Characterisation of object behaviour in Standard ML of New Jersey*

Darko Stefanović J. Eliot B. Moss
Department of Computer Science
University of Massachussets

Abstract

We describe a method of measuring lifetime characteristics of heap objects, and discuss ways in which such quantitative object behaviour measurements can help improve language implementations, especially garbage collection performance. For Standard ML of New Jersey, we find that certain primary aspects of object behaviour are qualitatively the same across benchmark programs, in particular the rapid object decay. We show that the heap-only allocation implementation model is the cause of this similarity. We confirm the weak generational hypothesis for SML/NJ and discuss garbage collector configuration tuning. Our approach is to obtain object statistics directly from program execution, rather than simulation, for reasons of simplicity and speed. Towards this end, we exploit the flexibility of the garbage collector toolkit as a measurement tool. Careful numerical analysis of the acquired data is necessary to arrive at relevant object lifetime measures. This study fills a gap in quantitative knowledge of the workings of heap-based compilers and their run-time systems, and should be useful to functional language implementors.

1 Introduction

Having fine-grained data on heap object lifetime behaviour is a requirement for the analysis of garbage collection algorithms performance and for optimal heap configuration in parametrisable collectors. Previous research has used object statistics, but those were often too coarse or incomplete. Taking advantage of increased processing power available today, we have developed a framework for finer-grained volumetric heap object analysis.

Applying our methodology to Standard ML of New Jersey, we characterised quantitatively the behaviour of several classes of allocated objects. The results indicate that under the current heap-based compilation model the allocation region should be kept at a fixed size of around one megabyte, but otherwise justify Appel's simple garbage collection scheme [1].

The advantages of using type-safe languages with a well-defined formal semantics, such as the mostly functional language Standard ML, are well recognised. The one disadvantage often perceived as an insuperable obstacle is the inefficiency of programs written in higher-order functional languages. More precisely, the implementation techniques currently available for these languages lag behind those for traditional imperative languages, such as C. Sometimes, programs performing the same task, employing the same algorithm, and even similarly coded (when this is possible) nevertheless exhibit significantly better performance when written in a traditional language.

Apart from suboptimal code generation, an important factor limiting the overall speed of SML programs is memory management and memory subsystem performance. The SML/NJ system using its standard garbage collector (generational garbage collector with two dynamically sized generations) is well-known for its bad paging behaviour when running on platforms with insufficient memory (where sufficient memory may be a large amount by conventional language standards). Even when there is enough physical memory to avoid paging entirely, the pauses caused by "major collections" are intolerably long [9, 20].

Cache performance of SML/NJ was also suspected to be bad. It was speculated that 40% of execution time was spent waiting for main memory access, a 66% overhead. Recent work [11] has shown that this is not altogether true, at least for some current architectures and memory subsystem organisations. In particular, the cache performance on the DECStation 5000/200 is reasonably good across all benchmarks reported, at a 17% overhead. Although object behaviour has a bearing on cache performance, cache simulation studies are beyond the scope of this paper.

Intensive allocation characterises most implementations of functional programming languages. The allocation consists of objects visible to the programmer and of function closure objects (corresponding to activation records in the parlance of block-structured languages). The latter may in fact dominate due to the small size of individual functions and high frequency of calls. Allocation (of function closure objects) on the stack is often perceived as giving deallocation for free, the only cost being the adjustment of a stack frame pointer.[1] Furthermore, the semantics of the language may preclude imposing a stack discipline for closures (the *upward funarg* problem), so that at least some must be allocated on the heap. Allocation on the heap carries a higher price of deallocation, usually garbage collection. SML/NJ avoids using the stack entirely, and allocates all closure records on the heap. Whether this kind of strategy is better or worse than stack-based implementations remains an open question [4]. As a result, the allocation rate is greatly increased (Section 3.2), and so is the burden on the garbage collector. For example, running the standard collector accounts for 5 – 24% (10% on the average) of total execution cost in the SML/NJ system [25]. Consequently, the importance of having a good garbage collector

*This work is supported by National Science Foundation grant CCR-9211272. The authors can be reached at Internet addresses {stefanov,moss}@cs.umass.edu.

To appear in the Proceedings of the 1994 ACM Conference on Lisp and Functional Programming.

[1] There is a hidden cost of processing the stack when the heap is collected, but this need not be great.

becomes even greater.

1.1 Object dynamics

The effectiveness of generational garbage collectors hinges on the assumption, termed the *generational hypothesis*, that young objects die more quickly than old objects [18, 26]. Hayes refined this into a weak and a strong generational hypothesis [14]. Qualitatively expressed, the *weak* hypothesis is that newly created objects have a much lower survival rate than objects that are older; the *strong* hypothesis is that even if the objects are not newly created, the relatively younger objects have a lower survival rate than the relatively older objects. The weak hypothesis has been confirmed repeatedly in several heap-allocating systems, and our results below do it again in the context of SML. It justifies the use of a simple generational collector with a *new* space and an *old* space: the collection effort is concentrated in the new space where there is high gain; the effort is reduced (collections less frequent) in the old space. However, having more generations is justified only under the strong hypothesis. We now turn to an analytical model of the temporal behaviour of heap-allocated objects, following the lead of Baker [6].

The intensity of heap allocation – the rate at which new objects are created – varies from one language implementation to another, from one application program to another, and from one program execution phase to another. For a class of performance considerations, the latter variation is important. For example, *opportunistic* garbage collection [30] takes advantage of idle periods in an interactive application to run the collector. In such a setting, heap behaviour must be characterised in terms of real, wall-clock time. Otherwise, the load on the collector can be defined by the amount of data created. Thus, the *time* variable in our model is quantified as the amount allocated since the start of execution. The *age* of an object is the amount allocated since object creation.

The most direct study of object dynamics tracks each object from creation through promotions to the point when it becomes unreachable[2] to the point when it is collected [31]. This technique demands an enormous computational expense, especially in a language such as SML/NJ, with a very high allocation rate; we have not attempted it. Instead we study the dynamics of objects by *volume*, as a useful approximation. We do this by tracking groups of objects of similar age as a unit.[3]

