An Advisor for Flexible Working Sets

Rafael Alonso Andrew W. Appel

Department of Computer Science
Princeton University

Abstract

The traditional model of virtual memory working sets does not account for programs that can adjust their working sets on demand. Examples of such programs are garbage-collected systems and databases with block cache buffers. We present a memory-use model of such systems, and propose a method that may be used by virtual memory managers to advise programs on how to adjust their working sets. Our method tries to minimize memory contention and ensure better overall system response time. We have implemented a memory "advice server" that runs as a non-privileged process under Berkeley Unix. User processes may ask this server for advice about working set sizes, so as to take maximum advantage of memory resources. Our implementation is quite simple, and has negligible overhead, and experimental results show that it results in sizable performance improvements.

1 Introduction

Algorithms for managing page replacement in a virtual memory system are designed and analyzed using a model of program behavior in which each process is assumed to have a particular pattern of page accesses. The future accesses of the process cannot be known in advance, but can be predicted (to some extent) by analyzing the past accesses. The access pattern is assumed to be dependent only on the program and its input data. A virtual-memory management algorithm can be designed and evaluated with respect to a given set of processes and their page-access patterns. There are many examples of page-replacement algorithms in the literature, and much work has been done in evaluating their effectiveness[6].

However, the usual assumption that the page-access pattern is an external variable, unmodifiable by the virtual-memory management algorithm, turns out not to be justified (at least for some classes of jobs). As we will show, systems that use compacting garbage collection can (at some cost) modify their access patterns to use a smaller working set. This leads to a new and interesting kind of virtual-memory management: instead of paging a process out to disk, the operating system can advise a process to use a smaller working set.

2 Copying garbage collection

In its simplest form, a copying garbage collector [7][4] works in two equal-sized memory spaces, only one of which is in use at a time. The allocator creates new cells in the *active* space. When it is time to collect, the garbage collector traverses all of the reachable cells in the active space, and copies them into the inactive space. Then the spaces are switched; that is, the other space is now used, and the formerly active space is left empty until the next garbage collection.

The traversal of reachable cells can be done using a breadth-first or depth-first search, which takes time proportional to the number of reachable cells. The copying takes a constant overhead per cell, in addition to some time proportional to the total size of the cells copied.

A most important property of the copying garbage collection algorithm is that it never visits a garbage cell, so that the execution time of the garbage collector is dependent only on the number (and size) of reachable cells, and is independent of the amount of garbage.

The cost of garbage collection may be computed as follows (see [1] for the details). Let A be the number of reachable cells and M be the size of each of the two spaces. We'll assume for this analysis that all cells are the same size. The traversal and copying requires a constant number of operations per cell: $t = cA$.

We can amortize the cost of one garbage collection over all the allocations that can occur before the next collection. Each space can hold M cells, but after a garbage collection the new active space will already con-

tain A cells; so the number of allocations before the next collection is $M - A$. So the cost for every allocation is just

$$\frac{cA}{M-A}$$

Now, the program allocates new cells at a certain rate, say once every k instructions. The total proportion of garbage-collection overhead (relative to the amount of actual computation) is

$$\text{overhead} = \frac{1}{k}\frac{cA}{M-A}$$

Clearly, the parameters k, c, and A are completely determined by the program and its input. But M, the amount of memory used, is independent: we can give the program a larger amount of memory and the overhead will go down, or a smaller amount (subject to $M > A$) and the overhead will go up. Furthermore, it is easy for the program to adjust to a changed memory size. After each garbage collection, the program can conveniently change the size of the active space before resuming allocation.

We now compute the minimum space a program needs to be able to run at all. A program using garbage collection will typically have some areas of memory (of size S_{fixed}) not subject to garbage collection (used for executable code, the runtime system, etc). In addition, each of the active and inactive areas must be large enough to hold the A cells of live data. So the *minimum space requirement* $S_{\text{req}} = S_{\text{fixed}} + 2A$. The *total space used* is just $S_{\text{tot}} = S_{\text{fixed}} + 2M$. The difference between these two is the "discretionary" or "flexible" memory; the more discretionary memory given the program, the less overhead it will incur. Clearly, $S_{\text{flex}} = 2(M - A)$.

Since the amount of overhead a program incurs varies according to S_{flex}, we will denote the time spent on overhead (per unit of useful computation) as t_{flex}. The non-garbage-collection time requirement we will call t_{req}; clearly this is dependent on the program and its input, and not dependent on memory used.

