Estimating the fault rate function

by T. Jennings

Paging activity can be a major factor in determining whether a software workload will run on a given computer system. A program's paging behavior is difficult to predict because it depends not only on the workload processed by the program, but also on the level of storage contention of the processor. A program's fault rate function relates storage allocation to the page fault rate experienced while processing a given workload. Thus, with the workload defined, the fault rate function can be used to see how the program's storage allocation is affected by varying levels of storage contention, represented by varying fault rates. This paper presents a technique to represent program workloads and estimate the fault rate function, and describes how these results can be used in analyzing program performance.

Programs usually do not reference their storage uniformly; they exhibit locality by concentrating storage references on subsets of their storage. A program's degree of locality determines how much processor storage it needs to run efficiently. The storage reference behavior of programs, sometimes simply referred to as *program behavior*, has been studied for nearly three decades. Much of the early attention was in the development of storage management algorithms for operating systems,[1-3] while other early efforts yielded restructuring techniques that improve program locality.[4,5] More recent work has recommended design techniques to enhance locality.[6] *Reference strings* (the string of memory references that the program generates during its execution) have frequently been used in these studies, as have component connectivity graphs. These representations focus on references to program modules and their data areas. With the increasing size of processor memories and the corresponding growth in application program size over the last decade, the storage required for program modules and their local data areas can be a smaller contributor to a program's total storage requirement than large data areas, which vary in size with environment parameters such as network configuration. For this reason, this study treats the storage for large data areas (referred to as *data pages*) separately from the storage for program modules and their local data areas (referred to as *base pages*).

Many programs process multiple workloads or transaction types. The storage reference behavior of a program depends on the particular workloads it is processing and the rate of arrivals for each workload. To characterize a program's demand on the system's storage and paging subsystems, one must know the arrival rates for each workload and the amount of storage contention on the processor. The *fault rate function*, which maps the page fault rate experienced by the program to the program's storage allocation, is a succinct representation of the program's demand on the system's storage and paging subsystems for a given workload mix. The fault rate function is closely related to the *lifetime function*, which plots storage allocation versus the average time between faults. These two functions were used during the development of virtual storage systems to compare the efficiency of different memory management policies.[1-4,7] Algorithms for generating the fault rate function from a program's reference string are well documented.[1,7] The technique presented in this study is unique in that it allows the fault rate function of a program to be approximated using information available during the program's design, rather than deriving the function from a measured reference string obtained after the program has been implemented. Instead of using the fault rate function to compare operating system memory management policies, it is used here to assess the efficiency of program design alternatives and to project the amount of resident storage required by the program to run efficiently in different system environments.

The first section contains a brief description of related work in this area, and provides definitions for important concepts in program behavior. Following that is a description of the modeling approach, including the important simplifying assumptions, and then separate sections describe base page and data page analysis, the algorithm for constructing the fault rate function, and examples to illustrate how this technique has been applied in analyzing program performance at IBM's Raleigh, North Carolina, Networking Laboratory.

Background

On most operating systems (including IBM's Multiple Virtual Storage [MVS], virtual machine [VM], and Virtual Storage Extended [VSE]), processor storage is divided into fixed-size pieces called *frames*. The operating systems allow multiple programs (i.e., jobs, virtual machines, or partitions) to run concurrently. Each program has virtual storage that contains its modules and data areas, and the operating system divides each program's virtual storage into fixed-size pieces called *pages*. Virtual storage pages and processor storage frames are the same size. In order to run, each program must have some of its virtual storage pages allocated to processor storage frames. The set of a program's pages assigned to processor frames at a given time is referred to as its *resident set*. Frequently, the total virtual storage for all currently running programs exceeds the amount of processor storage available, so the operating system uses a memory management scheme to manage the assignment of pages to frames. The operating system relies on the characteristic of programs to exhibit *locality of reference*, which Madison and Batson describe as "the experimentally observed phenomenon that, for relatively extended periods of time, a program references only some subset of its . . . virtual address space."[8]

If, as a program is running, it tries to access a page that is not assigned to a frame, a *page fault* occurs and a *page-in* is necessary to bring in the necessary page. The operating system uses a page replacement policy to select a page from all of the currently resident pages for replacement. MVS, VM, and VSE systems use a *least recently used* policy;[9,10] they select a page that has not been referenced for a long period of time for replacement, since it is highly probable that the page will not be referenced again soon. If the page that is selected for replacement has been modified since it became resident, a *page-out* is necessary to write the contents of the page to auxiliary storage. Page-ins usually require separate input/output operations, but page-outs are less frequent and the operating system can combine multiple page-outs per input/output operation.[9] Page-ins, therefore, are a more important consideration in evaluating program performance. Note that the program's page-in rate is equivalent to its page fault rate.

