

Predictive Garbage Collection

Andrej Bogdanov Kenji Obata

December 17, 2002

1 Introduction

With increasing amounts of software being written for managed execution environments, such as Sun's Java Virtual Machine or Microsoft's .NET platform, achieving high performance in implementations of these runtime environments is of growing importance. A common feature of such systems is the use of a garbage collector for reclaiming dynamically allocated memory. Standard garbage collection algorithms, such as those used in the above runtimes, identify dead objects by traversing the objects in the heap and removing objects not touched by the traversal. This process consumes system resources proportional to the number of pointers in the traversed region of the heap.

In this project, we consider a radically different approach to garbage collection, which we call *predictive garbage collection*. We attempt to design garbage identification algorithms which can accurately determine object lifetimes by studying the statistical properties of the application under consideration. This approach allows a corresponding garbage collection algorithm to reclaim memory without the costs and inefficiencies associated with heap traversal. Our approach can also be used to enhance the efficiency of traditional conservative garbage collection algorithms.

In traditional garbage collection, an object is considered "garbage" as soon as it becomes inaccessible from the stack by a sequence of pointer references. This is a pessimistic view, as the last access time to an object may occur well before the object becomes inaccessible. Instead, we propose the following semantic view of garbage collection: Say an object is *garbage* at time t if no accesses to this object occur after time t . The operational value of our definition is that it allows reclaiming the heap space occupied by an object as soon as we can guarantee that no accesses to this object will occur in the future. Reference analysis, which is used by traditional garbage collectors, is merely one of several possible approaches towards obtaining this property.

In the spirit of our view of garbage collection, we define the *actual lifetime* L_x^* of object x to be the length of the time interval between the allocation of object x and the last access to object x . Our investigation indicates that for most of the objects allocated on the heap, their actual lifetimes can be predicted accurately by analyzing their access patterns. The heart of the predictive garbage collection technique is an algorithm that, for each object x , determines a *predicted lifetime* L_x representing an estimate of its actual lifetime L_x^* . A traditional garbage collector provides the conservative guarantee that for all objects x , $L_x^* \leq L_x$. In contrast, a predictive garbage collector makes no such claim; it allows for the possibility that an object is garbage collected *before* the end of its lifecycle. Even though

this may seem unusual in the context of garbage collection, such pre-emptive strategies have been successfully exploited in other areas of systems design, including virtual memory and optimistic concurrency control [3].

Analysis framework. The framework we used for our analysis and evaluation of predictive garbage collection consists of three components: An injection tool, an analyzer and a simulator. The injection tool takes an arbitrary CLR¹ source executable and outputs a *profiled executable* which is functionally equivalent to the original, but logs all object and method accesses that may be pertinent to garbage collection. The analyzer takes the log produced by a source executable and outputs a model of object lifetime predictions based on the observed data. The simulator evaluates the prediction model against a *cost model* that quantifies the discrepancy between the actual and predicted lifetimes.

Predicting object lifetimes. Our algorithm for determining the predicted lifetime of objects was derived by evaluating several models of object behavior throughout a variety of applications: A program development environment, two games and a web server. Throughout these applications, the principal factor that determined the lifetime of an object turned out to be the object’s type. Most of the types were found to consist of objects that exhibit a roughly constant lifetime. For certain types that exhibited large lifetime variances across objects, it was helpful to classify the objects by *allocation context*, namely the point in the code at which the object was allocated. Roughly 80% of the objects that we observed exhibited almost constant lifetime when conditioned on either type or allocation context.

Among the types whose objects did not have almost constant lifetimes, a different model turned out to be widely applicable. These are types for which the ratio

$$\frac{\text{number of live objects of type } T \text{ at time } t}{\text{total number of objects of type } T}$$

remains small throughout the execution of the application. Even though the lifetime of such objects is difficult to estimate explicitly, we found out that the liveness of these objects is surprisingly well modeled by the “least recently used” ordering. In conclusion, we expect an LRU-based garbage collection strategy to perform well on this class of objects.

