Decrypting The Java Gene Pool

Predicting objects' lifetimes with micro-patterns

Sebastien Marion
University of Kent, Canterbury, UK
sm244@kent.ac.uk

Richard Jones
University of Kent, Canterbury, UK
R.E.Jones@kent.ac.uk

Chris Ryder
University of Kent, Canterbury, UK
C.Ryder@kent.ac.uk

Abstract

Pretenuring long-lived and immortal objects into infrequently or never collected regions reduces garbage collection costs significantly. However, extant approaches either require computationally expensive, application-specific, off-line profiling, or consider only allocation sites common to all programs, i.e. invoked by the virtual machine rather than application programs. In contrast, we show how a simple program analysis, combined with an object lifetime knowledge bank, can be exploited to match both runtime system and application program structure with object lifetimes. The complexity of the analysis is linear in the size of the program, so need not be run ahead of time. We obtain performance gains between 6–77% in GC time against a generational copying collector for several SPEC jvm98 programs.

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

General Terms Design, Performance, Algorithms

Keywords Pretenuring, Micro-Patterns, Java

1. Introduction

The goal of this research is to explore reusable yet application-specific mechanisms to reduce the cost of managing long-lived objects. By application-specific, we mean that we wish to be able to identify and take advantage of the application's particular pattern of object allocation. By reusable, we mean that the technique should be easily applied to any program, without for example having recourse to instrumenting and profiling runs of that program.

Tracing garbage collectors spend much of their time following the graph of live objects, either copying objects to a separate region or setting mark-bits. In other words, most effort is spent on rescuing live objects. However, the lifetimes of objects vary. Some may be immortal (live from the moment of their birth to the end of the program), some may live for a long time, but most typically die young [25].

The most widely adopted strategy, generational GC, exploits this observation by segregating objects by age into different regions, *generations*, of the heap. Younger generations are collected more frequently than older ones, and objects that survive sufficiently long are promoted (copied) or *tenured* to an older generation[1].

Generational collection has two corollaries. By concentrating on that region of the heap where most objects are likely to be dead, the youngest generation (or nursery), the most space is reclaimed for the least effort of promoting objects. Further, objects in older generations are given more time to die. This strategy has two shortcomings. First, longer-lived objects must still be copied at least once, on their promotion from the nursery in which they were first allocated. Second, immortal objects may be repeatedly processed when older generations are collected. The solution to the first problem is to *pretenure* long-lived objects by allocating them directly in an older generation [11]. The solution to the second problem is to allocate immortal objects into a region that is never collected [6], although references from immortal objects to objects in other spaces must be discovered.

Blackburn et al. [9, 7] use object birth and death statistics, gathered from instrumented programs in an off-line phase, to advise the allocator as to whether particular *allocation sites* (points in the program which allocate new objects) are expected to create objects with *short*, *long* or *immortal* lifetimes. Acquiring advice for *self-prediction* [4] of particular programs is extremely expensive. Blackburn et al. also combine advice common to all programs and provide this at Jikes RVM's build time. Such *build-time* advice provides *true prediction*, applicable to any program, and can reduce costs of generational copying by 40–70% on average in Jikes RVM.

ISMM '07 October 21–22, 2007, Montréal, Québec, Canada.

[1] For the sake of simplicity, we shall assume that all generations are managed by copying; however, our discussion is equally applicable to configurations in which generations are managed by other strategies.

However the scope of the advice is restricted to sites of JVM classes, and is, by definition, not program-specific.

We might seek allocation advice from the programmer, who has some understanding of the lifetimes of their objects. Unfortunately, their intuition may be unreliable, either because programmers make mistakes or because the context in which their code executes has changed. However, we show here that the way in which programmers write code can be mined to offer object lifetime hints to the allocator. Although programs are written in different styles, programmers tend to adopt similar practices and idioms. Indeed, they may be constrained to do so by standard libraries and frameworks, or encouraged to do so by the influence of what is seen to be good practice (educators, design patterns, books, etc. [12, 10]).

We use a simple program analysis to identify coding patterns. We match such micro-patterns [13] against a knowledge bank of historical object lifetime behaviour: rules about patterns are associated with expected lifetimes (short, long or immortal) and confidence levels. The knowledge bank is created by profiling a wide range of programs, in our case the DaCapo benchmark suite [5]. While this is extremely time-consuming to do, like Blackburn's build-time advice it only has to be done once. On the other hand, our program analysis and pattern lookup is quick: the analysis' complexity is linear in the size of the program. Furthermore, the knowledge bank can be extended at any time by processing further programs and improvements in the rule sets applied to any program, new or existing. Our contributions are as follows:

- We show that object lifetimes are often related to code patterns.
- We provide a knowledge bank that associates patterns with object lifetime.
- We provide a pretenuring allocator that can exploit such program-specific lifetime advice.
- We demonstrate that GC times can be reduced by up to 77%.
- We provide *MemTrace*, a comparatively cheap mechanism for gathering object lifetime data.

Note that our interest is in *program-specific* pretenuring. Thus, throughout we consider only such advice and not build-time advice, which we have not yet incorporated into our system. However, we could add build-time advice to the boot image, and intend to do so. We expect this to further enhance performance, especially in tight heaps or small benchmarks where Jikes RVM dominates.

Organisation of this paper We discuss related work in Section 2. We explain pretenuring in Section 3, micro-patterns in Section 4 and data mining in Section 5. We discuss our methodology in Section 6 and implementation details in Section 7. Our results are presented in Section 8 and evaluated in Section 9 where we also make suggestions for future work. We conclude in Section 10.

2. Related work

The advantages of treating different categories of object with different memory management policies have long been recognised. For example, large objects are commonly allocated into a separate region, managed by a non-moving collector. The most common form of regional organisation is generational GC [25]. Generational collectors typically have two generations, but may have more; more sophisticated regional organisations are also possible [8, 24]. It is important to decide when an object should be promoted from a younger to an older generation. Too early risks objects not only dying soon after promotion (thereby causing too frequent collections of the older generation) but also encourages nepotistic promotion of the referents of tenured (i.e. promoted) garbage; too late demands either larger young generations (and consequently longer pause times) or more frequent young generation collections. Some collectors can dynamically adjust promotion thresholds [26].

Cheng et al. [11] (CHL) observed that savings might be made if long-lived objects are allocated directly into an older generation (pretenured). Identification of suitable candidates may be through profiling programs [11, 9, 7], dynamic sampling at run-time [16, 21], or program analysis [15].

We consider off-line, profiling approaches first. CHL tag objects with the *allocation site* in the program that allocated them and inspect the tags of dead objects at each collection. From this profile, they identify those sites that allocate promoted objects consistently (i.e. where the volume exceeds a threshold) in their collector. This advice is then used to allocate objects from those sites directly into the old generation, thereby reducing GC times by 12–50%. The generality of this approach is limited because the pretenuring threshold is a function of a particular collector configuration.

