Dynamic Space-Sharing in Computer Systems

L. A. BELADY AND C. J. KUEHNER
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

A formalization of relationships between space-sharing, program behavior, and processor efficiency in computer systems is presented. Concepts of value and cost of space allocation per task are defined and then value and cost are combined to develop a single parameter termed value per unit cost.

The intent is to illustrate a possible analytic approach to the investigation of the problems of space-sharing and to demonstrate the method on sample problems.

KEY WORDS AND PHRASES: space-sharing, storage allocation, memory allocation, storage management, memory management, program behavior, multiprogramming, computer system design, allocation strategies, replacement strategies, demand-paging, time-sharing
CR CATEGORIES: 4.32, 6.2, 6.20

1. Introduction

Multiprogramming aims at increasing computer system productivity by sharing system resources among a number of concurrently active tasks. Such sharing should increase hardware utilization and decrease average turnaround time and cost per job when compared with batch systems. The processing units, for example, may be made available to one task while an uncompleted predecessor engages in some input or output activity. The extent of increased productive resource utilization, however, is dependent on the degree to which control of the system requires the consumption of the same resources.

The greater the number of jobs resident in the system, one would hope, the greater the probability that the active resources will be exploited. Unfortunately, the number of jobs that can be concurrently active is also a function of the passive system resources and in particular of the capacity of high-speed storage. Cost will limit the size of this store. Thus the sum total of storage actually required during the entire execution process by a number of jobs running concurrently will generally exceed the actual capacity of the store. It is this circumstance, herein termed space-squeezing, that limits the number of jobs concurrently resident. (One may choose, however, to artificially limit the number of jobs concurrently resident, for efficiency reasons, prior to reaching this limit.) Because of this space-squeezing, processor utilization, in general, may not be maximized by permitting the entire procedure and data base of any one job to be in core throughout the entire execution process. Instead, portions will be brought into store, perhaps on a demand basis, and be retained there beyond the period of their actual use. Sooner or later they will be removed from store in order to make space for the data or procedure required by another job or, for that matter, by that same job.

The conflict implied by the desire to increase the number of jobs concurrently active, on the one hand, and the need to fulfill the storage demands of individual jobs, on the other, is the root source of many of the difficulties that current multiprogramming and time-sharing operating systems face. Here we examine the problem of optimizing program execution in a space-shared, space-squeezed environment by developing relationships between factors influencing storage strategies. No attempt is made to study the effects or cost of the overhead introduced by multiprogramming and attention is restricted to the following:

—program behavior as a function of allocated storage space.

—program behavior as a function of the number of jobs concurrently resident in store.

—processor efficiency as a measure of the quality of storage management strategies.

—cost of task residence in storage in terms of space-time product.

—value of additional space to a task, competing with other tasks for storage occupancy.

This attempt at an analytic approach to the storage management problem is based on the treatment of system parameters as continuous variables. In fact, they are mostly discrete. However, this difference does not influence the basic characteristic of the relationships developed in the paper. We also assume a fixed system environment with constant characteristics of architecture, coding style, and programming tools. Other general assumptions under which the relationships are developed are:

1. Tasks are executed in a space-squeezed environment.

2. During its lifetime in the system each task has a stationary behavior, such that the average length of the uninterrupted processing intervals is a function of only the storage space occupied.

3. The time necessary to swap a page (i.e. an arbitrary cleavage of informational items into equal sized units) between execution and backup storage including queuing delays is constant and greater than the mean processing interval. We emphasize that this assumption holds for many current systems.

4. The amount of processing time necessary to handle page-swapping is assumed to be negligible.

2. System Parameters

2.1 *Processor Efficiency.* In a real computing system one cannot keep the central processing unit busy on useful work 100 percent of the time. It is therefore convenient to observe the amount of processor idle time and to define processor efficiency η_p, a measure of throughput, as:

$$\eta_p = \frac{\text{elapsed time} - \text{idle time}}{\text{elapsed time}} \tag{1}$$

$$\eta_p = \frac{\text{actual processing time}}{\text{actual processing time} + \text{time spent waiting}} \tag{2a}$$

$$\eta_p = \frac{e}{e + w} \tag{2b}$$

If several tasks contribute to processor efficiency each individual contribution will be indicated as $(\eta_p)_i$. The upper bound to processor efficiency is then:

$$\eta = \sum_{i=1} (\eta_p)_i, \qquad \eta < 1 \tag{3}$$

$$\eta = 1, \qquad \text{otherwise} \tag{3a}$$

where

$$(\eta_p)_i = e/(e + u) \tag{3b}$$

and (u) is the unproductive time delay for program (i).