Thus for copying garbage collection as analyzed above, t_{flex} is inversely proportional to S_{flex}:

$$t_{\text{flex}} = \frac{1}{k}\frac{cA}{M-A} = \frac{2cA}{k}\frac{1}{S_{\text{flex}}}$$

At this point it should be pointed out that modern garbage collectors use *generational* algorithms [8], which are much more efficient than the simple stop-and-copy collectors analyzed here. However, generational collectors still have the property that the overhead goes down as memory size goes up. And it is not difficult to organize a generational collector so that it can adjust its memory size after a garbage collection [2]. Analysis of generational collection is difficult; in our concluding section we speculate about how to incorporate it into our model.

3 Flexible working sets

One way to analyze virtual memory management algorithms is to note that any program at any period in its execution references only a subset of its pages. The *working set* model[5] assumes that in adjacent periods the referenced subsets will be similar. Given a period size Δt, the working set of a process at time t is that subset of its pages references between times $t - \Delta t$ and t. An operating system using the *working set algorithm* will ensure that all of these pages are in main memory before resuming execution of the process.

The very notion of "virtual memory" implies that the process is not aware of how much physical memory is allocated to it. Any decisions made by the operating system about which pages are kept in main memory are assumed to be invisible to the process. This implies that the process' pattern of page-accesses is independent of the memory-management algorithm, and of the amount of physical memory available.

But garbage-collected programs need not obey these assumptions. A program solving a given problem with a certain amount of live data can run in an arbitrarily large heap; and the larger (and thus sparser) the heap, the larger the working set. (This is probably true for all values of Δt, and it is provably true for large values of Δt, e.g. ten times the interval between garbage collections.)

A "conventional" (non-garbage-collected) process does not have much choice about how much virtual memory to use; this is determined by the algorithm it runs. But a garbage-collected process can decide (after every garbage collection) how much virtual memory to use in the next round[2]. Clearly, the more virtual memory it uses, the less garbage collection overhead it will incur, and the more paging it will do.

A good heuristic might be to use as much memory as is available without paging. But there might be several garbage-collectible processes using the same physical memory, each trying to obtain as much memory as it can. We claim that the allocation of garbage-collection heap sizes should be centrally managed, just as the allocation of physical memory is managed by a conventional page-replacement algorithm. In the following section we discuss a number of strategies that may be used for managing the memory requirements of flexible working set jobs.

4 Strategies for flexible working set management

Suppose we have two garbage-collectible processes P_1 and P_2 with minimum memory requirements S_{req_1} and S_{req_2} respectively, running on a computer with a physical memory of size S_{phys}. Then let $T = S_{\text{phys}} - (S_{\text{req}_1} + S_{\text{req}_2})$ be the amount of "discretionary" memory that can be divided between $S_{\text{flex}1}$ and $S_{\text{flex}2}$. We know that

$$t_{\text{flex}i} = \frac{\alpha_i}{S_{\text{flex}i}}$$

where α_i is a constant calculated from k_i, c_i, and A_i (as shown in Section 2).

We want to make the most effective use of the discretionary memory T. We can vary the parameter $x = S_{\text{flex}1}/T$ from 0 to 1. There are different assumptions we can make about what to minimize, however:

Assumption 1: We want to complete processes 1 and 2 in the minimum time. The optimum strategy is to halt process 2, run process 1 to completion, and then run process 2. That is, set $x = 1$ so that $t_{\text{flex}i} = \frac{\alpha_i}{T}$, which is as low as possible. (A similar trick works as a page-replacement algorithm for non-garbage-collection processes; it's called "batch execution.")

We reject this solution on the grounds that we want each process to make steady progress.

Assumption 2: We will give each process equal time-slices for t_{req_i} (that is, execution not including garbage-collection time), and minimize $\sum t_{\text{flex}i}$ (the total garbage-collection time). Thus, we minimize

$$\frac{\alpha_1}{xT} + \frac{\alpha_2}{(1-x)T}$$

This has a minimum at

$$x = \frac{\sqrt{\alpha_1}}{\sqrt{\alpha_1} + \sqrt{\alpha_2}}$$

that is, each process gets discretionary memory proportional to the square root of its α. (Assuming similar k_i and c_i, a process gets discretionary memory proportional to the square root of its live data.)

