

Meaningful Type Names as a Basis for Object Lifetime Prediction

Jeremy Singer Mikel Luján Ian Watson

University of Manchester, UK
{jsinger,mlujan,iwatson}@cs.man.ac.uk

Abstract

Object lifetime prediction can identify short-lived and long-lived objects at or before their heap-allocation point. This prediction is an increasingly important technique to enable optimization of runtime decisions for high-performance garbage collection. This paper shows that extracting information from object type names can give a reliable and efficient way to make such object lifetime predictions. We use this approach to optimize a generational garbage collector in the Jikes RVM system.

1. Introduction

It is well-known in the *program comprehension* community that important semantic information is encoded in the human-readable, pseudo-natural-language descriptions that programmers use for variable identifiers and type names [1, 2]. However the *program analysis* community generally disregards this information for several reasons.

1. Imprecision: The flexible and unstructured nature of natural language causes problems for automated analysis techniques, due to issues like synonymy and homonymy [3, 4].
2. Inaccuracy: Since different software engineers modify and maintain the source code over time, inconsistencies or contradictions often arise in the naming conventions [5, 6, 7].
3. Incomprehensibility: There is no widely available ontology for program variable names, which means that automated tools cannot easily infer semantic information from human-generated identifier names [8].

This paper explores the use of information contained in Java object type name strings for grouping objects that may have common lifetime characteristics. It is extremely useful to be able to identify cheaply objects that have similar lifetimes, since this can improve garbage collection (GC) performance dramatically [9, 10, 11].

Our new approach avoids the above disadvantages of natural language information by adopting the following strategies.

1. It never attempts to interpret the meaning of type name identifiers. Instead it merely finds lexicographic similarities between different type names.
2. It always uses ahead-of-time profiling to determine whether types with similar identifiers have common object lifetime characteristics. Optimization only occurs when the profile information confirms that there are similarities in object lifetimes.

1.1 Motivating Example

This section presents a simple example to motivate the idea of *type name suffix* as a basis for object lifetime prediction. Throughout the paper, we define *suffix* to be the last word in an identifier composed of multiple words, or the only word in a one-word identifier. Section 3 gives a detailed explanation for how we locate type name suffixes in Java. For the following example, it is sufficient to realise that `StringBuffer`, `RecordBuffer` and `VertexBuffer` all share the common `Buffer` type name suffix.

Now consider a hypothetical database¹ program written in Java. The program uses `StringBuffer` objects to construct and manipulate alphanumeric data fields for each record. These objects are created for ephemeral use. Once their data is transferred to immutable `String` objects, the `StringBuffer` objects are no longer needed. The database program also has a `RecordBuffer` type, which it uses when processing database queries. Such objects are also ephemeral, since they are created specifically for each query

¹The `StringBuffer` and `RecordBuffer` classes occur together in several open-source database systems, according to the *Krugle* online source code search engine.

and not needed after the query has completed. Neither of these types is related by inheritance, other than at the Object level. The `StringBuffer` class belongs to the Java standard library, whereas the `RecordBuffer` class is specific to the database application. Their object lifetime characteristics in this database program are similar, since both `StringBuffer` and `RecordBuffer` objects are generally short-lived. Human programmers have encoded this information in the `Buffer` type name suffix that is common to both classes.

There are two scenarios in which a GC system may be able to use this information. The first involves sharing lifetime information across *program phases*. Suppose the database program has an initialization phase, where it allocates a large number of `StringBuffer` objects. The GC system observes that these objects are mostly short-lived. Then the database program enters a query phase, where it starts to create `RecordBuffer` objects. Although the GC system has not seen many `RecordBuffer` objects so far, it can make predictions about their lifetimes based on the extensive information gathered about `StringBuffer` objects.

The second scenario involves sharing lifetime information across *different programs*. For instance, say the system has previously executed an *image editor* program, and noticed that most objects of type `VertexBuffer` have short lifetimes. The GC system records this trend, and uses it to make lifetime predictions about the `Buffer` classes of the database program during a subsequent program execution

1.2 Contributions

This paper makes four key contributions.

1. It uses semantic information encoded in human-generated identifiers to create ‘hints’ for garbage collection. To the best of our knowledge, this is the first time that meaningful identifier information has been harnessed for memory management.
2. This novel approach enables object lifetime predictions to be made about a previously unseen type, if it shares the same suffix as a previously seen type.
3. Section 3 presents an extensive analysis of garbage collection traces from the DaCapo benchmark suite, to confirm that type name suffix is a good basis for object lifetime prediction.
4. Section 4 reports on sample optimizations implemented for a generational garbage collector in Jikes RVM. The new scheme uses object lifetime predictions provided by type name suffix information. We demonstrate significant performance improvements for the DaCapo benchmark suite in Jikes RVM.

2. Background

2.1 Garbage Collection

Mismanagement of heap-allocated data is common in systems with explicit memory deallocation. Modern programming languages such as Java and C# employ *garbage collection* (GC) for automatic memory management within a managed runtime environment. The programmer allocates objects on the heap. When heap memory usage crosses some threshold, the garbage collector is invoked to identify objects that are no longer reachable via any chain of active pointers. Free heap space is reclaimed in this way.

2.2 Generational GC

Although the widespread prevalence of GC is only a phenomenon of the last decade [12], it has been a topic of research for fifty years [9]. Thus many GC algorithms have been developed and refined. Ungar [13] makes a significant observation that ‘most objects die young.’ This is known as the *weak generational hypothesis* and has been confirmed for many programs, languages and systems by large empirical studies. Only a small proportion of objects survive for a significant amount of time after their allocation. (In keeping with most GC literature, time is measured in overall number of allocated heap bytes, rather than executed processor cycles.) From this observation, Ungar and others propose a *generational* GC scheme [13, 14, 15]. In the simplest case, the heap is subdivided into two regions which are known as *nursery* and *mature* spaces, respectively. Objects are initially allocated in the nursery space, which is collected frequently as it is generally small and fills up rapidly. Objects that survive a threshold number of nursery collections are *promoted* to the mature space. Since the mature space fills up more slowly, it is collected less frequently. Objects in the mature space are presumed to be long-lived, so the GC system does not waste time unnecessarily scanning them at each collection. Nursery-only collections are known as *minor* collections. When the mature space is full, a *major* collection is required. This scans the entire heap at once.

One problem with generational GC is that long-lived objects are always initially allocated in the nursery space, so they will inevitably be scanned and copied to the mature space later. If the GC system can identify long-lived objects ahead of allocation time, it can *pretenure* such objects by initially allocating them directly in the mature space [16]. So long as object lifetime predictions are accurate, this should reduce nursery GC copying overhead. Pretenuring is regarded as a good optimization opportunity for high-performance generational GC systems.

Another problem with generational GC is that short-lived objects are sometimes promoted to the mature space, where they remain alive until the next major collection even if they effectively die almost immediately after their promotion. The situation is worse if such an object has pointers to other short-lived objects, since their lifetimes are also unnat-

urally extended [17]. This extra garbage in the mature space will cause an increased number of mature space collections, which are expensive. A potential solution is to identify short-lived objects in the nursery and never promote these objects to the mature space.

Therefore it seems that the key to effective generational GC is *accurate object lifetime prediction*.

2.3 Object Lifetime Prediction

An object lifetime predictor takes some allocation context (such as object type or allocation site identifier) as input. It produces a lifetime prediction as output, which may be as specific as an integer value [18] or as general as a binary short/long indicator [16].

A prediction scheme usually profiles some program execution, records object lifetime data, then extrapolates from these profile results. The standard kinds of prediction are based on object *type* (cites) or allocation *site* (cites). An example type-based prediction is ‘all objects of class `Foo` are short-lived.’ An example site-based prediction is ‘all objects created by statement `f = new Foo()` at line 42 in program *p* are long-lived.’ Site-based predictions are more fine-grained, since objects of the same type may have different lifetime behaviour at different allocation sites. Although there is some correlation between object type and lifetime, most pretenuring researchers have abandoned type-based prediction, since allocation site is a better indicator of lifetime [19, 20]. However this paper shows that *some* types may be extremely useful for indicating object lifetimes.

Type-based prediction has two clear advantages over site-based prediction. First, type-based prediction has fewer cases to consider than site-based prediction since the number of object types is in practice always much smaller than the number of allocation sites. This leads to a significant *efficiency* saving, particularly in terms of lookup table size, which is an important consideration for runtime predictors.

The second advantage is *generalization*. Object allocation sites are specific to a single program, and in some cases they may be specific to an individual program execution due to different runtime optimization settings. In contrast, types are often *re-used* across different programs, particularly library types and runtime system types. This paper considers suffixes of type names. It shows that these generalize over many programs, and that the object lifetime behaviour is similar across these programs. This enables the use of object lifetime predictions generated from one run of program *p* to make predictions about object lifetimes in a run of an entirely different program *q*. This generalization is the most attractive feature of the novel object lifetime prediction technique proposed in this paper.

