Pretenuring for Java

Stephen M. Blackburn Sharad Singhai Matthew Hertz
Kathryn S. McKinley J. Eliot B. Moss

Architecture and Language Implementation Laboratory, Department of Computer Science
University of Massachusetts, Amherst, MA 01003-4610

ABSTRACT

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 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 applications and the Jalapeño JVM, a compiler and run-time system for Java written in Java. Our results demonstrate that building combined advice into Jalapeño from different application executions improves performance regardless of the application Jalapeño is compiling and executing. This *build-time* advice thus gives user applications some benefits of pretenuring without any application profiling. No previous work pretenures in the run-time system. (3) We find that application-only advice also 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 and Older First collection, illustrating that it is *collector neutral*.

General Terms

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 because of premature reclamation. The growing use and popularity of Java, 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* [17] and demonstrate the generality of our approach with the *Older First* collector [15].

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 only collect successively older generations if necessary.

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.

Without an oracle, pretenuring advice can be gleaned from application profiling on a per allocation-site [8] or call-chain [4, 14] basis. For our suite of Java programs, we show that allocation-site advice results in accurate predictions, and these predictions are robust over different input data. ML programs are similar [8], whereas C programs need the additional context of a call-chain [4, 14].

We use two object lifetime statistics (measured in bytes allocated): *lifetime* and *time of death*. Object lifetime is how long an object lives (in bytes of allocation), and time of death is the point in the allocation history of the program at which the object becomes unreachable. Our advice 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.

*This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, NSF grant EIA-9726401, NSF Infrastructure grant CDA-9502639, DARPA grant 5-21425, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Cheng, Harper, and Lee (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* [8]. CHL profile a given application and generational collector configuration to generate pretenuring advice. We instead use frequent full-heap collections to generate the lifetime statistics from which we derive our advice, a more costly process. Because our statistics are collector- and configuration-neutral, they are more general.

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 internal to the JVM, and libraries). (2) We can and do use the advice to improve two distinct collectors, an Appel-style generational collector [3] and an Older First collector [15], on five benchmarks, three from SPEC JVM98.

In our experiments, we use the Jalapeño JVM [2, 1], a compiler and run-time system for Java written in Java, extended with an Appel-style generational collector. We profile all our benchmarks, and then combine their pretenuring advice to improve the performance of Jalapeño itself; we call this system *build-time pretenuring*. Because CHL profile advice is specific to both the application and collector configuration, they cannot readily combine advice for this purpose. When measuring the effectiveness of our build-time pretenuring, we omit information from the application to be measured from the combined advice. Such advice is called *true* advice [4]. We show that build-time pretenuring improves the performance of Jalapeño running our benchmarks an average of 8% 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. Building pretenuring into the JVM before distribution means users will benefit from pretenuring without profiling their applications. Just using our application-specific profile advice always improves performance, too: up to 3.5% on average for tight heaps. Our advice is also on average comparable to using CHL advice, and is significantly better for tight heaps. Combining our build-time and application-specific advice always yields the best performance: it decreases garbage collection time on average by 20% to 32% for most heap configurations. It improves total execution time on average by 7% for a tight heap.

The remainder of this paper is organized as follows. Section 3 discusses our approach to pretenuring and the collection and generation of pretenuring advice. It also analyzes the lifetime behaviors of objects in our Java applications. We then describe our experimental methodology and setting in Section 4. Section 5 presents execution time results for pretenuring with generational collection for the Jalapeño JVM 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. We then compare related work with our approach, and conclude.

2. Background

For this paper we built an Appel style generational collector [3] that partitions the heap into a nursery and a second, older, generation. It also has a separate, permanent space that is never collected. The total heap size is fixed. The nursery size is flexible: it is the space not used by the older generation and the permanent space. 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, it 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.

3. Pretenuring Advice

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 and Generating Pretenuring Advice

Any algorithm for generating pretenuring advice must consider the two major cost components: *copying* and *space rental*. The copying cost includes scanning and copying an object when it survives a collection. The space rental cost is the space in memory occupied by objects over time. On the two extremes, pretenuring advice that recommends pretenuring *all* objects into permanent space minimizes copying costs but incurs a high space rental cost; and advice that recommends pretenuring *no* objects minimizes space rental cost 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 [8], 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.

Following the garbage collection convention of equating time to bytes allocated, we normalize age with respect to *max live size*. *Max live size* refers to the maximum amount of live objects in a program execution, which indicates the *theoretical* minimum memory requirement of a program. 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. Object age is 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 occurs.

The relationships between object age, time of death, max live size, and total allocation are illustrated in Figure 1 for a Java version of health running a small input set, where one point is plotted for each object's age and time of death. The top and left axes normalize 'time' with respect to total bytes allocated for that program, while the bottom and right axes show time with respect to the program's max live size, which relates to a 'heap full' of allocation. This figure shows that a large number of objects have short lifetimes, and the vertical 'lines' of points indicate that throughout the life 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 $0.6 \times$ max live size).

[1] The relationship between max live size and total allocation is a function of allocation behavior. In our Java programs, total allocation ranges from 11 to 113 times max live size.