The maximum amount of data that can be allocated in between successive garbage collections is determined by the size of the nursery. For any given collection, the *nursery survival ratio* (sometimes termed *promotion ratio*) is the ratio of the amount of data promoted out of the nursery to the amount of data originally in the nursery. In a copying collector this ratio is closely tied to the cost of collection, since the overwhelming component of this cost is due to copying promoted objects. The survival rate is certainly not uniform during a program run, but for the purpose of analysis we introduce an aggregate measure defined for a program run, the *nursery survival rate*, equal to the ratio of the total amount promoted out of the nursery over all collections to the total amount allocated in the run. The variation along the additional axis of time (phase of execution) is beyond the scope of this paper. As defined, the nursery survival rate depends on the choice of collection points in the particular program run. We assume further that under uniform nursery size (i.e., uniform spacing of collection points) this rate is determined by nursery size, and little influenced by the positioning of points. Therefore we consider the nursery survival rate as a function of nursery size. Furthermore, we abstract from the discrete (whole number of words) values of nursery size, amount allocated, and amount promoted, and assume continuous, smooth functions. Experimental results show that for a broad range of values, this is a reasonable assumption. For a given age, the *object survival rate* is the fraction of objects (by volume) which survive to be that age or older. Thus, object survival rate is a function of age. As above, we abstract to a continuous model. There is a simple relationship between the *nursery survival rate* $s(x)$ and the *object survival rate* $s_o(x)$. Just before a collection, a nursery of size M contains objects whose age ranges between 0 and M . The amount promoted out of the nursery is $\int_0^M s_o(x)dx$. The nursery survival rate is $s(M) = \frac{1}{M}\int_0^M s_o(x)dx$. Differentiating with respect to M we obtain the solution $s_o(x) = s(x) + xs'(x)$. Note that both survival rates are nonincreasing functions by definition, so that $s'(x) \leq 0$ and therefore the object survival rate is everywhere lower than the nursery survival rate.

We define *object mortality*, a function of object age, to be the probability of objects of that age dying within the next infinitesimal time increment. Formally, $m_o(x) = -\frac{s_o'(x)}{s_o(x)} = -\frac{d}{dx}\ln s_o(x)$. By analogy, we define *nursery mortality* to be $m(x) = -\frac{s'(x)}{s(x)} = -\frac{d}{dx}\ln s(x)$. We interpret nursery mortality, from the implementor's standpoint, as the marginal yield in the volume of dead data as the nursery size is increased. We are currently working on the problem of finding closed-form expressions which fit experimentally observed forms of the above functions, and which are based on an understanding of the structure of computation.

Having defined the mathematical models offered above, we proceed to describe how one can set up experiments to derive them numerically for particular programs. A flexible garbage collector will be our experimentation instrument.

2 Experimental setup

In the study of heap behaviour, one might choose to track the history of individual objects. This approach requires generating program traces and then extracting various statistics. An alternative is to collect less precise, aggregate data, thus tracking groups of objects. In the absence of object type information, the only interesting parameters which guide object grouping are object age and size. We note below that size can be abstracted away if we consider allocation by volume, not by number of objects. With age as the remaining relevant parameter, it turns out that a generational garbage collector, properly instrumented, can serve as the analysis tool we need [22].

2.1 Toolkit collector

The garbage collector used for the studies reported here is the UMass Language-Independent Garbage Collector Toolkit, to which we added language-specific code to interface it with the SML/NJ system. To summarise the relevant features of the toolkit: the heap is organised into a number of generations, 0 being the youngest; each generation consists of a number of steps, and each step may have any number of blocks. A block is a contiguous, fixed-size, piece of memory. The number of blocks in a step may vary over time. A separate space hosts large objects which do not fit in a block. A contiguous, arbitrarily large part of memory (the *nursery*) serves as the allocation region. It may be guarded on either side by write-protected pages. The nursery logically belongs to a step.

[2] To be precise, one could consider the object useless immediately after the last reference to it is made, even if it remains reachable. This leads into the domain of compile-time garbage collection which is beyond the scope of this study.

[3] It has been suggested that a hybrid scheme – tracking newer objects by group, and older objects (which are fewer) individually – could be used.

2.2 Interfacing the toolkit with SML/NJ

Data presented to the garbage collector in SML/NJ come in three varieties: 1) objects allocated on the heap by the ML code, 2) static data allocated by the run-time system, and 3) precompiled ML code (in the text segment of the executable image).

SML/NJ allocates in a contiguous allocation region. The rate of allocation is very high: on a DECStation 5000/200, a typical value is 16 megabytes/second. A large part of allocation is due to function closure records and callee-save register records; less than a quarter is due to data records, and a large fraction of these are three words long, that is, cons-cells [12]. There is no check for allocation overflow on individual allocations; instead, checks are performed at function entry points only. A function in the SML/NJ intermediate representation has one entry point, several exit points and no loops. The compiler can statically determine the maximum possible allocation in a function, and prefix each with an overflow check.[4] If the remaining space is not sufficient, the garbage collector is invoked. Upon return from the collector, the check is repeated. We implemented ML's allocation region as a nursery in step 0 (generation 0). All objects, regardless of size, are allocated in this region.

For each *non-initialising store*, that is, whenever it writes a word into an already existing object, SML/NJ allocates a special 4-word record on the heap, containing the address of the object and the offset of the word. All such records are linked together into the *store list*. A dedicated register points to the head of the list. This mechanism makes assignments very expensive [25], and discourages imperative-style programming, in keeping with the philosophy of functional programming. With programmers avoiding the imperative features, the frequency of updates in ML is extremely low relative to the rate of allocation, and the store list becomes a feasible way to implement the write barrier of generational collection. We decided to keep this mechanism intact, but had to augment it with intergenerational remembered sets for our multi-generational setup.

We equipped the collector with statistics gathering code which we execute just before launching an actual collection, and just after it is complete. We go over all the generations and all the steps in each. To compute the amount of data in a step, we follow its list of blocks, and add up the used space in each block. We follow the list of large objects associated with the step and add them as well. The resulting step size statistics are analysed off-line.

2.3 Experiments using the toolkit

Above we have seen how SML/NJ works with the collection toolkit. Here we show how the flexibility of the toolkit is exploited to devise special configurations, which, although not recommended for efficient program execution, are useful for object statistics measurements.

Here we describe two main experiments which we performed on all our benchmark programs; we shall later mention other experiments. The first is designed to measure the object behaviour of newly allocated objects, whereas the second measures long-term behaviour. Measuring promotion rates for newly alocated objects serves to determine how large to make the allocation region: if it is too small, objects have not had a chance to die, so too many are copied; if it is too large, too much memory is used. We shall see that there is a range of sizes giving good performance across all studied benchmarks. Measuring long-term behaviour serves to find good arrangements of older generations in a multigenerational copying collector: the paramount desire is to balance excessive copying against excessive garbage retention.

[4]It can also perform across-function analyses to eliminate redundant checks.

2.3.1 Young objects

The analysis in Section 1.1 shows that the relevant object characteristics can be obtained from the nursery survival rate. We designed the SML-toolkit interface to allow setting the size of the nursery to an arbitrary number of bytes. Unfortunately, the construction of ML precludes running these experiments for very small nurseries. Specifically, the nursery must be large enough so that each allocation request can be met without having to change the size dynamically. Indeed, the absolute lower limit is 32K bytes, since this size is required whenever ML calls C functions.