3. Analysis of GC Traces

The standard Java naming convention is to merge multiple words into a single word, with each word in ‘initial caps’

format [21].² An example type name is `ByteArrayBuffer`. This type is a conjunction of three words: `byte`, `array` and `buffer`. The last word in a type name is known as the *suffix*. We identify the suffix by searching backward from the end of the type name string, until we reach a capital letter character or an alphanumeric character that is preceded by a separator character such as `_` or `$`. The substring from this position to the end of the string is its suffix. In the above example, the type name suffix is `Buffer`. Our hypothesis is that *type name suffix is an indicator of object lifetimes for that type*.

This section provides experimental support for the above hypothesis, based on the analysis of real-world GC traces. We obtain these traces from Jikes RVM v2.9.1 [22, 23], which is instrumented to record nursery object allocation and promotion events in the default GenMS generational GC scheme. Given this information, it is possible to track each allocated object and whether it is short-lived (dies in the nursery space) or long-lived (promoted to the mature space). We execute six of the DaCapo benchmarks [24] using the default size input sets. For each benchmark, we use a fixed heap sized at twice the minimum heap size in which that benchmark will execute, including a fixed nursery sized at one eighth of the total heap. We use the standard DaCapo test harness and take measurements on the fifth iteration for each benchmark. This should provide a steady-state view of execution, without being overwhelmed by Jikes RVM runtime compilation activity which is more prominent in earlier iterations.

We postprocess each GC trace to group object allocations according to type name suffix. From now on, we refer to such groups as *suffix groups*. For instance, recalling the example database program from Section 1.1, the `Buffer` suffix group would contain all allocations of `RecordBuffer` and `StringBuffer` objects. We analyse the object allocations in each suffix group to determine the proportion of objects with each lifetime. From now on, we say that a suffix group is *skewed* if more than 95% of object allocations in the group have the same age type (either young or old).

The rest of this section investigates the suffix groups for the DaCapo benchmarks, and addresses five important questions about the properties of these suffix groups.

3.1 Are Skewed Suffix Groups Important?

We investigate how many object allocations arise from objects belonging to skewed suffix groups. We present these results as proportions of the total number of object allocations, for each benchmark. This enables us to assess the significance of allocations from skewed suffix groups. Skewed suffix groups can only be a reasonable basis for optimization if they are commonplace. Figure 1 shows that allocations from skewed suffix groups account for the majority of object allocations in all benchmarks. Importantly many allocations come from suffix groups that have more than one

² also known as ‘Camel Case’.

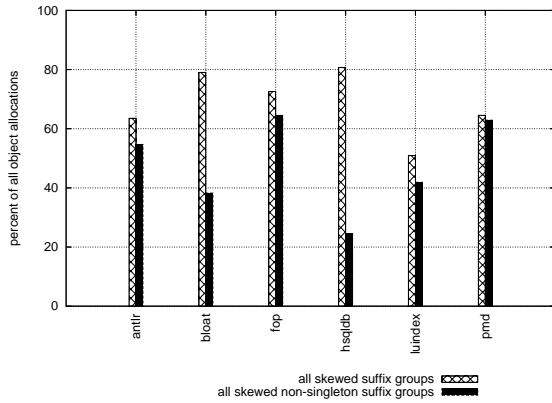


Figure 1. Proportion of all object allocations that are objects belonging to skewed suffix groups

class, which strengthens the case for type name suffix generalization. We refer to suffix groups with more than one class as *non-singleton* suffix groups. Note that singleton suffix groups are a degenerate case. If they are responsible for many object allocations then they are merely prolific types, as already identified by Shuf et al [25]. However it is important to remember that we can still exploit type name suffix commonalities from singleton suffix groups across different programs. Shuf et al do not investigate this possibility.

3.2 Which Skew Type is More Popular?

There are two different kinds of skewed suffix groups. The first kind is skewed so that more than 95% of the allocations are short-lived (*young-skew*). The second kind is skewed so that more than 95% of the allocations are long-lived (*old-skew*). A different optimization is appropriate for each skew type. Figure 2 shows that young-skew suffix groups are far more prevalent than old-skew suffix groups, for all benchmarks. The graph measures the absolute number of object allocations for objects that belong to a skewed suffix group. It gives figures for young-skew and old-skew, both for all suffix groups and for non-singleton suffix groups. Note the logarithmic scale on the *y*-axis. Apart from hsqldb, all the benchmarks have around two orders of magnitude more allocations from young-skew suffix groups than from old-skew suffix groups. This is clear confirmation of the weak generational hypothesis, except for hsqldb.

3.3 How Homogeneous are Skewed Suffix Groups?

Given that a non-singleton suffix group contains object allocations from several different classes, we must investigate whether it is fair to generalize the behaviour of the classes. That is to say, is the skew of the suffix group shared by the skews of all the classes when considered separately? For this analysis, we take each suffix group and calculate its overall skew. Then for skewed suffix groups, we examine all the classes that make up the group and see if the ob-

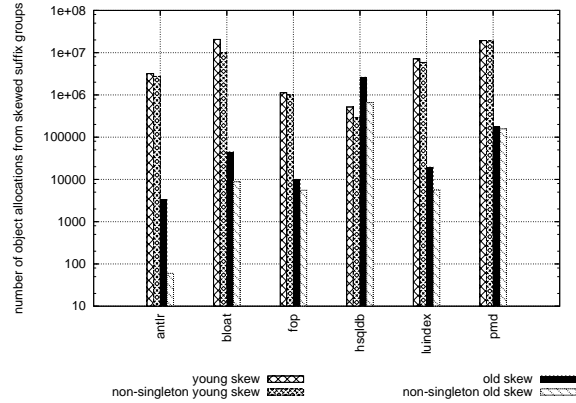


Figure 2. How object allocations are divided between young-skew and old-skew groups

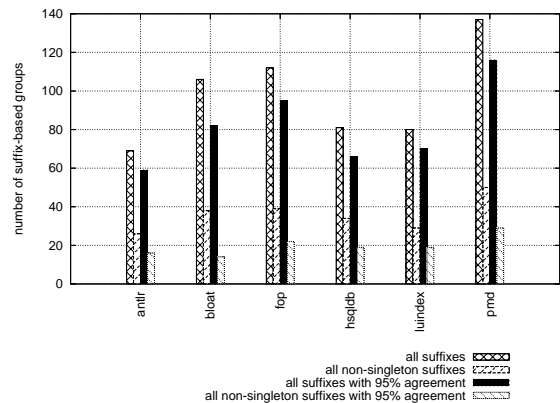


Figure 3. How individual class object lifetime distributions relate to suffix group lifetime distributions

ject allocations for each individual class have the same skew (more than 95% short-lived or long-lived) as the whole suffix group. Figure 3 presents the results of this analysis. First, we give an idea of the limit since we report the number of all skewed suffix groups and all non-singleton skewed suffix groups. (These are the leftmost two columns for each benchmark.) Then we report the number of suffix groups where 95% or more of the classes that comprise this group have the same skew as the group itself (third column for each benchmark). Finally we do the same for non-singleton suffix groups (rightmost column for each benchmark.)

The relative heights of columns 1 and 3 indicate the homogeneity of all suffix groups. It seems that around 85% of the suffix groups contain classes that mostly agree with the overall suffix skew. Only a small minority of suffix groups contain inter-class disagreement. Columns 2 and 4 present the situation when we eliminate single-class suffix groups. Now around 60% of suffix groups contain classes that mostly agree with the overall suffix skew.

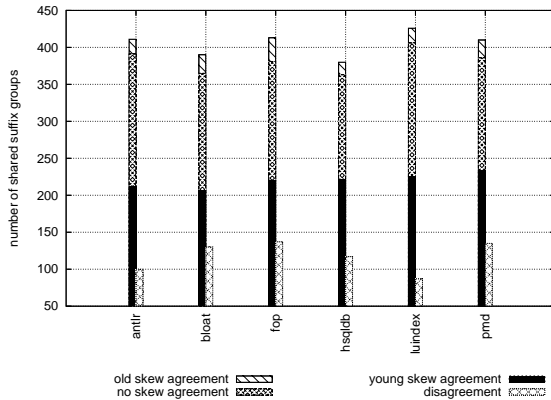


Figure 4. How suffix groups are shared between different benchmarks in the DaCapo suite

3.4 How General are Skewed Suffix Groups?

In order to strengthen the case for sharing lifetime predictions across different programs, it is necessary to show that suffix groups generalize across the DaCapo benchmarks. For this, we record the suffix name for each skewed suffix group in each benchmark, then count how many times one of these names appears in a list of suffix names in another DaCapo benchmark. We also record whether there is skew agreement or disagreement between the suffix groups in different benchmarks. Note that the same suffix for benchmark b is counted N times if it appears in N other benchmarks. For example, suppose the `Foo` suffix group appears in benchmarks `antlr`, `bloat` and `fop`. In `antlr` and `bloat`, the `Foo` suffix group has the same young-skew, so this counts as +1 for the young-skew agreement section in both these benchmarks. However in `fop`, the suffix group is not skewed (i.e. between 5% and 95% of the objects are short-lived) so this counts as +1 for disagreement for `antlr` and `bloat`, and +2 for disagreement in `fop`. Figure 4 shows this analysis. It is clear to see that agreement clearly outweighs disagreement for all benchmarks. It is also clear that young-skew agreement is much more common than old-skew agreement across the benchmarks.