Assumption 3: Each process is given equal timeslices for its execution including garbage collection, and we want to minimize collection overhead. Thus, if we let z_i be a variable that determines how much of process i gets executed in each timeslice,

$$z_1(t_{\text{req}_1} + \frac{\alpha_1 T}{x}) = z_2(t_{\text{req}_2} + \frac{\alpha_2 T}{1-x}) = t_{\text{slice}}$$

and we want to minimize

$$z_1\frac{\alpha_1 T}{x} + z_2\frac{\alpha_2 T}{1-x}$$

This is minimized when

$$x = \frac{\sqrt{\alpha_1/t_{\text{req}_1}}}{\sqrt{\alpha_1/t_{\text{req}_1}} + \sqrt{\alpha_2/t_{\text{req}_2}}}$$

Intuitively, the more time a process spends doing useful work (t_{req}), the less it garbage-collects; and if it doesn't collect much, then it doesn't need a lot of discretionary memory.

Assumption 3 is better than assumption 2 because it "charges" effectively for garbage collection. A process that does more garbage collection will get less work done, in contrast to the cost assumptions in strategy 2.

We can generalize these strategies for more than two processes. We will spare the reader the formal analysis, but the result is that

$$S_{\text{flex}i} = u\sqrt{\alpha_i/t_{\text{req}_i}}$$

where u is such that

$$\sum_i S_{\text{flex}i} = T$$

5 Implementing the advisor

When one of the authors (Appel) runs a big program on a time-shared machine in ML (a garbage-collected language)[3], he first runs the UNIX command **vmstat** to see how much memory is free, and manually tells ML to use that amount as its heap size. This works quite well, except when other big-memory jobs are subsequently started on the same machine. We wanted to improve and automate this procedure.

One way to do this is to integrate the management of flexible-memory processes into the operating system's page-replacement algorithm. However, this turns out not to be necessary (at least for an initial implementation). We have implemented a "working set advisor" that can tell collectible processes how to size their garbage-collection spaces. Our advisor runs as a non-privileged (i.e. user level) process in Berkeley Unix; it watches the virtual memory statistics using the **vmstat** program, and it responds to advice queries from collectible processes.

A collectible process asks for advice after each garbage collection (every minute or so). It passes the parameters of its current time and space usage:

t_{req} The amount of non-garbage-collection CPU-time since the last advice call.

t_{flex} The amount of garbage-collection time since the last advice.

t_{real} The amount of real (wall-clock) time since the last advice.

S_{req} The minimum (inflexible) memory the process requires; equal to fixed-space plus twice the live data.

S_{flex} The amount of discretionary memory the process is currently enjoying.

The advisor responds with ΔS indicating how S_{flex} should be changed.

The advice parameters can be easily calculated. Before and after each garbage collection, the collectible process makes a system call to learn the cumulative CPU time of the process, and subtracts accordingly. The real (wall-clock) time is also learned from a system call. The time parameters passed are time used *since* the last advice call. Since a garbage collection effectively measures the amount of live data, it is easy to calculate S_{flex} and subtract to get S_{req}.

From these parameters, the advisor can calculate α, the proportionality constant between t_{flex} and S_{flex}^{-1}.

According to assumption 3 (described in the previous section), we should make S_{flex} for each process proportional to $(\alpha t_{req})^{-.5}$. However, some processes spend much of their time idle (waiting for user I/O), or have low priority; these processes should be given less memory because it doesn't matter if they garbage collect a bit more. We can formalize this by changing our interpretation of k, the time between allocations. We have implicitly assumed that k is measured in CPU-seconds; but if we measure it in real (wall-clock) seconds, then by assumption 3 we have

$$S_{flex} \propto \sqrt{\frac{\alpha}{t_{real} - t_{flex}}}$$

We will allow non-garbage-collected processes to run on the same computer, and we will allow parts of the "fixed" (non-collectible) pages of each process to be paged out if they are not in the working set.

We have the problem of realistically determining T, the total amount of memory allocable to the S_{flex_i}. We could say that T is the size of physical memory minus the "fixed" part of all processes (where the "fixed" part of a non-collectible process is the entirety). However, this ignores the fact that the resident set of a process is a subset of its pages; we could use a larger value of T and still not page very much.

What we desire is a value of T that is large enough so that there isn't much garbage-collection overhead, but small enough so that there isn't much paging. We can say that T is just the size of physical memory minus the sum of the resident set sizes of all processes. There is no guarantee that this is optimal, however, since the notion of the working set is a heuristic, and is dependent on an arbitrary parameter Δt. Furthermore, this method of determining T first "optimizes" the paging, then optimizes the garbage collection subject to that amount of paging; we want to consider both simultaneously.

The tradeoff between paging and garbage collection can be adjusted empirically. A virtual-memory manager can measure the paging and garbage-collection overheads for one value of T, can accurately predict the garbage-collection overheads for any value of T, and can adjust T accordingly until the paging and collection overheads balance. We have chosen a simpler approach.