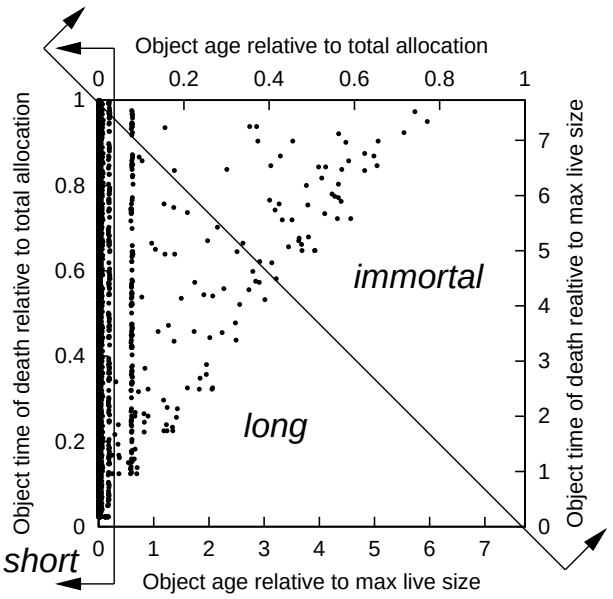


Figure 1: Object Age and Death Distributions for health (3-128)

Object Lifetime Profiling

We analyze age and lifetime statistics using an execution profile for each application. The profile takes the form of an object graph mutation trace, which records all object allocations, pointer mutations, and object deaths. We log all allocations, all heap pointer mutations, and when the collector frees an object. To obtain accurate object death information, we trigger a (non-generational) full heap collection frequently (once every 64KB of allocation). We gather age and time of death statistics for each object, as well as the max live size and total allocation for the application.

Binning

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

1. If an object dies more than halfway between its time of birth and the end of the program, we bin it as *immortal*.
2. Otherwise, if an object's age is less than $T_a \times$ max live size bytes, then it is binned as *short*.
3. In all other cases, an object is binned *long*.

We use $T_a = 0.2$ in our experiments below. Our *immortal* classification criterion is based on the 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), an object should be classified as immortal if *dead time*/*lifetime* ≤ 1 for that object, where dead time is the time from when the object dies to the end of the program. Figure 1 illustrates this categorization.

Allocation Site Classification

Using the bins, we then classify a site. Given an allocation site that allocates a fraction S_f of short-lived objects, L_f of long-lived objects, and I_f of immortal objects, we classify it using a homogeneity threshold H_f as follows:

1. If $S_f + H_f > L_f + I_f$ we classify the site *short*.
2. Otherwise, if $S_f + L_f + H_f > I_f$, we classify the site *long*.
3. In all other cases, we classify the site *immortal*.

Thus we make a conservative classification (short) for the site if that is the most common case, and the less conservative classifications (long or immortal) when they are sufficiently more common than the other choices. For example, given fractions of 0.1 short, 0.35 long, and 0.55 immortal, we classify the site 'long' if $H_f > (.55 - .35 - 0.1) = 0.1$, but if $H_f \leq 0.1$, we classify it 'immortal.' We use $H_f = .33$ in our experiments. We find our binning and classification fairly insensitive to reasonable choices of T_a and H_f.

Combining Classifications from Different Program Executions

We are also able to combine data from different program executions to generate pretenuring advice. Our trace combining algorithm works as follows. For each site, we generate new combined bins s_c, l_c, i_c. 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 following combined bins for all sites with trace information. Let $w_c = \sum_{t=1}^{n} w_t$.

$$s_c = (\sum_{t=1}^{n} s_t * w_t)/w_c$$

$$l_c = (\sum_{t=1}^{n} l_t * w_t)/w_c$$

$$i_c = (\sum_{t=1}^{n} i_t * w_t)/w_c$$

With these bins, we then use the same classification algorithm as above but with a different homogeneity factor, which we call the combining homogeneity factor, H_{cf}. We found that it was important to be particularly conservative when combining traces, so we used $H_{cf} = 0.9$.

3.2 Testing the Homogeneity Premise

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' [8]. C programs are not homogeneous at each call site, but require the dynamic call chain to predict similar classes of lifetimes [4, 14]. We show in this section that the allocation sites in our set of Java programs have homogeneous lifetimes with respect to our new classification scheme.

We use 3 benchmarks from the SPEC JVM98 suite: _202_jess, _213_javac, and _228_jack, plus IBM's pBOB [6], on which SPEC JBB 2000 is based, and health, an object-oriented Java version of the Olden C program that models a health care system [13]. We choose these programs because they exercise the garbage collector. We explicitly exclude other SPEC JVM98 benchmarks such as _201_compress because they have a low ratio of total heap size to

max live size and thus do not exercise garbage collection. Table 1 contains the total allocation in bytes, maximum live size in bytes, and the ratio between the two, for each benchmark.

Benchmark	Live	Alloc	Alloc/Live
jess	4,340,224	493,363,764	113
javac	12,450,043	651,452,676	52
jack	7,216,975	517,214,752	72
pBOB	36,272,138	678,600,124	18
health (6-128)	3,503,011	39,679,440	11

Table 1: Benchmark Characteristics: (Live) is maximum live size in bytes, (Alloc) is total allocation in bytes.