Blackburn et al. [9, 7] extend this approach. They remove implementation dependency by normalising object lifetime as a multiple of the maximum volume of objects live at any time, *live size*. They also add an immortal space for objects that are never to be collected. Sites are classified as generating predominantly short-lived, long-lived or 'immortal' objects, using thresholds expressed as a fraction of maximum live size; objects are defined as immortal if they die more than halfway between their birth and the end of the program (and hence not worth keeping *copy reserve* space for). Because their advice is implementation-independent, they can also combine advice from runs of different programs by ignoring application-specific data. Just using such *build-time* advice improved performance in tight heaps particularly. Our scheme, on the other hand, leverages program-specific knowledge. Object lifetimes are recorded at a 64KB granularity in [9], as we do, but precisely in [7] using Merlin [17]. For combining advice in their algorithm, Blackburn et al. cite the use of precise ages, plus ranking the importance of allocation sites by the volume they allocate rather than the

product of the objects' volumes and ages, as the reason for the better consistency of results [7].

Huang et al. [19] use type as a predictor, based on per-class rather than per allocation site lifetimes. However, they find type-based predictions be poor predictors. As they report experiments using the Jikes RVM BaseBase compiler, rather than the optimising compiler as we do, their results may exaggerate the benefits of object placement.

Harris [16] pretenures objects in a Java virtual machine using dynamic feedback from statistics gathered online. Dynamic pretenuring allows adaption to phase behaviour, by reversing and re-enabling pretenuring decisions. A disadvantage of his mechanism is that it is specific to Java as it uses weak pointers. Whereas Harris samples objects on local allocation buffer overflow, Jump et al. [21] sample at every 2^n bytes of allocation. Both mechanisms skew sampling, either by introducing a regular stride (Jump) or by over-sampling large objects (Harris). However, Jump et al.'s mechanism has lower time and space requirements than Harris's. Both provide only small benefits for jvm98 [23] benchmarks, and degrade some significantly (e.g. raytrace). Furthermore, there is evidence that, in Jikes RVM at least, a large fraction of long-lived objects are allocated at the start of the program [7], for which sampling-based pretenuring would be too late.

On the basis that connected objects share similar lifetimes [18], Guyer and McKinley [15] seek to colocate them in the same space. They combine a compiler analysis, that identifies the object to which a new object might be connected, with a specialised allocator, that places the new object in the same space as the connectee. The analysis is neither required to be sound nor must allocation sites be homogeneous. As well as reducing copying, colocation also reduces pressure on the write barrier. Experiments with jvm98 show that GC time can be reduced by up to 75%. In work most similar to ours, Singer et al. [22] suggest using metrics designed to measure the object-oriented nature of source code as a guide to longevity; they do not measure performance.

3. Pretenuring

By segregating objects by age and concentrating effort on the youngest, generational garbage collection [25] has proved remarkably successful. Yet its management of longer-lived objects is not optimal: all long-lived objects will be copied at least once into an older generation, and immortal objects may be copied many times as older generations are collected.

However, if we know ahead of time how long an object will live, we can allocate objects more efficiently. Those with a short lifetime would still be allocated in the nursery, but objects with a longer lifetime could be allocated directly into an older generation. Further, immortal objects can be allocated into a space which will never be collected. Thus, unnecessary copying between generations is avoided and the cost of a young-generation collection is potentially reduced to that of tracing roots (if all the objects allocated in the nursery are dead at the time of the collection). Moreover, young generation pause times should be reduced.

However, it is essential to predict object lifetimes conservatively, because incorrect pretenuring decisions harm performance. First, wrongly pretenuring short-lived objects leads to increased pressure on the older generation, and hence more frequent old generation collections. Second, in the case of an Appel-style collector [2], allocation in the older generation reduces the space available for the nursery; conversely, allocation into the immortal space eases memory pressure as there is no need to reserve space (the copy reserve) for copying it. Third, incorrectly pretenuring an object leads to nepotism [26] as no referent of a dead yet tenured object will be collected: it is particularly important to avoid incorrect allocation into the immortal region.

Blackburn et al. [9, 7] predict lifetimes by analysing program trace files in order to associate each allocation site with an expected lifetime (short, long or immortal). Their trace files were gathered with Merlin [17] and provide, inter alia, an object's times of birth and death, and allocation site. However, gathering traces with Merlin is slow: for example, instrumented javac may run for a week, whereas our *MemTrace* tool gathers the trace in 3.5 hours. MemTrace modifies the Jikes RVM compiler to cause allocation site IDs to be written into the header of each object allocated; unlike Merlin, we do not scan the stack on each allocation. On the other hand, although MemTrace allocation times are accurate, because we determine deaths by periodically collecting the full heap, object death times are only accurate to *N*=64 KB bytes whereas Merlin death times are accurate. Although this granularity exaggerates the lifetime of short-lived objects, this is not problematic, as no practical tracing collector could take advantage of greater precision.

Blackburn et al. classify lifetime behaviour as follows [7]. The first step is to classify objects. Note that they use *max live size*, the maximum volume of data live at any point in the program's execution, as a normalising factor.

1. If an object dies later than halfway between its time of birth and the end of the program, it is classified *immortal*.
2. Otherwise, if an object's age is greater than $T_a \times$*max live size*, then it is classified as *long-lived*. They use $T_a = 0.45$.
3. Otherwise, the object is classified as *short-lived*.

Immortality is defined in this way because objects that might be copied need space reserved for the copy, whereas immortal objects do not; the copy reserve accounts for half the heap in an Appel-style generational collector. They therefore adopt the heuristic that an object is immortal if it is dead for less time than it is alive.

The second step is to compute lifetimes for each allocation site, based on the fraction S_s of short-lived, L_s of long-lived and I_s of immortal objects it allocates. Homogeneity factors H_{lf} and H_{if} determine the conservatism of decision to pretenure in the mature and immortal spaces respectively:

higher values of H lead to fewer sites pretenured. They adopt $H_{lf} = 0.6$ and $H_{if} = 0.0$.

1. If $I_s > S_s + L_s + H_{if}$, the site is classified *immortal*.
2. If $I_s + L_s > S_s + H_{lf}$, the site is classified *long-lived*.
3. Otherwise, the site is classified *short-lived*.

Throughout this paper, we compare how pretenuring advice improves the performance of the MMTk [6] GenCopy collector for Jikes RVM, version 2.4.4 [1]. GenCopy is an Appel-style [2] generational collector with a variable-sized nursery (i.e. it expands to occupy all memory not used by other spaces). GenCopy's mature space is managed by semi-space copying. It also has an immortal space, used for VM objects, and a large object space managed by a treadmill [3]. For pretenuring, we modify the Jikes RVM compilers and the GenCopy allocator to exploit advice (see Section 7).

The improvements in GC time and overall execution time offered by Blackburn's self advice can be seen in Figure 3 and Figure 4 respectively. Measurements were taken on a Dell Optiplex GX270, with a Intel Pentium 4 2.6 GHz processor with 800 MHz FSB and Hyper-Threading, 512 KB 8-way set-associative L2 cache, 1 GB RAM, under Debian Linux, kernel 2.6.12. Performance is shown as the ratio of self-advice pretenuring time to that without (lower is better). Heap sizes are shown as multiples of the smallest heap in which the program would run. GC time is improved, often very substantially, for all programs except jack, where it performs significantly worse in large heap sizes, and compress, for which is neutral. On the other hand, execution time is not improved for db, jack or compress.