To span the range of nursery sizes up to 20 megabytes, we used about 480 points with spacing close to uniform on a logarithmic scale. For each point we ran the benchmark recording exact amounts allocated and promoted. The nursery survival rate was estimated as the ratio of total promotion out of the nursery to total program allocation. The approach has its drawbacks. Assuming deterministic execution, a given benchmark allocates a definite discrete sequence of objects, regardless of the garbage collector setup. A choice of nursery size imposes a partition of these objects into groups. As a result of unevenness of these groups, we have some *discretisation noise* in the observed nursery survival data – occasional small increases, which are not possible in theory. Careful numerical analysis (Section 2.5) takes care of this essentially random noise. Also, in the age region around 5 megabytes and above, as the total number of collections becomes small for certain benchmarks, we encounter the behaviour characteristic of older objects, where survival ratios greatly depend on the choice of collection points. For this reason, we shall only display the results for the region below 3 megabytes as representative of young object behaviour.

2.3.2 Older objects

To examine dynamics of longer-lived objects, we need to know not only the time of object allocation, but also the time of object demise; we can do that if we do a full collection (collecting the *entire* heap) frequently enough. We devised a setup in which a nursery of some size M is followed by a large number N of steps, each allowed to grow up to a maximum size of M. Thus each of these steps can safely contain the contents of the nursery. We set up promotion policies so that objects are promoted from the nursery to the first of these steps, and from the i-th step into the i+1-st step on each collection. All these steps belong to generation 0 and are always collected. Thus, the age of objects in step i is roughly proportional to i. In choosing the parameters M and N, we had to ensure that $MN \geq T$, where T is the total amount allocated by the program at hand. Therefore, objects never need to be promoted beyond step N. We chose nursery size M based on the temporal granularity desired and the computational cost we could afford (since each collection collects the entire heap, this cost is inversely proportional to M). For the benchmarks reported here, we had a nursery of 100 kilobytes and 10000 steps in generation 1, which allowed up to 1 gigabyte of allocation.

For each collection, we record the size of each step before and after. A run with N collections and thus N steps requires $O(N^2)$ numbers to be recorded; we use a differential encoding and compression scheme to make this feasible for values of N in the thousands. We note that the size of step i at collection j can be recorded as a *delta* from the size of step $i-1$ at collection $j-1$. Furthermore, since most of the mortality is for the youngest steps, many of the deltas will not only be small, but zero. So we truncate the all-zeroes tail of the series of deltas. This already reduces the space required to quasi-linear in the number of collections. We use standard compression techniques to reduce the remaining redundancy.

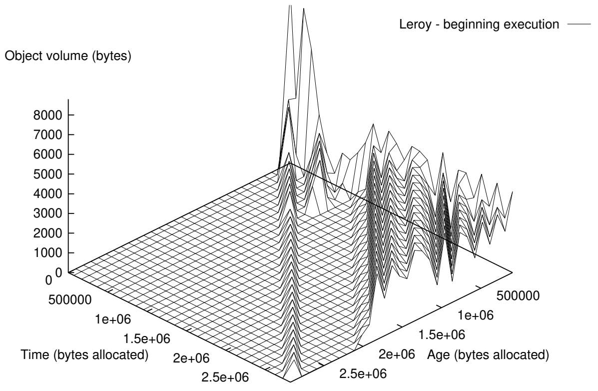


Figure 1: Joint time–age distribution for a segment of Leroy.

A study of object dynamics necessarily gathers large quantities of object statistics data. A way to display the entire data set, for modest values of N, is a three-dimensional plot, as illustrated in Figure 1. Here the execution time flows along one horizontal axis, and object age along the other; total object volume is the vertical axis. This kind of plot is useful for noting macroscopic behaviour. In the plot, a section perpendicular to the time axis (such as the foreground section) shows the instantaneous distribution of objects by age. A section perpendicular to the age axis shows live objects of a certain age (such as the rear-most section, which shows the live objects which just survived the nursery). The diagonal view (downward) corresponds to the evolution of groups of objects allocated at the same time. This is a good way to distinguish long-lived data structures. An important characteristic of long-term behaviour is the profile of total live data against time; it is here that programs will show striking differences – see Figures 3, 4a, 7a.

2.4 Experiments using the modified SML compiler

The run-time data format in the SML/NJ system has a header word for each heap-allocated object. The word contains a 4-bit tag field, which distinguishes different kinds of objects, and a length field. The tags, however, do not distinguish between "user" records and records introduced by the compiler in the closure conversion phase. We rebuilt the compiler with a modified descriptor scheme, adding 1 bit to the tag, and we now separate from ordinary records the closure records for escaping functions, closure records for known functions, and callee-save continuation closures. Since only the header word is changed, the modified system executes the same instructions as the original and has the same allocation behaviour.[5] Statistics for individual object classes can be obtained in the same manner as above.

2.5 Data analysis

As outlined above, of the several interesting object lifetime functions, the nursery survival rate is the only one we measure directly, and we only measure it at a necessarily limited number of points. In the presence of round-off noise for small nursery sizes (relative to object granularity) and opportunism interference for large nursery sizes (relative to total benchmark allocation), we must use good data analysis to derive the remaining object lifetime functions. In physical sciences, the existence of an underlying law of nature and its simple mathematical model justifies the use of interpolation and smoothing. We cannot predicate such a model yet, and the justification for smoothing must rest in the apparent goodness of fit. Starting with the nursery survival rate data, we re-express the nursery size on a logarithmic scale since we span three decades ($35 \cdot 10^3$ – $20 \cdot 10^6$ bytes) [19]. We then compute a fitting cubic spline, carefully adjusting the minimisation functional for a balance between goodness of fit and degree of smoothness [17, 21]. Using the analytical expression of the spline, we compute its derivatives, and then, inverting the logarithmic scale, compute the nursery mortality, object survival and object mortality rates.

2.6 Benchmark programs

The benchmark suite we used draws upon Appel's collection, and adds some further scientific programs. Table 1 (adapted from [2, 11]) summarises the individual benchmarks. Due to space constraints, results will be presented and discussed for a subset of the benchmarks consisting of Leroy, Yacc, and ML .

3 Results

We shall look at the results obtained from our main two experiments, first the promotion analysis for young objects and then the lifetime analysis for older objects. We shall discover, as expected, that object decay is faster than exponential for new objects. In other words, the weak generational hypothesis is confirmed. The investigation in Section 3.2 will further qualify this result, showing that it is the function closure objects which exhibit the generational behaviour most strongly. Our analysis of older objects is not conclusive. On the one hand, the benchmarks have wildly different behaviour patterns, with no immediately perceptible common characteristics. On the other, these benchmarks are still too short-running to reveal true long-term behaviour. Further work will be needed to elucidate these points.

3.1 Behaviour of young objects

Let us examine the behaviour of young objects as revealed in the survival and mortality curves. For example, let us look at the Leroy benchmark, Figure 2. We see the fast drop in survival in the region of new objects (Figure 2a). From a value of 100% at 0, the nursery survival is down to 2% for the smallest size we measured, 35 kilobytes. The curve slowly levels and is close to flat by 2 megabytes. The nursery mortality curve (Figure 2c) correspondingly falls from its initial high values, and by the time the nursery is sized at 500 kilobytes to 1 megabyte, it is quite low. The object survival curve (Figure 2b) has a sharp knee even earlier than nursery survival, at the age of around 100 kilobytes.