We present two types of results in the remainder of this section. For javac 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 site to demonstrate the quality of our advice.

Binning and Classification

Table 2 shows our per-site bins and object classification for javac and our combined advice for the Jalapeño build-time system. We include the top 14 sites ranked by their space rental costs, where space rental is the size $\times$ lifetime product for each of the objects allocated at that site as a percentage of total space rental, and present a cumulative total of space rental costs. Clearly, we exclude many sites in this presentation. The low cumulative total for space rental demonstrates that there are very many sites contributing to the total allocation.

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 $T_a = 0.2,\ H_f = 0.33,$ and $H_{cf} = 0.9$, we show our resulting classification. Notice that many allocation sites are 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.

Table 3 summarizes the level of classification accuracy for each of our benchmarks. We classify objects as (short (s), long (l), and immortal (i)) on both a *per-object* and *per-site* basis. We examine the per-object (exact, indicated with subscript o) and per-site (representative, indicated with subscript s) decisions for each object to establish the level of error in the per site decisions.

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 ($\langle s_o, s_s\rangle$, $\langle l_o, s_s\rangle$, and $\langle i_o, s_s\rangle$). Bad pretenuring advice allocates objects into a longer lived region than appropriate ($\langle s_o, l_s\rangle$, $\langle l_o, i_s\rangle$, and $\langle s_o, i_s\rangle$). Following bad advice tends to waste space. Good pretenuring advice allocates objects into longer lived regions, but not too long lived ($\langle i_o, i_s\rangle$, $\langle l_o, l_s\rangle$, and $\langle i_o, l_s\rangle$). Following good advice reduces copying without wasting space. Table 3 indicates that on average, 44.3% of our advice is "good", 52.3% is "neutral", and only about 3.4% is "bad".

4. Methodology

This section begins by describing how we use pretenuring advice, then it overviews the Jalapeño JVM and the GC toolkit we built for this exploration. We then discuss how we measure and configure our system.

4.1 Using Pretenuring Advice

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

We have modified the Jalapeño compiler 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 $\langle$site string, advice$\rangle$ pairs, where the site string identifies a particular bytecode within a class. By providing advice to the compiler at *build time* (when building the Jalapeño boot image [1]), allocation sites compiled into the boot image, including the Jalapeño run-time system, 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, one needs to provide advice only for long-lived and immortal sites.

In application-specific pretenuring, we use *self* advice [4], 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 [12]. Similarly, IBM's Jikes Bytecode Toolkit[2] allows bytecode manipulation. Since our pretenuring advice is implemented inside Jalapeño, we manipulate the intermediate representation directly. Also, for build-time pretenuring, we avoid modifying a large number of Jalapeño class files by using just one simple text file for all pretenuring advice.

4.2 The Jalapeño JVM and the GC Toolkit

We use the Jalapeño JVM for our implementation study [1]. Jalapeño is a high performance JVM written in Java. Because Jalapeño 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 Jalapeño means that the addition of pretenuring to Jalapeño (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 have developed GCTk, a new GC toolkit for Jalapeño. It is an efficient and flexible platform for GC experimentation, that exploits the object-orientation of Java and JVM-in-Java property of

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

				% space rental %		% bin %			
	site	objects	volume	site	total	short	long	immortal	classification
javac	137	465394	9307880	13.082	13.082	55.74	35.22	9.04	s
	3301	145636	4077808	8.727	21.809	2.64	77.13	20.23	l
	3364	148676	2378816	5.556	27.365	38.10	51.28	10.62	s
	3361	96696	1547136	5.553	32.918	4.85	78.83	16.32	l
	3308	48328	1159872	3.783	36.701	0.56	54.70	44.74	l
	3310	49812	1793232	3.501	40.202	1.33	64.43	34.24	l
	3331	46924	791408	3.198	43.400	1.10	64.47	34.44	l
	3330	40156	1766864	2.734	46.134	1.10	65.09	33.81	l
	29	435580	14588780	2.256	48.390	93.53	3.61	2.86	s
	3327	382616	7652320	2.046	50.436	93.00	4.11	29.0	s
	3340	32956	763504	1.843	52.279	3.39	81.67	14.94	l
	3303	22684	635152	1.670	53.949	4.37	52.81	42.81	l
	3339	23980	575520	1.519	55.468	1.84	73.91	24.25	l
	103	4787	114888	1.491	56.959	5.31	16.94	77.75	i
combined	1992	725186	14503720	4.801	4.801	77.79	15.15	7.06	s
	1862	106212	1699392	3.595	8.396	65.62	23.03	11.35	s
	2442	45537	1466549	3.367	11.763	56.74	18.86	24.39	s
	2893	515773	19759078	3.358	15.122	51.86	15.62	32.52	s
	3266	1663676	53687928	3.214	18.336	72.42	26.03	1.55	s
	3111	127833	7877744	2.953	21.289	20.15	25.76	54.09	s
	2333	1191535	38129120	2.398	23.687	69.86	28.38	1.76	s
	1727	9813	235512	1.796	25.483	8.75	11.54	79.71	s
	1377	5461	89956	1.667	27.150	0.09	0.00	99.91	i
	3583	5453	567112	1.667	28.818	0.00	0.00	1.00	i
	424	15489	309780	1.633	30.451	47.22	15.75	37.03	s
	2934	38758	775160	1.628	32.079	40.06	32.54	27.40	s
	1499	5294	460012	1.620	33.699	0.00	0.00	1.00	i
	2182	5273	84368	1.610	35.309	0.00	0.00	1.00	i