Many techniques have been identified to enhance a program's locality of reference by restructuring its virtual storage layout. (Ferrari offers a good overview of these techniques.[5]) These methods require analyzing the program's reference string. References in the string are associated with blocks of storage, and the blocks are depicted in a restructuring graph, where the blocks form the graph nodes and the edges between nodes have a number indicating the value of placing the two blocks on the same page. The edge values are computed by a restructuring algorithm, and a revised assignment of blocks to pages is generated by a clustering algorithm. Smith[6] identifies simple heuristic clustering techniques that use component size, frequency, and connectivity information, rather than a reference string. Ferrari indicates that "most of the programs for which restructuring is convenient are not very data-dependent, and for them the effectiveness of restructuring algorithms should not be expected to be data-dependent either."[5] By treating large data areas (data pages) separately from program modules and their local data areas (base pages), this paper presents a method of studying programs whose reference behavior is data-dependent.

While there is no related work on data page analysis explicitly, data pages can be viewed as a database residing in the program's virtual storage. Database reference behavior has been the subject of several studies. Rodriguez-Rosell shows that the database references generated by a pool of interactive database users exhibits "weak" locality in referencing blocks of a database.[11] Kearns and DeFazio, however, show that the reference behavior of individual batch database application programs differs from the behavior for "interleaved" requests of multiple interactive users, and that locality of reference is exhibited.[12] Easton describes a stochastic model for database references, which uses page reference and residence probabilities to compute the average number of pages in the first level of a storage hierarchy (the buffers) and the miss ratio given a working set window size.[13]

The assumption is that pages referenced will not be paged out, allowing us to consider only the unique page references.

Modeling approach

Because the number of virtual storage pages required by a program may depend on external parameters (e.g., system configuration, network configuration, etc.), this modeling approach separates the set of virtual storage pages into two groups: a set of base pages, whose number is independent of the environment, and a set of data pages, whose number is dependent on the environment. To estimate the fault rate function, it is first necessary to calculate the reference rate (in references per second) for each page of virtual storage. This is accomplished by dividing the base pages and data pages into subsets, such that the pages within each subset have the same reference rate (or reference probability). The reference rates are calculated from workload arrival rates and data page reference information. The subsets are then ordered by reference rate, and the subset sizes and reference rates are used to generate *demand points* that form a "piecewise linear graph"[1] approximating the fault rate function.

The determination of workloads is an important first step in this modeling approach. A workload can represent an individual transaction type or a class of transaction types. The workload selection should identify the dominant processing of the program. The following criteria should be considered when selecting workloads.

- Frequency. The most frequently processed transaction types should be identified.
- Processing requirements. The transaction types that contribute most to the program's resource usage should be identified.
- Path variations. A conditional variation in a path that adds significant processing and that references a significant number of additional modules can be treated as a separate workload. Distinguishing "secondary" workloads in this manner is one way to increase the granularity of the base page analysis.
- Data reference behavior. Transactions that reference large data groups or that generate a significant number of data group references should be identified.

Since workload selection is a manual process, the number of workloads should be less than about 20, in order to minimize complexity. (Our models have included up to ten workloads.) For programs with many more transaction types, it would be desirable to have workloads represent transaction classes instead of individual transaction types where possible.

This modeling approach is useful for estimating the *phase behavior* of a program. According to Denning, phase behavior is exhibited by a program when, "over extended periods, the program concentrates all references in small, fixed subsets of pages. Each maximal such period is called a *phase*; the associated pages constitute the *locality set* of the phase." Transition behavior is exhibited when, "in the intervals between phases, the nicely localized phase patterns are broken; the reference pattern is discontinuous, unordered."[1] Programs spend a large majority of their time in phase behavior.[8] Phase behavior can be associated with steady state processing, where items of work arrive at a steady rate.

As mentioned earlier, virtual storage subset sizes and reference rates are used to generate an approximation for the fault rate function. Pages with low reference rates have a low reference probability, and are more likely candidates for replacement (page-out) than pages with high reference rates. In generating the fault rate function, a *least frequently used* (LFU) page replacement algorithm is assumed, although most operating systems use a least recently used (LRU) algorithm.[9,10] For analyzing phase behavior, LFU replacement should closely approximate LRU replacement.

This paper also assumes that pages referenced by a workload item will not be paged out before the processing of the workload item is completed, an assumption noted by Smith as suitable for short-duration, transaction-type processing.[6] This assumption allows us to consider only the unique page references in processing a single workload item; subsequent references will not cause page faults. Easton makes a similar distinction between primary and secondary references: "the first access to a page after a 'long' period of inactivity" is a primary reference; "the references that follow, within a 'short' time, are called secondary references."[13] Also similar is Ferrari's distinction between critical and noncritical references.[5]

The next section describes how base pages are divided into subsets, and how the page reference rates are calculated.

Base page analysis

The important simplifying assumption made in base page analysis is that the local data used by a module are highly likely to be allocated on the same page as the module. With this assumption, it is only necessary to count module references for the base pages. (One way to relax this assumption would be to use an instruction trace for each of the program's workloads and distinguish between local data and module references.)

In processing an item of work from a workload, the flow of control of the program will pass through several modules, and will likely reference several of the base pages. Some modules, such as service routines, will be used by multiple workloads. Viewing the modules of the program as a set, the subset of modules used by one workload is likely to intersect with the subset used by another workload.