We compute T, the total discretionary memory, as follows: **vmstat** tells the number of free physical pages. To this we add the sum of S_{flex_i}. Then we subtract a "headroom" amount (perhaps a megabyte on a 16-megabyte system) to allow the operating system some free space without paging.

We implement the tradeoff between paging and collecting by decreasing T proportionally to the amount of paging. That is, for each unit of page-outs per second, we decrease T by C_{po} pages. This will decrease the pages used for collecting processes and thereby free some pages for use by paging processes, which will reduce the amount of paging. The page-out rate is available from **vmstat**. Unfortunately, **vmstat** doesn't tell the number of pages in swapped out processes, and there's no good way of computing this short of employing the expensive UNIX **ps** command (which returns process status information); so we will make do by using the exponentially weighted average of page-out activity over the last minute or two as an indication that there are swapped-out processes. Thus,

$$T = S_{free} + \sum_t k_1 e^{-k_2 PO(t)}$$

for appropriate k_1 and k_2.

The advisor is easy to implement. It maintains a socket for advice calls; a process needing advice just connects to this socket and periodically sends requests. The advisor maintains a table of all the t_{real_i}, S_{flex_i}, etc., and updates this table on an advice request. When process j requests advice, the advisor recomputes the constant of proportionality

$$u = \sum_i \sqrt{\frac{t_{flex_i} S_{flex_i}}{t_{real_i} - t_{flex_i}}}$$

and then gives the advice

$$\Delta S = \frac{T}{u} \sqrt{\frac{t_{flex_j} S_{flex_j}}{t_{real_j} - t_{flex_j}}} - S_{flex_j}$$

This advice is a bit inconsistent; we really want the advisor to recompute ΔS_i for all processes simultaneously. But the other processes are not in a convenient

position to receive advice at the moment. As all the processes eventually request advice, the system will evolve to an approximately optimal configuration.

Furthermore, when a process is to be given ΔS additional flexible memory, we will instead give it only a portion of its deserved increase (e.g. $\Delta S/3$); this prevents the first process that asks advice from getting all the resources. Of course, that process will eventually ask advice again, and if there is little competition for memory, it will be given an additional 1/3 of available resources, so that all processes should eventually converge to the optimum sharing. There is an important reason for being conservative about giving out memory shares: the larger a process's heap, the longer it will take before the next garbage collection (and hence the next advice request). If a process is given too little memory, it will soon garbage-collect and seek advice again, at which point the mistake can be corrected. But if a process is given too much, it will be long time before advice is requested and an adjustment can be made.

After giving the advice, the advisor assumes the advice will be taken, and updates $S_{\text{flex}j}$ accordingly. It also computes values $t_{\text{req}j}$ and $t_{\text{real}j}$ consistent with the change in $S_{\text{flex}j}$ and enters them in the table.

A process is expected to tell the advice server when its execution completes. In the case that a process exits abnormally, the advisor will continue to believe that it is using resources. This distorts the advice given to the other processes, but not fatally: the dead process' pages will show up as free pages reported by **vmstat**, and can be given to other processes. Of course, in doling out these pages, the advisor will attempt to save a share for the dead process; but each time it does this, the dead process won't use them and they will show up in the free pages reported by **vmstat**. Thus, over time the number of wasted pages will diminish exponentially. Finally, when a process has not requested advice for several hours, it is assumed dead and removed from the table.

The client program's runtime system must be modified to seek advice. Obviously, the client must keep track of the time at the beginning and end of each garbage collection, and must communicate with the advisor. In addition, we found the following change was helpful: In the generational garbage collector, advice is requested only after a major-cycle garbage collection; but if a really large heap is used, there could be hundreds of minor cycles before another major cycle; therefore we limited the number of minor cycles per major cycle to 200.

6 Performance measurements

We ran our advisor on a 32-megabyte VaxStation-III workstation running Ultrix. The processes requesting advice were implemented in Standard ML of New Jersey[3], which uses two-level generational garbage collection[2]. We use four different benchmark jobs:

pig A compute-bound process that allocates frequently but uses little live data (it solves a real-number partition problem using backtracking).

hog A compute-bound process that uses a lot of memory (the ML compiler).

dog An io-bound process that uses little memory (it solves a real-number partition problem, then sleeps for several seconds).

hippo An io-bound process that uses a lot of memory (it allocates large arrays and sleeps).

All processes allocate new cells very frequently, but some have a larger amount of live data at any given instant.