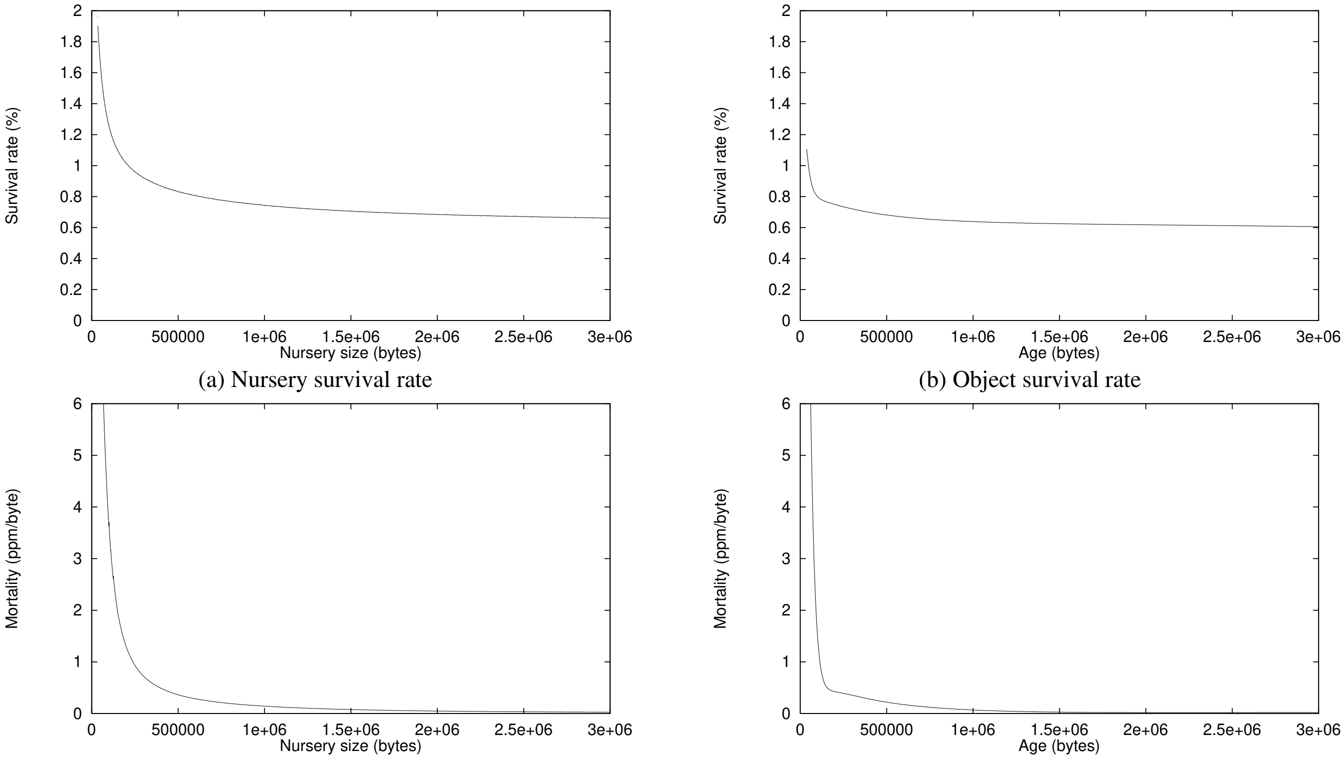
Looking at other benchmarks, we note qualitatively the same behaviour. What is different is the age at which the survival curve starts levelling off, and what level is reached. For instance, ML (Figure 4) has a much higher nursery survival rate and the knee in the object survival rate curve, although present at around 150 kilobytes, is not as sharp. Likewise, the corresponding mortality curves lie much lower than for Leroy.

The conclusion is that the newly allocated objects behave in substantially the same way across all benchmarks. We explain this as a result of the overwhelming number of function-closure objects as opposed to user-level objects. These objects are for the most part

[5] The length field lost one bit, halving maximum record length, but this did not cause any difficulties.

Program	Description
CW	The Concurrency Workbench [8] is a tool for analysing networks of finite state processes expressed in Milner's Calculus of Communicating Systems, processing a sample session input.
Leroy	An implementation of the Knuth-Bendix completion algorithm, implemented by Gérard Huet, processing some axioms of geometry.
Lexgen	A lexical-analyser generator, implemented by James S. Mattson and David R. Tarditi [3], processing the lexical description of Standard ML.
ML	The SML/NJ compiler compiling the Leroy benchmark.
Modula	A compiler translating a Modula-like language [23] into GNU C. The input is a 1400-line source program.
PIA	The Perspective Inversion Algorithm [28] decides the location of an object in a perspective video image.
Simple	A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [10], translated into ID [13], and then translated into Standard ML by Lal George.
VLIW	A Very-Long-Instruction-Word instruction scheduler written by John Danskin.
Yacc	A LALR(1) parser generator, implemented by David R. Tarditi [24], processing the grammar of Standard ML.

Table 1: Benchmark Programs



(c) Nursery mortality (d) Object mortality

Figure 2: Leroy: lifetime statistics.

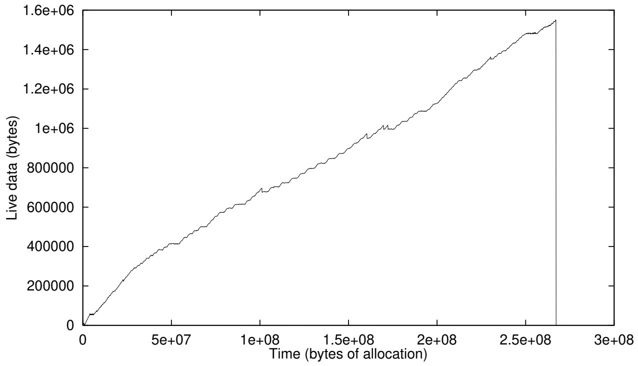


Figure 3: Leroy: live data profile.

short-lived, as they represent what in other implementations would be the active area on top of the stack. Hence their statistics mask those of the user-level data, which are application-specific. The lesson for collector setup is that the marginal utility of increasing the nursery size diminishes very quickly. The designer will weigh this fact against available memory, and will not spend more than 1 or 2 megabytes on the nursery even if the most demanding applications (such as the compiler) are expected. A much smaller area, 500 – 700 kilobytes will suffice in many cases.

3.2 A refinement: object statistics by class

There are many dimensions to explore in heap behaviour; we have undertaken additional experiments in some promising directions, including simulations, cost models, cache performance, opportunism analysis, and refined promotion analysis. Some of these are done by further examination of the data already described, whereas some require new data to be gathered beyond what is straightforwardly available from the toolkit. Due to space constraints, we shall only consider the last point here.