To optimize throughput, one should maximize the efficiency, equivalent to minimizing idle time. Idling occurs when there is no program, together with its associated data, in the execution store ready for processing. Thus multiprogramming may be viewed as an attempt to increase the probability that at least one program is in main storage and awaiting availability of the processor.

2.2 *Cost of Storage.* When several tasks co-exist in the execution store, the quantity allotted to each will, in general, vary with time. We define the cost C of a storage section occupied by a task as the space-time product:

$$C = \int_{t_0}^{t_1} s(t)\, dt \tag{4}$$

where $s(t)$ is the varying amount of storage occupied during the real time interval (t_0, t_1). It is important to note that the real time occupancy of information in store can be much larger than the amount of processing time given to the associated task during that time interval.

3. Program Behavior Under Dynamic Storage Management

In systems employing demand-based dynamic storage management, only a fraction of the task is placed in the execution store at initial load time. Then, as the need for additional procedures or data is encountered, they are loaded dynamically. This scheme creates an alternating sequence of intervals of execution (e) and delays (u), the detailed pattern of alternation varying with the amount of execution storage allocated to the particular task. Belady [4] has proposed that the variation in this pattern of alternation is also a function of the *locality* of storage references—a basic program property. *Locality* is defined as the *total range* of storage references during a *given execution interval.* (Denning [3] has recently referred to this notion by the term *working set* having chosen to redefine locality as the nonuniform scattering of references over the entire program during execution.) As originally defined and as used in this paper, *locality* (or working set) is an abstract measure of the spread of storage references.

A related but distinct concept, which forms the basis for the remainder of this paper, is that of the *lifetime function.* If one selects any instant during the execution of a program, in a given system, there is a fixed amount of storage space containing a specific set of informational items which are associated with this program. The amount and precise content of the storage space allotted determines the execution time which will be obtained before additional information (i.e. page exception) is required. In any given system the *lifetime function* relates the average length of the execution intervals (e) to the different storage sizes (s) in which the program can be compelled to run. Hence, once one defines an amount of storage associated with a particular program, its capability for execution is also defined. The precise *lifetime function* of particular programs will be influenced by such factors as programming style, problem type, and the system replacement algorithm employed. On the basis of incomplete data, it is conjectured that the effect of specific replacement algorithms on the lifetime function for general program sets will be one of perturbation of local values and not one of radical change in the general relationship. For the present study we consider the average execution interval (e) as a unique function of space for a given task. (See assumption 2 in the introduction. The study of the actual variation in real life programs is one of many possible extensions of this paper.)

One can make some intuitive statements about the characteristics of the lifetime function. If the storage allotment is small, the accommodated informational items quite probably produce but very short execution bursts. This is evident in the extreme case of a one word or even a one page allotment. On the other hand, if the storage accommodates the entire program the average execution interval is on the order of the total running time. One would expect that the execution capability is monotonically increasing between these extremes. The important question arises whether a linear approximation is justified in describing the lifetime function.

The simplest assumption about the referencing pattern of a program is that it is randomly distributed. Let (r)

designate the storage range referenced by the program and (s) be the eoecution storage allotment ($s < r$). Then the expected number of consecutive references to the execution store before new data must be transmitted can be shown to be:

$$e = q/(1 - q) = q + q^2 + q^3 + \cdots \quad (5)$$

where

$$q = s/r. \quad (5a)$$

Thus e is the mean length of an execution interval in units of storage references of a "random" referencing task. *It is apparent that the above function can be represented by a convex curve.*

Real life programs in general do not follow a random distribution in referencing storage. Nevertheless, several investigations have shown that the (e) versus (s) relationship of many programs is nonlinear (Figure 1) and that a section of it can be approximated by:

$$e = as^k \quad (6)$$

where (a) varies with the individual program[1] and (k) has been observed to take values in the vicinity of 2. Insofar as (e) represents the *average* number of consecutive storage references, (a) is constant for a given task.