3.5 What Kind of Code Causes Skewed Suffix Groups?

The previous study demonstrates that skewed suffix groups are shared consistently across benchmarks. Now we endeavour to establish the cause of this sharing. On one hand, the common suffixes could be due to shared code from Jikes RVM and the standard libraries. On the other hand, the common suffixes could arise from entirely independent user source code. Figure 5 shows how the object allocations from skewed suffix groups are divided between these various code sources.

As may be expected, there is a large amount of library code in skewed suffix groups. One of the strengths of the Java programming language is its extensive and easy-to-use

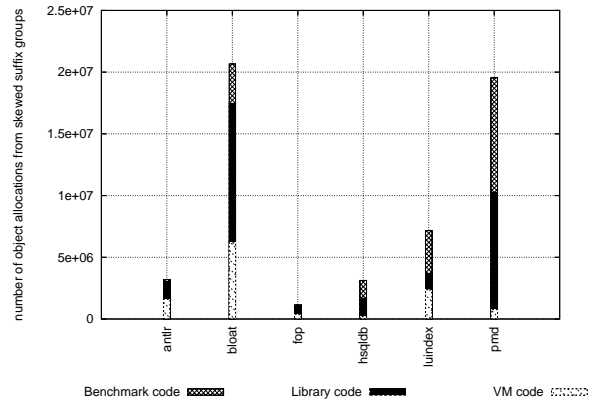


Figure 5. How skewed object allocations are shared between different source code bases

library code base. In every case, object allocations from VM code are also prevalent. Since Jikes RVM is written in Java, it is straightforward to optimize allocation decisions for these types. Blackburn et al [26, 27] demonstrate that significant optimization is possible by pretenuring VM objects without analysing or modifying user source code or standard libraries.

However four of the benchmarks also have a reasonable proportion of skewed object allocations arising from user code, which confirms our idea that type name suffix may provide a basis for *generalized* object lifetime prediction.

4. Evaluation of GC Optimizations

This section describes how we use suffix-based object lifetime predictions to optimize GC performance. We evaluate our optimizations in a customized version of Jikes RVM.

We study optimization of short-lived and long-lived objects separately. It seems that the programs we studied can benefit from either one or the other optimization, but not both.

4.1 Infrastructure

We conducted these experiments using an extended version of Jikes RVM v2.9.1, which is built from the same code base that we used to obtain the GC profiling statistics in Section 3. We adapted the default MMTk generational garbage collection scheme as follows. Our modified heap structure is split into two copying nursery semi-spaces, and a mark-sweep mature space. This setup is reminiscent of Sun's default HotSpot JVM heap configuration, without the initial Eden pre-nursery phase [28].

An ahead-of-time profiling run identifies young- and old-skew suffix groups for a *training* benchmark set and records these type name suffixes. An automated script adds new Java source code to the `VM_Type` class to recognise type names for skewed suffix groups at VM type resolution time, by simple string comparison of `VM_Atom` character strings. Each

Object Allocation

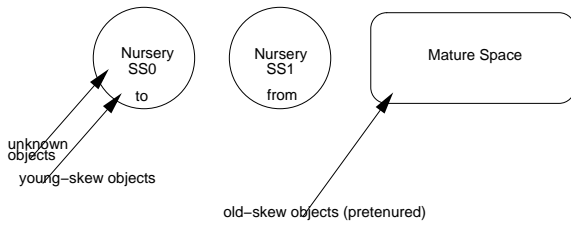


Figure 6. Allocation of objects in our GC scheme

Nursery Collection

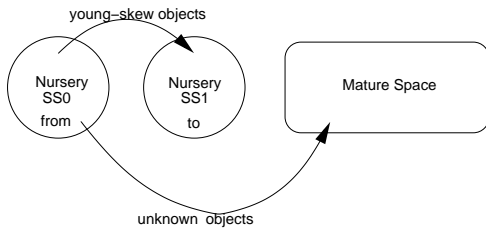


Figure 7. Nursery Garbage Collection in our GC scheme

type belonging to skewed suffix group is marked as such, by writing to a field in the corresponding `VM.Type` object.

We then build this specialised, profile-guided version of Jikes RVM, and run it on a *testing* benchmark set. Every time the VM allocates an object at runtime, it checks the type to determine whether it is designated as short- or long-lived and therefore should be handled specially. Figure 6 illustrates where different kinds of objects are initially allocated. All undesignated objects are allocated in the nursery. Long-lived objects are initially allocated in the mature space. Short-lived objects are allocated in the nursery, and have a reserved *juvenile* object header bit set to indicate that they should remain in the nursery after collection. The GC copies such juvenile objects to the other nursery semispace at collection time, whereas all non-juvenile objects are copied to the mature space. Figure 7 illustrates how different kinds of nursery-allocated objects are moved at GC time.

MMTk generational schemes use slot-based remembered sets to track inter-generational pointers [29, 30]. The original scheme required remset entries to be accumulated during mutator execution, when nursery references are written to non-nursery objects. This is a simple write barrier. The remset is processed, with slots being updated, at each minor collection. The remset is flushed without processing at each major collection. Either way, the remset is empty at the end of a garbage collection phase.

Our new scheme requires additional remset entries, since it retains juvenile objects in the nursery after collections. So slots containing references to such juvenile objects have to be in the remset at the end of a garbage collection phase. This

requires minor changes to the GC tracing code. Blackburn et al present a study to show that the performance impact of write barriers and remembered sets is generally minimal [31].

4.2 Result Reporting Rules

We use the production build of Jikes RVM, disabling all assertions and extraneous logging. All timing results from our experiments are taken as the mean of 10 tests, together with quoted confidence intervals for one standard deviation either side of the mean. The raw mutator and collector timings are obtained from the DaCapo MMTk harness. All benchmarks are supplied with their default input sets.

As recommended by Georges et al [32] we take results from a steady state iteration of the benchmark to minimise the overhead of adaptive compilation activity. We use fixed size heaps and nursery spaces. All heap sizes are reported as multiples of the minimum heap size in which each particular benchmark would execute successfully with no `OutOfMemoryException` messages. All nursery sizes are reported as proportions of the current total heap size.

4.3 Optimizing Short-Lived Objects

Short-lived objects are much more common than long-lived objects. This is the basis of the weak generational hypothesis [13]. It is confirmed by our earlier studies, see particularly Figure 2. The default behaviour of a conventional generational collector is to assume that all objects are short-lived, and therefore should be allocated in the nursery. At collection time, survivor objects are promoted to mature space. However, some short-lived objects will have only just been created and are likely to die imminently. They are *accidentally* promoted since they were allocated too near to the beginning of collection. Xian et al [33] refer to such objects as *surviving new objects*. Such objects waste mature space and nepotistically drag other referents to mature space, often with implications for performance [17]. One potential solution is to *retain* some objects in the nursery over a collection. As described in Section 4.1, these nursery-retained objects should be explicitly marked as *juvenile* at their allocation point, then they are copied to the other nursery semi-space at collection time.

This is the short-lived object optimization we will apply in the following experiments (Sections 4.3.1 and 4.3.2). Note that in these sections, we only apply the short-lived object optimization. We *do not* pretenure any long-lived objects. We aim to quantify the performance impact of the short-lived and long-lived optimizations in *isolation*.

4.3.1 Self-Prediction Experiments

The simplest evaluation of GC optimizations is to profile and test on the same benchmark with the same parameters. In machine learning terms, this means using the same benchmark for the training set as for the testing set. Such *self-prediction* [34] should give an upper bound on the per-

<i>benchmark</i>	<i>skew threshold</i>	<i>allocation threshold</i>
antlr	97%	3%
bloat	93%	3%
fop	93%	1%
hsqldb	97%	1%
luindex	99%	1%
pmd	99%	3%

Figure 8. Optimal parameters for short-lived object optimization with self-prediction

formance improvements possible from GC optimizations. Blackburn et al generally use self-prediction in their pre-tuning studies to give performance limits [26, 27].

In our self-prediction experiments, we identify young-skew suffix groups for a particular benchmark, then create a Jikes RVM build specialised to optimize objects with these suffixes. Finally we run the same benchmark in the modified VM and gather performance figures. We compare the performance of the modified VM against a *default* VM that has the same heap configuration but does not perform any suffix-based optimization.