The first step in base page analysis is to partition the modules into subsets based on the workloads that reference them. This requires only a list of the modules used in processing each workload.

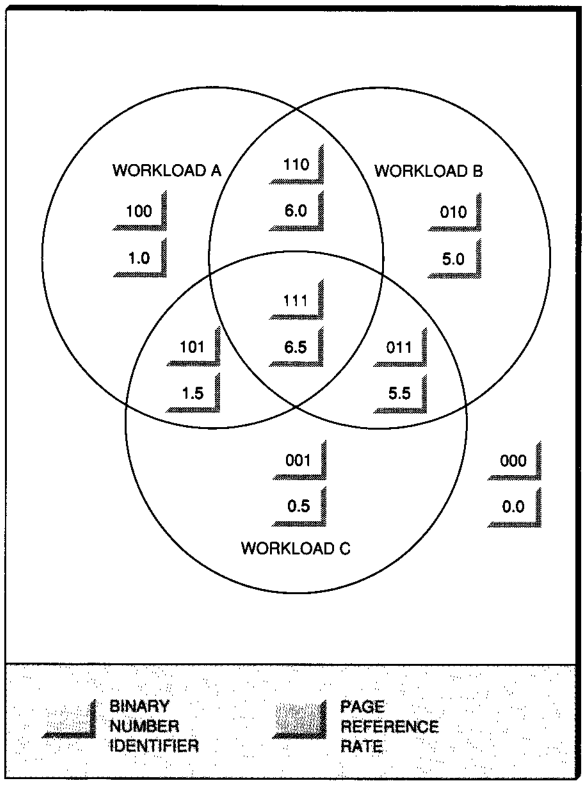
Once this is done, the mapping of modules to base pages needs to be represented. This can be obtained from a virtual storage map, consisting of a list of module names, the starting addresses, and the module lengths. (Ideally, the virtual storage map of the program should be taken when no data pages have been allocated by the program, so that only base pages show up in the map.) Mapping modules to pages is trivial: page identifiers for 31-bit addressing systems are five bytes in length, and are simply the first five bytes of the module starting addresses for modules within the page. For 24-bit addressing systems, the page identifier is three bytes long. Modules will frequently span page boundaries (this is determined by the length of the module). For this analysis, it is assumed that the entire module is referenced in processing each workload, implying that every page that module spans is referenced when the module is referenced. (To relax this assumption, one could use an instruction trace and record the offsets that are referenced within each module.)

The next step is to determine which workloads reference each base page. The first step identified which workloads reference each module, and the second step identified which modules are on each base page. For a given base page, it is referenced by each workload that references a module contained on that page.

The final step in the base page analysis is to determine the reference rates for each of the base pages. Recalling the assumption that only unique references need to be counted, each base page is counted once per workload item that references it. Given mean arrival rates for each of the program's workloads, the reference rate for a base page is simply the sum of the arrival rates for the workloads referencing that base page.

This analysis technique is a subsetting process, where the base pages in each subset are referenced by the same combination of workloads. If there are n workloads for the program, the subsetting process yields 2^n subsets of the base

Figure 1 Base page subsets



pages, some of which will probably be empty. The subsets can be enumerated using a binary numbering scheme, as shown in Figure 1. This figure is an example with three workloads and eight base page subsets. The three-digit identifier indicates which of the three workloads references the subset. The following workload arrival rates are used to calculate the page reference rates for the subsets:

- Workload A = 1.0 items per second
- Workload B = 5.0 items per second
- Workload C = 0.5 items per second

Given a storage map, a list of modules used by each workload, and workload arrival rates, this technique lends itself quite easily to automation.

To illustrate the base page references generated by a given workload, consider the subset sizes shown in Figure 2. When an item of work from workload A is processed, each page within base page subsets 100, 101, 110, and 111 are referenced, resulting in 74 unique page references.

This section described a method of subsetting base pages which yields subsets containing pages with the same reference rates. As will be shown in the next section, data pages are treated differently from base pages. The goal, however, is still to divide the pages of virtual storage into subsets containing pages with the same reference rates.

Data page analysis

Data pages contain major control blocks or data areas, and vary in number depending on external parameters (system configuration, network configuration, etc.). The number of data pages may be quite large, and may even dwarf the number of base pages for some programs. The set of data pages for the program may contain several types of control blocks. A *data group* is made up of a set of control block types that are allocated on the same data pages. For example, if two control block types are the same length, then the program's storage management scheme might allocate those control blocks from the same data pages. The two control block types would then comprise a data group.

To determine the number of data pages that a program will have, it is necessary to know how many of the major control blocks will be needed based on external parameters. For each data group, the number of data pages can be computed based on the storage requirements for each of the control block types within the group for the given environment.

In addition to estimating the size of the data groups, it is also necessary to estimate the number of unique references to the groups by items from each workload. As with base pages, only unique references will be considered. The number of references to a data group for a workload item could be quite large. For example, if in processing a workload item the program performed a sequential search of a linked list, the number of unique references to the data group containing the linked list would be equal to the average number of list elements searched.

