The Garbage Collection Toolkit as an Experimentation Tool

Darko Stefanović*
Fox Project
School of Computer Science, Carnegie Mellon University
`darkos@cs.cmu.edu`

September 10, 1993

Introduction

The UMass Garbage Collection Toolkit[4] was designed to facilitate language implementation by providing a language-independent library of collection algorithms and policies, and auxiliary data structures.

Having integrated the toolkit collector into Standard ML of New Jersey, we found that the functionality of the toolkit allowed us to perform experiments revealing the nature of object allocation and object dynamics in the SML/NJ system. We explored ways to visualize the large quantities of data our instrumentation gathers. We believe that the techniques developed can be of use to the language implementor in reviewing overall performance, and to the application writer in tracking down the space behavior of the program (which, for functional languages, is often not intimately related to the source program).

In the following we briefly describe the collector interface with SML, discuss the methodology of experiments, and outline the outcome of the experiments. Although we have examined a larger suite of benchmark programs, we limit the presentation here to the Leroy benchmark, one of the benchmarks in [1].

For a detailed description of the toolkit collector see [4]. To summarize the relevant features: the heap is organized 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 don't 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.

For SML/NJ, we let step 0 (in generation 0) have a nursery, with the same function as the variable-sized allocation region of the original collector. Since ML code performs explicit nursery overflow checking, we didn't need guard pages. Large objects are allocated here as well, and moved to the large object space on the first collection; however, this inefficiency can be ignored since there are few large objects in the SML/NJ system.

The SML/NJ compiler produces code which maintains the write barrier by linking onto a list of stored-into locations. To maintain root sets for multiple generations, we dump this store-list into remembered sets on each collection.

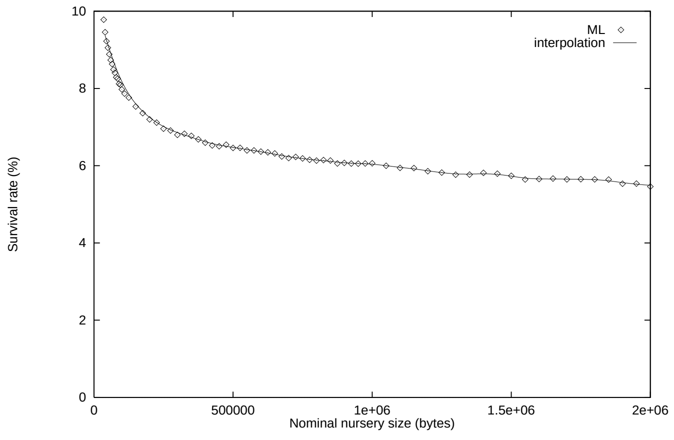
Having rebuilt the run-time system around the toolkit compiler, we added instrumentation to record, prior to and following each collection, the amount of data contained in each generation and in each step. (Exact values, taking into account block fragmentation, are obtained by inspecting all allocated block data structures; quick estimates are immediately available, since each step records the number of blocks it contains.) This includes the large object space, since large objects logically belong to individual steps as well.

Survival rate and mortality

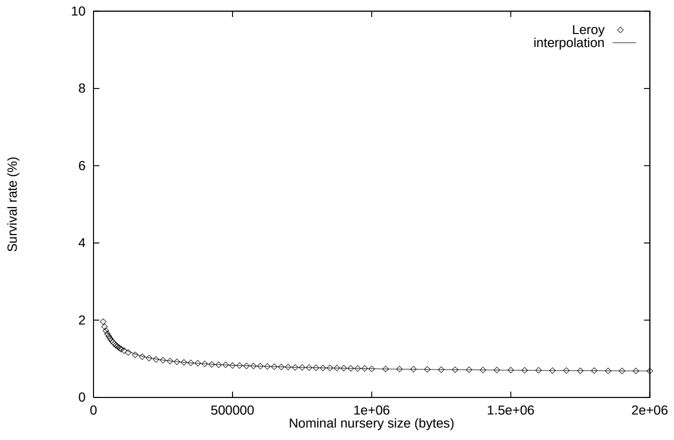
The simplest experiment is to examine the survival rate of objects. When sizing the nursery region, the user

*Current address: Department of Computer Science, University of Massachusetts (`stefanov@cs.umass.edu`) wants to make it as large as possible to allow objects to die and never be promoted (given the constraints of available memory and memory subsystem effects (paging, caches, etc.)). It is important to discover if there is a threshold which ensures low promotion, and if there is, how application-specific is it. Furthermore, object mortality needs to be a decreasing function of object age for the generational hypothesis to be borne out [2].