There are many parameters to vary in the experiments. During the identification of young-skew suffix groups, we can change the percentage of young objects required to skew the suffix group (known as *skew threshold*). In all our profiling tests in Section 3 we set the skew threshold to 95%. We can also set a threshold on the number of object allocations in a suffix group before we consider it sufficiently prolific to warrant optimization. We measure this as the proportion of object allocations in a single suffix group compared to the total number of object allocations throughout the benchmark run (known as *allocation threshold*). These two parameters affect how many suffixes are to be optimized by the modified Jikes RVM.

When we evaluate the performance of the modified Jikes RVM, we can vary the heap size parameters to change the stress levels of the GC subsystem. In small heaps with small nurseries, there will be more frequent collections than in larger heaps with larger nurseries. When there are more collections, the performance impact of any GC optimizations is more strongly emphasized.

Since we intend to use these self-prediction results to show the upper bound on the potential of suffix-based object lifetime prediction for GC optimization, then we conduct a coarse-grained automated search of the parameter space outlined above. We found that we get greatest performance gains for benchmarks running in their minimum heapsize, with a fixed nursery size of 5% of the total heap (in practice, rounded to nearest MB). Figure 8 shows the optimal parameters chosen (from our coarse-grained exploration) for each benchmark to generate best performance.

Given these optimal parameters, Figure 9 shows the GC times for four of the DaCapo benchmarks, comparing the *de-*

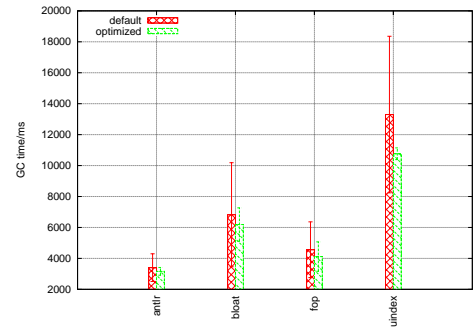


Figure 9. Benchmarks for which juvenile optimizations reduce GC times

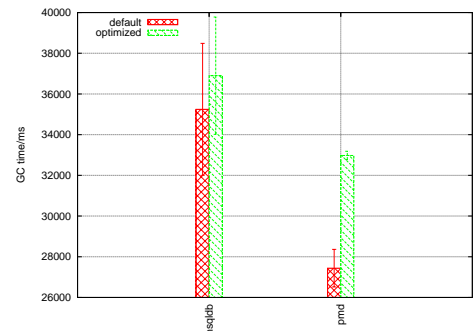


Figure 10. Benchmarks for which juvenile optimizations do not reduce GC times

fault no-optimization time with the suffix-based prediction *optimized* time. For each of these benchmarks, the optimized version gives a speedup in GC time, up to a maximum of 23% for luindex. In contrast, Figure 10 shows the GC times for the two DaCapo benchmarks for which the optimization does not cause improvement. We note that these are longer running benchmarks. The hsqldb program is a particularly notable exception to the weak generational hypothesis, as Figure 2 shows.

While it is interesting to consider GC time improvements, the optimization is only significant if it also leads to overall execution time improvements. Figure 11 shows that for the same four benchmarks as above, there is a corresponding reduction in overall execution time. These speedups are due to the reductions in GC time, since GC time is a large proportion of overall execution time in tight heaps (up to 69% in bloat, for instance). In contrast, Figure 12 shows the two benchmarks for which the optimization increases overall execution time. This degradation is largely since GC time is increased by the optimization.

Next we study how the impact of this short-lived object optimization varies as the heap size increases. Recall that in all the experiments reported above, each benchmark has executed in its minimum heap size. Figure 13 shows how the GC time changes for both default and optimized VMs

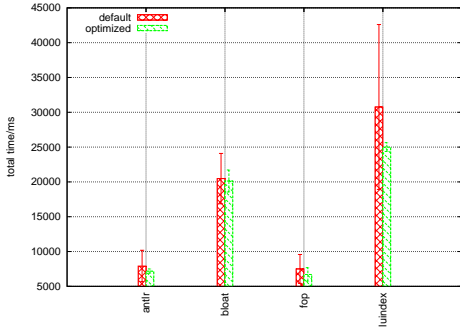


Figure 11. Benchmarks for which juvenile optimizations reduce total execution times

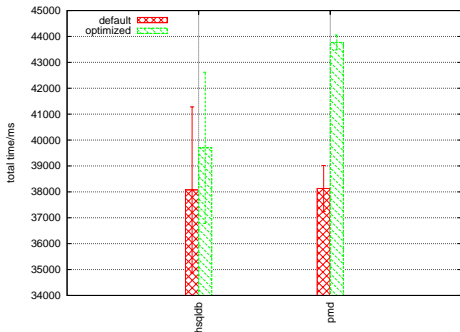


Figure 12. Benchmarks for which juvenile optimizations do not reduce total execution times

as the heap size is increased in proportion to the minimum heap size for each benchmark. First, it is clear to see that GC time reduces significantly as the heap size increases. (Since the nursery is still fixed at 5% of the total heap size, then the nursery size is also increasing from left to right in each graph.) Second, it seems that the impact of the short-lived object optimization is reduced until it becomes negligible, as the heap size increases.

Discussion At this point, it is appropriate to mention that the standard deviation measurements (shown as confidence intervals in all the graphs in this section) are relatively large. This could imply that our results would not hold under rigorous statistical analysis [32]. However we present two arguments in our defence. First, it is a common concession that GC measurements always contain significant noise levels [35] since GC is inherently non-deterministic, especially in an adaptive runtime environment like Jikes RVM. This noise will always increase the variance of any set of GC time measurements. Second, we note that many existing *published* GC optimizations demonstrate smaller performance gains than our optimization, and do not even consider variance in their results reporting method. We advocate that our presentation must be ‘at least’ as convincing as this literature.

The results show that our optimization only works on four out of six DaCapo benchmarks. This leads us to suggest

that the optimization should be enabled as adaptive runtime behaviour which can be triggered or disabled depending on how effectively it works. The optimization is only really applicable in tight heaps. One problem in larger heaps is that remembered sets grow quickly and contain many repeated entries, since collections are less frequent. We confirm this by extensive profiling. Significant GC time can be wasted in processing these repeated remembered set entries. One of our future objectives is to remove such repeated entries.

4.3.2 True Prediction Experiments

The most realistic evaluation of GC optimizations is to profile and test on different benchmarks. In machine learning terms, this means using disjoint sets of benchmarks for the training and testing sets. Such *true-prediction* [34] should give a realistic idea of the likely performance improvements from GC optimizations. We employ the *leave-one-out cross-validation* methodology (LOOCV). Given a set of N benchmarks, LOOCV trains on $N - 1$ benchmarks and tests on the N th. This can be repeated N times, rotating the test benchmark each time.

Given the levels of inter-benchmark suffix commonality we found in Section 3, we anticipate that true-prediction should lead to performance optimizations for the benchmarks in which self-prediction worked well. Recall from the previous section that these benchmarks are antlr, bloat, fop and luindex.

In our true-prediction experiments, we identify young-skew suffix groups for all except one of the four benchmarks, then we build a Jikes RVM build specialised to optimize objects with these suffixes. Finally we run the last benchmark in the modified VM and gather performance figures. We compare the performance of the modified VM against an equivalent VM that does not perform any suffix-based optimization. We use the same thresholds and heap parameters for each benchmark as in Section 4.3.1.

Figure 14 shows the effects of juvenile object optimizations on the four benchmarks with LOOCV at the minimum heapsize for each benchmark. The only benchmark to show noticeable improvement is antlr. The bloat and fop benchmarks have negligible improvements. The luindex benchmark performance slightly degrades with the optimization. When we checked the suffix predictions, luindex has one key suffix `Match` that is not included in the other three benchmarks, so this accounts for its poor performance under LOOCV. Figure 15 shows how the total benchmark execution time changes for the same experiments. Now both antlr and bloat have improved execution times with the optimization. We attribute the antlr performance to the improved GC time. This is not the case for bloat. We surmise that the optimization, while not saving GC time, has had some second-order memory effect, such as better caching, that improves overall execution time.