To extend the base page analysis example from the previous section, Table 1 shows two data groups for the example, giving their size in the number of pages in each data group and the number of unique references to control blocks for each workload.

When the program references a data group, one control block within the data group is referenced, implying that one data page of the data group is referenced. Some control blocks within the data group may be referenced more frequently than others. If this is the case, a discrete distribution can be specified to represent the reference behavior to a data group. This discrete distribution defines the relative size and reference probabilities for *data subgroups*. References to a subgroup are assumed to be distributed uniformly among the pages in the subgroup. The discrete distribution can be made more granular by increasing the number of subgroups.

A convenient way to specify this discrete distribution and define the data subgroups is with a size distribution matrix and a reference distribution matrix (see Table 2). The size distribution table on the left contains the percentage of the data pages for a data group that falls within each subgroup. In the example, subgroup 1-a consists of 50 percent of the data pages that are in data group 1. The reference distribution table on the right contains the percentage of the references to the data group that fall within each subgroup. The table shows that when data group 2 is referenced, 20 percent of the time the reference will be to a page within subgroup 2-b. To simplify automating this analysis technique, each data group is given the same number of subgroups. Notice that data subgroup 2-c has zero percent of the data pages from data group 2, and gets zero percent of the references to data group 2. In other words, data group 2 has only two subgroups: 20 percent of the pages are referenced 80 percent of the time, and 80 percent of the pages are referenced 20 percent of the time. Zero entries in the size and reference distribution matrices indicate that the subgroup is empty and is not used.

The following input is therefore required to compute the reference rates to the program's data pages.

- Workload arrival rates
- Data group definition
  —Data group sizes
  —Number of unique control block references to each data group for each workload item

Figure 2 Base page subset sizes

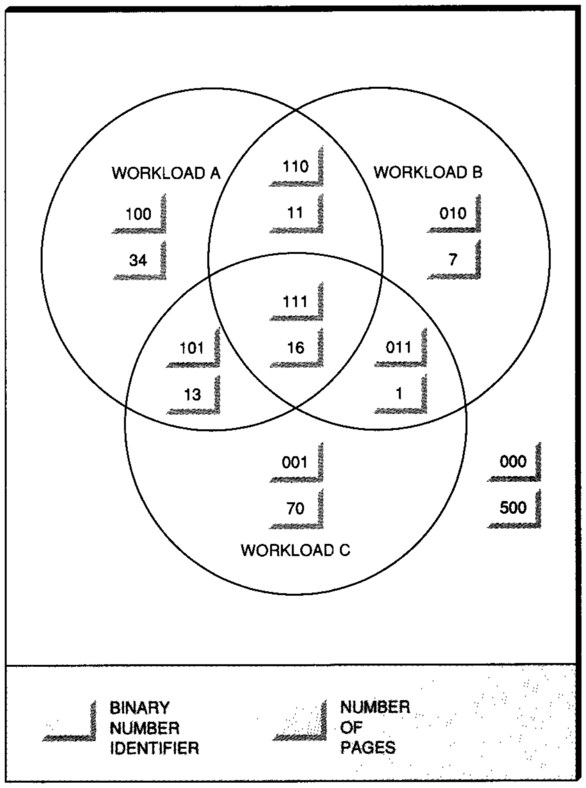


Table 1 Data groups

Data Group	Size	Unique References		
		Workload		
		A	B	C
1	80	4	15	1
2	50	0	2	6

- Data subgroup definition
  —Size distribution
  —Reference distribution

Given the above input, the calculation of subgroup sizes and subgroup reference rates is simple and intuitive. For compactness, the formulas are expressed below mathematically. Given the following where:

w	the number of workloads
AR_w	the workload arrival rate
g	the number of data groups

Table 2 Data subgroup size and reference distributions

Size Distribution				Reference Distribution			
Data Group	Subgroup a	b	c	Data Group	Subgroup a	b	c
1	50%	40%	10%	1	40%	20%	40%
2	20%	80%	0%	2	80%	20%	0%

Table 3 Data subgroup size and reference rate

Size				Reference Rate			
Data Group	Subgroup a	b	c	Data Group	Subgroup a	b	c
1	40	32	8	1	0.795	0.497	3.975
2	10	40	0	2	1.040	0.065	0.000

GS_g	the data group size
$UR_{w,g}$	the unique references by workloads to data groups
s	the number of data subgroups
$SGSD_{g,s}$	the subgroup size distribution
$SGRD_{g,s}$	the subgroup reference distribution

we compute:

$SGS_{g,s}$	the subgroup size (in pages)
$SGRR_{g,s}$	the subgroup reference rate (in references per page per second)

using the following two formulas:

$$SGS_{i,j} = GS_i \cdot SGSD_{i,j}$$

and

$$SGRR_{i,j} = \sum_{k=1}^{w} \frac{(AR_k \cdot UR_{k,i} \cdot SGRD_{i,j})}{SGS_{i,j}}$$

Table 3 shows the subgroup sizes and reference rates for the example. This table shows the size in pages and the reference rate in references per page per second for each subgroup in the example. The data group sizes are shown in Table 1, and the size and reference distribution data are shown in Table 2.