The experimental setup described so far dealt with properties of objects *en masse*; we would like to have information at least somewhat more refined. A straightforward modification in the language-collector interface allows gathering allocation and promotion statistics based on any criterion encoded or derivable from object contents at run-time. We will concentrate on two such criteria: object size and object tag, both directly encoded in the object header word as described in Section 2.4 above.

The distribution of allocated objects by size is shown by means of histograms (Figures 4b and 7b). Small objects completely dominate allocation in these programs. This situation is favourable to a simple collector which minimises per-object overhead [1]. The distribution of allocated objects by tag is shown in Figures 4c and 7c.

Table 2 gives the allocation as determined by scanning the heap and counting all objects actually allocated. (The entry ε means that there are some objects in the given category, but the number is below reportable.) Tables 3, 4, and 5 give the breakdown of allocation by size (sizes contributing more than 0.1% of volume are listed).

Previously, there have been measurements of the distribution of allocated objects by kind for SML [25]. In addition to this, we are in a position to assess the difference in temporal object behaviour by kind. The same analysis as in Section 1.1 applies here, but for individual object classes. The nursery survival data classified by object tag are shown in Figures 5 and 6, for those tags contributing more than 5% of the allocation in the particular program.

We can now see clearly that user records survive much longer than closures; over 8% are still alive after 2 megabytes of allocation in Yacc. This part of the allocation more closely resembles that found in conventional language implementations. On the other hand, closures have extremely short lifetimes. We speculate that the differences amongst them (namely, 16- and 20-bytes large closures live considerably longer than 40- and 44-bytes large ones) are due to iterative (tail-recursive) execution where particular size closures are associated with particular functions. Thus, closures for “inner” functions are shorter-lived than those for “outer”. The nursery survival rate for callee-save continuation records is already below 1% at 80000 bytes of total allocation, which corresponds to 43600 bytes of allocated callee-save continuation records. If an alternative implementation allocated callee-save continuation records on the stack, then the active top of the stack area could be estimated to be well within 45000 bytes. Note that this size is sufficiently small to fit in present-day data caches.

3.3 Behaviour of older objects

Each program leaves a characteristic signature in its live data profile. Inspection of these curves reveals phases of program execution, building-up of temporary data structures and their disposal, etc. For example, the curve for ML (Figure 4d) shows the parsing and type checking phases first, followed by repeated optimisation passes, and code generation and instruction scheduling passes. On the other hand, Leroy (Figure 3) shows a steady accumulation of data. It would be worthwhile to explore the reference pattern to the data: are objects kept because they are truly used in computations, or just because they are pointed to by some long-lived object (a dictionary or symbol table)?

4 Directions for further exploration

The studies reported here point to several promising directions for further in-depth investigation. We showed the applicability of the garbage collection toolkit in the context of a functional language with intensive heap allocation. We developed a methodology for gathering object statistics, taking advantage of the flexibility inherent to the toolkit. This methodology and the test-bed design will carry over to other languages, but the behaviours we expect to observe, in object-oriented languages for example, may be quite different. We identified the patterns of heap behaviour characteristic of SML/NJ, and confirmed that the weak generational hypothesis uniformly holds. We investigated ways to take advantage of it by applying different heap management policies. We found that uniformly sizing the nursery in the range of one megabyte gives satisfactory results. This global observation is supported and explained by the analysis of individual object classes. More exploration is needed, however, in the area of older space management. We touched briefly on cache behaviour issues, and we plan to investigate the interaction of heap organisation and cache configuration more fully in forthcoming research. Finally, with more efficient simulation techniques, it should be possible to use longer-running benchmark programs than we have been able to.

5 Related work

Many authors have examined issues of garbage collection performance at the macroscopic level, while some have tried to characterise it theoretically in terms of statistical properties of object allocation. We believe that both approaches, in addition to finer granularity measurements described here, are needed to inspire theoretical models, and should be used to validate them. Ungar [27] reported on the performance of garbage collection in a Smalltalk system and investigated the tradeoffs in nursery size selection. Theoretical models of behaviour have been proposed by Baker [5]. Zorn conducted statistical studies in the context of Lisp [31]; his methodology is based on object-level simulation, and lifetimes are estimated from object reference points. We improve on his Discrete Interval Simulator by taking advantage of complete liveness information. For a more detailed discussion of the UMass Garbage Collector Toolkit consult the original design report [16] and a later article on the Smalltalk system [15]. For a broader overview of garbage collection algorithms and generational techniques, consult Wilson's survey paper [29].

6 Acknowledgements

In the beginning of this work, Scott Nettles and David Tarditi provided us with some of the benchmarks, and with valuable discussions. Brian Milnes, Andi Stephens, and Craig Marcus helped set up the operating system environment. Tony Hosking developed the toolkit and improved it as we encountered the need for additional functionality. Rick Hudson, Amer Diwan, and Chris Weight participated in the design of the toolkit. A preliminary version of this report was presented at the OOPSLA '93 Workshop on Memory Management and Garbage Collection. We thank Ben Zorn for his comments on the draft submission, and all the participants for their discussion. We thank Kathryn McKinley, Tony Hosking, Amer Diwan, Rick Hudson, and the anonymous referees for their comments. This work originated during our stay at Carnegie Mellon University, which was made possible by Peter Lee and Bob Harper.

References

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–83, 1989.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, first edition, 1992.

[3] Andrew W. Appel, James S. Mattson, and David Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[4] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Dept. of Computer Science, Princeton University, Princeton, NJ, 1994.

[5] Henry Baker. The thermodynamics of garbage collection — a tutorial. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, September 1993.

[6] Henry G. Baker. 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

[7] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[8] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[9] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *1992 ACM Conference on Lisp and Functional Programming*, pages 43–52, San Francisco, California, June 1992.

[10] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA, February 1978.

[11] Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for Publication, October 1993.

[12] Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[13] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[14] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46, Phoenix, Arizona, October 1991. *ACM SIGPLAN Not. 26*, 11 (November 1991).

[15] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Not. 27*, 10 (October 1992).

[16] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.

[17] Peter Lancaster and Kęstutis Šalkauskas. *Curve and Surface Fitting*. Academic Press, 1986.

[18] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[19] Frederick Mosteller and John W. Tukey. *Data Analysis and Regression*. Addison-Wesley, 1977.

[20] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [7], pages 357–364.

[21] G.A.F. Seber and C.J. Wild. *Nonlinear Regression*. Wiley, 1989.

[22] Darko Stefanović. The garbage collection toolkit as an experimentation tool, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

[23] Darko Stefanović. Implementing a small imperative language with safe dynamic allocation. Memo, April 1993.

[24] David Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[25] David Tarditi and Amer Diwan. The full cost of a generational copying garbage collection implementation, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

[26] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. *ACM SIGPLAN Not. 19*, 5 (May 1984).