4. Micro-patterns

Gil & Maman [13] provide a framework for the statistically valid comparison of Java coding styles. A *micro-pattern* is "a non-trivial, formal condition on the attributes, types, name and body of a class and its components, which is mechanically recognisable, purposeful, prevalent and simple". Micro-patterns are similar to design patterns [12] but closer to implementation: a set of micro-patterns may implement a design pattern. Rather than describing an interaction between classes, a micro-pattern is a property of a single class (although a class may exhibit more than one pattern). Not all implementations of the same specification will use the same combination of patterns (though they are statistically highly likely to do so), but they are likely to use some combination.

Gil & Maman's catalogue captures a wide spectrum of common coding practices, including particular uses of immutability, wrapping, restricted creation and emulation of different programming paradigms with object-oriented constructs. They identify 29 different patterns (Table 1). Statistical analysis of a very large corpus, drawn from a wide variety of application domains, indicates that use of these patterns is not random. Consider an example.

Sampler is a 'controlled creation' micro-pattern. It defines classes which have a `public` constructor and one or more `public static` fields of the same type as the class. Such classes provide clients with pre-made instances of the class as well as being able to make their own. `java.awt.Color`, which provides pre-defined colours, is a Sampler. Sampler objects turn out to be very likely to be immortal.

$$
\begin{array}{rcll}
\mathcal{MP} & = & \{p_0, p_1, \ldots, p_n\} & \text{(Set of all patterns)} \\
\mathcal{LT} & = & \{\textit{short}, \textit{long}, \textit{immortal}\} & \text{(lifetime)} \\
\mathcal{SL} & = & \textit{SiteID} \rightarrow \mathcal{LT} & \\
\mathcal{S} & = & \textit{SiteID} \rightarrow \mathbb{P}(\mathcal{MP}) \times \mathbb{P}(\mathcal{MP}) & \\
\mathcal{PL} & = & \mathbb{P}(\mathcal{MP}) \times \mathbb{P}(\mathcal{MP}) \rightarrow \mathcal{LT} &
\end{array}
$$

Table 2. Site, micro-pattern and lifetime mappings.

We associate micro-patterns with allocation sites. For the code snippet below, we call class `Src` in which the allocation occurs the *source* and class `Dst` of the object allocated the *destination*. Both source and destination classes define sets of micro-patterns. We associate this allocation site (represented by a unique site ID) with these sets (implemented as a bit-vector), thus obtaining the mapping $\mathcal{S}$ in Table 2.

```
public class Src {
    IDst bar = new Dst();
}
```

We now require a mapping $\mathcal{PL}$ from sets of patterns to expected site lifetimes. We adopt the same approach as Gil and Maman, namely, to analyse a large corpus of classes. Because we are interested in object demographics, we chose to use DaCapo[5], a suite of real-world Java benchmarks with non-trivial memory loads (Table 3). We profiled each benchmark with MemTrace, and classified each DaCapo, Jikes RVM and library site using Blackburn's heuristics, giving mapping $\mathcal{SL}$. We used Gil and Maman's analysis to associate each of these sites with a set of patterns, giving $\mathcal{S}$. $\mathcal{PL} = \mathcal{SL} \circ \mathcal{S}^{-1}$ provides the mapping from sets of patterns to lifetimes. We use $\mathcal{PL}$ as a historical guide to pretenuring.

However, $\mathcal{PL}$ is a very large relation, and worse, it is not clear which patterns influence object lifetime and which do not. Rather than a large relation, we require rules that identify probable lifetimes from the patterns exhibited by an allocation site. Each rule should have a confidence level. We obtain these rules by data mining the relation.

5. Data mining

Data mining extracts information from large data sets which would be very hard, if not impossible, for a human to find, and can often discover unexpected relationships between its attributes. Choosing the best suited algorithm depends on the nature of the data to be analysed. We use *Clementine C5.0* [14], a powerful and widely used algorithm designed to analyse databases of thousands to hundreds of thousands of records along with tens to hundreds of attributes. C5.0

	Micro-pattern	*Definition*	All *Src-Dst*	SPEC *Src-Dst*
Degenerate classes	*Designator*	*Interface with no members.*	*0-0*	*0-0*
	Taxonomy	Empty interface extending another interface.	4-0	1-0
	Joiner	*Empty interface joining two or more superinterfaces.*	*0-1*	*0-0*
	Pool	Class which declares only static final fields, but no methods.	8-0	3-0
	Function Pointer	Class with a single public instance method, but with no fields.	1-3	1-1
	Function Object	Class with a single public instance method, and at least one instance field.	9-5	4-3
	Cobol Like	*Class with a single static method, but no instance members.*	*1-0*	*0-0*
	Stateless	Class with no fields, other than static final ones.	15-6	5-4
	Common State	Class in which all fields are static.	18-0	7-0
	Immutable	Class with several instance fields, which are assigned exactly once, during instance construction.	5-6	3-3
	Restricted Creation	Class with no public constructors, and at least one static field of the same type as the class.	6-4	5-2
	Sampler	Class with one or more public constructors, and at least one static field of the same type as the class.	8-9	7-3
Containment	Box	Class which has exactly one, mutable, instance field.	6-14	2-10
	Compound Box	Class with exactly one non primitive instance field.	32-22	17-12
	Canopy	Class with exactly one instance field that it assigned exactly once, during instance construction.	4-13	2-6
	Record	*Class in which all fields are public, no declared methods.*	*0-0*	*0-0*
	Data Manager	Class where all methods are either setters or getters.	1-10	0-5
	Sink	Class whose methods do not propagate calls to any other class.	11-9	4-5
Inheritance	Outline	Class where at least two methods invoke an abstract method on `this`.	7-4	5-2
	Trait	Abstract class which has no state.	4-0	2-0
	State Machine	*Interface whose methods accept no parameters.*	*0-0*	*0-0*
	Pure Type	*Class with only abstract methods, and no static members, and no fields.*	*0-0*	*0-0*
	Augmented Type	*Only abstract methods and three or more static final fields of the same type.*	*0-0*	*0-0*
	Pseudo Class	Class which can be rewritten as an interface: no concrete methods, only static fields.	5-2	3-1
	Implementor	Concrete class, where all the methods override inherited abstract methods.	19-11	9-3
	Overrider	Class in which all methods override inherited, non-abstract methods.	27-20	11-14
	Extender	Class which extends the inherited protocol, without overriding any methods.	0-8	3-5
	Limited Self	Subclass that does not introduce new fields and all self method calls are to its superclass.	13-9	6-5
	Recursive	Class that has at least one field whose type is the same as that of the class.	7-11	3-7

Table 1. Gil & Maman's micro-patterns [13]. The table includes counts of patterns used in all rules with confidence greater than 75%, generated (*All*)and discovered in the SPEC benchmarks (*SPEC*), as either sources or destinations. Micro-patterns never exploited (including those only discovered in classes compiled at build-time) are shown in italic.

classifiers can be expressed as decision trees or as sets of rules, the latter being easier to interpret.

C5.0 captures the field giving the best information gain and, from this, splits the data into different bins. Each bin is further subdivided using a different field, and the process is repeated until no more splitting can be performed. When the binning is complete, the final bins are analysed again, and those with insufficient influence on the results are removed. A decision tree or a ruleset is then constructed from these different bins. C5.0 also offers an *adaptive boosting* facility able to further refine previously generated rule sets by focusing on the incorrect predictions of the previous model. A voting system is used to determine the final prediction.