There is a complication to consider when computing subgroup reference rates. Recall the assumption that only unique page references need to be counted. If a workload item generates more page references to a data subgroup than there are pages in the subgroup, then some pages will be referenced more than once for a given workload item. To account for this, the number of references to a data page within a subgroup is not allowed to exceed one for a given workload item. The subgroup reference rate calculation is modified as follows.

$$SGRR_{i,j} = \sum_{k=1}^{w} AR_k \cdot \min\left(\frac{(UR_{k,i} \cdot SGRD_{i,j})}{SGS_{i,j}}, 1\right)$$

The subgroup size and reference rate matrices (SGS and $SGRR$) define a subsetting of the program's data pages into subgroups, such that each page within a subgroup is referenced at the same rate (or with the same probability).

Constructing the fault rate function

Using the subset sizes and reference rates from the previous two sections, an estimate of the fault rate function can be generated using the following algorithm.

1. Sort the subsets by increasing reference rate, discarding all empty or unreferenced subsets. The sum of all subset sizes would equal the virtual storage allocation of the program. Some of the storage may be unreferenced by

the workloads under consideration, however, and unreferenced pages are prime candidates for "stealing." The program should encounter page faults only when there is enough storage contention on the system to remove unreferenced pages. For the purposes of this analysis, unreferenced pages will be ignored.

2. For the first demand point (f, s), where f is a fault rate and s is a storage allocation, f is zero and s is sum of the sizes of the subsets that appear in the sorted list (the referenced subsets). To run without page faults, the program must have all of the pages referenced by its current workload mix resident.
3. To calculate each subsequent demand point from the current demand point, use the next subset in the ordered list and:
   - Increase f from the current demand point by the total reference rate for the next subset, calculated by multiplying the page reference rate by the size of the subset.
   - Decrease s from the current demand point by the size of the next subset.

The final demand point in the fault rate function has a storage allocation s of zero pages, implying that all pages referenced must be paged in. The fault rate f will be equal to the sum of the total reference rates for each subset. Demand points with high fault rates, where almost all pages have to be paged in, are likely to underestimate the fault rate, since the assumption that pages are resident throughout processing of a workload item is optimistic in that case.

Table 4 shows the demand points for the example.

Once the demand points have been calculated, they can be plotted to approximate the fault rate function. Figure 3 shows the graph for the example.

The curve in Figure 3 consists of a set of line segments joined end to end, with each segment representing a subset of the virtual storage pages. For each segment, the slope is the negative inverse of the page reference rate, and the segment length is directly proportional to the number of pages in the subset. Since the graph was built using a list sorted by reference rate, and since the slope of each segment is inversely proportional to the reference rate, the slope gets closer to zero and the curve gets flatter as the fault rate increases. The interesting part of the curve lies to the left of the curve's "knee," which in this case appears at around 150 faults per second. Beyond the knee, a decrease in the program's storage allocation requires a relatively large increase in the fault rate.

