GARBAGE COLLECTION VERSUS SWAPPING[*]

Jeffrey A. Barnett[†]
USC/Infromation Sciences Institute

August 25, 2015

1 Introduction

Relationships among dynamic memory management, swapping, and equipment costs are investigated in this paper. Very primitive first-order models are used and all results are in closed form. Therefore, the conclusions apply directly only to very simple-minded situations. However, the phenomena addressed are important. It is suggested that an analysis of more realistic systems employing more accurate models can be beneficial. Of course simulation may then be necessary.

Section 2 introduces the models, their parameterization and justification, and the simplifying assumptions that have been made. Section 3 investigates an optimization from the viewpoint of an applications task. The optimization is on program size to minimize throughput time and the constraints are the penalties introduced by a garbage collector and a swapping mechanism. Section 4 looks at another optimization from the viewpoint of a computation center manager who wishes to provide the best possible services given a fixed equipment budget. The degree of freedom in his choices is the amount to invest in the CPU versus the amount to invest in the swapping device. Section 5 treats the problem of installation optimization where the users control the sizes of their programs and the computation center manager selects the configuration mix. This is the problem of doing the two above optimizations together. It is shown that a stable policy exists, given the simplified models, and that everybody benefits.

[*]Originally appeared in *Operating Systems Review*, 13 (3), 12–17, 1979.
[†]Current email: jbb@notatt.com

The major results are

- Small programs complete execution more quickly than large programs even though they garbage collect more frequently.
- The optimal investment policy for a CPU and swapping device is independent of the cost-effectiveness of the CPU but is sensitive to the cost-effectiveness of the swapping device.
- Much can be gained through cooperation of the users and the computation center manager because they share a common objective—better resource utilization.

2 System Models

The models defined below represent first-order approximations of the system components they describe. Because of simplifying assumptions inherent in such approximations, results are derivable in closed form. It is acknowledged that such results cannot be directly generalized or applied. However, they can be used to indicate a system's gross performance characteristics. The system components modeled below are

- Dynamic memory management — garbage collector overhead as a function of size of the program's memory pool.
- User program — total computation time of the program including garbage collections as a function of its memory utilization.
- Operating system — a simplified time-slicing round robin system and its resource allocation policy to programs as a function of their memory requirements.
- Hardware — performance of the CPU and the swapping device as a function of their respective costs.

As the component models are introduced, an attempt is made to justify the assumptions set forth as reasonable approximations to gross behavior characteristics.

2.1 Garbage Collector Timing

A dynamic memory management scheme is a method of allocating cells from a memory pool to various tasks as the need arises. If the total number of cells allocated exceeds the pool size, then some method must be employed to return to the pool for reallocation those cells no longer needed. There are two general ways of returning cells to the pool: (1) erasure schemes and (2) garbage collector schemes. In an erasure scheme, the program itself is responsible for returning cells to the availability pool when they are no longer needed and can, therefore, be reallocated. In a garbage collector scheme, cells are allocated until the pool is exhausted. When the pool is exhausted, the garbage collector algorithmically and automatically determines, by examining the program's memory, which cells are available for reallocation because the program can no longer reference them. Obviously, a garbage collector scheme is more flexible and easy to use than an erasure scheme. Experience has also shown that there is little difference in overhead between the two approaches. Therefore, it is assumed here that the dynamic memory management scheme employed is a garbage collector.

The runtime overhead exacted by a garbage collector is a function of the number of cells in use and the total size of the memory pool. Experiments by this author and experiments and analysis by others have established that the time required to perform a single garbage collection, γ_1, is a linear function of the number of cells in use, m_u, and the number of cells reclaimed, m_r. (Note, the total pool size is $m_u + m_r$.) The equation for this is

$$\gamma_1 \quad = \quad \alpha m_u + \beta m_r + \gamma_i \tag{1}$$

where γ_i is the time needed to initialize the garbage collector, and α and β are respectively the average incremental times needed to process one more cell of used memory and one more cell of reclaimed memory. The linearity result assumes the mix of "shape," "connectivity," and "blocking" (i.e., the statistical measures of intercell pointer references) of the used structure is relatively constant over different instantiations of the garbage collector.

The analysis and experimentation leading to equation 1 have generally shown α to be significantly larger than β by a factor of ten or so. This is explained by the fact that cells in use must be marked and/or copied, relatively expensive operations, and those reclaimed need not be.