To measure the survival rate averaged over a program run, we only need to record the amount of data in the nursery prior to each collection and the amount promoted out of it. We can set up the remaining steps arbitrarily. We chose an efficient configuration, and then ran the experiment repeatedly, varying nursery size. (One could, in principle, derive the same information from the experiment described below, but at a much greater computational cost.)



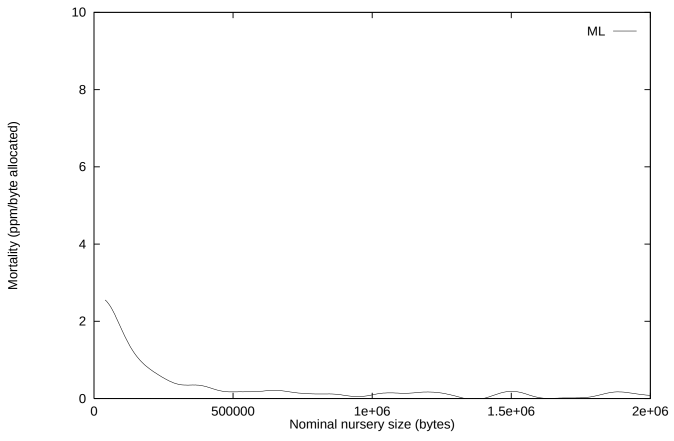
(a) compiler



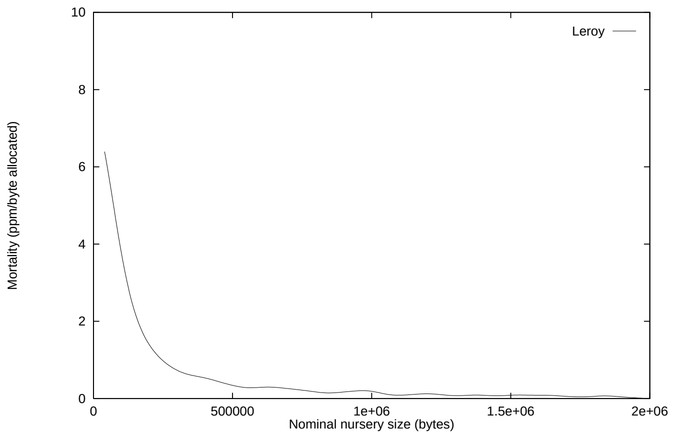
(b) user program

Figure 1: Nursery survival rate.

The mortality *m* (naturally expressed in parts-per-million per byte) is the likelihood of objects dying within an infinitesimal increment of their age, as a function of age. (Age being measured in terms of cumulative allocation.) It is derived from the survival rate *s* as $m = -\frac{ds/dt}{s} = -d/dt \ln s$.



(a)



(b)

Figure 2: Nursery mortality.

Both compilation and execution show rapidly decreasing mortality rates (figure 2). Discretization effects, due to the finite grain of both objects and of the increment of nursery size, show up in the high end as occasional increases in the survival rate (figure 1). Overall, a nursery capturing the high mortality region, about 700 kilobytes large, will perform well on this program.

Lifetimes

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 of the 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 scavenged. Thus, the age of objects in step *i* is roughly proportional to *i*. In choosing the parameters *M* and *N*, we must 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 choose *M* based on the temporal granularity desired and the computational cost we can afford (since each collection collects the entire heap, this cost is inversely proportional to *M*).

For each collection, we record the size of each step before and after. A run with *N* collectionand thuss *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.

A way to display the entire data set, for modest values of *N*, is a three-dimensional plot, Figure 3. 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 behavior. 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 the best way to distinguish long-lived data structures.