Table 2: Per-site Object Binning and Classification

	% good %			% neutral %			% bad%		
benchmark	$< i_o, i_s >$	$< l_o, l_s >$	$< i_o, l_s >$	$< s_o, s_s >$	$< l_o, s_s >$	$< i_o, s_s >$	$< s_o, l_s >$	$< l_o, i_s >$	$< s_o, i_s >$
jess	41.6	0.1	0.0	47.3	3.9	5.8	0.0	0.4	0.7
javac	10.6	37.0	15.5	23.7	8.1	3.1	1.6	0.3	0.2
jack	29.6	7.6	0.8	46.2	7.8	5.6	1.3	0.2	0.9
health	25.5	3.5	1.7	42.0	19.5	5.8	1.4	0.3	0.3
pBOB	37.7	3.3	6.8	33.9	4.9	4.0	2.5	6.0	0.9
average	29.0	10.3	5.0	38.6	8.9	4.8	1.4	1.4	0.6

Table 3: Per-program Pretenuring Decision Accuracy (weighted by space rental cost)

Jalapeño. We have implemented a number of GC algorithms using GCTk and found their performance to be similar to that of the existing Jalapeño GC implementations. Our Appel-style generational collector is well tuned and uses a fast address-order write barrier [15]. We extend the algorithm in a straightforward way to include an uncollected region (for immortal objects). We recently implemented the Older First GC algorithm [15] using the GCTk, and added an uncollected region to it as well.

4.3 Experimental Setting and GC Configuration

We performed our experimental timing runs on a Macintosh Power Mac G4, with two 533 MHz processors, 32KB on-chip L1 data and instruction caches, 256KB unified L2 cache, 1MB L3 off-chip cache, and 384MB of memory, running PPC Linux 2.4.3. (We used only one processor for our experiments.)

As indicated in Section 3.1, a time-space trade-off is at the heart of each pretenuring decision. In order to better understand how that trade-off is played out and to make fair comparisons, we conduct all of our experiments with 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 a Appel-style generational collector without pretenuring. This amount is at least twice the max live size, we determine it experimentally.

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.

5. Results

This section presents execution time and other results using generational collection for build-time pretenuring, application-specific pretenuring with our advice and CHL advice, and the combination of build-time and application-specific pretenuring. Finally, we

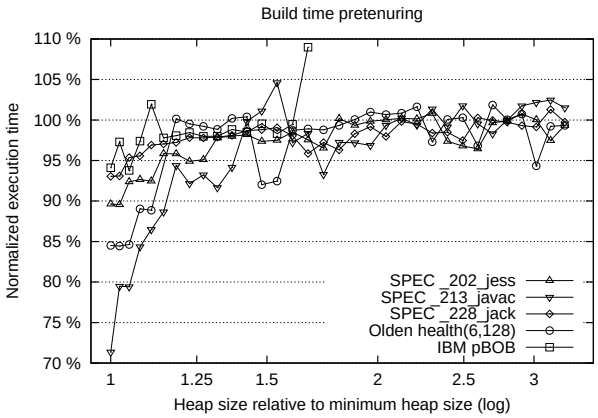


Figure 2: Relative Execution Time for Build-Time Pretenuring

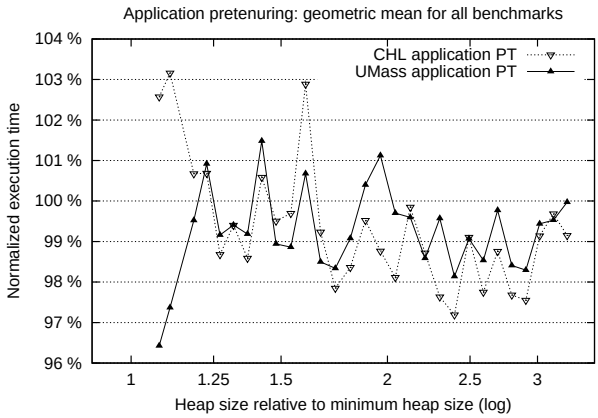


Figure 3: Relative Execution Time for Application-Specific Pretenuring

demonstrate that our advice is collector-neutral by showing that it improves a very different collector as well, the Older First collector. In all of the experiments, we use the pretenuring advice parameters $T_a = 0.2$, $H_f = 0.33$, and $H_{cf} = 0.9$ as described in Section 3.1.

We use the times reported by the SPEC JVM98 benchmarks themselves; health similarly reports its execution time. The pBOB benchmark runs for a fixed period and reports transactions per second, which we invert, giving time per transaction as our measure of time. We always report times normalized with respect to the non-pretenured case.