2.2 Program Timing

The model of the program runtime is simplified here by making the following assumptions:

- The total size of the memory pool is fixed.
- The total demand for cell allocation is many times larger than the size of the memory pool. Therefore, there will be many garbage collections.
- Cells are taken from the memory pool at a nearly constant rate.
- Cells become free at the same rate.

Given these assumptions, it is obvious that the number of cells in use, m_u, and the number of cells reclaimed by the garbage collector, m_r, can be treated as constants, i.e., their values do not vary significantly from one garbage collection to the next. The below analysis also ignores initial effects—those occurring before the first garbage collection. This is safe because the total computation is assumed to be large.

The total size, m, of the program is defined by

$$m \;=\; m_u + m_r + m_i \tag{2}$$

where m_i is the number of cells in the program that are not part of the memory pool and therefore not relevant to the operation time of the garbage collector.

Since cells are allocated at a nearly constant rate, a CONS (the cell allocation function) clock can be substituted for the CPU timer. Let λ be the average time between the start of one CONS call and the start of another, not including garbage collection overhead. Them, if a computation needs W cell allocations, the program runtime exclusive of garbage collection is λW. Also, the total number of garbage collections, n_{gc}, and the total time required in the garbage collector, γ, for the computation are $n_{\text{gc}} = W/m_r$ and

$$\gamma \;=\; n_{\text{gc}}\gamma_1 \;=\; W(\alpha m_u + \beta m_r + \gamma_i)/m_r \tag{3}$$

Therefore, the total runtime, T, for the calculation including garbage collection is

$$T \;=\; \lambda W + \gamma \;=\; W(\alpha m_u + (\beta + \lambda)m_r + \gamma_i)/m_r \tag{4}$$

and T decreases with increasing m_r as should be expected.

2.3 Operating System Policy

The modeled operating system employs a simple round-robin scheduler. A program needing CPU service is placed in the run queue. When the program reaches the top of the queue, it is given one quantum of service and if it has not completed its calculation, the program is reinserted at the bottom of the queue. The length of a quantum is q time units and includes the time necessary to swap the program into and out of high-speed memory. It is assumed that q is a task-independent system constant.

Let s be the average (incremental) time needed to transfer one cell between the high-speed memory and the swapping device—delay times for seek, etc., are averaged into s—then the time needed to swap a program of size m into high-speed memory and back again is $2sm$. Further, the execution time per quantum available to the program is $q - 2sm$. Therefore, if a program needs T time units to complete its calculation, then the necessary number of quanta, n_q, to complete the job is given by

$$n_q \;=\; T/(q - 2sm) \tag{5}$$

It is further assumed that the user is charged a fixed amount per quantum of service; that is, the full penalty for large programs is exacted by lack of service and the resulting increased charges because more quanta are used. Note, equation 5 shows increasing n_q for increasing m.

The assumptions made here about scheduling and swapping are extremely old-fashioned. No provision is made for more modern approaches such as multiprogramming, multiprocessing, paging, or multiqueue scheduling. Elaboration of the model to cover such cases would likely rule out the possibility of closed form, easily interpreted results. To be tractable to simulation, another equally rigid model would need to be substituted, restricting the results to one particular set of assumptions in any event. Therefore, a choice has been made to use the simplest set of reasonable assumptions exhibiting the expected general behavior pattern—namely that responsiveness decreases and cost increases as programs get larger.

2.4 Hardware Performance

The only equipment considered here is the CPU and the swapping device. High-speed memory is not discussed because the assumptions made in the previous subsection about the operating system do not allow sufficient freedom to explore the possible savings and costs for a varying core size. The simplifying assumptions made about hardware performance are

- The transfer rate of the swapping device is directly proportional to its cost.
- The CPU's execution time of an average or typical instruction obeys Grosch's law.