Figure 3 Fault rate function

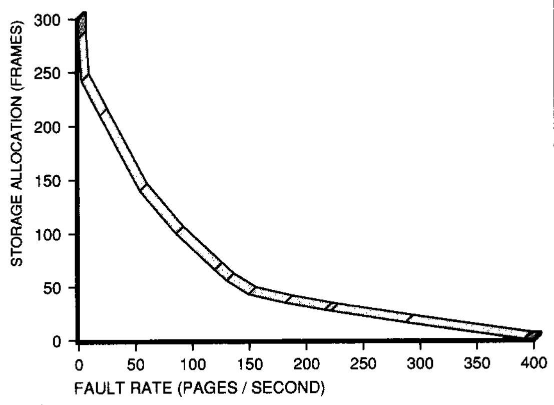


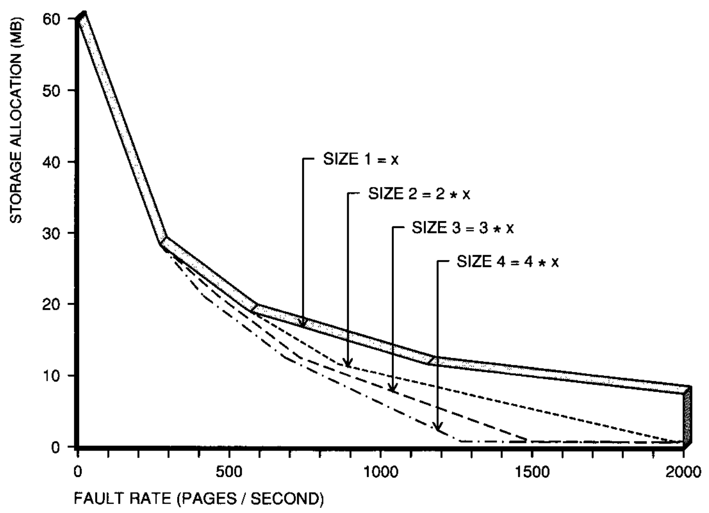
Table 4 Demand point calculation

Subset	Reference Rate	Size	Fault Rate	Storage Frames
			0.0	282
2-b	0.065	40	2.6	242
1-b	0.497	32	18.5	210
001	0.500	70	53.5	140
1-a	0.795	40	85.3	100
100	1.000	34	119.3	66
2-a	1.040	10	129.7	56
101	1.500	13	149.2	43
1-c	3.975	8	181.0	35
010	5.000	7	216.0	28
011	5.500	1	221.5	27
110	6.000	11	287.5	16
111	6.500	16	391.5	0

Examples

This section describes two examples of how this modeling technique has been applied in analyzing

Figure 4 Varying hash table size



program performance in IBM's Raleigh Networking Laboratory. Both examples were taken from a study of a very large network with high workload arrival rates. The program under study monitors the state of sessions in the network (a *session* is a conversation between two logical units in an IBM Systems Network Architecture environment). The program maintains control blocks for each logical unit and each session in the network, and therefore has large data groups that are comprised of these control blocks. The program has three workloads that are reflected in the model: one for the start of new sessions, one for the end of existing sessions, and one for individual traffic items flowing on the sessions.

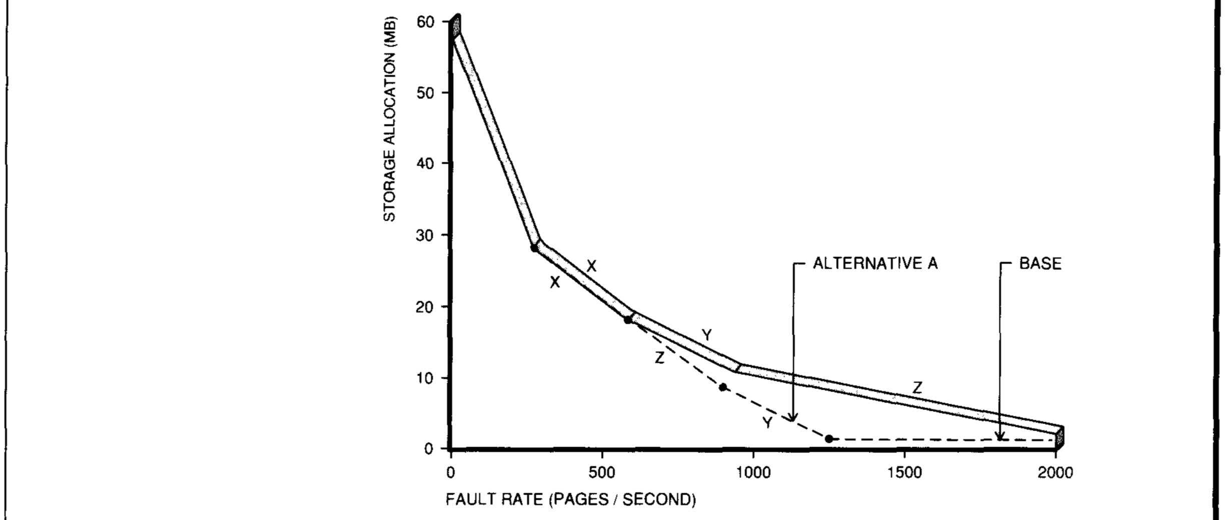
Since the program needs to support very large networks, the large pools of control blocks are accessed using a hashing technique to minimize the search time for locating a particular control block. The hash tables are allocated at program initialization, and an initialization parameter is used to control the size of these tables. With larger tables, fewer hash "collisions" will occur, and fewer control blocks are referenced in the search for a particular control block.

Figure 4 shows an estimate of the program's fault rate function for a given network size and workload arrival rates. Four different values of the initialization parameter are shown. Each large line segment in the plots represents one of the large pools of control blocks. As the hash table sizes increase, the number of unique references made to the data groups by individual workload items decreases, since fewer control blocks are searched. As a result, the reference rates for those data groups decrease. In the fault rate function, this behavior is shown by the slope of the line segments. With increasing hash table size, the line segments associated with the affected data groups have a steeper slope; in other words, fault rate activity is not affected as greatly by those data groups.

The following points should be considered in interpreting graphs of a program's fault rate function.