5.1 Build-Time Pretenuring

Build-time advice is true advice; in these experiments we acquired it by combining advice (Section 3.1) from each of the *other* benchmarks. Because pretenuring will only occur at sites pre-compiled into the Jalapeño 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 Jalapeño optimizing compiler), build-time advice has the distinct advantage of delivering pretenuring benefits without requiring the user to profile the application.

Figure 2 shows for each benchmark the total performance improvement using build-time pretenuring normalized with respect to the generational collector without pretenuring. It plots on the x-axis the heap size in multiples of the minimum heap size for 32 points between 1 and 3.25 on a log scale versus relative execution time. All our results use the same x-axis.

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.

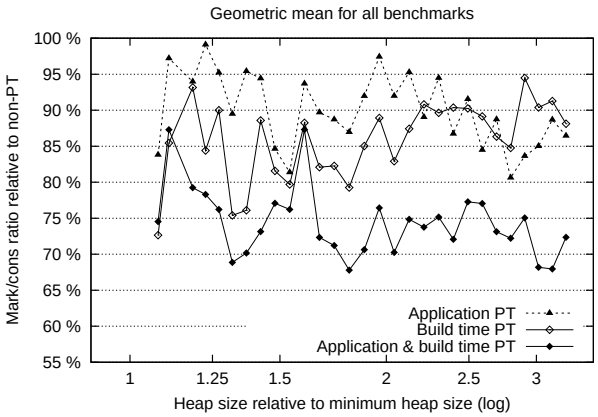
In some cases, build-time pretenuring degrades total performance by a few percent, but for most configurations, programs improve, sometimes significantly. Improvements tend to decline as the heap size gets larger because the contribution of garbage collection time to total time declines as the heap gets 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 2.5% even for large heaps. All programs improve on average, and for javac and health, with a number a configurations improvements are more than 15%. These improvements are a result of reducing copying in the garbage collector, and the signficant decrease in GC time improves overall execution time. Section 5.3 presents these measurements as well.

5.2 Application-Specific Pretenuring

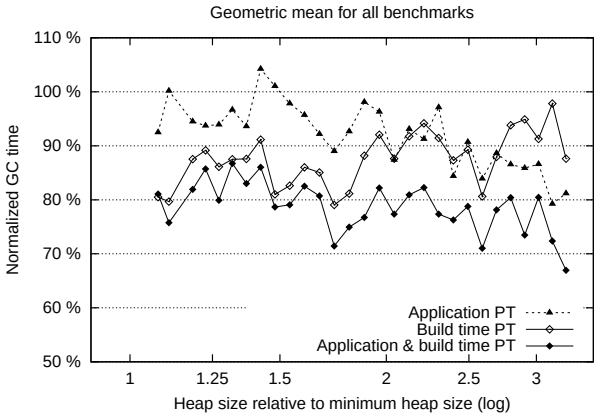
In this section, we compare our classification scheme to the CHL scheme [8] using application-specific (self) advice. Given an application running with a generational collector with a fixed nursery size, CHL advice generation first measures the proportion of object instances that survive at least one 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 3.2 for our Java programs), their approach is fairly robust to the threshold.

The biggest difference between the two classification schemes is that we include an immortal category and our collector puts immortal objects into a region that it never collects. With respect to space rental cost, pretenuring allocates on average 29% of objects into the immortal space (see Table 3), 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 reduces the portion of the heap the generational collector manages in our scheme (see figure 5).

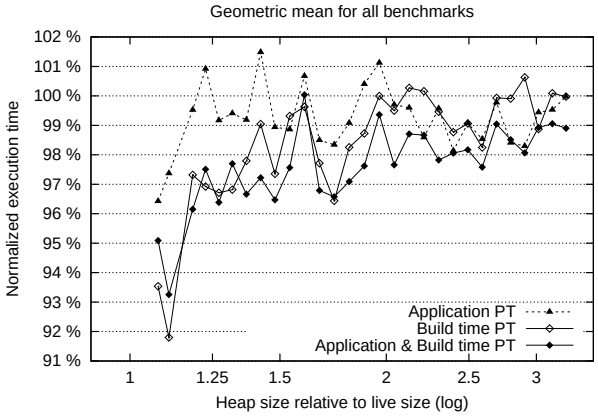
Figure 3 compares CHL and UMass application-specific pretenuring, using the generational collector, which has a flexible nursery



(a) Relative Mark/Cons Ratios



(b) Relative Garbage Collection Time



(c) Relative Execution Time

Figure 4: Comparing UMass Application-Specific, Build-Time, and Combined Pretenuring

size. The figure shows the average relative execution time using a geometric mean of our benchmark programs. On average our advice performs about as well as CHL, except in a tight heap where the impact of immortal objects is highest and our advice performs significantly better. With UMass pretenuring, immortal objects are never copied, and because they do not require reserved copy space they are twice as space efficient as regular heap objects. With CHL these objects typically go into the second generation. Given a tight heap and a flexible nursery size, the amount of space available to the nursery is thus reduced, which triggers more full heap collections for CHL. This cost is greater than the savings of avoiding one copy out of the nursery. Our scheme is sometimes subject to the same behavior, but clearly less frequently. With a larger heap, both schemes avoid these additional full heap collections.