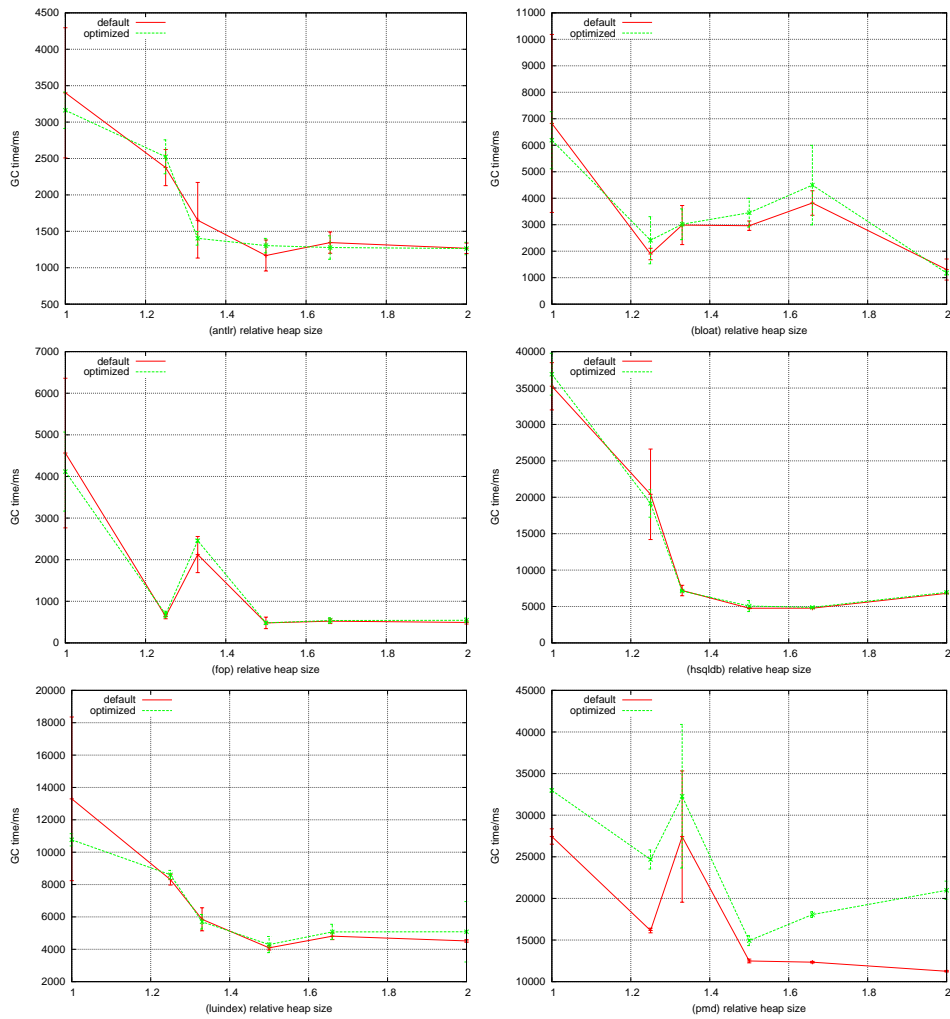


Figure 13. How the impact of the juvenile optimization is reduced as the heap size increases, for each benchmark

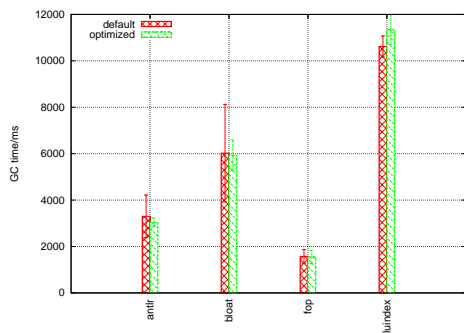


Figure 14. Leave-one-out cross-validation of juvenile optimization, GC times

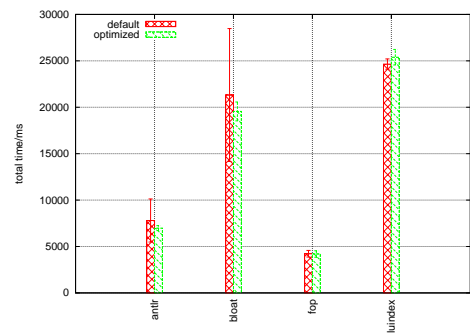


Figure 15. Leave-one-out cross-validation of juvenile optimization, total execution times

4.4 Optimizing Long-Lived Objects

As already noted, long-lived objects are much less frequent than short-lived objects. Figure 2 shows that the hsqldb and pmd benchmarks have the highest relative proportion of old-

skew suffix groups, out of all of the DaCapo benchmarks we examined. These are the only benchmarks for which we can significantly improve GC performance by long-lived object optimization, since they have high numbers of objects that

can potentially be pretenured. Recall from Section 4.3 that hsqldb and pmd were the only benchmarks that we could not improve by short-lived object optimization.

The standard optimization for long-lived objects is to allocate them initially into the mature space, to save copying time. This is called pretenuring [16]. In the experiments below, we enable pretenuring based on suffix names in Jikes RVM. We measure the performance impact of pretenuring in isolation, so we do not optimize short-lived objects in these experiments.

4.4.1 Self-Prediction Experiments

We analyse the profile information to determine old-skew suffix groups for each benchmark. We use an allocation threshold of 1% and a skew threshold of 50%.³ We build a specialized Jikes RVM for each DaCapo benchmark, that pretenures objects with old-skew suffixes. Then we re-execute each DaCapo benchmark and gather timing information using the same results reporting rules as above. We compare the times of our optimizing Jikes RVM runs against execution in a non-pretenuing version of Jikes RVM, using exactly the same runtime settings and experimental sampling policy. We find that speedups are greater in large heaps, so we fix all benchmark heap sizes to be four times the minimum heap size for that benchmark. We fix the nursery to be 12.5% of the total heap size.

Figure 16 shows the effects of pretenuring optimizations on GC times. The hsqldb benchmark has the most dramatic improvement, with a speedup in GC time of 90%. The pmd benchmark has a more modest GC time speedup of 14%. The other four benchmarks do not show GC time improvements. The other benchmark results are disappointing. This is presumably due to the relatively small number of long-lived objects in the heap. Since we have set an extremely aggressive skew threshold of 50%, we are probably pretenuring too many objects that are not genuinely long-lived. This will cause noticeable increases in GC time, as is the case for bloat and fop. Ideally we should conduct a more disciplined exploration of the threshold parameter space.

Figure 17 shows the effects of pretenuring optimizations on total execution times. The hsqldb performance is almost unchanged, since GC time is only a small proportion of overall execution time. The pmd benchmark does show an overall speedup of 7% from pretenuring optimizations. The antlr benchmark shows some overall improvement too, perhaps due to secondary effects like caching. The other benchmark performances degrade, due to over-application of pretenuring optimizations on short-lived objects.

4.4.2 True Prediction Experiments

We did not perform any true-prediction experiments for long-lived object optimization. From the self-prediction

³ We use this lower threshold, since most of the benchmarks, except hsqldb, have few long-lived objects, therefore few old-skew suffix groups, as we noted in Section 3.2.

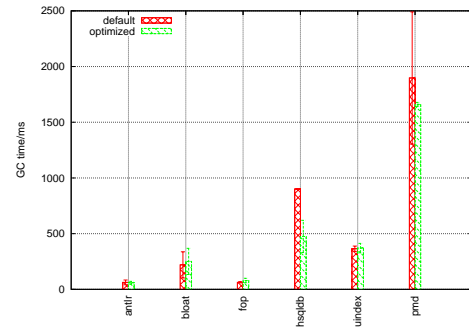


Figure 16. Effect of pretenuring optimizations on GC times for the DaCapo benchmarks

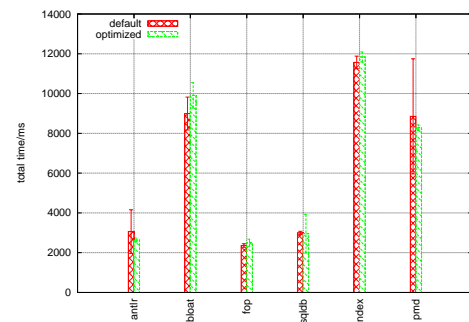


Figure 17. Effect of pretenuring optimizations on total execution times for the DaCapo benchmarks

study in Section 4.4.1 the only significant suffixes that have any performance effect are in hsqldb. These suffixes are not found in other DaCapo benchmarks. Therefore true-prediction experiments would not show any performance improvement. We require a much larger training set of benchmarks with pretenuring potential to be able to show good performance results from true-prediction.

5. Manual Examination of Suffixes

In this section, we open the ‘black-box’ and examine the type name suffixes that are representative of young- and old-skew suffix groups, in Section 5.1. We also consider whether natural language analysis problems actually apply in the programming language domain, in Section 5.2.

5.1 Intuitive Explanations

In our opinion, the most useful application of this kind of data mining in systems research is not the auto-generation of heuristics, but rather the *hindsight* gained from manual post-processing of these heuristics.

Therefore, we attempt to select some example suffix names and explain why such objects should be short- or long-lived? Our explanations should seem intuitive to experienced programmers.

Short-lived objects: One of the most common suffix for short-lived objects is `Buffer`. It is present in each of the LOOCV true-prediction VMs in Section 4.3.2. The most prolific class in this suffix group is `java.lang.StringBuffer`. This is typically a short-lived object, since it is created for some short-term character manipulation, then the data is transferred to an immutable `String` object and the `StringBuffer` dies. Another suffix present in each true-prediction VM is `1`. This indicates that anonymous inner class objects are also generally short-lived, which is also intuitive.