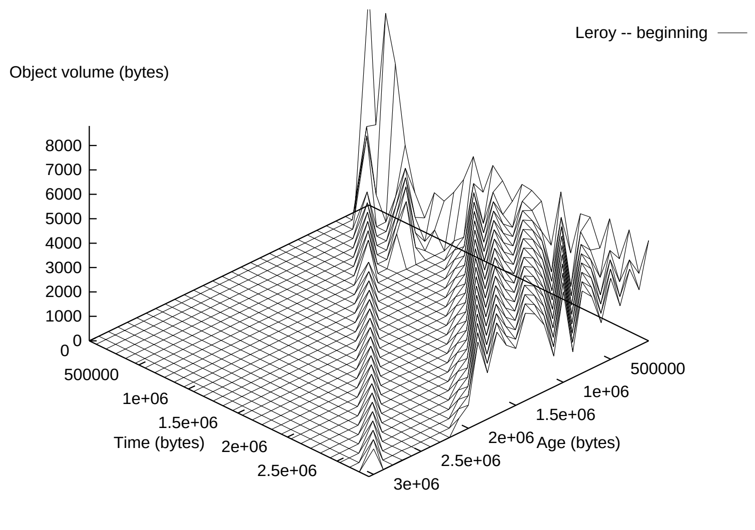


Figure 3: Initial part of execution.

A number of statistics can be extracted from the data set and displayed individually.

Since the age of an object is implicit in its step, whenever a step shrinks we know the age of the objects that died. Inverting this relation, we obtain the object lifetime distribution, Figure 4.

Age distribution is obtained by averaging live data age over the entire run. This gives us a crude extension to the plots described above, figure 5.

Total live data plot is the primary characteristic of long-term behaviour and it is here that programs will show striking differences. Figure 6 has the plots for the ML compiler and the Leroy benchmark.

Age layers (not shown) are another way to visualize creation of longer-lived data. The top line in an age layer plot corresponds to the total amount of live data as a function of time. Each layer corresponds to the contribution of live objects up to a given age.

Finally, an application writer will benefit from seeing an animated picture of heap dynamics. We built an X Windows interface representing the nursery and the steps; the SML image equipped with statistics instrumentation fires up the display and maintains a scrolling self-scaling bar graph representing the sizes of the nursery and all the steps.

An auxiliary program can be used to display the same bar graph off-line from the statistics files.

In figures 4b and 5b a somewhat atypical behavior of the benchmark is revealed; it allocates a large quantity of data at a uniform pace and hangs onto it until the end. On the other hand, the compiler, figures 4a and 5a, clearly has several groups of objects of different duration, as is to be expected in a multi-pass design. In both cases, however, we note the preponderance of short-lived objects.

Cache performance

In conjunction with a tracing tool and a cache simulator, we have examined the effect varying nursery sizes and varying cache sizes can have on cache performance. Cache performance is measured as *cycles per instruction (CPI)*. We show two graphs from our preliminary investigation: CPI as a function of nursery size with a cache size of 64K bytes (figure 7a), and CPI as a function of cache size, with a fixed nursery size of 10^6 bytes (figure 7b). The cache organization is the one found on the DECStation 5000/200.

In figure 7a, we note that varying the nursery size has little effect on the cycles-per-instruction measure, for this organization. This is to be expected, as described in [3]. The actual performance, however, is significantly affected by the choice of nursery size — see figure 8.

In figure 7b, most of the effect is due to the instruction cache; generally, SML programs' instruction working sets take a few hundreds of kilobytes.

Collection cost model

We used the cache simulator to derive a precise garbage collection cost model. It turns out that for generation 0 collections, the cost, including any memory subsystem effects, is very close to being directly proportional to the amount of data promoted, as one would expect in a copying collector.

Further analysis

The lifetime patterns are seen to be decoupled: short-term behavior is qualitatively the same for all programs, being a more or less rapid decay of (mostly closure record) objects; long-term behavior is program-dependent. In other words, short-term behavior is independent of long-term behavior. Choosing a good generational setup is thus decoupled into two tasks: short-lived objects must be filtered out with a good choice of nursery size, and other, much longer-lived objects must be managed by a well-tuned (possibly application-specific) choice of generations and steps.

Choosing the nursery size

Across all benchmarks studied, the mortality rate at the object age of about 500 kilobytes has dropped to a very low value (an insignificant part of the initial value) and stays low as the age increases further. Hence a nursery about that size will be a good choice in general. Of course, some programs, such as the compiler, have lower absolute mortality, or higher survival, and will benefit from increased nursery space — but not much.