In our case, we data mine the site–lifetime relation $\mathcal{PL}$ to discover which attributes (combinations of patterns) are good lifetime predictors. Unlike the classical data mining approach, we do not clean our data set by removing duplicate or contradictory data, because (a) we expect $\mathcal{PL}$ to be 1-many and (b) we want to take account of reinforcement of a prediction (i.e. if many instances make the same prediction, this strengthens the merit of the prediction).

```
//Rule 57 for IMMORTAL (351.712, 0.996)
if (Src=10 & Dst!=14 and Dst=10) IMM
```

Outputs are rules in disjunctive normal form. The rule above, true in *99.6%* of the cases, states *"If the source is immutable (pattern 10) and the destination is not a compound box (pattern 14) and is also immutable, then instances allocated by this site are immortal in 99.6% of the cases"*.

C5.0 thus provides a set of rules associating patterns with lifetimes and confidence levels. These constitute a historical *knowledge bank* which sites can query by matching pattern-sets; it can be refined at any time as further programs are analysed. Such queries can be built into the compiler or performed off-line and stored as (*allocation-site,lifetime*) pairs. We describe our implementation in Section 7.

6. Methodology

6.1 Program-specific, true prediction

Our purpose is to explore the extent to which micro-patterns can provide program-specific prediction to guide pretenuring. However, it is important that such program-specific prediction nevertheless be true prediction rather than self-prediction [4], that is, the prediction should not be derived simply from a past execution of the same program.

In Section 5, we explained that we use a data mining algorithm to derive a rule set, matching patterns found at allocation sites to expected lifetimes, from off-line profiling

Program	Input	Max. live	Allocated	Sites
antlr	large	8.21	649.6	2014
bloat	large	12.72	3218	1940
fop	large	17.61	132	2336
hsqldb	default	19.8	1025	1265
jython	large	9.02	1724	1432
pmd	large	16.56	1533	1393
ps	large	15.39	1676	1166

Table 3. Allocation (MB) by the DaCapo benchmark suite, v. 051009, baseline compiled. *Sites* is the number of sites used at run-time by the benchmark, Jikes RVM or libraries.

of the DaCapo suite. Unlike other pretenuring approaches that exploit program-specific prediction, our off-line analysis is performed just once. We use DaCapo to construct rulesets because it is the best representative of 'real-world' programs: it comprises a set of benchmarks with a large number of classes, written in an object-oriented style, and generates intensive memory loads, and provides a large knowledge base for our rule sets. However, the need for true prediction requires that we do not use it for performance experiments. To evaluate our approach, we thus apply the results of our analysis to the jvm98 suite [23]. For each benchmark, we identify the patterns at each allocation site — our approach is program-specific. To obtain lifetime advice for a site, we compare its patterns against the rule set derived from DaCapo — this is true prediction. Thus, our advice is applicable to JVMs other than Jikes RVM.

At run-time, we compile this advice into the allocation sequence for each site (details in Section 7). Note that not all sites with advice are used. First, our advice may be applied to sites in code paths that are never executed. Second, we do not incorporate advice to classes compiled at build-time.

We profiled DaCapo with MemTrace, using the base compiler at both build- and run-time (BaseBase). We profile with BaseBase because our focus is on application objects rather than optimising compiler data (which has a more pronounced effect on smaller programs). We also wished to minimise the effect of compiler-allocated data on exaggerating object lifetimes (which are measured in bytes — the size of any allocation, e.g. by the compiler, between an object's birth and death contributes to its lifetime). Further, BaseBase MemTrace configurations generate comparatively smaller, although still several gigabyte, trace files.

We ran our tests on the jvm98 suite using a FastAdaptive configuration (using the optimising compiler at both build- and run-time). We use compiler replay to avoid allocation and mutator variations due to non-deterministic invocation of the adaptive compiler. For each benchmark, we take the best of 5 performance runs.

6.2 Immortal classification

The output of the Clementine C5.0 data mining algorithm is a set of rules, each an expression in disjunctive normal form, a lifetime and a confidence level. In our experiments, we accept a rule as pretenuring advice only if its confidence level exceeds a fixed threshold. In general, this threshold determines how many sites will pretenure objects: higher values lead to more conservative schemes — less chance of making bad decisions — but less opportunity for gains.

One aspect of prediction is how immortality is defined. We explored both Blackburn's heuristic definition of immortality (dying more than halfway between time of birth and the end of the program) and a true immortality (living until the end of the program). In general, where pretenuring advice provides performance gains, then the benefit is sometimes greater with the heuristic definition than with the true definition. Otherwise, results are similar. Our pretenuring scheme is as follows:

1. We exclude any rule with confidence less than a fixed threshold.
2. For any allocation site exhibiting a set of patterns *mp*, we select the rule r that matches *mp* with the highest confidence level.

Theoretically, conflicting rules with identical confidences are possible. In practice, we find that advice with high confidence never conflicts.

6.3 Results

Column 3 of Table 5 shows the number of *sites* in the jvm98 suite for which we obtain predictions with confidence greater than 75%, using the heuristic definition of immortality. At this confidence level, most predictions are of immortal sites and there are no predictions for long-lived sites (although these do appear at lower confidence levels). However, of the 97 rules (confidence $\geq 75\%$) in the knowledge bank, less than half are found in jvm98. Figure 1 shows the number of times each rule is used in each benchmark.[2] Most are applied to very few sites, and a small number apply to very many sites; most rules are used the same number of times in each benchmark, and different benchmarks are distinguished by use of only a very small number of rules.

Consider an example. GC times for raytrace/mtrt are improved by advice (Figure 3). Rule 25 in Figure 1 is more prevalent (72 instances) in these benchmarks than others (1–3): the source is *Immutable* as is the destination, which is also not a *CompoundBox*. These sites provide graphical components of the scenes to be raytraced: the `Scene` (the source) and objects within it (the destinations) are never changed, and common components like `Points` have only primitive fields.

Figure 2 compares the accuracy of micro-pattern prediction with self prediction for sites compiled at run-time with the baseline compiler. The height of each bar is the total number of sites that allocated data. The shaded blocks are micro-pattern predictions. Although micro-patterns make

[2] Excluding those found only in classes compiled at build-time for which we do not currently provide advice.

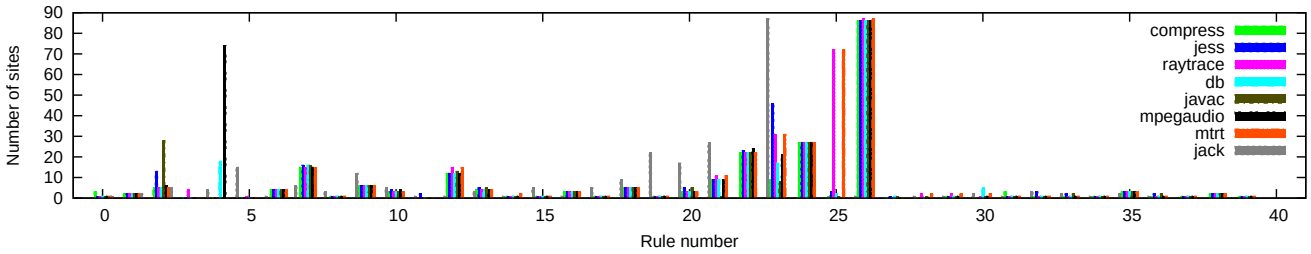


Figure 1. Instances of rules used by SPEC jvm98 sites at confidence 75%.