- The fault rate function is useful in estimating the program's resident storage requirement. In this example, if a fault rate of 300 pages per second (which could correspond to page movement be-

Figure 5 Design alternative A

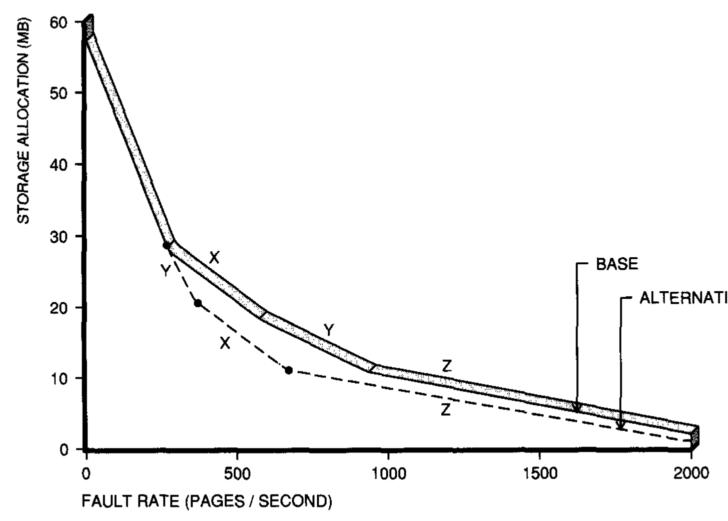


tween central and expanded storage) was acceptable for this system, the fault rate functions in Figure 4 indicate that the minimum central storage requirement would be about 27 megabytes. The hash table size parameter has little effect at this fault rate, since the first large data group is not affected by the parameter.

- The fault rate function is also useful in estimating the fault rate experienced by the program if its storage allocation is constrained. In Figure 4, if the program is restricted to using 10 megabytes of storage, its fault rate is estimated at around 1500 pages per second with size 1 of the hash table size parameter. If the parameter is increased by a factor of four (size 4), the program's fault rate is estimated at around 800 pages per second, for a 47 percent reduction in the fault rate. The fault rate function should be helpful in performance management exercises to guide storage isolation decisions. For MVS systems with expanded storage, the fault rate can be interpreted as the rate of moving the program's pages from expanded storage to central storage. For storage allocations beyond the capacity of processor storage (central plus expanded), the fault rate can be interpreted as the program's page-in rate from auxiliary storage.
- An improvement in the fault rate function brings it closer in to the x and y axes. The worst case for a fault rate function is when the program does not exhibit any locality of reference, but instead references its pages randomly. For the worst case, the fault rate function would consist of a single line segment that extends from the y axis intercept to the x axis intercept.
- The number of referenced data pages in this model dwarfs the number of referenced base pages. Only 1 megabyte of storage was required to keep the referenced base pages resident in Figure 4. In our studies, referenced base pages were usually beyond the knee of the curve. For programs with a large amount of data in virtual storage, this might often prove to be the case. This underscores the value of separating data page analysis from base page analysis. Data pages can be analyzed during the initial design stages, whereas base page analysis is usually done once implementation is underway (since a virtual storage map is needed).

The first example showed the effect of a parameter change, illustrating that this type of model is useful both during program development (to assess the performance of a design) and after the program is installed (for performance management and capacity planning use).

Figure 6 Design alternative B



The next example uses the same model and the same workload arrival rates. In this example, two design alternatives are compared to see their effect on the program's fault rate function. Alternative *A* improves the locality of data group *z* by moving the referenced control block to the top of its hash table collision chain. This would have the effect of shortening the average collision chain search depth, and would reduce the number of unique references made by the three workloads to data group *z*. Figure 5 shows how this design change affects the program's fault rate function. Data groups *x* and *y* are not affected by this change. Data group *z* has moved in its relative location on the fault rate curve, because after the design change, the reference rate per page for data group *z* lies between the reference rates for data groups *x* and *y*.

Design alternative *B* (see Figure 6) is similar to alternative *A*, except that it improves the locality of data group *y* (with the same mechanism of placing the referenced control block at the top of its collision chain). With alternative *B*, the reference rate per page for data group *y* is less than that for data group *x*, so its relative position in the fault rate curve changes. Note that the vertical component of line segment *y* does not change with design alternative *B*, because the size of data group *y* has not changed. Only the horizontal component changes, because the page reference rate for data group *y* has been reduced.

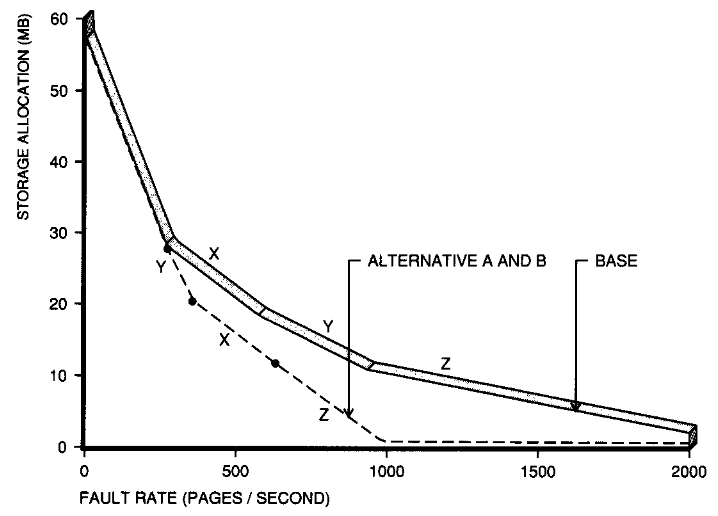
Figure 7 shows the effect of implementing both design alternatives. The change in length and slope of the line segments for data groups *y* and *z* remains the same when both alternatives are implemented. In Figure 6, with alternative *B* implemented, data group *x* was below data group *y*; this relative position does not change if alternative *A* is implemented. Likewise, in Figure 5, data group *z* was below data group *x* with alternative *A* implemented, and it remains below *x* in Figure 7.