In the above we assumed that the nursery is kept at a constant size throughout the program run. This is in contrast to the original SML/NJ collector, where the allocation region size oscillates from very large (just after a major collection) to *very* small (just prior to the next major collection). Very large sizes are not too useful, and the very small sizes are detrimental: the overhead of collector invocation is paid too often, and too many objects are promoted.

Another arrangement where nursery size, i.e., the frequency of collections, is not constant is one that attempts to collect at *opportune* moments: when the amount of live data that needs to be promoted is low. This is where long-term behavior comes into play, and such strategies may better be applied to older generations. Nevertheless, it's instructive to see how much opportunism exists at this level. We used our live data information, together with the precise cost model described above, as input to an optimizing algorithm. The optimization target was a set of collection points spanning the program run with minimum total collection cost, subject to the constraint on the maximum distance between successive points (the maximum nursery size). Figure 9 shows how increasing the maximum nursery size opens more opportunities for savings over the uniform (always maximum size) strategy, but the absolute improvement never exceeds 30% for the compiler, or 8% for the Leroy benchmark. Of course, any realistic scheme exploiting opportunism can only approach the (omniscient) optimal, and will require compiler support and/or user-level language features to be realized.

Conclusions

The flexibility of the toolkit has allowed us to configure the collector and gather the data with relative ease, once we had built the language-dependent interface for SML.

This opens the possibility of exploring the efficacy of different heap configurations by simulation in addition to direct (and more expensive) measurement.

We are preparing such an analysis for this and other, more relevant, benchmarks.

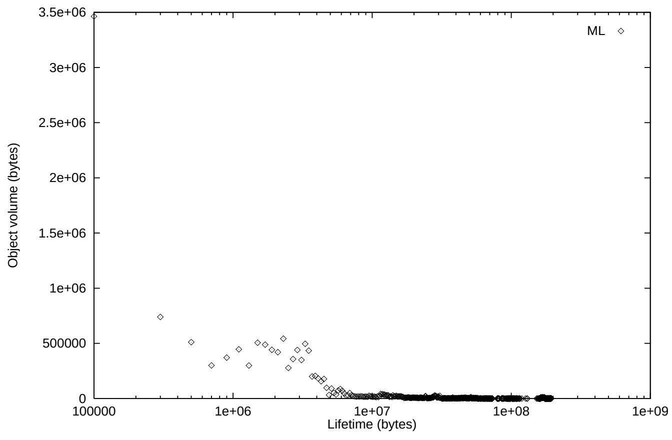
References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

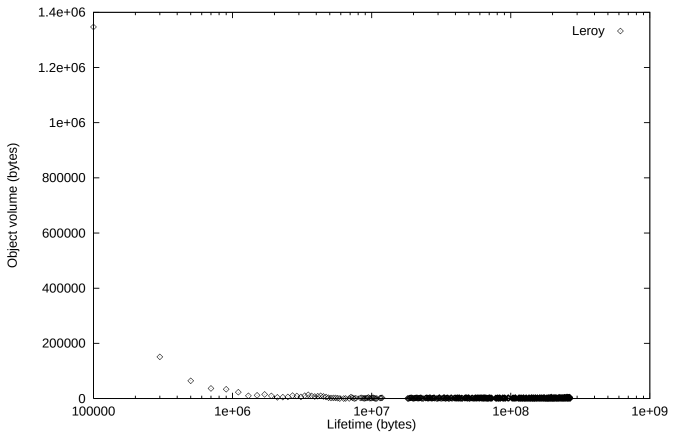
[2] Henry G. Baker. 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

[3] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for publication, July 1993.

[4] 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, MA 01003, September 1991.