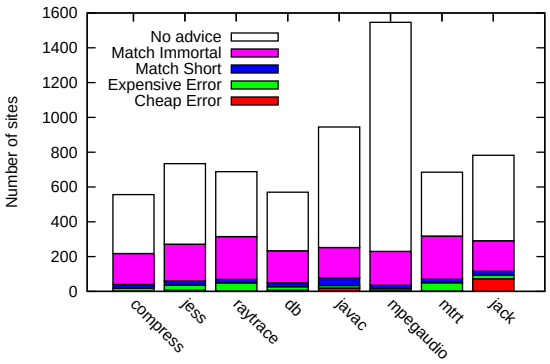


Figure 2. Comparing self prediction with micro-pattern advice at 75%.

fewer predictions than self-advice, on average they match self-prediction advice for 81% of the sites for which they give advice. However, occasionally (9% on average of the sites we predict) our advice makes a potentially 'expensive' error by pretenuring a site when it should not; more rarely, we make 'cheap' errors by failing to pretenure.

7. Implementation

To evaluate our pretenuring analysis, we implemented a pretenuring mechanism in Jikes RVM, an open source Java virtual machine written in Java. Its well-defined memory management toolkit, *MMTk* [6], allows the easy implementation of new garbage collectors by composing reusable components, and by defining a common base for the main garbage collector algorithms available.

Our system passes *allocation advice* to the JIT compiler, allowing it to insert different allocation paths according to the advice. Advice is contained in a file specified to the JVM using a command line option. Loading advice from a file at run-time has the advantage of allowing easy experimentation without having to rebuild the JVM. However, advice generation could be done on-line instead by the class loader.

7.1 Advice file format

An allocation advice file comprises one line for each allocation site, specifying the fully qualified name of the allocating method, the offset from the start of the method to the `new` instruction, and the lifetime advice: `Class:Method:Offset Advice`, for example (wrapped over multiple lines here):

```
Ljava/util/TimeZone;
  :timezones()Ljava/util/Hashtable;:1123 2
```

This entry describes an allocation site contained in the method `timezones()` of `java.util.TimeZone`, returning a `Hashtable` object. The allocation site is at offset 1123 within the method. The pretenuring advice for this site is '2' (immortal), which will be used by the compiler to generate an appropriate allocation sequence.

7.2 Exploiting advice files

When the JVM starts up it reads the allocation advice file into a `HashMap`. The keys used to insert and lookup advice are derived from the allocation site descriptor. The class loader in Jikes RVM maintains a collection of `VM_Atom` objects that represent class names, method names, type descriptors, etc. These `VM_Atoms` are singletons, shared between all uses. Our advice system exploits this property by using the class name, method name and method type `VM_Atoms`, along with the offset, as the hash key. This avoids duplicating class or method names. A side effect of this mechanism is that some of the `VM_Atoms` will be constructed ahead-of-time by virtue of their inclusion in an allocation advice file, but this has a negligible impact on space usage.

During method compilation, the modified compiler looks up the advice for each site it finds, and uses that to generate a suitable allocation sequence. For example, if the advice for a site is *immortal*, the compiler will generate instructions to allocate the object in the immortal region. If no allocation advice is found for a site, it is allocated in the default region — typically the nursery. Jikes RVM already performs a minimal form of pretenuring — instances of certain MMTk memory management classes are allocated in the immortal space. These decisions might conflict with our pretenuring advice. Table 4 summarises how conflicts are resolved.

It is important to note that this pretenuring mechanism performs all its work in the compiler, which produces specialised allocation sequences depending upon the pretenuring advice. Hence this mechanism has no runtime overhead, other than a slight increase in compilation time. This is im-

Alloc \ Advice	*Short*	*Long*	*Imm*	*No advice*
Nursery	Nursery	Mature	Imm	Nursery
Mature	Mature	Mature	Imm	Mature
LO Space	LOS	LOS	Imm	LOS
Immortal	Imm	Imm	Imm	Imm

Table 4. Combining MMTk and pretenuring allocation policies.

portant because any overhead on allocation results in a significant slowdown in program execution.

One potential concern about our pretenuring mechanism is the size of the `HashMap` that is used to store the allocation advice. This `HashMap` must be retained for the lifetime of the program because we do not know when new classes may be loaded, or methods recompiled at new optimisation levels. Measurements of the space and time overhead of loading our advice shows that it has minimal impact (Table 5).

7.3 Immortal objects

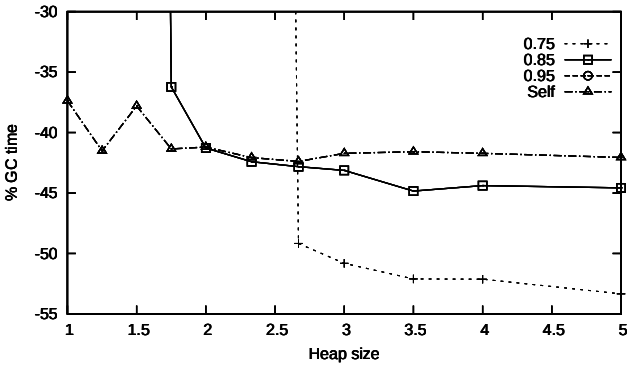
A consequence of pretenuring is that many more objects are allocated in the immortal region of the heap than would otherwise be the case. Although immortal objects are never reclaimed, it is essential to treat any reference fields they contain as roots for other spaces. In MMTk, these fields are discovered from the remembered sets at minor collections, but by tracing the immortal space at full collections. This ensures that only reachable objects in the immortal space are considered in the latter case, hence minimising nepotism. However, this policy increases the work done to allocate in the immortal region compared with allocation in the nursery or mature space. Allocation in those regions is performed by a simple bump pointer, while allocation of an object in the immortal space requires in addition setting the object's mark-bit to the appropriate value. However, the cost of this extra work is negligible.

8. Results

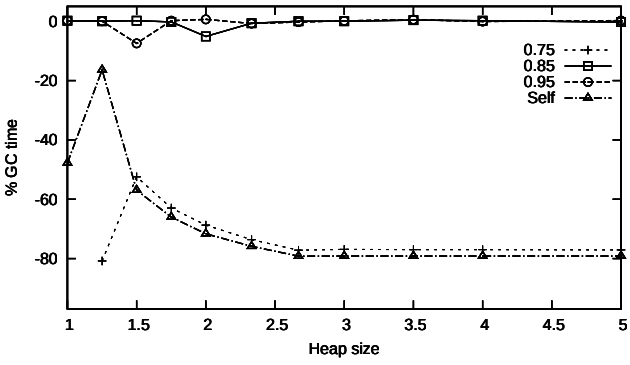
Here, we compare the effectiveness of the pretenuring schemes described in Section 6. We first consider the effect on GC time (both overall and pause time distribution). We then show the effect of pretenuring on overall run-time and object placement (number of sites identified for pretenuring, volume pretenured, etc.). We explore in detail the cost in terms of time and space of loading and exploiting advice.