In the absence of advice, a process could be expected to use an amount of flexible memory proportional to the amount of live data. A conservative heuristic would be to use $S_{\text{flex}} = S_{\text{live}}/2$, and a liberal heuristic would use $S_{\text{flex}} = S_{\text{live}} * 3$. These heuristics don't need an advisor at all, but we implemented them as versions of our advisor so that we could collect statistics in a consistent way.

The cost of the advisor is negligible. Over a 3200-second interval with 4 jobs requesting frequent advice, the advisor took 8 seconds of cpu time, **vmstat** took 7 seconds, and our **vmstat** interface program took 5.

Figure 1 illustrates the performance of the advisor when four processes (a pig, a hog, a dog, and a hippo) are contending for memory, and figures 2 and 3 illustrate the performance of the conservative and liberal heuristics. The top graph in each figure shows the amount of free memory and the amount of paging activity as reported by vmstat over the half-hour interval. The bottom graphs indicate the S_{req} (in black) and S_{flex} (in white) of each process. These are only measured at garbage collections, when advice is taken (the vertical lines), and between these times are only interpolations.

Clearly, the conservative policy is not making good use of the available memory, and is suffering increased garbage-collection overhead (as one can tell by counting the number of vertical lines in plots of the individual jobs, or by looking at Table 1); the liberal policy is thrashing the paging device. The advisor is able to avoid both of these problems.

Job Mix	Policy	t_0	g.c.	faults	sum	real-time
pig+hog+dog+hippo	Advice	1603	275	5390	1892	1961
	Conserv.	1603	477	10	2080	2181
	Liberal	1657	223	138,750	2213	2131
pig+pig+hog	Advice	2626	144	10	2781	2847
	Conserv.	2612	519	10	3132	3229
	Liberal	2620	239	10	2870	2935
hog	Advice	451	98	10	550	578
	Conserv.	455	178	10	633	663
	Liberal	454	107	10	561	587
pig	Advice	1083	7	10	1090	1121
	Conserv.	1075	179	10	1255	1292
	Liberal	1079	65	10	1144	1173
hog+hog+hog	Advice	1393	475	96,455	2100	2032
	Conserv.	1387	538	77,500	2111	2243
	Liberal	1449	362	173,605	2228	2819

Table 1: Benchmark data

Table 1 tabulates the performance of the three algorithms on a variety of job mixes. The "t_0" column shows the non-garbage-collected part of the computation (summed over all jobs in the mix); the "g.c." column shows t_{flex}, the garbage collection overhead; the "faults" column shows the total number of page-outs (estimated). We estimate that the microVax-III takes 2.4 milliseconds of CPU time to process a page fault, so we can compare paging time to garbage collection time using that multiplier. The "sum" column shows $t_0 + t_{flex} + .0024 * faults$. Finally, the last column shows the wall-clock time to completion of the last job in the mix; ideally, this would be equal to the "sum" column, but our measuring techniques are not perfect. All times are in seconds.

Figure 4 shows the breakdown of garbage collection and (estimated) paging overhead (summed over all the jobs in each mix), and figure 5 shows the wall-clock time to completion of all the jobs in the mix, and also shows the non-overhead (computation) time for all jobs. In both of these figures, **A** indicates the advice policy, **C** indicates the conservative policy, and **L** indicates the liberal policy.

In all cases, the advice policy is the best policy. In some cases, it is only marginally better than the conservative policy, and in some cases it is only marginally better than the liberal policy; but it is always better than the best of those two.

Consider a "moderate" policy, which used a fixed ratio of heap size to live data between that of the conservative and liberal policies. Clearly, it would have an amount of garbage collection between that of the liberal and that of the conservative, and an amount of paging between that of the liberal and the conservative. Clearly, for second, third, and fourth job mixes in the table, the moderate policy would not perform as well as the "advice" policy (since there was no paging in these runs); for the first and fifth job mixes, the moderate policy might be competitive with the advice policy. Overall, however, the Advisor seems likely to give better results than any policy that does not take global information into account.

In most cases, the amount of overhead is not very high. This is a consequence of the remarkable efficiency of generational garbage collection. If the benchmarks did not use generational collection, then the garbage collection times would be a bigger proportion of run times, and the differences between the policies would be magnified.

7 Conclusion

In this paper we have explained how garbage-collected jobs do not follow the usual assumptions about fixed working set sizes and can indeed expand or contract their memory demands at will. This presents the opportunity to operating system designers of creating a centralized service that will manage heap requests by garbage-collected processes so as to alleviate memory contention. We have implemented such an advisory service, and our experimental results show that it can substantially improve performance of garbage-collected systems.