[27] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[28] Kevin G. Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, August 1990.

[29] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [7].

[30] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana, October 1989. *ACM SIGPLAN Not. 24*, 10 (October 1989).

[31] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, December 1989. Available as Technical Report UCB/CSD 89/544.

Program	Allocation (megabytes)	record (%)	array (%)	string (%)	bytearray (%)	realdarray (%)	pair (%)	reald (%)	escaping (%)	callee-save (%)	known (%)
Leroy	268.34	0.15	0.02	ε	ε	0	12.90	0	36.72	50.08	0.11
ML	194.78	12.30	1.51	0.06	ε	0	12.36	0	12.40	51.79	9.58
Yacc	69.45	5.15	0.11	0.54	ε	0	22.13	0	2.38	56.54	13.14

Table 2: Allocation characteristics of benchmark programs: breakdown by kind of object

Size	12	16	20	24	28	32	36	40	44	48	64
Volume (%)	19.65	25.65	11.32	7.87	5.31	13.04	0.66	0.66	15.18	0.27	0.12

Table 3: Allocation characteristics of benchmark Leroy: breakdown by object size

Size	8	12	16	20	24	28	32	36	40	44	48	52
Volume (%)	1.20	16.76	14.12	8.10	7.08	5.35	7.67	4.42	2.08	7.14	3.50	1.67

Size	56	60	64	68	72	76	80	84	88	92	96	112
Volume (%)	3.34	1.45	6.74	0.44	0.96	0.77	1.40	0.10	1.05	0.21	0.13	1.61

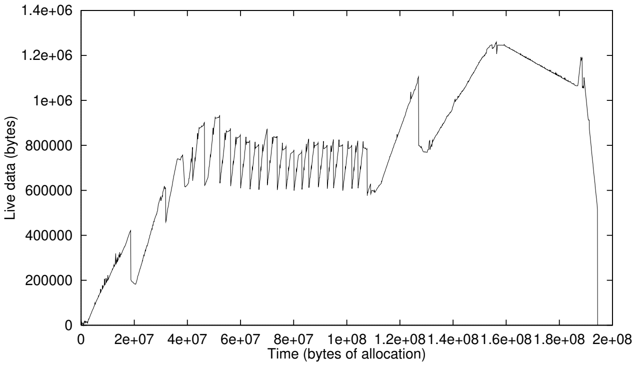
Size	120	132	140	144	156	164	276
Volume (%)	0.58	0.72	0.12	0.10	0.17	0.18	0.45

Table 4: Allocation characteristics of benchmark ML: breakdown by object size

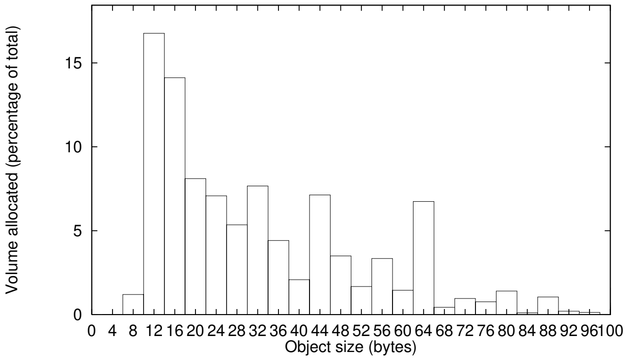
Size	8	12	16	20	24	28	32	36	40	44	48	52
Volume (%)	0.27	23.07	10.17	6.91	2.09	3.96	3.23	4.13	8.79	15.18	6.38	0.55

Size	60	64	68	72	80	84	88	92	96	104
Volume (%)	0.37	3.39	0.34	0.16	0.86	3.75	3.89	0.84	0.89	0.11

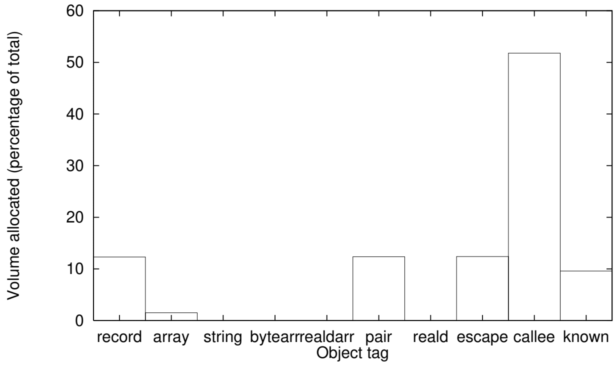
Table 5: Allocation characteristics of benchmark Yacc: breakdown by object size



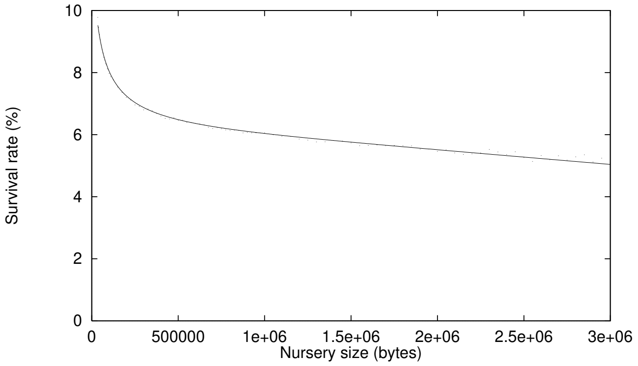
(a) Live data profile



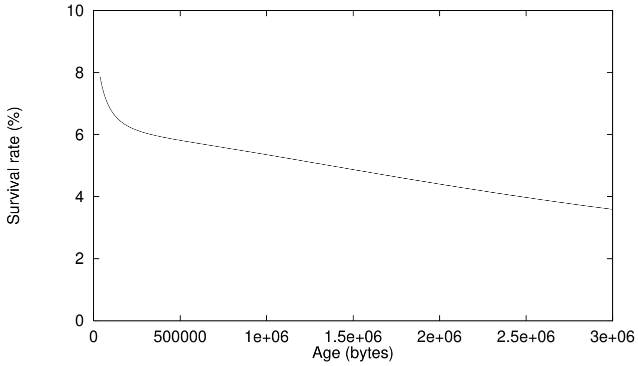
(b) Allocation segregated by object size