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 similar performance to the CHL two-way scheme on average and much better performance than CHL for tight heaps.

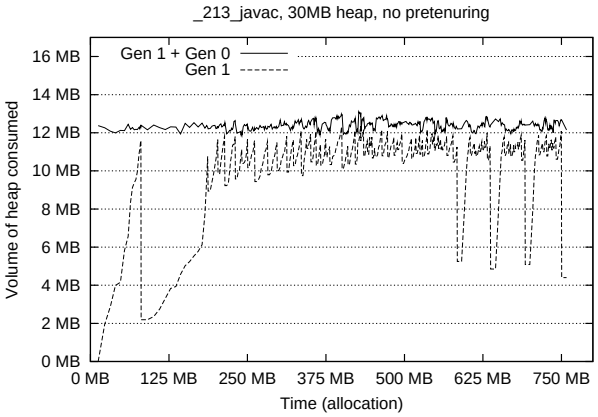
5.3 Combining Build-Time and Application-Specific Pretenuring

This section shows 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 5 benchmarks for relative mark/cons ratio in Figure 4(a), the geometric mean of the relative garbage collection time in Figure 4(b), the geometric mean of the relative execution time in Figure 4(c), and the relative execution time for each benchmark in Figure 6.

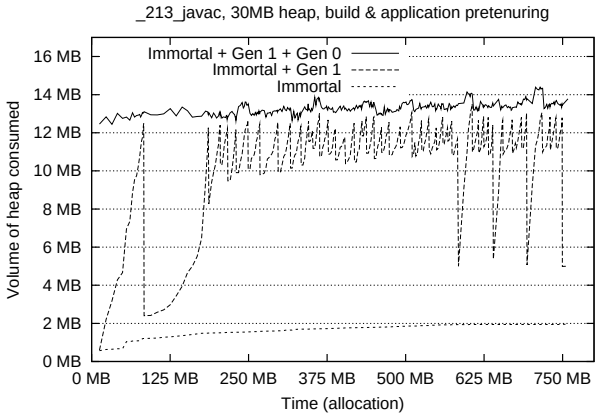
Figure 4(a) shows the mark/cons ratio for each pretenuring scheme, relative to non-pretenuring. The mark/cons ratio is the ratio of bytes copied ("marked") to bytes allocated ("cons"). The figure explains *why* pretenuring works: it reduces copying. In all cases, pretenuring reduces the number of objects the collector copies. Reductions range from 1% to 33%, which is quite significant when minimum heap sizes can be as large 150MB (pBOB).

Figure 5 offers additional insights. Figure 5(a) shows heap usage over time for a run of the javac benchmark without pretenuring, and Figure 5(b) shows it with pretenuring. 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 (always zero in Figure 5(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. This makes the nursery effectively larger. A larger nursery delays the growth of the older generation and defers older generation collections. The lowest points in the second line are very similar in both graphs, which shows that pretenuring does not allocate many immortal objects inappropriately (if it does, 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



(a) Heap Profile Without Pretenuring



(b) Heap Profile with Build-Time and Application-Specific Pretenuring

Figure 5: Heap Usage Over Time By Region for a Run of javac

of the figures. When not pretenuring, the bottoms of the troughs are flat, showing that there is no direct allocation to the older generation. 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 4(b) 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 GC time between 20% and 30% for most heap sizes. Application-specific pretenuring is on average usually the least effective of the three, but it occasionally improves over build-time pretenuring, mirroring the mark/cons results. Combined pretenuring is virtually always the best of the three schemes.

Collector time is of course a fraction of total execution time and that fraction ranges from around 10% on large heaps to 45% on very small heaps for our programs. Figure 4(c) shows that all the pretenuring schemes improve performance. Average improvements are usually between 1% and 4%, but as shown in Figure 6, individual programs improve by as much as 29%.

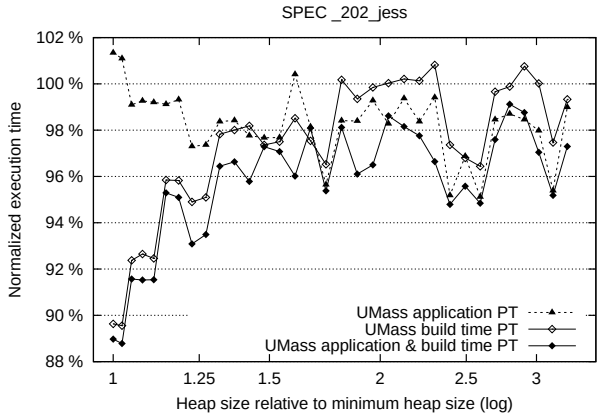
Figure 6 compares the three schemes with respect to each of our benchmark programs. Notice again that pretenuring improves performance more in tighter heaps. For application-specific pretenuring and a tight heap, jess, javac, and jack exhibit degradations due to the phenomenon described in Section 5.2. We get our best improvements on javac and health. Tables 1 and 3 show that these programs have very different lifetime characteristics and receive very different pretenuring advice. For javac, pretenuring allocates objects with 11% of the space rental cost into the immortal space and 54% into the long-lived space. For health, 26% go into immortal and 5% into the long-lived space. These differences further emphasize the value of a three-way classification.