Our current advisor implementation runs as a user level process under Berkeley UNIX, but it is clear that it would be advantageous to implement such a process as part of the operating system itself. The advantages of such an approach are three-fold. First, there is reduced overhead in obtaining the advice as part of a system call

(as opposed to using the interprocess communication features of Berkeley UNIX). Furthermore, our advisor does not know whether the amount of free memory in the system is really available or whether a job (who was told it could increase its heap size) is in the process of requesting it. Lastly, our server's advice is just that: a hint to the process; the advisor has no way of enforcing its advice.

Finally, for generational collectors, we believe that the techniques presented in this paper are most relevant to the *youngest* generations. Older generations are collected so rarely that the garbage collector cannot quickly respond to advice about their size, and have a locality of reference that can be exploited by a conventional paging system. Younger generations are collected frequently and exhibit very poor locality, so it makes sense to control their size using advice. We plan to experiment with a hybrid system.

Acknowledgements. Rafael Alonso was supported in part by an IBM Research Initiation Grant and an SRI David Sarnoff Research Center Grant. Andrew W. Appel was supported in part by NSF Grant CCR-8806121.

References

[1] Andrew W. Appel.
Garbage collection can be faster than stack allocation.
Information Processing Letters, 25(4):275–279, 1987.

[2] Andrew W. Appel.
Simple generational garbage collection and fast allocation.
Software—Practice/Experience, 1989.

[3] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–324. Springer-Verlag, 1987.

[4] C. J. Cheney.
A nonrecursive list compacting algorithm.
Communications of the ACM, 13(11):677–678, 1970.

[5] Peter J. Denning.
The working set model for program behavior.
CACM, 11(5):323–333, 1968.

[6] Peter J. Denning.
Working sets past and present.
IEEE Trans. Software Engineering, SE-6(1):64–84, 1980.

[7] Robert R. Fenichel and Jerome C. Yochelson.
A LISP garbage-collector for virtual-memory computer systems.
Communications of the ACM, 12(11):611–612, 1969.

[8] Henry Lieberman and Carl Hewitt.
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM, 23(6):419–429, 1983.

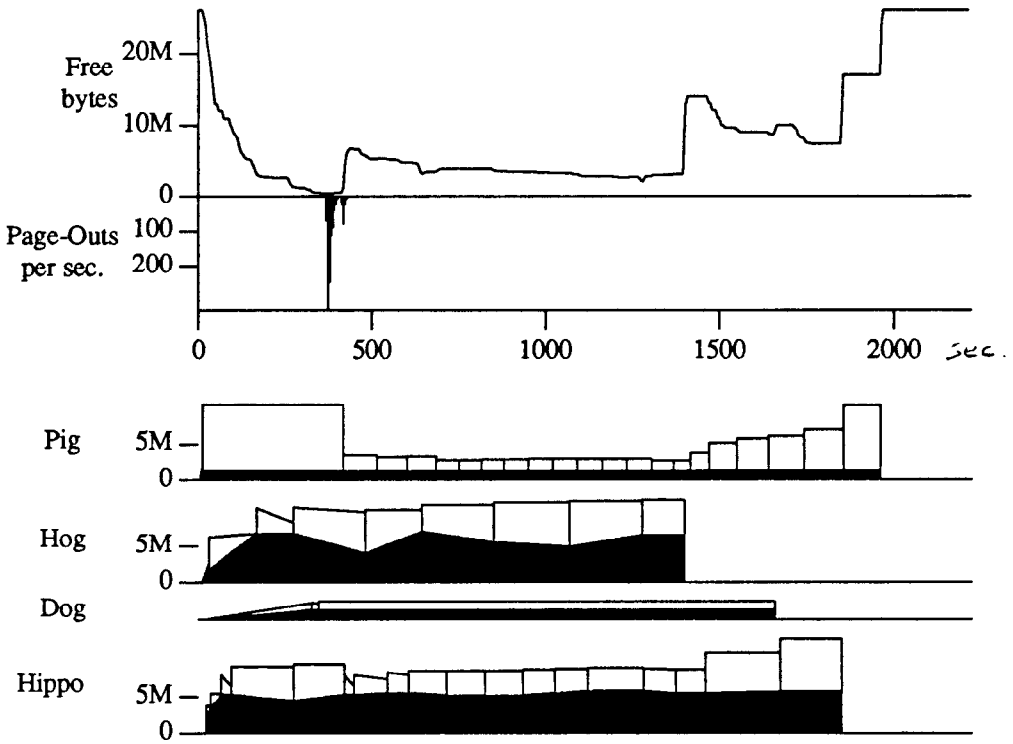


Figure 1. Four jobs using advice.