2 Previous Work

Garbage collection is a very old and thoroughly investigated subject, and we do not attempt to summarize the enormous research body on the topic here. For an excellent survey, we refer to [6]. We briefly mention a few major ideas in garbage collection as they are applicable to our results.

Lieberman and Hewitt ([4] cited from Wilson’s survey), and no doubt many others, observed that most allocated objects have a very short lifetime, while the remainder are likely to have much longer lifetimes. This suggests that it would be profitable to partition the heap according to the likelihood that the contained objects will be destroyed in a particular collection pass. Limiting collection to regions of the heap which are likely to contain a large number of dead objects allows memory to be reclaimed without wasting resources

¹Common Language Runtime, the .NET framework virtual execution environment defined in ECMA publication 335, available at www.ecma.ch.

traversing a large number of objects which will not be garbage. Partial heap traversal can be implemented by using a write barrier to track pointers which cross between partitions; the set of pointers with endpoints in a given partition form the base set for a heap traversal and collection can proceed as before.

A very effective and popular version of this approach, proposed in [4], is *generational garbage collection*. A generational garbage collector partitions the heap according to the number of collection passes a given object has survived. New objects are allocated in the *youngest* heap; as collections occur, objects either die or are promoted into an *older* heap. A generational garbage collector is described by a heap partitioning scheme, an object promotion policy, and a collection scheduling policy. For example, Microsoft's commercial implementation of the CLR [2] uses three generations, labeled 0, 1, and 2. Objects are always allocated in generation 0; an object in generation i which survives a collection is promoted to generation $\min(i+1, 2)$; a generation 0 collection is triggered whenever an allocation request cannot be satisfied within generation 0, and higher generations are collected according to a more complex policy incorporating various factors of system state. (In practice, the CLR combines generational partitioning with other techniques; for example, a separate heap is maintained for very large objects and per-processor memory arenas are used to minimize contention on SMPs).

Another way of interpreting generational collection is as a heuristic for estimating whether a set of objects is alive; the youngest objects are considered likely to be dead, while objects which have survived previous collection passes are considered likely to be alive, at least until the next collection in the corresponding generation. The partitioning and scheduling policies then attempt to focus collection resources on sets of objects likely to be dead according to this heuristic. Clearly, other choices are possible, and indeed many others have been explored. For example, Stefanovic, McKinley, and Moss [5] consider the opposite approach, which they call age-based collection. In this approach, collection efforts are focused on the oldest objects, with the motivation that the youngest objects may not yet have had enough time to be used and die while the oldest objects are more likely to have spanned their lifetime. We mention this *lifetime heuristic* interpretation of current garbage collection strategies to emphasize that a probabilistic object lifetime estimator can be used in concert with current collection algorithms and implementations to improve collection efficiency.

3 Applications of Predictive Garbage Collection

We envision our predictive garbage collector to be used either as a standalone tool, or in conjunction with a traditional garbage collector. In this section we briefly describe these two applications. While detailed consideration of these proposals is beyond the scope of our project, we believe these would be very fruitful applications of our prediction algorithms.

A standalone predictive garbage collector. Given a sufficiently accurate algorithm for predicting object lifetimes, one can envision an extremely efficient garbage collector which operates simply by expiring objects at whatever time the predictor believes the object will be dead with some appropriately high probability. The primary difficulty with such an approach is that any probabilistic garbage identification strategy will, at least occasionally,

make an error and prematurely expire a live object. What should one do when this occurs?