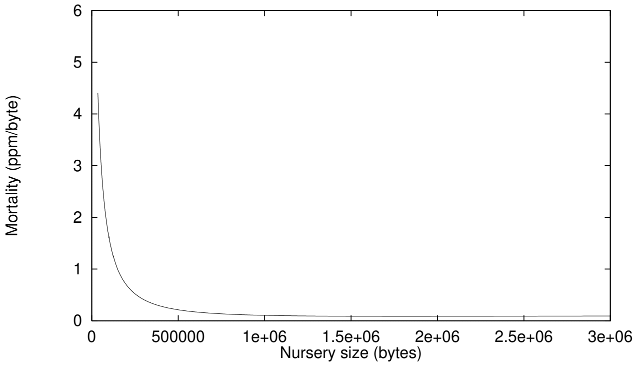
(c) Allocation segregated by tag



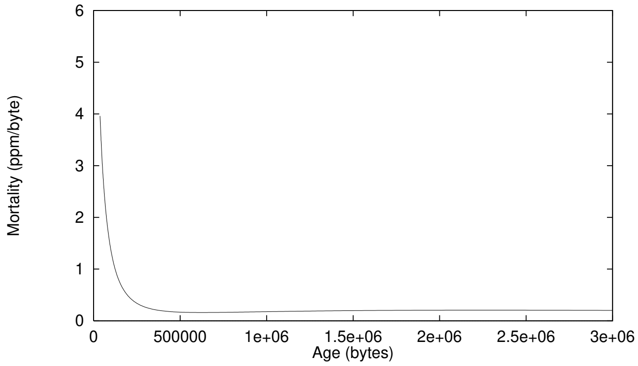
(d) Nursery survival rate



(e) Object survival rate



(f) Nursery mortality



(g) Object mortality

Figure 4: ML.

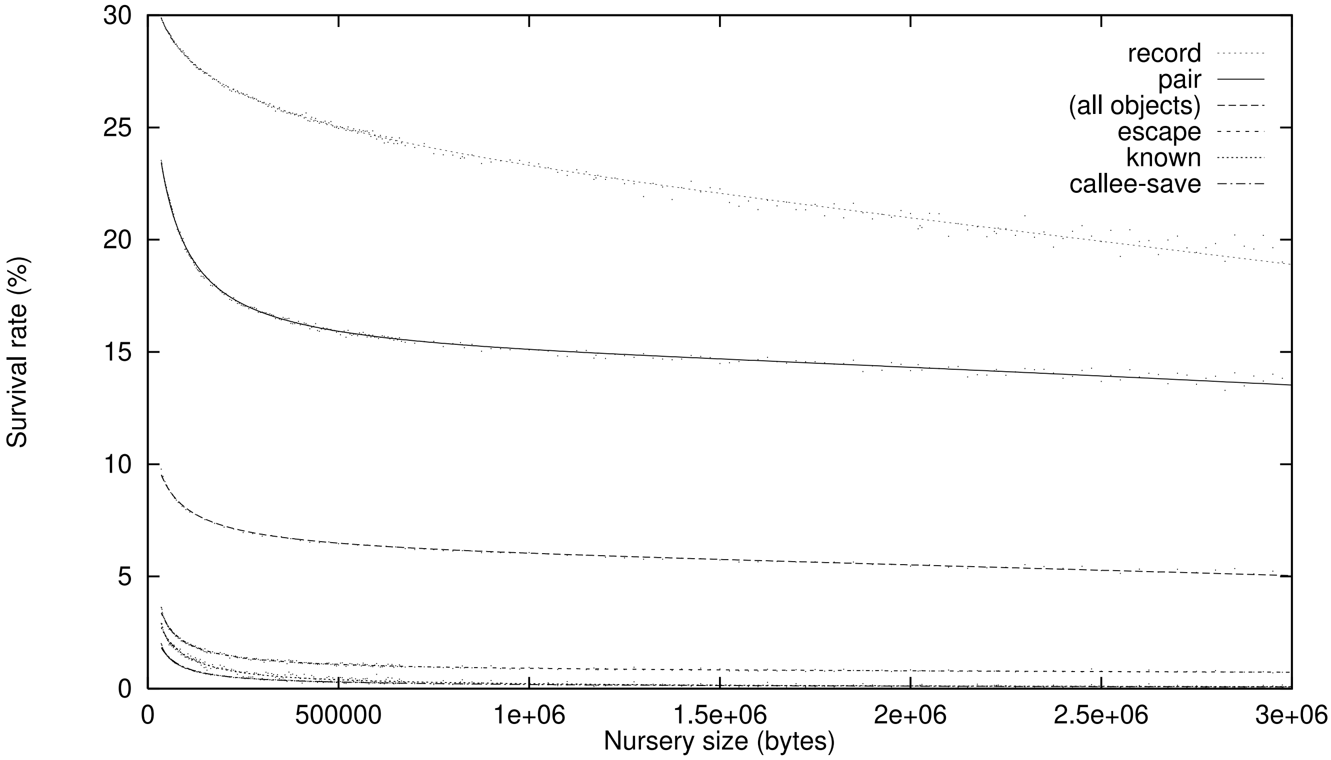


Figure 5: ML: nursery survival for various kinds of objects.