5.4 Improving Older First Collection

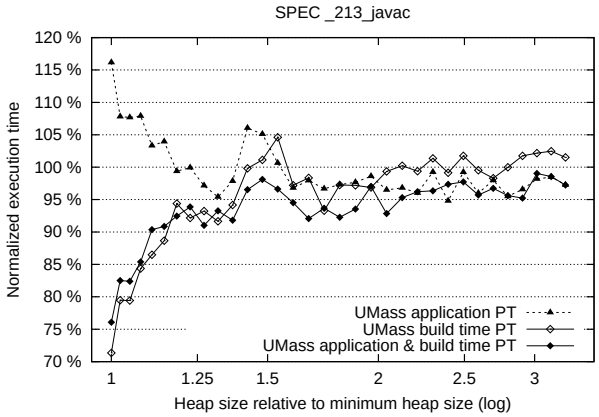
Using an Older First (OF) collector [15], we show the same advice can improve this collector as well. The OF collector organizes the heap in allocation order. View the heap as a queue; the oldest objects are at the tail and the OF allocator inserts newly allocated objects at the head of the queue. OF 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 [15].

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, g, to $0.3\times$ heap size.

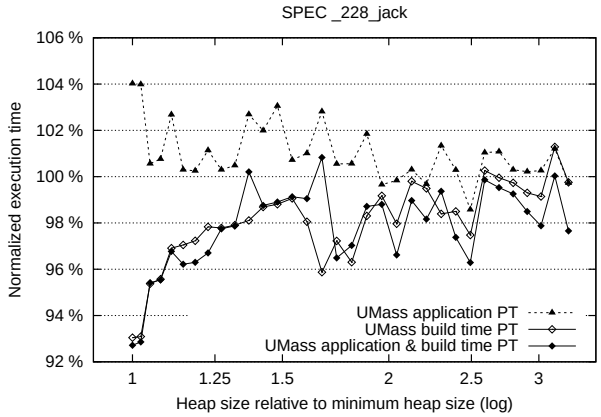
Figure 7 shows the geometric mean of the relative performance for all our benchmarks, normalized with respect to the OF collector without pretenuring, for build-time, application-specific, and combined pretenuring. Application-specific OF pretenuring is almost always a win, except for a number of heap sizes around 1.5 where javac sees a degredation of 20% for one heap size and some benchmarks see ocassional degredations as high as 10%. These degredations lead to degradations in the geometric mean at around 1.5 and 1.7. In these cases, the degradations are caused by a significant increase in the number of collections, most likely due to overzealous pretenuring. Again, build-time pretenuring improves performance, and additional improvements from combined pretenuring are consistent and significant, ranging from 2% to 23%.



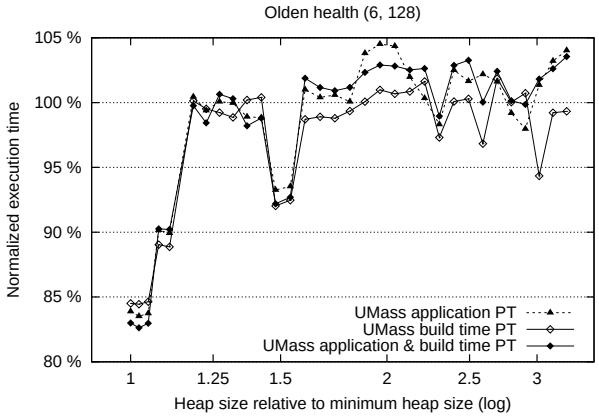
(a) jess



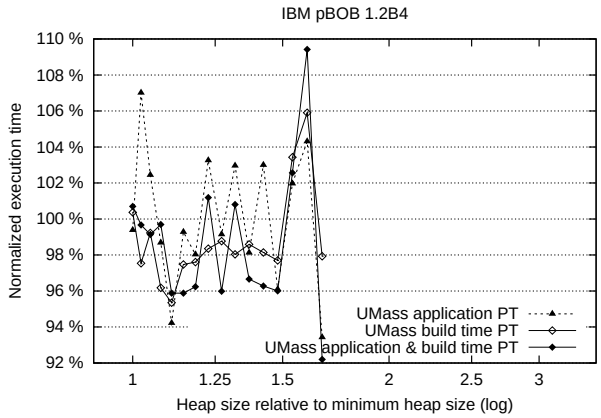
(b) javac



(c) jack



(d) health



(e) pBOB

Figure 6: Comparing UMass Application-Specific, Build-Time, and Combined Pretenuring Execution Time Relative to Non-Pretenuring

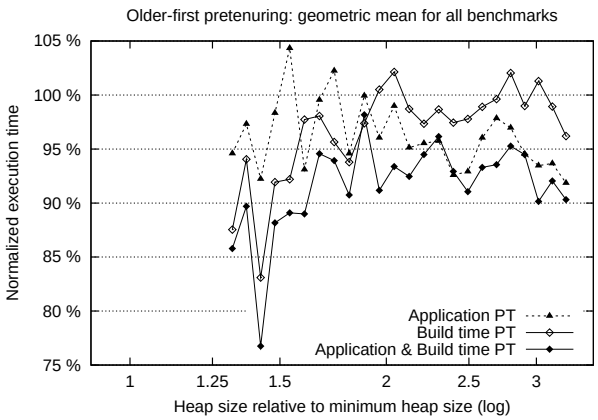


Figure 7: Relative Execution Time for Pretenuring with OF Collection

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 [15]. 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 this vastly different collector and it improves performance as well.