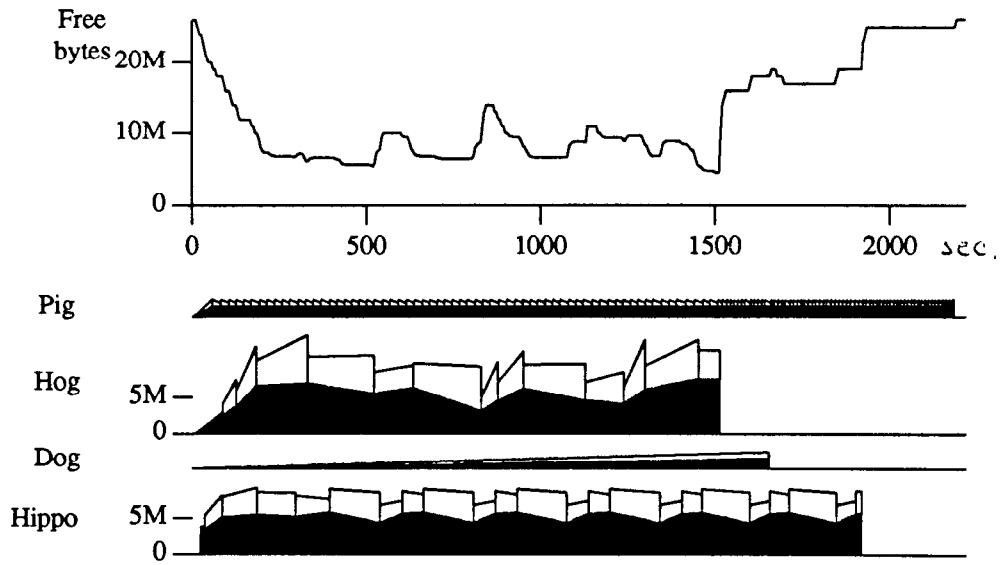


Figure 2. Four jobs using the conservative strategy.