We suggest two potential solutions to this problem. Our first proposal exploits the fact that secondary storage is extremely plentiful compared to main memory, albeit slow to access. Upon object expiration, a probabilistic collector can, instead of simply overwriting the memory associated with the object, write the object contents to a log file on disk. Then, in the event that a program references an object which has been expired, the runtime can reconstruct the object from the contents on disk. This will certainly be expensive but, assuming that the predictor only rarely errs, will not be a significant cost. The cost model that we used for evaluating our garbage collection algorithm (see Section 5) allows for this by charging a high constant cost for premature expirations. The log contents on disk can be deleted upon program termination or, if necessary, by a standard garbage collection traversal (which will be mostly confined to in-memory objects). Alternatively, one can interpret this as an object-oriented virtual memory system, which uses the predictor to efficiently emulate an infinite virtual address space.

Our second suggestion is simpler but modifies the semantics of the runtime. Upon object expiration, the probabilistic collector simply marks the associated memory as freed and allows it to be used to service other allocation requests. In the event that an expired address is referenced, the runtime generates an “object expired” exception, causing the execution stack to unwind according to the structured exception handling semantics of the CLR. While this is a quite disruptive method of dealing with errors, it is again acceptable if the event is rare and the application is designed to recovery gracefully from runtime-generated exceptions. Such a design is typical of server application software. A version of this approach is, in a sense, used in Microsoft’s Internet Information Server [1]; IIS can monitor the processes executing a user’s (typically buggy, memory-leaking) web application and simply kill and restart the application when the memory usage exceeds a certain threshold. This is roughly equivalent to an allocator which does not perform any collections and simply generates an exception when the heap is expired. When coupled with a probabilistic lifetime predictor, such an allocator can substantially extend the expected time before the heap is exhausted.

An improved traditional garbage collector. As we noted in the introduction, traditional garbage collection algorithms can be viewed as containing implicit notions of object lifetime prediction. It seems, then, that a reasonable way to go about constructing a collector is to begin with a lifetime predictor and optimize the collection policy around its predictions. For example, if one knows at allocation time that a given object is likely to be dead within a short time (as we observe to be the case with the large majority of objects), it is sensible to place the object in a short-lived partition of the heap. When these short-lived objects are all dead, the traversal collector can quickly reclaim the space, since traversal time is linear in the number of live edges in the subgraph to be traversed. As a bonus, the dead objects create large contiguous regions of free memory, minimizing the number of compaction copies required for heap defragmentation. Standard generational collectors can be interpreted as implementing this strategy with a very crude lifetime approximation (every object is likely to be dead by the time generation 0 is exhausted). For an arbitrary heap partition, a predictor can be used to estimate the amount of memory which will be released as the result of a traversal. A traditional collector can then employ this estimate when determining whether a traversal of a higher generation would be profitable.

4 Code Injection

In order to investigate the statistical properties of object births and lifetimes in real-world software applications, we needed a tool which would allow us to profile pertinent events in existing executable programs. To this end, we built a *code injection tool*, which we call simply Inject. While we used Inject to study garbage collection, this tool clearly has a wide range of potential applications. For this reason, as well as the considerable amount of effort which was expended to realize the tool, we feel that Inject is an independently interesting contribution of our project. This section describes the design and implementation of Inject and discusses some of the issues we encountered in its development.

Inject overview. Inject is a CLR console application consisting of approximately 10,000 lines of (well documented) C# source code. Inject takes as its input an arbitrary CLR source executable, which may be a stand-alone executable or a shared library, and a profiling executable. The profiling executable is expected to expose methods named `OnNew`, `OnTouch`, and `OnCall` with certain pre-defined signatures. Given these binaries, Inject produces a new profiled executable which is functionally equivalent to the source executable, but has calls to the profiling event handlers compiled into the appropriate locations along with profiling metadata, including context and type information. To facilitate analysis of profiling output, Inject also produces an XML symbol table which contains mappings from various internal identifiers to class and method names extracted by examining the source executable metadata.

Inject is used as follows:

```
inject source.exe prof.dll output.exe symbols.xml [/nocalls]
```

where `source.exe` is the source executable, `prof.dll` is the profiling code, `output.exe` is the output executable, and `symbols.xml` is the symbol table. Because method invocation tracking tremendously increases the size of profiling logs and this information is not necessarily desired, we include an optional `/nocalls` flag which suppresses injection of the `OnCall` handler.