8.1 Pretenuring schemes

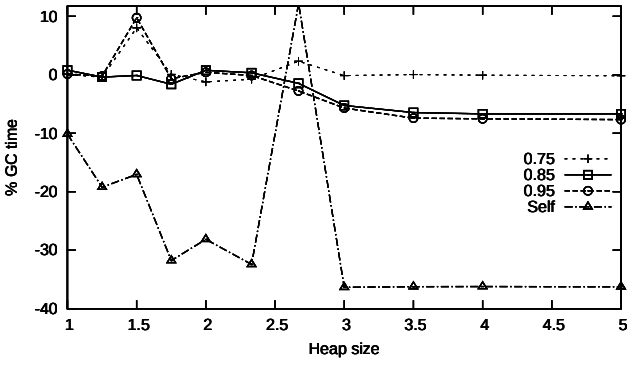
Our pretenuring schemes are determined by two main factors: how the lifetime of a site is classified and the confidence level used. We adopt Blackburn's heuristics for the proportions of short-, long-lived and immortal objects (Section 3). By and large, using a 'true' definition of immortality offers similar, if tending toward more conservative, results.



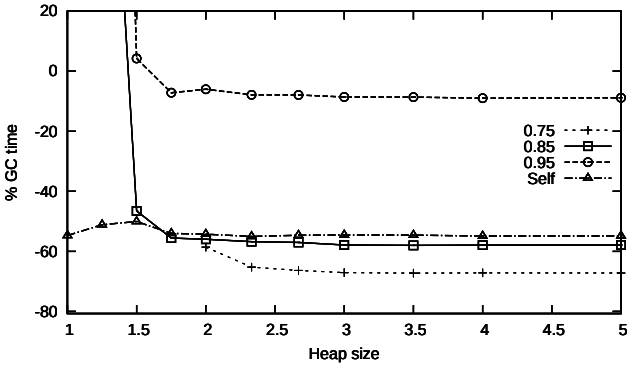
(a) raytrace



(b) db

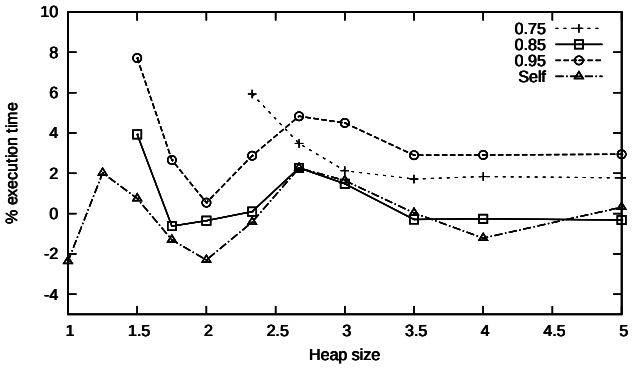


(c) javac

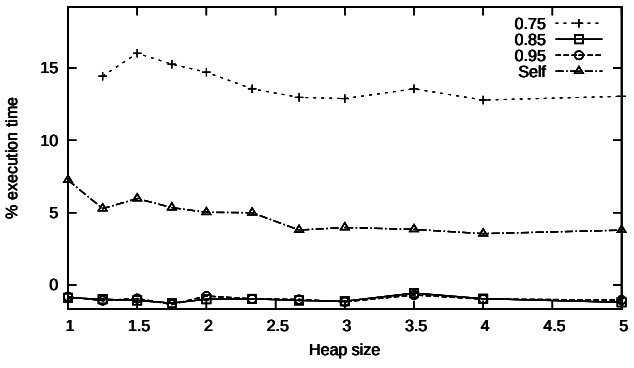


(d) mtrt

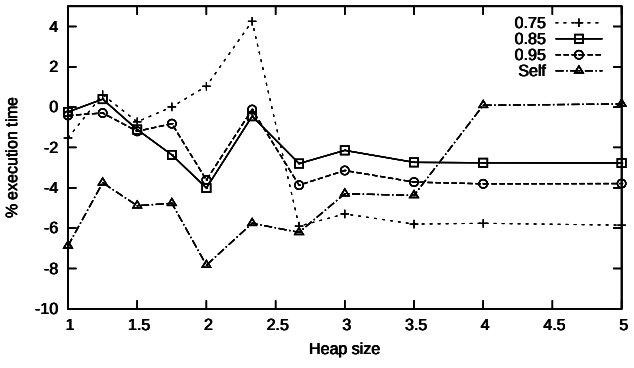
Figure 3. GC time relative to no advice for GenCopy configurations at 75, 85 and 95% confidence and for self-prediction (lower is better).



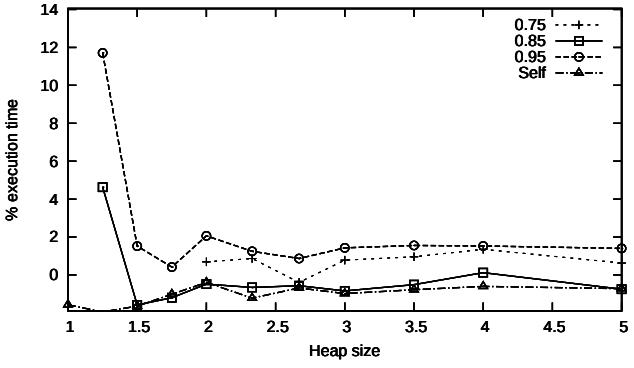
(a) raytrace



(b) db



(c) javac



(d) mtrt

Figure 4. Overall execution times relative to no advice.

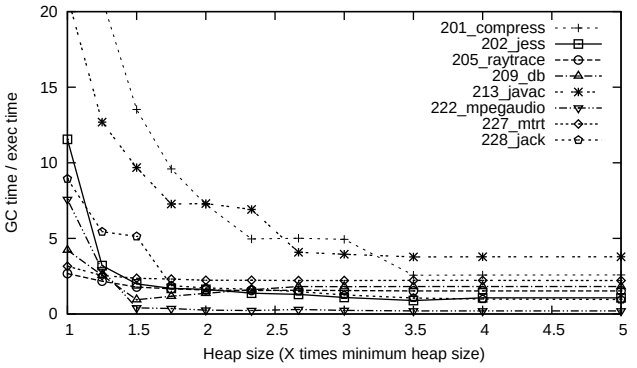


Figure 5. GC time as a fraction of overall execution time for MMTk GenCopy (without advice).

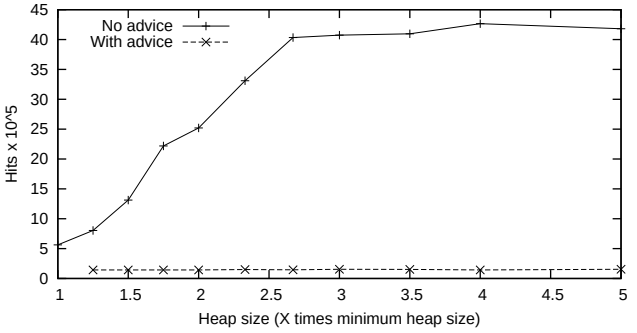


Figure 6. The frequency that the write barrier's slow path is taken, with and without advice, by db at 75% confidence.