Long-lived objects: The `hsqldb` program is the only benchmark for which long-lived object optimization is effective. The two most common long-lived object suffixes in this program are `Row` and `Transaction`. These both relate to classes that comprise the internal structure of the SQL database query system, which should live for the duration of the program execution. We could not discover any more generic long-lived object suffixes from our small corpus of benchmark programs.

5.2 Semantic Problems

There are many difficulties with using natural language information to drive program optimization, as outlined in Section 1. Two of the biggest problems are synonymy and homonymy. Sections 5.2.1 and 5.2.2 explore these respective problems to analyse how much they affect our analysis, and possible consequences for optimization.

5.2.1 Synonymy

Synonymy is the use of different words to mean the same thing. For instance, a software engineer may consider that the words ‘code’ and ‘text’ refer to the same underlying concept. A computing analogy is the memory *aliasing* problem, when multiple reference variables point to the same memory word.

A common instances of synonymy, which we discovered by manual inspection, is the use of abbreviated forms of long nouns. Examples include `Enum` for `Enumeration` and `Itr` for `Iterator`. A more interesting case is the use of genuine synonyms to describe the same concept. One example pair from our data is `Iterator` and `Traverser`. These suffixes both clearly refer to objects that sequentially visit each element in a collection. Both suffix groups contain mostly short-lived objects, according to our analysis.

In order to assess the extent of synonymy in our data, we require an automated synonym detection system. We use *WordNet* 3.0 [36] which is a lexical reference system designed according to psycholinguistic theory. Words are organized into synonym sets, with each set representing one underlying lexical concept. Various relations link the synonym sets. We use *WordNet* to provide an upper bound on the number of synonyms for each suffix, by treating each suffix as an English word and using it as a *WordNet* noun-synonym query. We measure the number of synonyms for each suffix, and sum this over all suffixes present in each

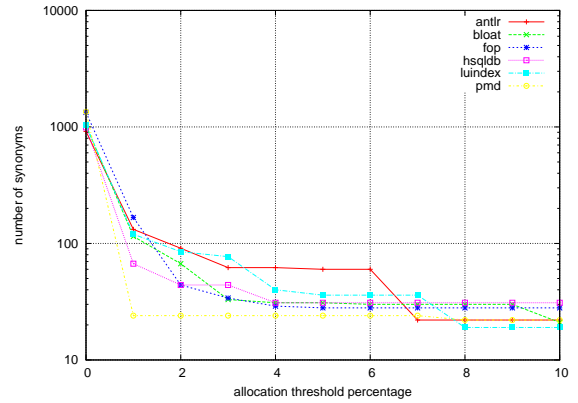


Figure 18. An upper bound on the number of synonyms for suffixes in each DaCapo benchmark program

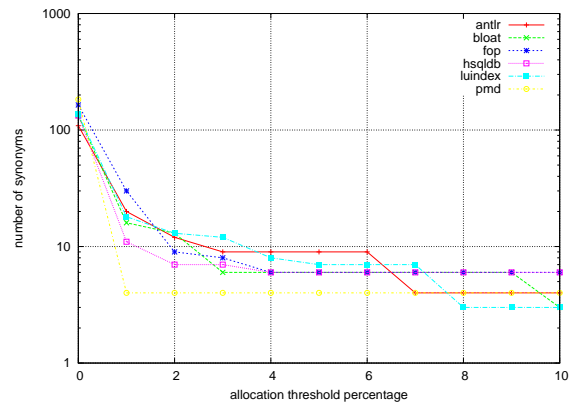


Figure 19. Actual number of synonyms for suffixes in each DaCapo benchmark program, as found throughout the set of benchmarks

benchmark program. We investigate how the number of synonyms changes as we filter suffixes based on their allocation threshold for each particular benchmark. Figure 18 shows this upper bound.

Note that this upper bound is not realistic, since many of the synonym words would rarely occur in a programming context. For instance, while `Container` is a common suffix in our data, *WordNet* indicates that this noun is synonymous with both *Bag* (reasonable) and *Lady* (misogynist!).

Thus we undertake a second study. Now, for each synonym that *WordNet* suggests, we check to see whether this synonym occurs in the source code (of any of the programs, including the program in which the suffix originally occurred). Note that for the moment, we only count the synonym once even if it occurs in multiple programs. Figure 19 shows the results of this study. Note that the number of synonyms is much lower now. Also, the number of synonyms drops considerably as the allocation threshold is raised.

Implications for Optimization: If two suffixes are synonymous, then there are two possibilities. Either (a) they have the same suffix skew and should be optimized in the same way, or (b) they have different skews, so they should be handled differently. If we fail to spot the synonyms in case (a) then we have a missed optimization opportunity.

Now we examine the synonyms in our data, to see whether they have the same suffix skew or not. For each benchmark, we consider all the suffixes in that benchmark, at a given allocation threshold. For each suffix, we consider all synonyms of that suffix. We see whether then synonyms are present in the same benchmark, or in other benchmarks in the set. We see whether the synonym skew (short, none or long) agrees with the original suffix skew, and record this information. Unlike the previous study, now if suffix A has synonym B, and B occurs in multiple benchmarks, then we count B multiple times since sometimes it might agree with A's skew and sometimes it might disagree! Figure 20 shows the agreements and disagreements between synonyms for each benchmark. It is apparent that disagreement outweighs agreement, both within a single program and across multiple programs. This disagreement becomes more likely at higher allocation thresholds.

From this study, we could conclude that WordNet is generally 'conservative' and overestimate synonymy. At least, we can be fairly sure that synonymy is less likely in programming language than in natural language on the evidence above. Since disagreements are more likely than agreements, according to Figure 20, it seems that we have not missed many optimization opportunities due to synonymy.

5.2.2 Homonymy

Homonymy is the use of the same word to mean different things. For instance, a hardware engineer understands 'RAM' to be a lump of silicon, whereas a farmer generally expects a 'ram' to be a male sheep. A computing analogy is the *false sharing* problem, where unrelated memory words may be located in the same cache line. We were unable to find any example homonym suffixes from a simple manual inspection of the benchmark source code.

We used WordNet to analyse the potential homonyms in the suffixes. As before, WordNet provides a conservative *overestimate* of homonymy since it only gives a *polysemy count*, i.e. it lists all the meanings for a word. When such meanings are related then they are not true homonyms. We could guess that such objects should probably have similar lifetime characteristics. This requires further investigation and confirmation.

Figure 21 indicates the number of suffixes for each benchmark, and how this number changes when we only consider suffixes above a certain allocation threshold. While there are a large number of suffixes in general, only a few prolific suffixes need to be considered at higher allocation thresholds. Figure 22 shows the mean number of homonyms per suffix,

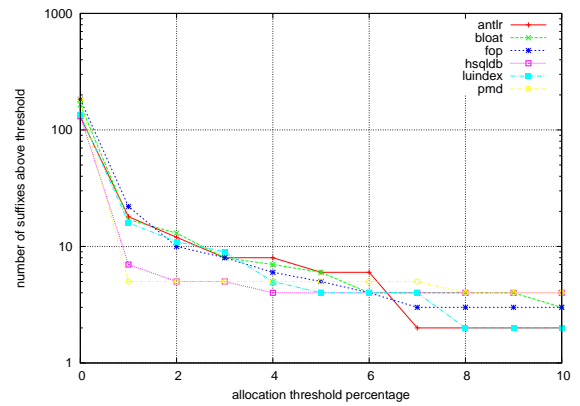


Figure 21. Number of suffixes to consider at each allocation threshold, for each benchmark

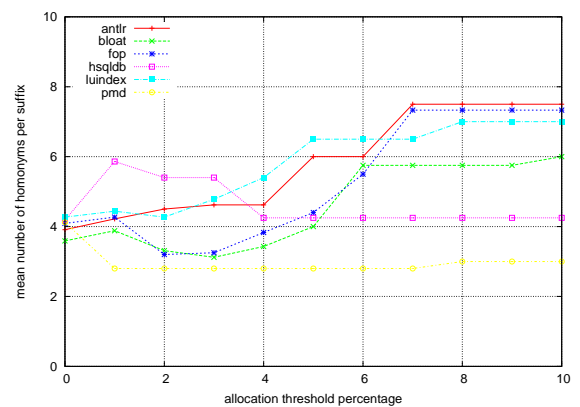


Figure 22. Mean number of homonyms per suffix at each allocation threshold, for each benchmark

for each benchmark, and how this number changes with the allocation threshold.

It is helpful to compare these figures against homonym counts for standard English text. We analyse the list of 400 common nouns in Ogden's Basic English [37] and find that the mean number of homonyms per noun is 5.74. The mean value for suffix names in the benchmarks is mostly lower than this, so it seems that homonymy is less of a problem in programming languages than in English natural language. (More investigation is required to confirm this.) Note in Figure 22 that the mean number of homonyms increases as the allocation threshold increases. This is mostly due to the suffix `String` being most prolific, and having polysemy count of 10 in WordNet. However these uses are related, therefore they are not true homonyms. In computer programs, `String` generally has a single meaning (character string). It has been postulated that in natural language, when a word is a homonym, one of the multiple meanings is much more likely than any of the others. This may also hold for programming languages.