This nonlinear property was first observed a few years ago when a collection of 7094 programs were interpretively executed and subsequently run on a simulated paging machine with varying page and storage sizes [4]. Later substantiation came while testing code written for the M44/44X system [5] and, independently, for the S360/Model 67. Additional simulation studies carried out at SDC [6] and at Princeton [7] imply the same tendencies.

Figure 1 illustrates the general lifetime function and the approximation offered. Point R represents the storage required to wholly contain the task. It has been observed that in the upper range between R and a point P the average capability for execution changes only slightly when the allotted storage capacity is reduced.[2] P may conveniently be selected as the point at which the second derivative is equal to zero or changes sign. If the task, however, is forced to run with a storage allotment less than P, the e-value drastically decreases in accord with eq. (6).

In view of the cost of fast storage devices it seems desirable that, in a shared environment, tasks not occupy space beyond point P of Figure 1. This clearly represents an area of diminishing returns in which additional storage space does not significantly improve performance. From a different point of view, if the task is highly "compressible", reduction of the space allotment will not cause significant degradation in execution potential. This fact has been corroborated on the M44/44X, where large programs have been observed to be generally more compressible than small programs due to their relatively longer P-R region.

4. Derived Relationships

4.1 *Value.* At any particular operating point $e_1(s_1)$ on the assumed (e, s) curve, the task has a potential for execution (e_1) based on its space share (s_1). If we now increase the space share of this task by allotment of an additional *page frame* (a uniform amount of storage space sufficient to contain a *page*) the potential for execution increases to (e_2), and in general the incremental contribution is:

$$\Delta e = e(s + \Delta s) - e(s) \quad (7)$$

Since the magnitude of (Δe) is dependent upon the local value $e(s)$, the value V of unit space appears to be

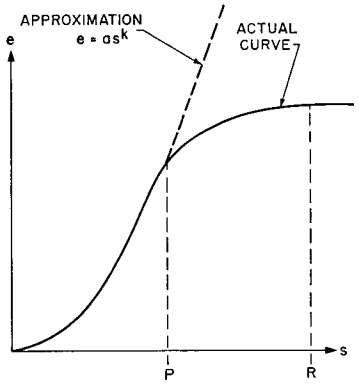


FIG. 1. General lifetime function

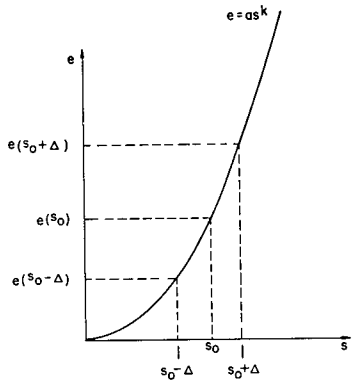


FIG. 2. Convexity and value

To be precise $a = a'a''$ where a' is strictly a program property for a given instruction set and a'' is the conversion factor from storage cycles to real time.

[2] The flattening of the (e vs s) curve in the [P, R] region does not at first glance, appear obvious. This effect comes about from two basic causes. First, (e) is defined as the average of all execution intervals. However, while the program goes through its initial loading phase, some very short execution intervals are created which contribute to the reduction of the average value of (e). Second, it has been observed that executing programs generally issue references to pages of a transient nature. Such pages are used for short intervals of time, and then are abandoned by the program. The program finally accumulates its locality (or working set), and if this set of pages can be accommodated in working storage, no paging is induced until another transient page is required *no matter how much additional storage has previously been available.*

directly related to the first derivative at the operating point of the (e, s) curve,

$$V \sim \Delta e/\Delta s. \quad (8)$$

When the space share of a task is small, an additional s amount of space does not significantly improve the processing potential—the associated e is small, as reflected by the derivative. With a larger space base, the processing increment induced by additional space improves rapidly, also indicated by the derivative. Finally, when the task acquires a sufficiency of space, (Δe) is approximately zero.