8.2 GC time

Figure 3 shows relative GC time for a generational copying collector (GenCopy) for the four jvm98 benchmarks for which we obtained performance improvements. Other jvm98 benchmarks showed no significant change in performance, except for jack which fares very poorly both at 75% confidence and, to a lesser extent, with self-prediction. At 85% confidence, no programs show significant GC time degradation in other than the tightest heaps. The graphs show GC time relative to running with no advice, using rules with confidences 75, 85, 95%, and self prediction, at a range of heap sizes, expressed as a multiple of the size of the smallest heap in which the benchmark would run. 75% confidence generally offers the largest performance gains, improving raytrace and mtrt by around 50% and db by 65% on average; however, in tight heaps 75% pretenures too much, causing the raytrace and mtrt to run out of memory. javac in contrast only shows improvements (around 7%) for confidences $\geq 75\%$, in large heaps.

8.3 GC pause time

Pretenuring aims to reduce the volume of data copied at each collection. It should reduce pause times for full heap collections as immortal data is traced rather than copied; moreover,

as a copy reserve is not needed for immortal objects, pretenuring increases the effective heap size, thereby giving objects longer to die, again reducing the volume to be copied. For minor collections, pretenuring increases the proportion of short-lived objects allocated in the nursery which again should reduce pause times. Figure 7 shows pause time distributions for the benchmarks for which pretenuring improves overall GC time. Pauses are improved in curves that are lower (fewer GCs) and lefter-most (shorter pauses). Where we improve overall GC time, we also reduce the pause time for both full collections (although jvm98 benchmarks do very few full collections) and minor collections.

8.4 Overall execution time

Figure 4 shows overall execution times relative to no advice for a range of confidences. At 75% confidence, gains in GC performance are generally not reflected in overall execution time. Given that GC time represents only a small fraction of overall execution time for the jvm98 benchmarks (Figure 5), it is not surprising that changes in GC time lead to much smaller changes in execution time. However, micro-pattern advice offers performance similar to self-prediction for raytrace and mtrt at 85%, and for javac at 75% in large heaps.

However, some results are counter-intuitive. For example for db, 75% confidence pretenuring improves GC time by 65% but gives extremely poor performance overall. We explore the reasons for this below.

8.5 Loading advice

Accounting for the space used by our advice is complicated because it shares VM_Atoms with the rest of the system (see Section 7). To estimate the space overhead, we measured heap usage at two points: just before a benchmarks starts running and just after it finishes. We trigger a full collection before each of these points to ensure that only live data is accounted. Columns 6 and 7 in Table 5 show the overhead (in 4 KB pages) of loading advice, comparing running with and without advice. Notice that the overhead at the end of the run may be smaller than that at the beginning; this is because some VM_Atoms created by loading the advice are subsequently shared by the compiler.

8.6 Object placement

Table 5 summarises the consequences of the pretenuring decisions made by MMTK and with 75% confidence advice (MMTk places many of its objects in the immortal space, Section 7). Pretenuring has three consequences, particularly for the immortal space. First, objects allocated in the immortal space must have a bit set to the current value of the marking bit. Second, although pretenuring increases the space used by the immortal space, this frees copy reserve, thereby increasing the effective heap available to other spaces, provided the pretenuring decision is correct and does not increase the volume of floating garbage. Pretenuring at 75% confidence places no objects in the mature space (although pretenuring at more aggressive, lower confidence levels does) but places many objects in the immortal space. Third, increasing the volume of objects in the immortal and mature spaces might change the number of cross-region references, which have to be trapped by the write-barrier's slow path and added to the remembered set.

We explored our most consistently contrary performer, db at 75% confidence, which showed improved GC time performance but reduced execution time performance, in order to analyse the causes of the extra mutator overhead.

To estimate a lower bound on the additional cost of allocating in the immortal region, we measured the cost of setting a header bit at *all* allocations and calculated the fraction of this cost proportional to the number of immortal allocations. This is a lower bound because we can assume that the value of the bit will stick in the cache, which it might not if bit-setting is unusual (as it would be for immortal allocation). Under this pretenuring scheme, db allocates 40,047 objects in the immortal space. We find that the initial mark-bit setting overhead is negligible.

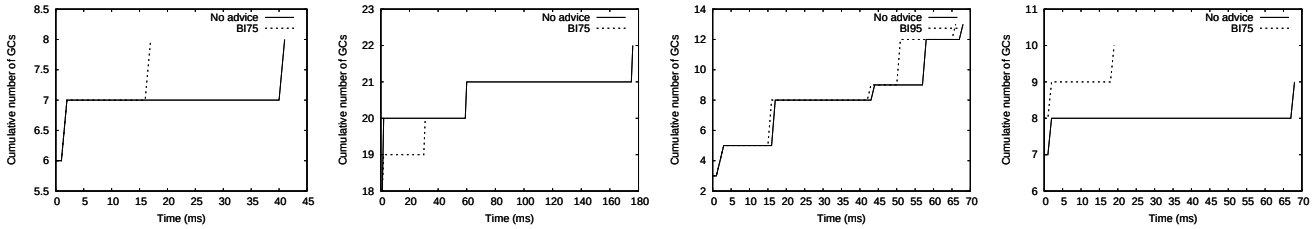
The number of times that the write-barrier slow path is taken can only be reduced by promoting an object to an older space; the larger the nursery, and hence fewer collections, the later that the object is promoted. This accounts for the no-advice curve in Figure 6. However, objects advised to be pretenured are immediately allocated together in the same space. We believe that this is the reason that, rather than increasing its cost, pretenuring has substantially reduced it.

However, we note that db is sensitive to the layout of data since it repeatedly traverses long singly-linked lists [7]. Moreover, GC time accounts for only a very small proportion of overall execution time, so even small mutator overheads will override any improvements in GC time. Examination of the hardware performance counters indicates that, although the number of instructions executed and the cache behaviour for the mutator is similar with and without advice, advice increases DTLB misses by 33%.

9. Evaluation

Data mining object and allocation site lifetimes, obtained from the DaCapo benchmark suite and classified with either the heuristic or true definition of immortality, has shown a link with the sets of micro-patterns [13] exhibited at sites. Data mining is most effective at identifying patterns that tend to lead to immortal or short-lived object; rules that identify long-lived objects are far less frequent, especially at high confidence levels.

For a choice of confidence that is neither too aggressive nor too conservative, rules derived from Clementine's C5.0 algorithm lead to improvements in GC times of between 6–77%. For all programs except jack at 75% confidence, such pretenuring never significantly degrades GC performance. GC performance is never significantly degraded at 85% or more in other than the tightest heaps. The option of running



(a) raytrace, 3×, conf. 75% (b) db, 1.25×, conf. 75% (c) javac, 3.5×, conf. 95% (d) mtrt, 2.33×, conf. 75%

Figure 7. Cumulative pause time distributions, compared with no advice. A point (x, y) on the curve indicates that y collections had a pause time less than x ms.