Profiling interface. The profiling executable is expected to define the following methods, matching the given names and signatures. For each method, we include as an example the implementation actually used in our project.

The `OnNew` method is invoked immediately following allocation of an object. This must come after the allocation since we pass to the profiling handler the object which was just allocated:

```
public static void OnNew (object o, int iContext, int iType)
{
    lock (_log)
    {
        // assign and store the object ID
        _hashObjects[o] = _ulObjectId;

        // log it
        _log.OnNew ((int) _ulObjectId, iContext, iType);
    }
}
```

```

        // next ID
        _ulObjectId++;
    }
}

```

The `o` argument is the object which was just allocated. Our profiler assigns and stores a unique 32-bit identifier for the object instance. The `iContext` argument is a unique integer assigned per allocation site in the source executable; this can be resolved to a class/method name by looking up the corresponding `OnNewContext` in the symbol table. The `iType` argument is an integer identifying the type of the object being allocated; a mapping to class names is also provided in the symbol table.

The CLR allows for complex value types, i.e. pass-by-value classes. While such objects are instantiated via the same CIL instruction as reference classes, they of course are not allocated in the heap. Since we do not want to track these (and could not, since attempting to pass such a type to the handler would cause an IL verification exception), Inject detects and suppresses profiling of value type classes.

Note that Inject does not introduce any synchronization into the output binary around the profiling calls. To allow profiling of multithreaded applications, the injected code must use appropriate synchronization (in this example, we use the C# `lock` operator).

The `OnTouch` method is invoked immediately prior to an access of a field in an object:

```

public static void OnTouch (object o, int iContext)
{
    lock (_log)
    {
        if (_hashObjects[o] != null)
        {
            // lookup ID
            UInt32 ulObjectId = (UInt32) _hashObjects[o];

            // log it
            _log.OnTouch ((int) ulObjectId, iContext);
        }
    }
}

```

The object in which a field was referenced is passed, along with the identifier of the reference site.

The `OnCall` method is invoked immediately prior to a method invocation:

```

public static void OnCall (int iContext, int iMethod)
{
    lock (_log)
    {
        // log it
    }
}

```

```

    _log.OnCall (iContext, iMethod);
}
}

```

As before, `iContext` identifies the method invocation site. The `iMethod` parameter is an integer identifying the invoked class/method pair. Note that the object on which the method is invoked is not passed to the handler; we do not treat a method invocation itself as a touch event. An object is considered touched only if a method invocation actually causes a field of the object to be accessed, in which case the `OnTouch` handler is invoked.

Injection alternatives. There are several ways one might go about gathering allocation profiling information from a CLR executable. We briefly discuss some of these alternatives.

In fact, our original intention in this work was to extend the shared source implementation of the CLR, so-called Rotor², to include gathering of profiling information directly within the runtime allocator implementation. We did not end up using this approach: As currently defined, the Rotor allocator does not handle any type or context information. Rotor is a JIT-only implementation (it does not provide an implementation of an IL interpreter) and there are many points of interaction between the JITed code and the allocator. Given the absence of detailed documentation of these interactions³ and the potential difficulties involved, we decided this approach was too risky given our timeframe.

We also considered using the facilities provided in the `System.Reflection` namespace of the framework class libraries. These classes provide a fairly powerful set of routines for reading and writing CLR executable files. Unfortunately, as we discovered, they are not powerful enough for our purposes; the read capability is limited to viewing assembly metadata (not actual IL content), and the write capability is tailored to creating new assemblies, not rewriting existing ones. Thus we ended up, in effect, implementing our own reflection library with the functionality necessary for our application.

5 Modeling and analysis of object lifetimes

In this section we describe the evaluation model for our garbage collection simulator and the algorithms we used for predicting object lifetimes. We also present experimental evidence that supports the validity of our modeling approach.