In practice it would be quite difficult to dynamically determine the operating points for a set of concurrently executing tasks in order to use "value" to control allocation. However, a less dynamic approach, based nevertheless on the "value" concept, comes from the realization that for any convex curve $e(s)$,

$$e(s_0) < [e(s_0 - \Delta) + e(s_0 + \Delta)]/2, \quad \text{for all } \Delta. \quad (9)$$

Thus for a fixed average store allotment (s_0), more instructions can be executed by running a task half-time in an $(s_0 - \Delta s)$ and half-time in an $(s_0 + \Delta s)$ size allocation than in an allotment of constant size s_0. (See Figure 2.)

4.2 *Biased Replacement Strategies.* The use of this relationship to improve the effectiveness of allocation and replacement strategies is illustrated by a scheme developed on the M44/44X. In this scheme a First-In-First-Out (FIFO) page replacement algorithm has been modified to effect a bias in a simple way [8]. Basically, what occurs is that one of the contending tasks is selected to be favored for an interval of time. The task so selected has all of its pages exempted from consideration by the replacement algorithm for a period of (p) replacements. This privileged state is then passed on to the other members of the competing set in like fashion. The effect achieved is that for some period of time (i.e. the amount required to accomplish (p) replacements) the space share of the favored task will not contract but possibly grow by (Δ) pages. Following this period the task will lose pages at a more rapid rate than if a simple FIFO replacement scheme was employed. This bias accomplishes the desired effect that the page frames are loaded with information of higher marginal value at the cost of removing information of lower marginal value.

The results obtained from a simple FIFO and a biased FIFO (BIFO) approach to replacement can be illustrated via a FLOW picture (see Figure 3). (This output is a chart of the system space sharing as it changes in the time domain. Each line is a description of the storage state during a given interreplacement interval; a new line is generated for each replacement. Identical characters within a line indicate the space share of a particular task.) The first experiments comparing biased[3] and simple FIFO on the M44 indicated a gain of 10 to 15 percent in rate of throughput.[4] Measurements in this area are still being carried out [5].

The important point to note concerning biased replacement algorithms is that the performance improvement obtained is due simply to the general adherence of programs to the convex relationship. Denning [9] has proven that "biasing" will improve the performance of any replacement algorithm.

The magnitude of this improvement depends quite strongly on the proper phasing of the bias with dynamic program space requirements. If one is to maximize storage effectiveness, clearly, one must closely track the dynamic fluctuations in storage required by individual executing programs. Denning [9] has proposed a theoretical approach which claims to be optimal by precisely tracking the dynamic fluctuations of individual program space requirements. Even to approach such an optimal policy, a great deal of dynamically varying information at the individual page level of detail is required. The feasibility and practicality of such an approach remains to be shown.

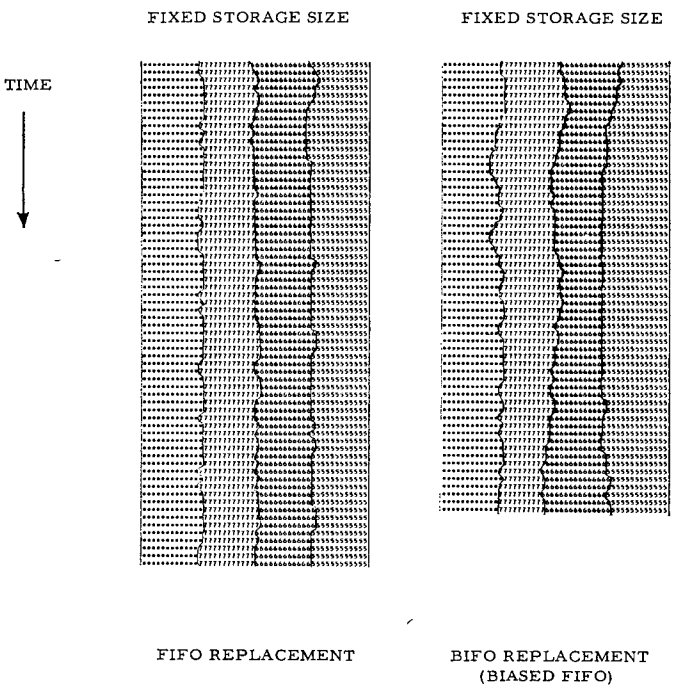


FIG. 3. "FLOW" illustration of FIFO and BIFO replacement strategies

4.3 *Value Per Unit Cost.* A single relationship including the concepts of value and cost as described above arises from the adoption of the processor efficiency as the performance indicator. We call the derivative of (η_p) with respect to (s) the value per unit cost (V_{uc}).