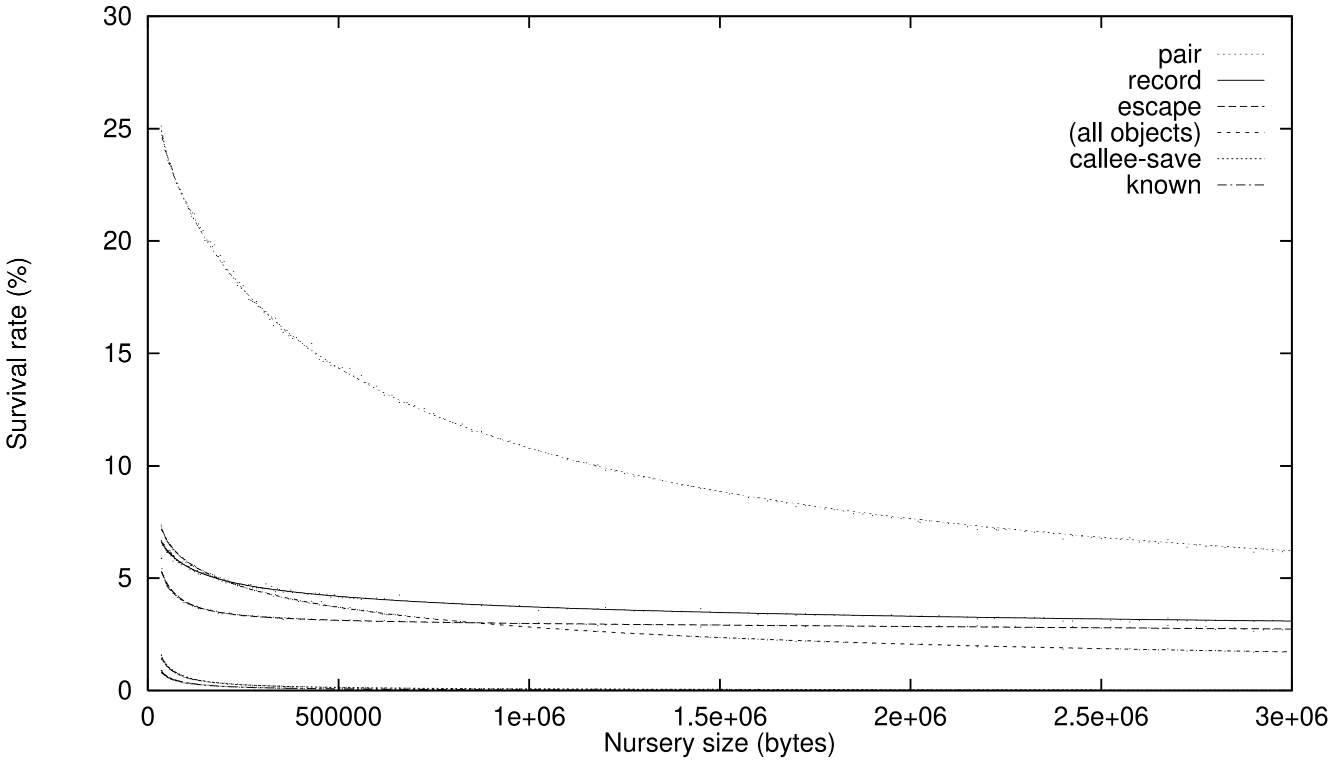
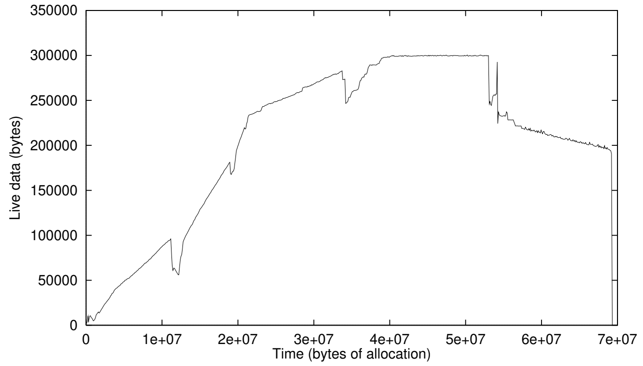
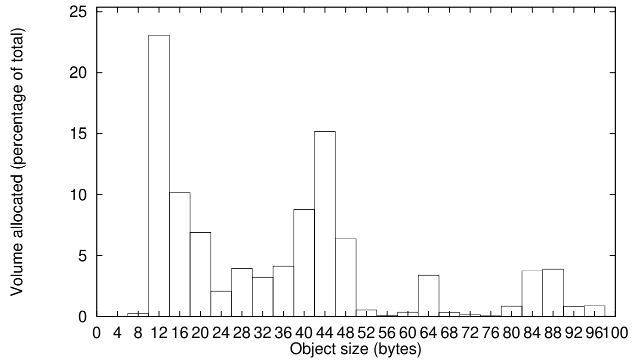


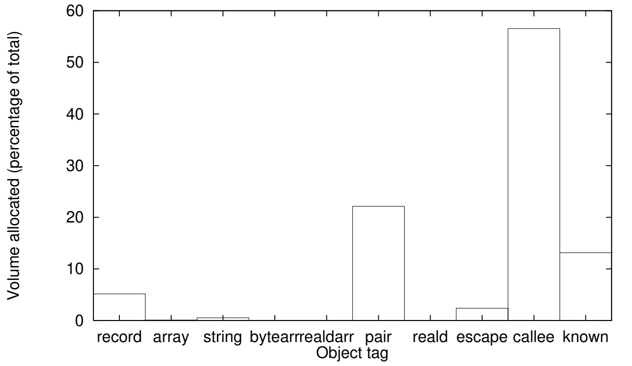
Figure 6: Yacc: nursery survival for various kinds of objects.



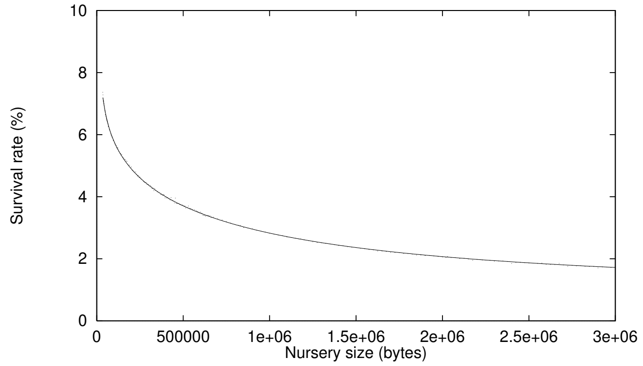
(a) Live data profile



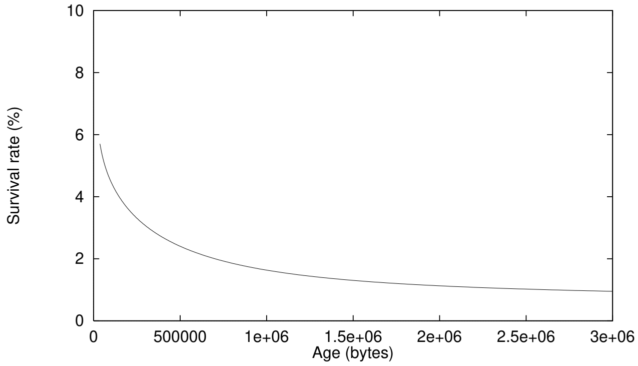
(b) Allocation segregated by object size



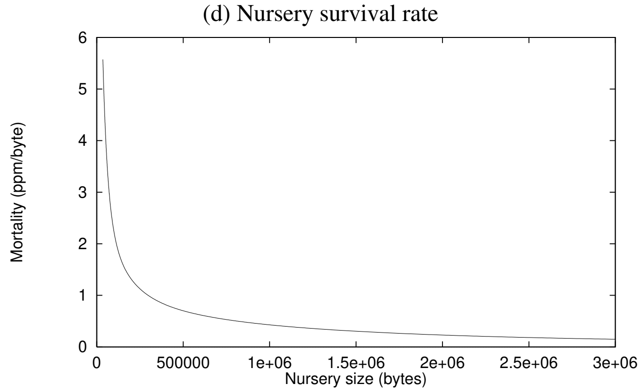
(c) Allocation segregated by tag



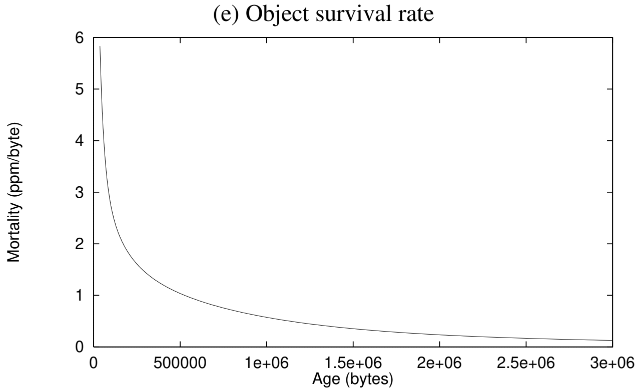
(d) Nursery survival rate



(e) Object survival rate



(f) Nursery mortality



(g) Object mortality

Figure 7: Yacc.