The cost model. The cost model for object expiration is described by the following function: For each object x ,

$$\text{cost}(x) = \begin{cases} c(L_x - L_x^*) & \text{if } L_x^* \leq L_x \\ K & \text{if } L_x^* > L_x \end{cases}.$$

Note that if the prediction is perfect, $L_x = L_x^*$ and the cost of x is zero. The model charges a small cost c for each time unit that the object spends in memory after it becomes garbage, and a large but constant cost K for expiring the object too early.

Almost constant lifetime objects. We call a collection of objects *almost constant lifetime* if the variance in lifetimes among these objects is a small fraction of the mean. Our

² Available at msdn.microsoft.com/downloads

³ A comprehensive book on Rotor is scheduled for publication by O'Reilly next year

experimental analysis revealed that more than 80% of the encountered objects, when classified according to type or allocation context, fall into this category. For example, these include objects that programs use locally for “scratch work.”

We detect almost constant lifetime objects by performing a statistical test on a collection of candidate objects with measured lifetimes L_1^*, \dots, L_N^* . The statistical test consists of the following:

1. Compute the sample mean $\hat{\mu} = \frac{1}{N} \sum_{i=1}^N L_i^*$ and sample variance $\hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^N L_i^{*2} - \hat{\mu}^2$.
2. If $N > N_{\min}$ and $\hat{\sigma} \leq a + b\hat{\mu}$, the test accepts, otherwise it rejects.

Here, N_{\min} , a and b are parameters of the test. The threshold N_{\min} ensures that the number of data points is sufficient to guarantee a statistically significant result. The purpose of the additive factor a is to ensure that objects with a small mean lifetime are not discriminated against. The multiplicative factor b determines the required concentration around the mean. In our evaluation, we used the following ranges of parameter values: $N_{\min} \in [5, 10]$, $a = 100\text{ms}$ and $b \in [0.3, 0.6]$.

We model almost constant lifetimes as independent samples of a Gaussian distribution with mean $\hat{\mu}$ and variance $\hat{\sigma}^2$. We are interested in computing a predicted lifetime L , depending on the distribution parameters $\hat{\mu}$ and $\hat{\sigma}$, that is, in some sense, optimal for this distribution. Intuitively, we would like to ensure that, for most objects x , $L \geq L_x^*$, but that the differences $L_x^* - L$ are not too large. We can formalize this intuition as follows: Define L_{opt} to be the value that minimizes the *expected* cost of a Gaussian sample with mean $\hat{\mu}$ and variance $\hat{\sigma}^2$ with respect to our cost model. Even though L_{opt} cannot be computed explicitly, we will show that it satisfies the equation

$$K e^{-(L_{\text{opt}} - \hat{\mu})^2 / 2\hat{\sigma}^2} = c \int_{-\infty}^{L_{\text{opt}} - \hat{\mu}} e^{-t^2 / 2\hat{\sigma}^2} dt.$$

This equation can be solved numerically for L_{opt} using standard software, such as Matlab or Maple. Figure 1 shows the value of L_{opt} for two settings of the cost model parameters.

To simplify notation, let $l = L - \hat{\mu}$, where L is the predicted lifetime for a collection of almost constant lifetime objects. Similarly define $l_x^* = L_x^* - \hat{\mu}$ for the actual lifetimes. Then $E[l] = E[L] - E[\hat{\mu}] = E[L] - \text{const}$, so it will be sufficient to compute the value of l that achieves the optimal expected cost. The expected cost of an object from the collection is given by the formula

$$\begin{aligned} E[\text{cost}(x)] &= E[\text{cost}(x) | L_x^* \leq L] \Pr[L_x^* \leq L] + E[\text{cost}(x) | L_x^* > L] \Pr[L_x^* > L] \\ &= E[c(L - L_x^*) | L_x^* \leq L] \Pr[L_x^* \leq L] + E[K | L_x^* > L] \Pr[L_x^* > L] \\ &= c(L - E[L_x^* | L_x^* \leq L]) \Pr[L_x^* \leq L] + K(1 - \Pr[L_x^* \leq L]) \\ &= c(l - E[l_x^* | l_x^* \leq l]) \Pr[l_x^* \leq l] + K(1 - \Pr[l_x^* \leq l]). \end{aligned}$$