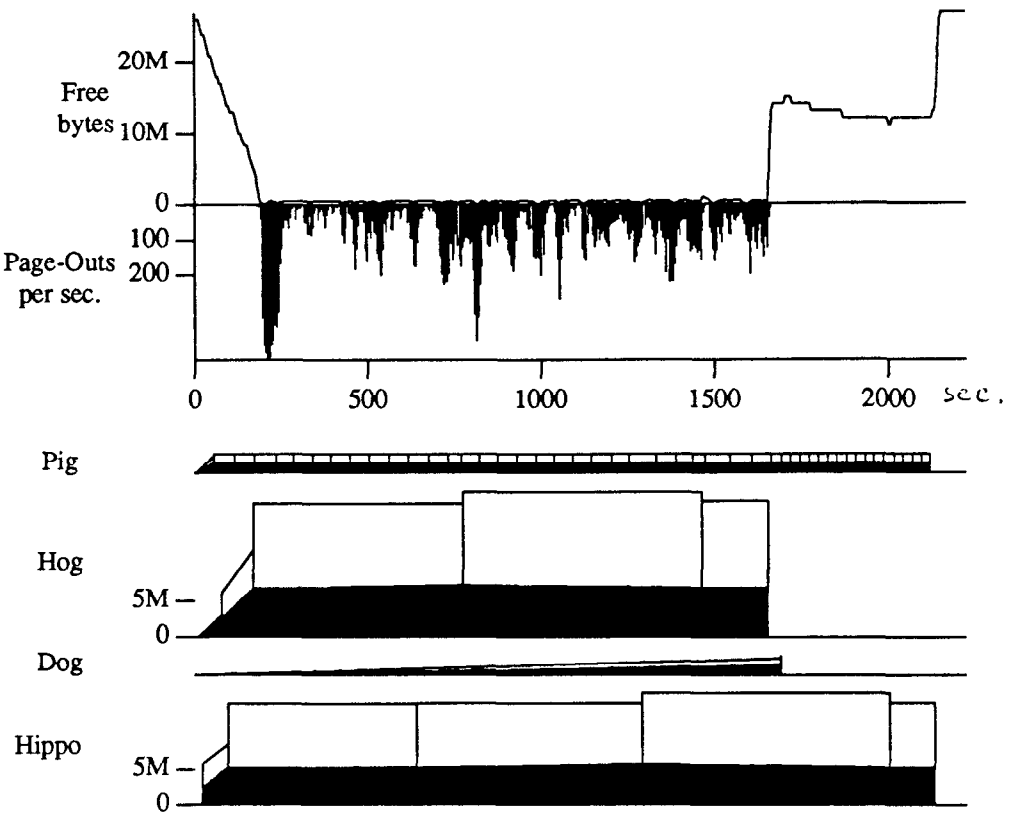
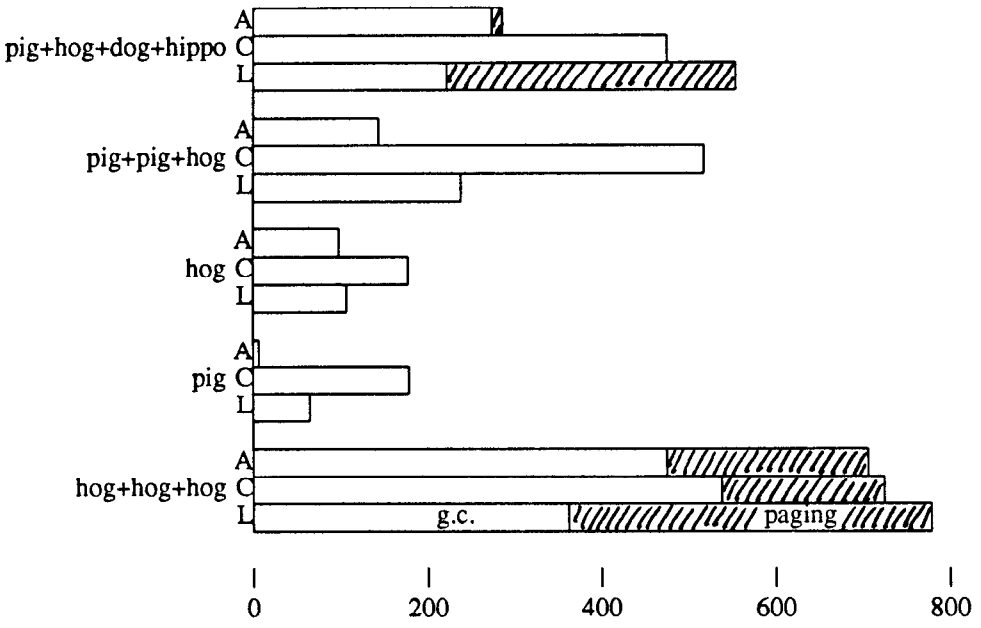
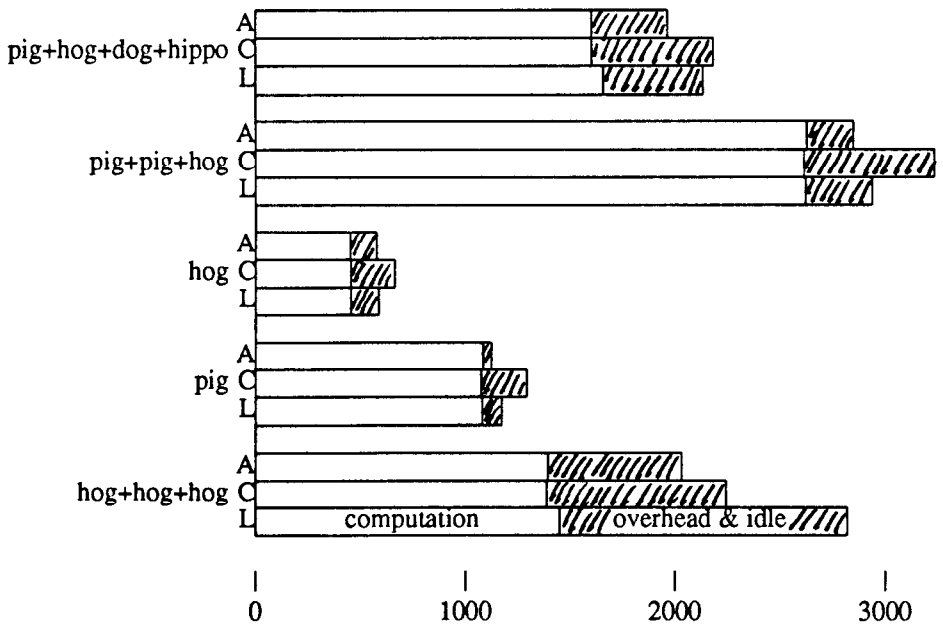


Figure 3. Four jobs using the liberal strategy.



Garbage-collection and Paging (sec)
Figure 4.



Real time to completion (sec)
Figure 5.