[3] There are better replacement algorithms than FIFO or BIFO. The presented investigation intends only to indicate the pure effect of the biased scheme.

[4] Throughput was measured by comparing the running times of a fixed program mix. In addition, the number of page exceptions generated by both schemes was recorded. The BIFO scheme consistently produced a smaller number of page exceptions than the FIFO scheme when given the same program load.

For a single program

$$V_{uc} = d(\eta_p)_i/ds. \tag{10}$$

More explicitly, from eqs. (3b) and (6)[5]

$$\frac{d(\eta_p)_i}{ds} = \frac{\partial e}{\partial s}\left[\frac{1}{u\left(\frac{e}{u}+1\right)^2}\right] = \frac{akus^{k-1}}{(as^k+u)^2}. \tag{11}$$

Our selection of the name value per unit cost (V_{uc}) is justified as follows. The notion of efficiency (η) as defined here is the amount of useful processing per unit time. The above defined V_{uc} is then the incremental amount of processing per unit time and per unit space. But the unit-time/unit-space product, by definition, eq. (4), is unit cost. Therefore V_{uc} is indeed "value per unit cost." In other words V_{uc} represents in a single quantity the incremental additional processing capability, normalized to the cost of the incremental space and time for the requesting task.

It is of interest to find the space allotment at which, for a given task, V_{uc} is maximum (i.e. $d^2\eta/ds^2 = 0$.) This defines the operating point around which additional space is most valuable to the task in order to contribute to system efficiency. Differentiating eq. (11) with respect to s,

$$\begin{aligned}\frac{d}{ds}\left[\frac{d(\eta_p)_i}{ds}\right] &= \frac{d}{ds}\left[\frac{akus^{k-1}}{(as^k+u)^2}\right] \\ &= kua\,\frac{u(k-1)s^{k-2} - a(k+1)s^{2(k-1)}}{(as^k+u)^3}.\end{aligned} \tag{12}$$

Solving s_{max} yields

$$s_{max} = \left(\frac{u}{a}\cdot\frac{k-1}{k+1}\right)^{\frac{1}{k}}. \tag{13}$$

Hence the space allotment defining the operating point of best space use of the program to increase system efficiency is a function of u (i.e. the delay in obtaining information from an auxiliary store) and (a), a program property.

To illustrate the effects of backing storage characteristics, represented here by the waiting time (u), we plot $d\eta_p/ds$ against (s) as in Figure 4.

It is clear from this illustration that the magnitude of the maximum V_{uc} increases with decreasing (u). On the other hand, the location of this maximum decreases (smaller s) as (u) decreases.

The quantity (u/a) may be considered the ratio of delay to processor speeds, since the value of (a) is proportional to the length of time of uninterrupted processing periods. For a given task then, the (u/a) value relates the access time to auxiliary information to the processor speed, and large (u/a) values, for example, characterize a relatively slow auxiliary store operating in conjunction with a relatively fast processor system.

[5] Following assumption 3 in the introduction, the second term of the equation $\frac{\partial \eta}{\partial s} = \frac{\partial \eta}{\partial e}\cdot\frac{de}{ds} + \frac{\partial \eta}{\partial u}\cdot\frac{du}{ds}$ is zero.

It is an important observation to note that, for any given task the maximum of (de/ds) and the maximum V_{uc} are not necessarily located at the same storage size. If we call the locality (or working set) at $(de/ds)_{max}$ the parachor[6] of the task, then the allotment of space to any single task might correspond to a space allotment smaller than the parachor, depending upon values of (u/a) and k.