The quantity $\Pr[L_x^* \leq L]$ is given by the probability mass function of the Gaussian distribution

$$\Pr[l_x^* \leq l] = \frac{1}{\sqrt{2\pi\hat{\sigma}^2}} \int_{-\infty}^l e^{-t^2 / 2\hat{\sigma}^2} dt.$$

To compute the conditional expectation of l_x^* , let $\nu_l(t)$ denote the Gaussian measure with mean zero and variance $\hat{\sigma}^2$ conditioned on $t \leq l$. Then

$$\mathbb{E}[l_x^* | l_x^* \leq l] = \int_{-\infty}^l t d\nu_l(t) = \frac{\int_{-\infty}^l t d\nu_\infty(t)}{\int_{-\infty}^l d\nu_\infty(t)} = \frac{\frac{1}{\sqrt{2\pi\hat{\sigma}^2}} \int_{-\infty}^l t e^{-t^2/2\hat{\sigma}^2} dt}{\frac{1}{\sqrt{2\pi\hat{\sigma}^2}} \int_{-\infty}^l e^{-t^2/2\hat{\sigma}^2} dt} = \frac{-\hat{\sigma}^2 e^{-l^2/2\hat{\sigma}^2}}{\int_{-\infty}^l e^{-t^2/2\hat{\sigma}^2} dt}.$$

Substituting and taking derivatives with respect to l , we obtain

$$\frac{d\mathbb{E}[\text{cost}(x)]}{dl} = \frac{c}{\sqrt{2\pi\hat{\sigma}^2}} \int_{-\infty}^l e^{-t^2/2\hat{\sigma}^2} dt - \frac{K}{\sqrt{2\pi\hat{\sigma}^2}} e^{-l^2/2\hat{\sigma}^2}.$$

Setting the left hand side to zero, we find that this equation has a single positive solution which corresponds to the optimal value $l_{\text{opt}} = L_{\text{opt}} - \hat{\mu}$.

It is instructive to compare the optimal lifetime prediction computed by the Gaussian model to the *empirical optimal lifetime*, i.e., the value L^* that minimizes $\sum_{i=1}^N \text{cost}(i)$ for the observed samples. Figure 1 shows the relationship between the predicted and empirical optimal lifetimes for two settings of the cost model parameters.

Figure 2 shows the frequency of premature object expirations across three runs of a sample application. We observe that in all cases, the fraction of prematurely expired objects is well under 1%.

Bounded live ratio objects. We say that a collection of objects satisfies the *bounded live ratio criterion* if the following is true at all times t of the execution:

$$\frac{\text{number of live objects of type } T \text{ at time } t}{\text{total number of objects of type } T} \leq \Delta.$$

Here, $0 \leq \Delta \leq 1$ is a parameter of the test. Bounded live ratio objects are intended to model certain user interface elements in an application, such as open documents in an editor. The user is expected to interact with only a few such documents at a time, even though the total number of opened documents throughout the lifetime of the application may be large. We found this behavior to be common in computer games as well.

Let N denote the total number of objects of type T . Our garbage collection algorithm for bounded live ratio objects does the following: When the number of objects of type T in the heap exceeds ΔN , expire the *least recently used* object of type T from the heap. Somewhat surprisingly, this strategy worked perfectly for all objects for all $\Delta < 0.6$: No live object was ever expired from the heap.