Program	Min. heap	All sites	Rules	Loading overhead time%	Loading overhead before	Loading overhead after	Advice sites Advice	Advice sites MMTk	Count Advice	Count MMTk	Volume Advice	Volume MMTk	Volume All
compress	21	6655	230	-0.55	8	8	203	83	1467	369	139	114	128
jess	22	6935	289	0.60	8	8	255	83	14871	647	535	133	299
raytrace	30	6757	328	0.63	8	8	303	83	2491889	416	54194	118	166
db	39	6665	246	0.13	8	6	215	83	140067	377	23784	114	101
javac	40	7269	271	1.16	16	13	204	83	25786	665	559	149	225
mpegaudio	18	7656	243	0.66	8	8	217	83	1463	461	140	120	44
mtrt	38	6756	331	0.60	8	8	306	83	2656466	419	57548	118	176
jack	22	6954	308	0.48	8	6	209	83	573176	469	13255	122	312

Table 5. Pretenuring placement at 75% confidence, all benchmarks with speed 100 input. *Min. heap* is the size in MB of the smallest heap in which the program would run. *All sites* is the total number of sites compiled (including Jikes RVM, application and library code used). *Rules* is the number of sites with advice. *Loading overhead* is the percentage execution time overhead to load the advice file, and the space overhead (in pages) before and after the run. *Advice sites* is the number of sites pretenured to the immortal space, either by advice or by MMTk; similarly, *Volume* and *Count* are the number and volume (in KB, or MB for *All*) of objects pretenured.

more aggressive pretenuring schemes gives the application user a tuning option that may, but is not guaranteed to, improve GC performance yet further. The degree of pretenuring aggression can be selected by a command-line switch, without need to rebuild Jikes RVM, nor recompile the application. Construction and data mining of the prediction knowledge bank need be done only once, e.g. by the JVM developer. Currently, we analyse program classes for patterns and match these against the rule set off-line (mainly for ease of experimentation). However, this process is quick and simple and we plan to move it into the class loader in future. Further, our rules can be improved at any time by processing further programs; the results can then be applied to any new or existing program.

Yet, improvements in GC time do not lead consistently in all programs to improvements in overall execution time. Pretenuring, and hence changes of object locality, may increase or reduce pressure on the mutator. If connected objects are pretenured into the same space, whereas they might otherwise have been placed in different spaces, the number of times the write-barrier slow path is taken and the size of the remembered set are reduced (e.g. [15]). Conversely, if connected objects are dispersed to different spaces, these will be increased. Dispersion of objects that might otherwise have been allocated close together affects DTLB behaviour, particularly in programs like db that are sensitive to data layout.

It is possible to identify a richer, yet domain-specific, set of micro-patterns than Gil and Maman's [13]. The MMTk allocator already treats specially allocation of instances of objects that belong to particular packages. We plan to explore this further, by treating certain Jikes RVM-specific packages as patterns to see if this improves treatment of Jikes RVM objects. We also intend to add advice at build-time, rather than just run-time, either using rules derived in this way or using Blackburn et al.'s techniques to combine build-time advice [7]. We expect significant performance improvements, especially in tight heaps.

Allocation sites in Java programs demonstrate a far more diverse object lifetime behaviour than the short, long, immortal classification used above [20]. In particular, it is common for a single site to allocate objects with a bimodal lifetime distribution. Based on the Beltway GC framework [8], we are investigating the extent to which we can capture and exploit these richer patterns.

10. Conclusions

Pretenuring long-lived and immortal objects into regions that are infrequently or never collected can reduce garbage col-

lection costs significantly. However, extant approaches either require extremely computationally expensive, application-specific, off-line profiling, or consider only allocation sites common to all programs, i.e. invoked by the virtual machine rather than application programs.

In contrast, we have shown how programmer's intentions can be captured with micro-patterns [13], applied to object allocation sites. By data mining a large corpus of Java programs, we find relationships between patterns exhibited at an allocation site and the lifetimes of the objects allocated by that site. This analysis is effective at discovering short-lived and immortal objects, but predicts fewer sites that allocate long-lived data.

Our analysis is cheap and could be provided in a class loader (though currently we load advice from files prepared off-line). We obtain performance gains between 6–77% in GC time against a generational copying collector for several jvm98 programs.

We are grateful for the support of IBM through its Faculty Partnership Awards and the EPSRC through grant EP/D078342. Any opinions, findings, conclusions or recommendations expressed in this paper are the authors' and do not necessarily reflect those of the sponsors. We thank Yossi Gil and Itay Maman for providing their pattern analysis tool, IBM Research for making Jikes RVM available, the MMTk Core Team for their GC framework, and the DaCapo group for their benchmark suite.

References

[1] B. Alpern, D. Attanasio, J.J. Barton et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

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

[3] H.G. Baker. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices*, 27(3):66–70, 1992.

[4] D.A. Barrett and B.G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Programming Languages Design and Implementation (PLDI'93)*, 187–196, 1993.

[5] S.M. Blackburn, R. Garner, K.S. McKinley et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Systems, Languages and Applications (OOPSLA'06)*, 2006.

[6] S.M. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *International Conference on Software Engineering (ICSE'04)*, 2004.

[7] S.M. Blackburn, M. Hertz, K.S. Mckinley et al. Profile-based pretenuring. *Transactions on Programming Languages and Systems*, 29(1):1–57, 2007.

[8] S.M. Blackburn, R.E. Jones, K.S. McKinley, and J.E.B. Moss. Beltway: Getting around garbage collection gridlock. In *Programming Languages Design and Implementation (PLDI'02)*, 153–164, 2002.

[9] S.M. Blackburn, S. Singhai, M. Hertz et al. Pretenuring for Java. In *Object-Oriented Systems, Languages and Applications (OOPSLA'01)*, 342–352, 2001.

[10] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[11] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Programming Languages Design and Implementation (PLDI'98)*, June 1998.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[13] J. Gil and I. Maman. Micro patterns in Java code. In *Object-Oriented Programming, Systems, Language and Applications (OOPSLA'05)*, 97–116, 2005.

[14] U. Grimmer. Clementine: Data mining software. In *Classification and Multivariate Graphics*,10, 25–31. Weierstrass-Institut für Angewandte Analysis und Stochastik, Berlin, 1996.

[15] S. Guyer and K. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *Conference on Object-Oriented Systems, Languages and Applications (OOPSLA'04)*, 2004.

[16] T. Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM'02)*, 2000.

[17] M. Hertz, S.M. Blackburn, K.S. McKinley et al. Error-free garbage collection traces: How to cheat and not get caught. In *Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, 2002.

[18] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Object-Oriented Systems, Languages and Applications (OOPSLA'03)*, 2003.

[19] W. Huang and W. Srisa-an and J.M. Chang. Adaptive pretenuring for generational garbage collection. In *International Symposium on Performance and Analysis of Systems and Software (ISPASS'04)*, 133–140, 2004.

[20] R.E. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'06)*, 2006.

[21] M. Jump, S.M. Blackburn, and K.S. McKinley. Dynamic object sampling for pretenuring. In *International Symposium on Memory Management (ISMM'04)*, 2004.

[22] J.Singer, G. Brown, M. Lujan et al.. Towards intelligent analysis techniques for object pretenuring. In *Principles & Practice of Programming in Java (PPPL'07)*, 2007.

[23] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03, 1999.

[24] D. Stefanović, K.S. McKinley, and J..B. Moss. Age-based garbage collection. In *Object-Oriented Systems, Languages and Applications (OOPSLA'99)*, 370–381, 1999.

[25] D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, 1984.

[26] D.M. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. *SIGPLAN Notices*, 23(11):1–17, 1988.