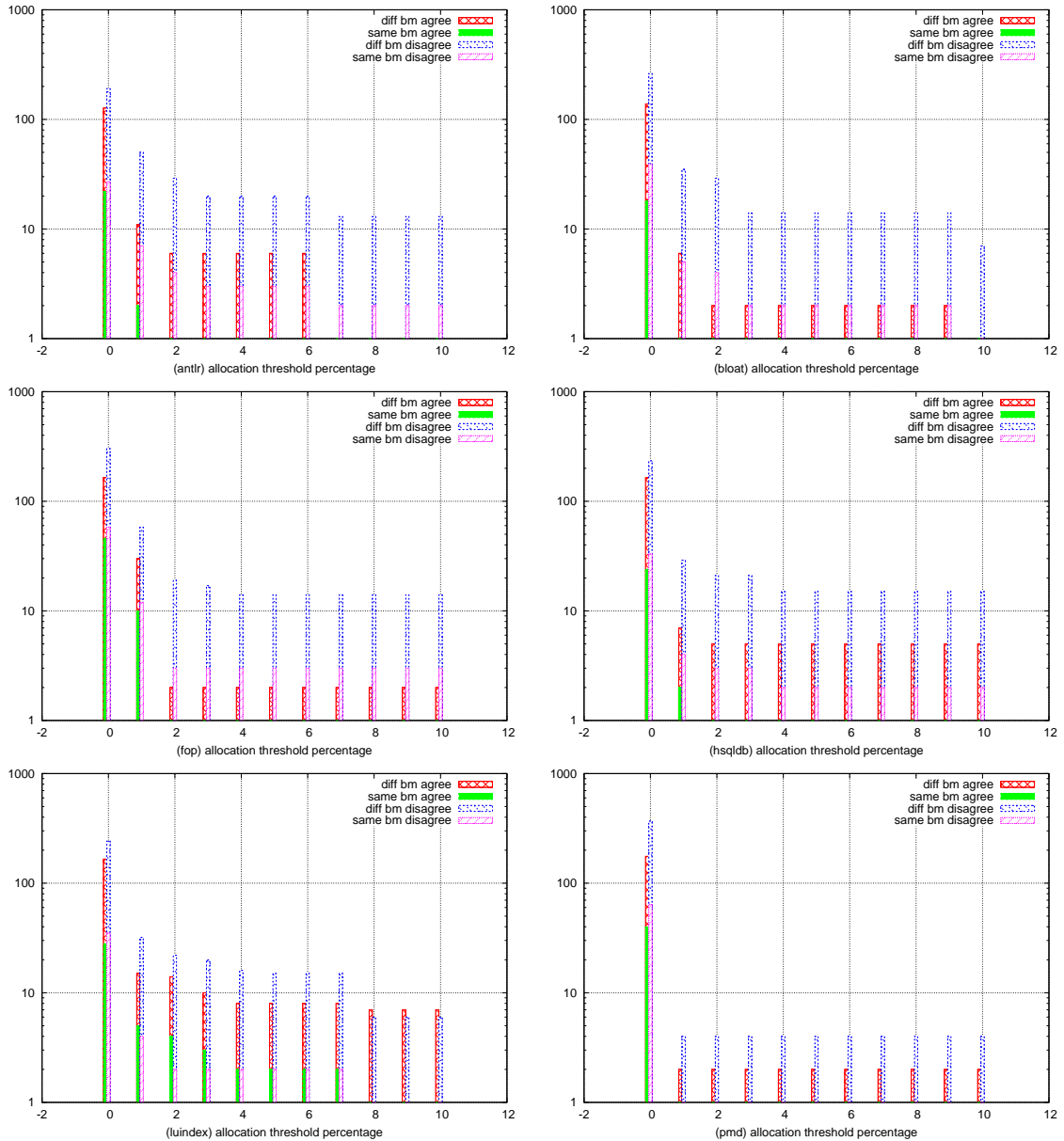


Figure 20. Level of agreement between synonyms in various DaCapo benchmarks

Implications for Optimization: The problem of homonymy is potentially much more serious than synonymy. Whereas synonymy simply results in *missed* optimization opportunities, homonymy can cause some objects to be incorrectly optimized and thus degrade performance. This scenario occurs if we see a word with one homonym meaning in the training phase, and with a different homonym meaning in the testing phase. The only sensible work-around for this problem is to use runtime feedback mechanisms to enable or disable suffix-based optimizations adaptively.

6. Related Work

6.1 Object Lifetime Prediction

The majority of existing object lifetime prediction schemes are site-based [26, 27, 19, 38, 16]. The scheme presented in this paper is type-based. We review some other type-based schemes [25, 39, 40] in detail below.

Huang et al [39] describe a dynamic type-based pretenuring scheme for Java programs implemented in Jikes RVM. For each object type, the VM records an *average survival ratio*, computed from the numbers of nursery allocated objects and promoted objects of that type. This ratio is updated at the end of each GC phase. When new objects are created,

the VM checks whether the average survival ratio for this type is above a threshold (heuristically set to 70%). If so, then the object is pretenured. They claim a maximum GC time reduction of 37%. However they only study a small set of benchmarks. They evaluate their scheme using the non-optimizing runtime compiler in Jikes RVM. This compilation inefficiency may inflate their performance figures.

Shuf et al [25] introduce the notion of *prolific types* to characterize those relatively few object types that account for a large proportion of heap allocation activity. Based on offline analysis of object lifetime traces for an extensive set of benchmarks, they propose the prolific hypothesis: objects of prolific type die younger than objects of non-prolific type. They present a *type-based garbage collection* system implemented for Java programs in Jikes RVM. They segregate objects into two different spaces. The P-space is for prolific objects, the N-space is for non-prolific objects. No objects are ever copied between spaces. The P-space is analogous to the nursery space in generational collectors, since it is collected more frequently than the N-space due to a higher allocation rate. Their aim is to improve GC performance by reducing both the need for inter-space write barriers and also the likelihood of short-lived objects polluting the space of long-lived objects. This type-based GC scheme gives an average reduction in GC time of 7%, compared to the standard generational setup.

Marion et al [40] have a hybrid approach that incorporates types and sites. Each allocation site belongs to a *source* type, which is the class containing the method in which the allocation occurs. Each allocation site creates an object with a *destination* type, which is the concrete type of the newly created object. They associate a set of general-purpose *micro-patterns* [41] with each class, based on its static properties. Thus they can match each allocation site with a set of micro-patterns for the source and destination types at that site. They profile a large number of standard Java benchmarks to build up an object lifetime knowledge base. Then they apply data mining to create a set of rules that associate allocation site micro-pattern sets with object lifetime predictions, which are used for pretenuring optimizations. They demonstrate performance gains between 6-77% in GC time over a standard generational copying collector in Jikes RVM.

The crucial advantage of our scheme is that we generalize object type names by processing the type name suffix only. We present offline analyses to show that this is a valid assumption. It enables our scheme to operate on a wider range of types, including those in previously unseen programs. In addition, simple type name string comparisons are cheaper to analyse at runtime than rules based on sets of micro-patterns. At present, we are only using the object lifetime predictions to optimize allocation of long-lived objects via pretenuring. The idea of retaining short-lived objects in nursery is most appealing. Shuf et al [25] show that optimizing the allocation of short-lived objects also leads to sig-

nificant performance improvements. Our type name suffix scheme should be an ideal framework for this optimization.

6.2 Automated Information Extraction from Identifiers

To the best of our knowledge, no existing GC algorithm uses semantic information encoded in human-generated programming language identifiers. As stated in Section 1, the systems community appears reluctant to take advantage of this fuzzy information.

The most active research areas for semantic information extraction are *software maintenance* and *reverse engineering*. These are large fields, so we only present a few representative examples.

Biggerstaff et al [42] show how source code identifier names can be used to build up a set of *program concepts* that encapsulate high-level design issues in a system. Their concept assignment process is grounded in artificial intelligence techniques. Once identified, the concepts are useful for program comprehension, documentation recovery, specification reverse engineering and refactoring.

Lawrie et al [43] analyse variable identifier names in large programs written in various imperative and object oriented languages. They aim to measure consistency of variable names (which is related to the problems of synonymy and homonymy outlined above). They also use the WordNet tool to aid semantic analysis. They also conclude that programming languages have a more limited use of vocabulary than natural languages.

Anquetil and Lethbridge [44] evaluate the relevance of Pascal record identifier names in a legacy telecoms application. They provide a framework to assess whether a program naming convention is reliable for use in program comprehension and reverse engineering tools.