6 Future work

The injection tool. The current code injection tool has some omissions which prevent it from profiling certain types of applications or gathering certain profiling information. While we believe that any of these issues could be fixed in a matter of days, we had already expended too much time on the injection tool and had to stop work on it in order to finish the remainder of the project within the semester. Given a little more time, Inject could be modified to deal with these cases.

Figure 1: Predicted optimal lifetimes vs. empirical optimal lifetimes for almost constant lifetime objects. The curve indicates the optimal lifetime predicted by the Gaussian model over $\hat{\sigma} \in [1, 10000]$. The points show the empirical optimal lifetimes for observed collections of objects that pass the almost constant lifetime test with $N_{\min} = 5, a = 100, b \in \{0.3, 0.6\}$. The cost model parameters are (a) $K = 1000, c = 0.1$; (b) $K = 10000, c = 0.01$.

Run	b	Premature Expirations
1	0.3	0.15%
1	0.6	0.18%
2	0.3	0.61%
2	0.6	0.61%
3	0.3	0.28%
3	0.6	0.29%

Figure 2: Frequency of premature expirations. The table shows statistics for three experimental runs of an application. The cost model parameters are $K = 1000, c = 0.1$. The almost constant lifetime test parameters are $N_{\min} = 5, a = 100, b \in \{0.3, 0.6\}$.

Strongly named assemblies. The CLR has a notion of a strongly named assembly, which is simply a mechanism for referring to shared libraries not just by name but by a combination of name, version, publisher, and digital signature. Unlike their typical usage today, digital signatures in the CLR are not restricted to code intended for deployment on the web. In fact, any assembly registered as a shared component is required to be strongly named.

The code injection process, of course, modifies the contents of an assembly and therefore invalidates its digital signature. If an executable attempts to load an injected strongly named assembly, the runtime throws an invalid signature exception. Unfortunately, more complex applications frequently use shared components, and therefore strongly named assemblies, and so cannot currently be profiled using Inject. To remedy this problem, one would simply need a tool to walk through the dependency tree of assemblies, update each injected assembly's digital signature, and update the expected signature in each referencing assembly. This is not an issue if one implements the profiling within the runtime itself.

Array operators. The CLR uses a distinct set of operators to handle allocation and referencing of arrays. These operators are not currently profiled by Inject, although references to individual objects within arrays are tracked as normal. However, arrays are always allocated within the heap, even if the constituent elements are value types, so for complete correctness one would need to track the array allocations themselves.

Mixed managed and native code. The CLR allows managed IL and native machine instructions to be mixed and interoperate within a single assembly. The Inject tool does not currently support these mixed assemblies. This is another scenario which arises in more sophisticated applications, which one would like to be able to profile. Of course, our injection and profiling code would only be able to process the managed components of a mixed assembly; the interiors of unmanaged routines and heaps are opaque to the injector.

Modeling and analysis. Our initial simulations of predictive garbage collection, as described in Section 5, lead us to believe that a small number of object lifetime models are sufficient to describe the behavior of most objects occurring in standard applications. Our study was limited to about a dozen sample runs over several representative applications. These runs were carried out with the explicit intent of collecting data for object lifetime analysis, but the observations may not necessarily generalize to typical user scenarios. To obtain a better understanding of object behavior, it would be interesting to deploy injected code among actual users, and observe object access patterns across longer runs of an application.

Another limitation of our models is the implicit assumption that the behavior of objects does not change much throughout the lifetime of the application. However, in many applications where efficient garbage collection is especially important (e.g., servers) object access patterns may change dramatically over time. Extension of our approach to dynamic, real-time environments should provide a rich area for future research.

References

- [1] Internet Information Server. Microsoft Corporation, 2002. Available at <http://www.microsoft.com/iis>.

- [2] Microsoft .NET Framework. Microsoft Corporation, 2002. Available at <http://www.microsoft.com/net>.
- [3] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
- [4] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.
- [5] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Conference on Object-Oriented*, pages 370–381, 1999.
- [6] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.