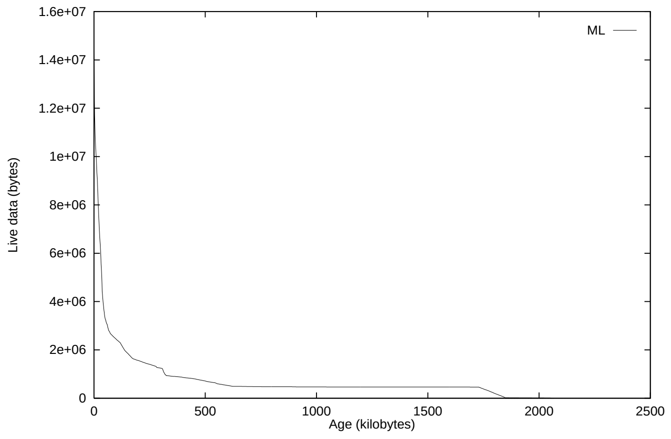


(a) compiler

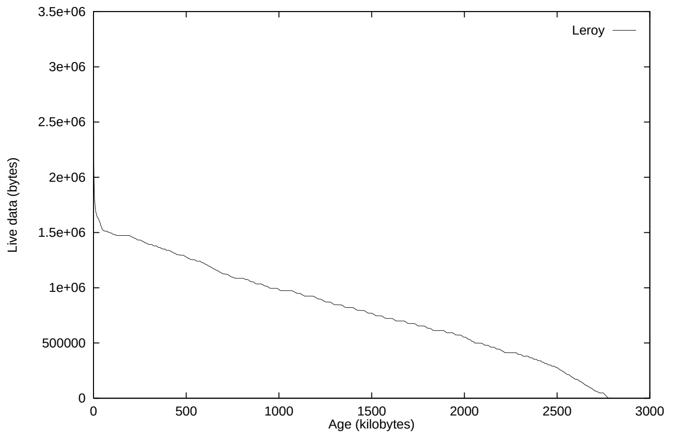


(b) user program

Figure 4: Relative distribution of objects by lifetime (note the log scale for the *x*-axis).

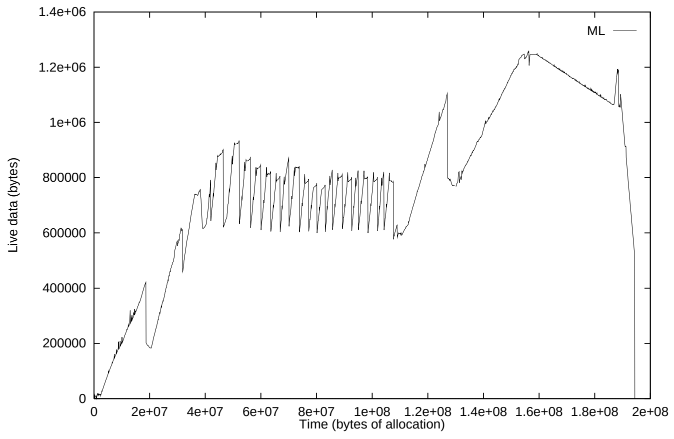


(a) compiler

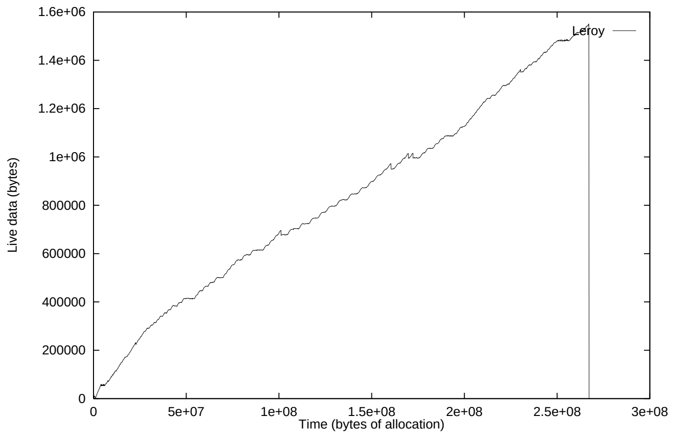


(b) user program

Figure 5: Distribution of live objects by age.

