring the cost of storage management. *Lisp and Symbolic Computation 9*, 4 (Dec.), 323–342.

UNGAR, D. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, Pittsburgh, PA, 157–167.

UNGAR, D. AND JACKSON, F. 1988. Tenuring policies for generation-based storage reclamation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, N. K. Meyrowitz, Ed. ACM Press, San Diego, CA, 1–17.

UNGAR, D. AND JACKSON, F. 1992. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS) 14*, 1, 1–27.

YANG, T., HERTZ, M., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. 2004. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 2004 International Symposium on Memory Management*. ACM SIGPLAN, ACM Press, Vancouver, BC, 61–72.

ZORN, B. 1989. Comparative performance evaluation of garbage collection algorithms. Ph.D. thesis, Computer Science Dept., University of California, Berkeley. Available as Technical Report UCB/CSD 89/544.

Received July 2004; revised December 2005; accepted January 2006