4.4 *System Relationships.* In Sections 4.1, 4.2, and 4.3 several descriptions of system characteristics and program structures were defined and derived. The purpose of the present section is to develop system relationships using these descriptors and to provide some intuitive insight into the meaning of such relationships via a geometric representation.

If one integrates V_{uc}, eq. (10), with respect to (s) the result is the task contribution to efficiency within the specified space allotment. The formula suggests that the first few pages allotted to a task only slightly increase the task contribution to efficiency. It is around the maximum V_{uc} that individual pages contribute the most to processing. Assignment to a task of at least this amount of space is thus advisable.

Whether one should increase the space share of a task beyond its maximum V_{uc} depends largely on the characteristics of other storage resident tasks. To illuminate this problem, let us assume that two tasks occupy the execution storage. Sooner or later a new page is required by one of the tasks, and a page frame must be emptied to accommodate it (i.e. the contents of a page frame is to be swapped). The proper policy then is to avoid increasing the space share of the task with the smaller instantaneous V_{uc} whether this task itself is the requestor for the new page or not. The idea behind this is that one would reduce the area under the V_{uc} curve by a smaller amount and hence take away a smaller potential efficiency contribution (Figure 5).

We do not presently know how to implement a viable space-sharing algorithm to operate in the manner described above, mainly because instantaneous characteristics of tasks are generally not known to the system. Nevertheless, one can see and understand from a system viewpoint the desirability of having tasks share the available space in such a way that they operate around their maximum V_{uc}, which implies a limit to the number of tasks concurrently resident.

If we again integrate eq. (10) with respect to real time

$$E(s,t) = \sum_i \int_{t_0}^{t}\int_{s_0}^{s} \frac{d(\eta_p)_i}{ds}\,ds\,dt, \tag{14}$$

the result is indicative of the amount of virtual processing time (E) to be obtained with the varying space allotment over the period of real time specified in the limits of integration.

Figure 6 presents an irregular object in a three-dimensional space, the dimensions being space, real time, and

[6] Parachor is an arbitrary term which came into use during the development of T.S.S./360 to designate the amount of core required for a task to achieve "reasonable" processing.

V_{uc}, whose volume is thus the processing time obtained. The object graphically portrays the history of a task under the space control mechanism of the operating system and the result in terms of the specified dimensions.

At time (t_0) the task is introduced into the execution store at (s_0). From this point the task acquired additional space (Δs), in this example, for the sake of simplicity, instantaneously. With this additional space the task is able to use the processor according to the η_p versus s relationship for the particular program.

If we allow this task to continue occupying space in the store for some period of real time (Δt), the task will (1) gain and/or lose space according to its own needs and subject to the dictates of the operating system, and (2) decrease and/or increase in terms of value/unit cost depending upon how much virtual processing time it accumulates relative to its size and real time expenditure.

The irregularity of the object is brought about by (1) and (2) above. The upper surface of the solid is a representation of mean values as derived from observed program properties without taking into account the local variations around these means. In real life the value of an additional page is not only the function of occupied space but also that of time. However the $d\eta_p/ds$ function here is considered constant for the running time of the task.