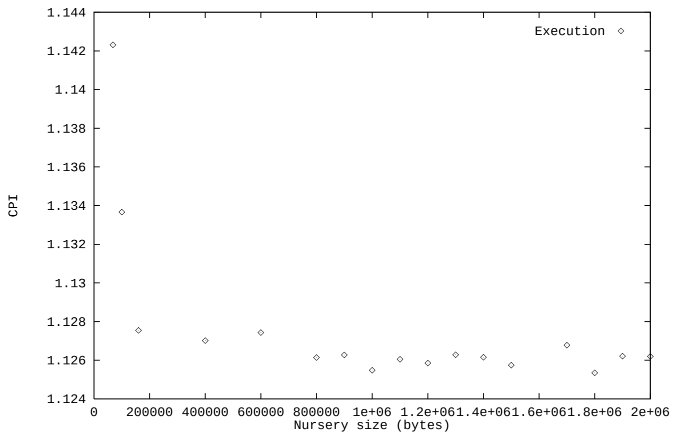


(a) compiler

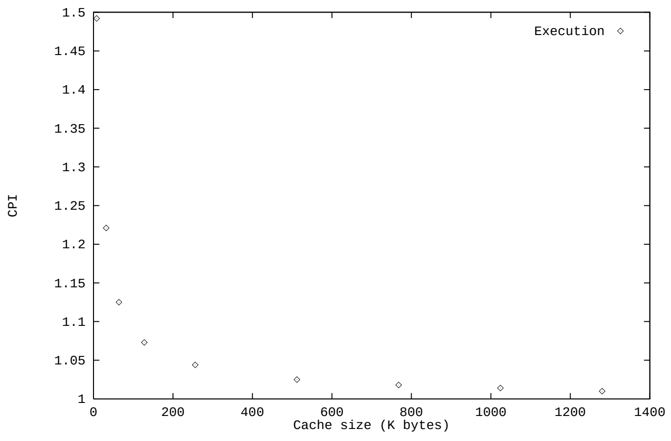


(b) user program

Figure 6: Live data.



(a) Varying nursery size



(b) Varying cache size

Figure 7: Cache performance.

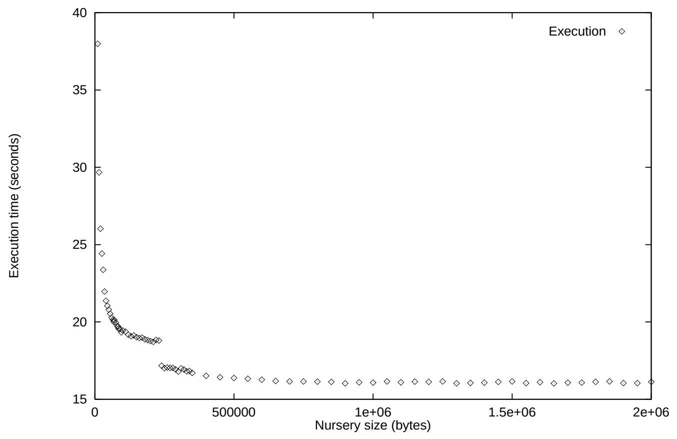
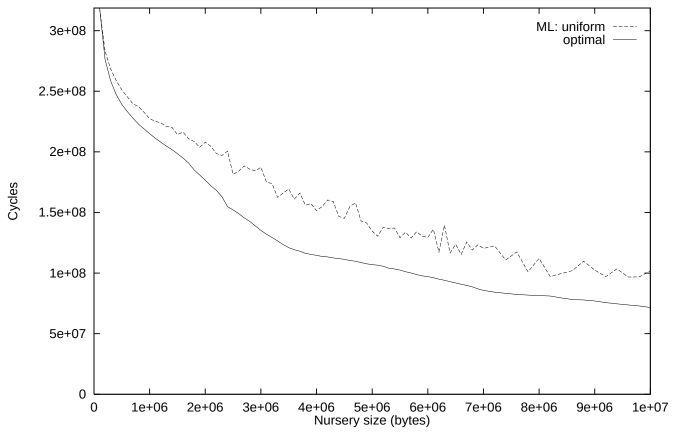
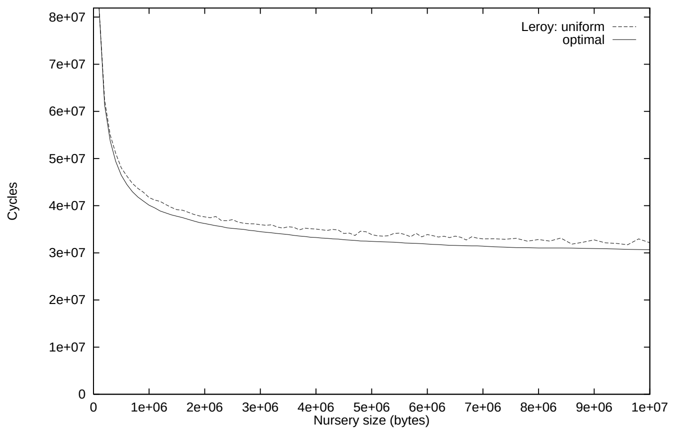


Figure 8: Running time.



(a) compiler



(b) user program

Figure 9: Available opportunism at the nursery level.