7. Conclusions

7.1 Summary

This paper has demonstrated the importance of type name suffix information. Extraction of information from meaningful identifiers is generally neglected in systems optimization. We have shown that it can be used for object lifetime prediction within and across programs. We have presented extensive analysis to justify this claim, together with preliminary optimization implementations for both short-lived and long-lived objects.

7.2 Limitations

Slavish reliance on human-generated identifier names has limitations. There are often problems caused by inconsistencies in naming conventions and coding styles. Sneed [45] claims that many programmers use their girlfriends' names as identifiers in COBOL programs. We hope this convention is not carried over to Java programs. An additional complication is that Java uses Unicode identifiers, so any natural

language from Kurdish to Klingon is permissible. These issues may reduce the applicability of our new technique. Obfuscated code also presents a problem, since identifiers may be mangled. However, in all these cases type name identifiers for VM and library types should be preserved, so these could still be optimized.

7.3 Future Work

So far we have only optimized a conventional generational collector. We aim to evaluate the potential of object lifetime prediction in other age-based garbage collection schemes, such as oldest-first collection [46] for which the prediction of a range of object lifetimes will be beneficial.

Further work involves extracting information from other parts of type names, rather than just their suffixes. Most Java type names have the form (adjective)* (noun)+. So far all our analysis concentrates on the last noun. There is no sound basis for this, other than sheer intuition. It would be useful to discover whether other parts of name correlate with lifetime, or with other predictable properties. We feel that the first adjective of a type name may be helpful in some circumstances. This extended analysis may also be relevant for other programming languages, such as C++, which have less rigorous naming conventions than Java.

It would be possible to data-mine large online open-source code repositories for type name information. Extending this idea, it would be appealing to have online repositories of GC traces for pretenuring research. In addition, each local JVM should maintain a continuously updated, long-term database of object lifetime demographics to enable profile-guided optimization for generational GC.

We hope to investigate the use of name-based similarity for something other than object lifetime prediction. One possibility is fault prediction in large software projects. Micro-patterns [41] have already been used for this purpose [47]. We assume that type name suffixes should be equally, if not more, helpful. Another possibility is the prediction of transactional conflicts in concurrent programs.

References

- [1] Etzkorn, L.H., Davis, C.G.: Automatically identifying reusable OO legacy code. *IEEE Computer* **30**(10) (1997) 66–71
- [2] Etzkorn, L.H., Bowen, L.L., Davis, C.G.: An approach to program understanding by natural language understanding. *Natural Language Engineering* **5**(3) (1999) 219–236
- [3] Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The vocabulary problem in human-system communication. *CACM* **30**(11) (1987) 964–971
- [4] Deissenboeck, F., Pizka, M.: Concise and consistent naming. *Software Quality Journal* **14**(3) (2006) 261–282
- [5] Parnas, D.: Software aging. *ICSE* (1994) 279–287
- [6] Malpohl, G., Hunt, J., Tichy, W.: Renaming detection. *Automated Software Engineering* **10**(2) (2003) 183–202
- [7] Kim, S., Pan, K., Whitehead Jr, E.: When functions change their names: Automatic detection of origin relationships. In: *WCRE*. (2005) 143–152
- [8] Caprile, B., Tonella, P.: Nomen est omen: Analyzing the language of function identifiers. In: *WCRE*. (1999) 112–122
- [9] Jones, R., Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley (1996)
- [10] Blackburn, S., Jones, R., McKinley, K., Moss, J.: Beltway: getting around garbage collection gridlock. In: *PLDI*. (2002) 153–164
- [11] Jones, R., Ryder, C.: Garbage collection should be lifetime aware. In: *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*. (2006)
- [12] Jones, R.: Dynamic memory management: Challenges for today and tomorrow. In: *Proceedings of the International Lisp Conference*. (2007) 115–124
- [13] Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In: *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. (1984) 157–167
- [14] Lieberman, H., Hewitt, C.: A real-time garbage collector based on the lifetimes of objects. *CACM* **26**(6) (1983) 419–429
- [15] Baker, H.G.: Infant mortality and generational garbage collection. *ACM SIGPLAN Notices* **28**(4) (1993) 55–57
- [16] Cheng, P., Harper, R., Lee, P.: Generational stack collection and profile-driven pretenuring. In: *PLDI*. (1998) 162–173
- [17] Ungar, D.M., Jackson, F.: An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems* **14**(1) (1992) 1–27
- [18] Inoue, H., Stefanovic, D., Forrest, S.: On the prediction of Java object lifetimes. *IEEE Transactions on Computers* **55**(7) (2006) 880–892
- [19] Jump, M., Blackburn, S.M., McKinley, K.S.: Dynamic object sampling for pretenuring. In: *ISMM*. (2004) 152–162
- [20] Singer, J., Brown, G., Luján, M., Watson, I.: Towards intelligent analysis techniques for object pretenuring. In: *Proceedings of the 5th International Conference on Principles and Practice of Programming in Java*. (September 2007) 203–208
- [21] Sun Microsystems: Code conventions for the Java programming language (Sep 1999) <http://java.sun.com/docs/codeconv>.
- [22] Alpern, B., et al.: The Jalapeño virtual machine. *IBM System Journal* **39**(1) (Feb 2000)
- [23] Alpern, B., et al.: The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal* **44**(2) (Feb 2005) 1–19
- [24] Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA*. (2006) 169–190

- [25] Shuf, Y., Gupta, M., Bordawekar, R., Singh, J.P.: Exploiting prolific types for memory management and optimizations. In: *POPL*. (2002) 295–306
- [26] Blackburn, S.M., Singhai, S., Hertz, M., McKinley, K.S., Moss, J.E.B.: Pretenuring for Java. In: *OOPSLA*. (2001) 342–352
- [27] Blackburn, S.M., Hertz, M., McKinley, K.S., Moss, J.E.B., Yang, T.: Profile-based pretenuring. *ACM TOPLAS* **29**(1) (2007) 1–57
- [28] : Memory management in the Java HotSpot virtual machine (Apr 2006) http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf.
- [29] Blackburn, S., Cheng, P., McKinley, K.: Oil and water? high performance garbage collection in java with mmtk. In: *Proceedings of the 26th International Conference on Software Engineering*. (2004) 137–146
- [30] Blackburn, S.M., Cheng, P., McKinley, K.S.: Myths and realities: the performance impact of garbage collection. *SIGMETRICS Performance Evaluation Review* **32**(1) (2004) 25–36
- [31] Blackburn, S.M., Hosking, A.L.: Barriers: friend or foe? In: *Proceedings of the 4th international symposium on Memory management*. (2004) 143–151
- [32] Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*. (2007) 57–76
- [33] Xian, F., Srisa-an, W., Jiang, H.: Microphase: an approach to proactively invoking garbage collection for improved performance. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*. (2007) 77–96
- [34] Barrett, D.A., Zorn, B.G.: Using lifetime predictors to improve memory allocation performance. In: *PLDI*. (1993) 187–196
- [35] Gu, D., Verbrugge, C., Gagnon, E.M.: Relative factors in performance analysis of Java virtual machines. In: *Proceedings of the 2nd international conference on Virtual execution environments*. (2006) 111–121
- [36] Miller, G.A.: Wordnet: a lexical database for english. *Communications of the ACM* **38**(11) (Nov 1995) 39–41
- [37] Ogden, C.K.: Ogden’s basic english <http://ogden.basic-english.org/>.
- [38] Harris, T.L.: Dynamic adaptive pre-tenuring. In: *ISMM*. (2000) 127–136
- [39] Huang, W., Srisa-an, W., Chang, J.M.: Dynamic pretenuring schemes for generational garbage collection. In: *IEEE International Symposium on Performance Analysis of Systems and Software*. (2004) 133–140
- [40] Marion, S., Jones, R., Ryder, C.: Decrypting the Java gene pool: Predicting objects’ lifetimes with micro-patterns. In: *Proceedings of the International Symposium on Memory Management*. (Oct 2007) (to appear)
- [41] Gil, J.Y., Maman, I.: Micro patterns in Java code. In: *OOPSLA*. (2005) 97–116
- [42] Biggerstaff, T., Mitbander, B., Webster, D.: Program understanding and the concept assignment problem. *CACM* **37**(5) (1994) 72–82
- [43] Lawrie, D., Feild, H., Binkley, D.: Syntactic identifier conciseness and consistency. In: *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. (2006) 139–148
- [44] Anquetil, N., Lethbridge, T.: Assessing the relevance of identifier names in a legacy software system. *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research* (1998)
- [45] Sneed, H.: Object-oriented COBOL recycling. In: *WCRE*. (1996) 169–178
- [46] Stefanović, D., McKinley, K.S., Moss, J.E.B.: Age-based garbage collection. In: *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. (1999) 370–381
- [47] Kim, S., Pan, K., Whitehead Jr, E.: Micro pattern evolution. (2006) 40–46