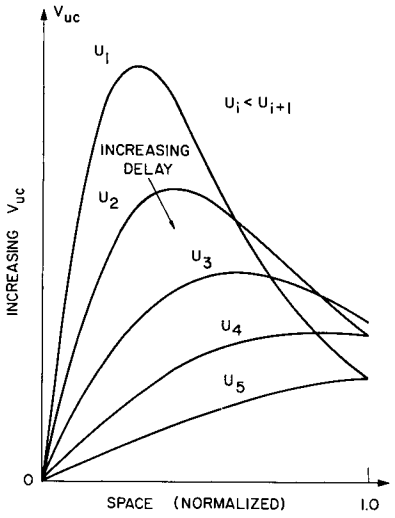


FIG. 4. V_{uc} as a function of space share

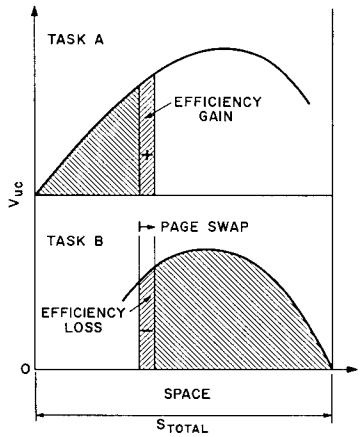


FIG. 5. Optimal page-swapping policy

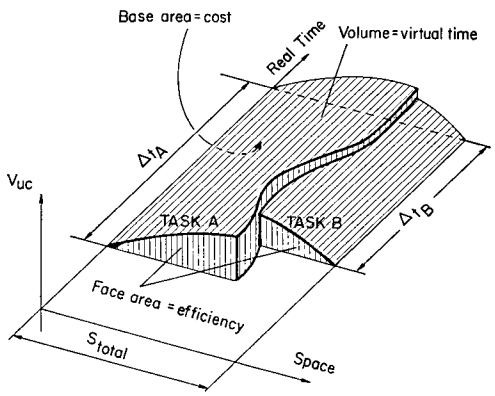


FIG. 6. System relationships

5. Replacement Strategies—An Application of System Relationships

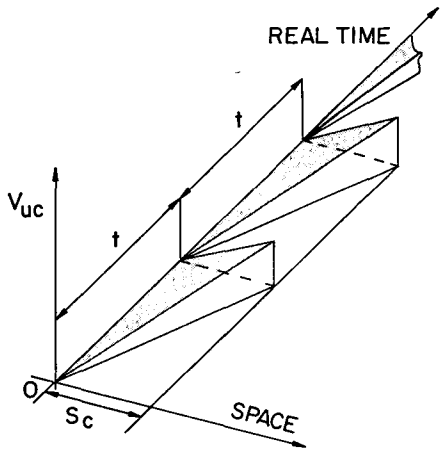
The relationships and concepts developed up to this point can be used to gain insight into, among various things, storage replacement strategies.

As an example we discuss a possible strategy whereby tasks are introduced into execution with a single page of information. They are then permitted to grow via demand-paging until they cause a preset maximum number of page frames to be occupied in the execution store. When this occurs, a task will be prohibited from further execution and its page set will be purged from the store. Subsequently, the task will be rescheduled and at the appropriate time reinitiated and allowed to grow again as described above. This cycle is repeated until the task is completed. With this artificial restriction on individual space shares, it is possible to temper the influence of large tasks on the rest of the system participants.

If we now turn attention to Figure 7(a) where this strategy, let us call it strategy (1), is illustrated in terms of the quantities developed earlier, we see a solid repeated twice (in the general case (n) times). Each solid represents the growth of the task until it consumes an amount of space (S_c) and its instantaneous removal. For the sake of simplicity, we have assumed that the space allotment increases linearly with real time, and that the V_{uc} versus (s) function is linear. This results in a pyramid (see Figure 7). Let us further assume that the time needed to achieve a size S_c is t.

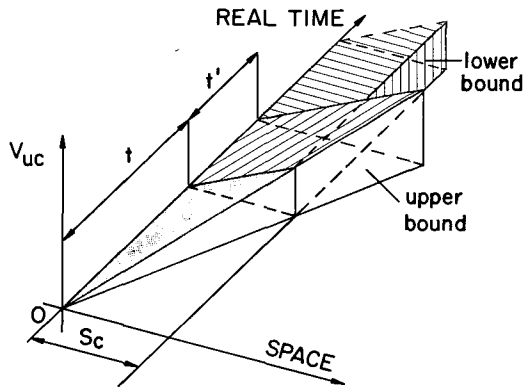
In order to process the task one has to produce an accumulated volume proportional to the total processing requirement (E) of the task. By assumption this is equivalent to repeating the pyramid n times. However, another technique is possible, let us call it strategy (2) (see Figure 7(b), which permits the continuation of processing, without reloading, beyond the point of removal of strategy (1). The imaginary extension of the pyramid can be of many different shapes. One would nevertheless expect that it falls between the following two bounds: the continuation of the pyramid by a prism (lower bound) and the continuation by an extended pyramid of the same slope (upper bound). (See Figure 8.)