6. 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 [17]. Performance studies with a variety of languages demonstrate well tuned generational collector performance ranges from 10% to 40% of the total execution time [18, 20, 19, 5, 16, 8].

Ungar and Jackson use profiling to identify what we call long-lived objects in a two generation collector for Smalltalk [18, 19]. The older generation in their system is the tenured, permanent space and is never collected. They do not allocate directly into this region, but copy into it objects that survive a given number of nursery collections. Their system keeps long-lived objects in the nursery, repeatedly collecting them to keep from tenuring them, which would result in tenured garbage. They 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 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 [8]. 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 an altogether different collector, the Older First collector.

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 pretenured 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 [8, 15], and as they point out, it reduces the effectiveness of pretenuring in their system. We never collect or scan 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 makes dynamic pretenuring decisions for Java programs in the context of a two generation collector and shows improvements by detecting long-lived objects [10]. This technique uses sampling based on overflow, and thus tends to sample more large objects, and can react to phase changes. Our scheme for build-time pretenuring is most similar to this work, and it achieves better performance improvements due to lower overhead. Profiling enables us to achieve even better performance. Our advice is neutral with respect to the collector, and we show that we can predict object lifetime based on the allocation site whereas Harris' work does not investigate how much context is needed to perform prediction.

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 [9]. Barrett and Zorn extend Hanson's algorithm by using profile data to predict short-lived objects automatically [4]. To achieve accurate results, their predictor uses the dynamic call chain and object size, whereas we show Java prediction does well with only the allocation site. Subsequent work by Siedl and Zorn predicts short-lived objects with only the call chain [14]. 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 allo-

cation site, we statically indicate 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.

A technique somewhat complementary to pretenuring is *large object space* (LOS) [7, 19, 11]. One allocates large objects (one exceeding a chosen size threshold) directly into a non-copying space, effectively applying mark-sweep techniques to them. This avoids copying these objects, and can noticeably improve performance. Our GCTk does not (yet) support LOS, so we cannot 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 obejcts 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 a criterion. The compress benchmark is an example of this: it allocates and discards large arrays.

7. 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 and reduces copying, garbage collection time, and total time, sometimes significantly. We are the first to demonstrate the effectiveness of build-time pretenuring, and we do so using *true* advice. Because Jalapeño 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. We further show that the combination of build-time and application-specific pretenuring offers the best improvements.

Acknowledgements

We thank Sara Smolensky who did the first studies that inspired this work. We also thank John Cavazos, Asjad Khan, and Narendran Sachindran for their contributions to various incarnations of this work, and our anonymous reviewers for their helpful comments. Finally, we thank the members of the Jalapeño team at IBM T.J. Watson Research Center who helped facilitate this research.

8. REFERENCES

[1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in java. In *ACM Conference Proceedings on Object–Oriented Programming Systems, Languages, and Applications*, Denver, CO, Nov. 1999.

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

[4] D. A. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, June 23-25, 1993*, pages 187–196, June 1993.

[5] D. A. Barrett and B. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, June 18-21, 1995*, pages 301–314, June 1995.

[6] S. J. Baylor, M. Devarakonda, S. J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM System Journal*, 39(1), Feb. 2000.

[7] P. J. Caudill and A. Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *OOPSLA'86 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 119–130, 1986.

[8] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, 17-19 June 1998*, pages 162–173, May 1998.

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

[10] T. L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the International Symposium On Memory Management (ISMM), Minneapolis, MN U.S.A, 15-16 October, 2000*, pages 127–136. ACM, 2000.

[11] M. Hicks, L. Hornof, J. T. Moore, and S. Nettles. A study of Large Object Spaces. In *ISMM'98 Proceedings of the First International Symposium on Memory Management*, pages 138–145. ACM, 1998.

[12] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting java bytecodes. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[13] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, Mar. 1995.

[14] M. L. Seidl and B. G. Zorn. 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, San Jose, California, October 3-7, 1998*, pages 12–23, Nov. 1998.

[15] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, November 1-5, 1999*, pages 379–381, Nov. 1999.

[16] D. Tarditi and A. Diwan. Measuring the cost of storage management. *Lisp and Symbolic Computation*, 9(4), Dec. 1996.

[17] D. Ungar. 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, Pittsburgh, Pennsylvania, April 23-25, 1984.*, pages 157–167, May 1984.

[18] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In N. K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), September 25-30, 1988, San Diego, California, Proceedings*, pages 1–17, Nov. 1988.

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

[20] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, Computer Science Dept., University of California, Berkeley, Dec. 1989. Available as Technical Report UCB/CSD 89/544.