The assumption about swap performance is very reasonable if the swapping device is disk because performance enhancement is normally obtained by buying more (not faster) disks, and the improvement is nearly linear because proportionally more transfers and seeks can overlap. Let t_s be the average time necessary to transfer a cell from or to a device costing one unit and let c_s be the actual cost of the swapping device, then by the proportionality assumption, the average time required to transfer one cell to or from the swapping device is

$$s = t_s/c_s \tag{6}$$

The derivations below assume that the quantum size, q, is held constant. Another plausible assumption is that q is proportional to s. However, this would not affect the fraction of the quantum available for program execution. Given a particular s and q, the available fraction is $(q - 2sm)/q$. If both s and q are increased by the same factor, say k, then the available time is $(kq - 2ksm)/(kq)$. But these two quantities are identical. Thus, the only effect of changing the quantum size proportionally to the swap time is to run through the queue at a different rate. If $k < 1$, the system is more responsive to short calculations. The opposite effect occurs if $k > 1$. In each case, the total computational service available to the users is the same.

Assume that t_e is the average execution time for an instruction on a CPU costing one unit and that a CPU costing c_e units is purchased, then Grosch's law states that average execution time on this CPU is t_e/c_e^z where z is an empirically determined constant with wide applicability. However, t_e may vary substantially from one family of computers (e,g., IBM 370s) to another. In prior sections, several timing constants have been defined, i.e., λ, γ_i, α, and β. Underlying each is an associated instruction count. For example, if I_λ is the average number of instructions executed from the initiation of one

CONS call to another, then

$$\lambda \;=\; I_\lambda t_e / c_e^z \tag{7}$$

An equation similar to this one can be written for each timing constant.

3 A User Optimization

Given the above models, an optimal policy for the user is to control the size of his program such that the necessary number of quanta, n_q, to complete the calculation is a minimum. This both maximizes the rate at which he obtains service and minimizes the cost. If a program is made smaller (i.e., m_r decreases), then a larger fraction of each quantum is available for computation. However, the garbage collector will run more often and the program will need more total CPU time. The opposite effects occur if the program is made larger. Thus, there is a tradeoff.

The minimum permissible program size, $m_{\min}$, is $m_u + m_i$. If the program is this size, then $m_r = 0$ (equation 2) and will spend infinite time in the garbage collector (equation 3), thus never reaching completion (equation 4). On the other hand, the maximum permissible program size, $m_{\max}$ is $q/(2s)$. If the program is this size, the calculation will take an infinite number of quanta to complete because each quantum is dedicated in toto to swapping (equation 5). Therefore, the permissible range of program sizes for terminating calculations is $m_{\text{range}} = m_{\max} - m_{\min}$ and hence, $0 < m_r < m_{\text{range}}$.

The optimal program size is derived by determining the optimal amount of free memory, M_r—this is the only control left to the user by the model. This is done by substituting equations 2 and 4 into equation 5, differentiating with respect to m_r and setting the result equal to zero. Also, the definition of m_{range} has been substituted.

$$(\beta + \lambda) m_r^2 / c_{\text{gc}} + 2m_r - m_{\text{range}} \;=\; 0 \tag{8}$$

where $c_{\text{gc}} = \alpha m_u + \gamma_i$, the per garbage collection overhead that is independent of the value of m_r.

The solution of equation 8 is $m_r = c_{\text{gc}}((1 + m_{\text{range}}(\beta + \lambda)/c_{\text{gc}})^{1/2} - 1)/(\beta + \lambda)$. The other root of the quadratic is discarded because $M_r > 0$. Therefore, the optimal program size, M, is $M_r + m_u + m_i = M_r + m_{\min}$

Let $f = M_r/m_{\text{range}}$, i.e., f is the fraction of the possible size range used when $m = M$, the optimal size. Substituting the definition of f into equation 8 gives $1 - 2f = f(\beta + \lambda)M_r/c_{\text{gc}}$. But all terms on the right-hand side are positive. Therefore, $1 - 2f > 0$, or more to the point, $0.5 > f$.

This result is particularly interesting since it disputes the oft-heard claim that programs run faster if given a larger workspace. The claim quickly becomes false because swapping imposes a larger penalty than the garbage collector.

4 A Configuration Optimization

From the viewpoint of the computer center manager, an important problem is balancing the capabilities of the swapping device and the CPU so that user's programs complete their calculations in minimum time, hence, so that the computer center can support the maximum number of users. The method of achieving the optimal balance is to split a fixed pot of money, C, between the purchase of the two devices. The optimal policy derived below assumes that other funds, not part of C, are spent in a fixed way on the rest of the configuration. The amount spent on the CPU is c_e and the amount spent on the swapping device is c_s Therefore, $C = c_e + c_s$.