By purely geometrical reasoning, it can be shown that the cost ratio (R_c) of the first to the second strategies falls between the limits: $3n/(1 + 2n) < R_c < n^{\frac{1}{2}}$ where (n) is the number of times the task is reintroduced as indicated by the shaded portion of Figure 8. With these assumptions and for $n > 1$ strategy (1) is increasingly costly from the storage point of view as (n) increases relative to strategy (2).



STRATEGY (1)

FIG. 7(a). Storage replacement strategy (1)



STRATEGY (2)

FIG. 7(b). Storage replacement strategy (2)

This result graphically illustrates why strategies attempting to restrict the space share of large tasks in order to reduce their effect on the overall system will fail unless the amount of storage space available is far greater than that normally required. Strategies which prolong the existence of large tasks in the execution store effectively increase the system load and in general yield the inverse of the desired result. This fact would be enforced even more strongly if consideration had been given to overheads in general and channel activities in particular, since purging and reinstatement of the task (n) times implies (n − 1) additional information transmissions.

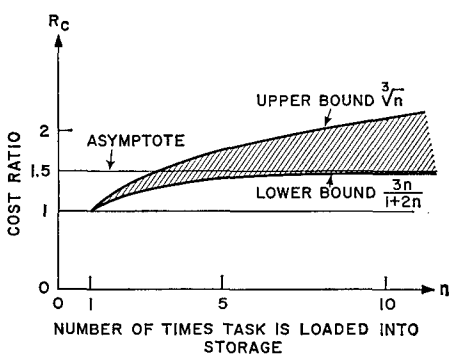


FIG. 8. Cost ratio versus (n)

6. Summary

A formalization of some of the relevant relationships which exist between program characteristics and system properties have been presented. In particular the parameters of processor efficiency (η) and the cost of storage (C) were chosen as the relevant system properties. The program attributes deduced included: (e), the mean length of uninterrupted execution periods and the parachor of a task.

The basic nonlinear relationship between the processor execution time (e) and allotted storage space (s), called the lifetime function, has been discussed in detail and an approximation offered for the region where programs operate with less than a sufficiency of space.

The discussion is extended to dynamic space-sharing in a multiprogramming environment where the concepts of value and value per unit cost (V_{uc}) of space allocation on a task basis are formally developed. V_{uc} represents in a single variable a powerful indicator which can be used to investigate overall storage management designs.

To provide insight into the meaning and use of these global relations, the paper concludes with an interpretation and discussion of the implications of such relations and a global analysis of two storage replacement strategies.

Acknowledgments. The authors would like to thank their colleagues at both the Thomas J. Watson Research Center and the IBM Time-Sharing Development Group for numerous discussions and suggestions in the preparation of this paper. Thanks are also due to the anonymous individuals who refereed this paper and provided many constructive suggestions.

RECEIVED MAY, 1968; REVISED DECEMBER, 1968

REFERENCES

1. RANDELL, B. AND KUEHNER, C. J. Dynamic storage allocation systems. *Comm. ACM 11*, (May 1968), 297-306.
2. CODD, E. F. Multiprogram scheduling. *Comm. ACM 3*, (1960), 347-350.
3. DENNING, P. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323-333.
4. BELADY, L. A. A study of replacement algorithms for a virtual storage computer. *IBM Syst. J. 5*, 2 (1966), 78-101.
5. O'NEILL, R. W. Experience using a time-sharing multiprogramming system with dynamic address relocation hardware. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30, Thompson Book Co., Washington D. C., pp. 611-621.
6. FINE, G. H., JACKSON, C. W., AND MCISSAC, P. V. Dynamic program behavior under paging. Proc. ACM 21st Nat. Conf., 1966, Thompson Book Co., Washington, D. C., pp. 223-228.
7. VARIAN, L. C. AND COFFMAN, E. An empirical study of the behavior of programs in a paging environment. Presented at the ACM Symposium on Operating System Principles, Gatlinburg, Tenn. October 1967.
8. BELADY, L. A. Biased replacement algorithms for multiprogramming. Rep. NC 697, IBM Thomas J. Watson Res. Center, Yorktown Heights, N. Y., Mar. 1967.
9. DENNING, P. Resource allocation in multiprocess computer systems. Doctoral Diss., MIT, June 1968.