Assume that a program performs a calculation that requires the execution of I instructions. Then the amount of CPU time necessary, by analogy to equation 7, is It_e/c_e^z. Further, the total fraction of each quantum available for calculation is $(q - 2sm)/q = (q - 2t_s m/c_s)/q = (qc_s - 2t_s m)/(c_s q)$, where m is the program size and equation 6 has been substituted. Therefore, the total system time used by the program (CPU plus swap), y, is the total CPU time divided by the fraction of time available for computation. Thus, $y = (It_e/c_e^z)/((qc_s - 2t_s m)/(c_s q)) = It_e c_s q/(c_e^z(qc_s - 2t_s m))$. Substituting $c_e = C - c_s$. gives $y = It_e c_s q/((C - c_s)^z(qc_s - 2t_s m))$. The optimal expenditure on the swapping device, C_s, is determined by finding a c_s such that $dy/dc_s = 0$. After differentiating and some algebra, it follows that

$$\begin{aligned} C_s &= (t_0(z-1) + (t_0^2(z-1)^2 + 4t_0 zqC)^{1/2})/(2zq) \qquad (9) \\ C_e &= C - C_s \end{aligned}$$

where $t_0 = 2t_s m$, the in-plus-out swap time on a device costing one unit, and C_e is the optimal expenditure on the CPU. Other roots of the differential equation are not relevant because $C_s > 0$. If Grosch's constant, z, is one, then the result has a more appealing form.

$$\begin{aligned} C_s &= (t_0 C/q)^{1/2} = (2t_s mC/q)^{1/2} \\ C_s/C &= (t_s/C_s)(2m/q) = 2sm/q \end{aligned}$$

because $s = t_s/c_s$—see equation 6. Since $2sm/q$ is the fraction of the quantum spent swapping, the optimal policy states (for $z = 1$) that the fraction of the total expenditure invested in the swapping device (C_s/C) should equal the fraction of the total system time spent swapping ($2sm/q$).

It is important to note that the optimal expenditure policy (equation 9) does not depend on t_e, the cost effectiveness of the CPUs in the computer line, but does depend on t_s, the cost effectiveness of the swapping device. A possible interpretation of the result is that the optimal policy dictates purchasing the right amount of swapping capability first and then using whatever funds are left over for CPU purchase.

The above calculations have assumed all programs are the same size, m cells, and all have the same computational needs, I instruction executions. This is of course unrealistic. However, if m and I are selected from *independent* non-point distributions, it would still follow that C_e and C_s do not depend on t_e.

5 An Installation-Wide Optimization

The separate policies developed for a computer center and its users share a common objective—minimize the amount of resources necessary to complete a job. The computer center manager does his optimization by observing the size of the jobs currently using the system and, over time, he adjusts the configuration. The users, on the other hand, quickly observe configuration changes and adjust the sizes of their programs to optimize system behavior on their own behalf.

In theory, this improvement cycle could go on indefinitely with each side looking out only for its own self interests. It is possible that this behavior is cyclic and does not converge in a useful direction. Fortunately, each of the players' selfish activities benefits all concerned because each change reduces the amount of resources necessary to complete a job. Also, there is obviously a lower bound on the amount of resources necessary so the behavior must converge because a monotonic non-decreasing series with a lower bound is guaranteed to have a limit. The limit can be found and the optimal values of m, c_e, and c_s can be determined by simultaneously solving equations 8 and 9 or the analogous equations for a different set of models.

The appropriate time to perform this combined analysis is when the installation is preparing to install a new system or modify an existing system. The computer center manager should query the members of the user community about the projected complexity of their programs, and cooperatively they can arrive at an estimate of the joint optimal policy. Such forethought and analysis is sure to be well rewarded, particularly at installations running programs with large dynamic memory demands such as LISP systems.

Another topic not covered above is control of the quantum size q. If control is left to the computer center manager, its value will be set large enough so that any program can complete its calculation in a single quantum. Thus, there is exactly one swap in and, if necessary, one swap out per job rather than a fixed fraction of the resources being consumed for swapping. The view taken here is that q is a fixed constant representing the responsiveness of the system to short calculations, i.e., if there are u users in the run queue, then a calculation is guaranteed a shot at the CPU in time no longer than uq.