With every locality improvement, note that the fault rate function gets closer to the *x* and *y* axes.

Concluding remarks

This paper has presented a method of estimating the fault rate function of a program given the arrival rates for each of the program's workloads and a representation of the page references made by each workload. The projected fault rate function can be used in estimating the program's stor-

Figure 7 Design alternatives A and B together



age behavior for varying workload mixes and in assessing program design alternatives. The process of grouping program pages into subsets reduces the complexity of treating each page separately. Easton made a similar observation in "grouping pages with similar behavior into classes."[13] To develop a modeling approach that would handle programs with large virtual storage requirements, the simplification offered by this subsetting process was instrumental, as was the separation between base page and data page analysis.

Methods for relaxing some of the assumptions in the base page analysis are identified in the paper. In our studies, however, the data page analysis has been the more important and enlightening, since data pages comprise most of our program's virtual storage in large network environments. Also, data page analysis can be done during the design phase, when the data structure and access techniques are first defined. As the development cycle progresses and more detailed data are available, the granularity of the data page analysis can be increased. Base page analysis is usually done during implementation, since a virtual storage map is needed. Optimizations resulting from the base page analysis would normally require changes to the low-level design or changes to the clustering of modules to pages.

The base and data page analyses and construction of the fault rate function are straightforward and can be automated easily. Inputs to the base page analysis (the virtual storage map and the list of modules used by each workload) can be generated using standard mapping and trace programs. Inputs to the data page analysis (the data group sizes, the number of unique references to each data group for each workload, and the optional subgroup definition) are determined manually.

An extension to this modeling approach is worthy of mention. If two or more programs are being studied together, it is possible to combine their fault rate functions into a single "aggregate" fault rate function. To do this combination, the base page and data page analysis would be done separately for each program to subset the programs' virtual storage by reference rate. The subsets for all of the programs would then be combined and sorted, and the aggregate fault rate function constructed using the same algorithm as for a single program.

Validating a program's projected fault rate function with measurement data is difficult because it requires measuring carefully regulated workload arrival rates on a system with controlled storage contention. (This difficulty is similarly noted for program lifetime data.[14]) One model validation exercise has been conducted on an IBM 3090* processor running the VM/XA* operating system with 5000 simulated conversational monitor system (CMS) users exercising a program development workload. A program with four steady workloads of its own was added to the system, and the system performance was monitored for 30 minutes. A demand point consisting of the program's measured page-in rate and resident set size was then compared to the program's projected fault rate function. The measured demand point was within 13 percent of the equivalent demand point on the projected fault rate curve. The accuracy of the projected results using this modeling technique will obviously depend on the accuracy and granularity of the program's representation.

The fault rate functions generated by this modeling approach have been useful in predicting the effect of program design and tuning changes. They have been especially useful in predicting when paging problems are likely to occur (i.e., the knee of the fault rate curve). Unless a program's projected fault rate function is validated with careful measurement data, however, one should avoid attributing a high degree of accuracy to its demand points, especially for high fault rates.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. P. J. Denning, "Optimal Multiprogrammed Memory Management," in *Current Trends in Programming Methodology III*, K. Chandy and R. Yeh, Editors, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978), pp. 298–322.
2. P. J. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engineering* **SE-6**, No. 1, 64–84 (January 1980).
3. J. R. Spirn, *Program Behavior: Models and Measurement*, Elsevier/North-Holland Publishing Co., New York (1977).
4. J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems," *IEEE Transactions on Software Engineering* **SE-2**, No. 1, 54–62 (March 1976).
5. D. Ferrari, "The Improvement of Program Behavior," *Computer* **9**, 39–47 (November 1976).
6. C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley Publishing Co., Reading, MA (1990), pp. 475–524.
7. G. J. Chastek and J. P. Kearns, "Efficient Calculation of the Space-Time Performance of the Working Set," *Performance Evaluation*, Netherlands **10**, No. 4, 281–294 (December 1989).
8. A. W. Madison and A. P. Batson, "Characteristics of Program Localities," *Communications of the ACM* **19**, No. 5, 285–294 (May 1976).
9. T. Beretvas, "Analysis of Processor Storage and Paging Configurations," *Proceedings of the Computer Measurement Group Conference on Computer Performance Evaluation* (December 1986), pp. 772–788.
10. W. Tetzlaff, "Paging in the VM/XA System Product," *Computer Measurement Group Transactions* **66**, 55–64 (Fall 1989).
11. J. Rodriguez-Rosell, "Empirical Data Reference Behavior in Data Base Systems," *Computer* **9**, 9–13 (November 1976).
12. J. P. Kearns and S. DeFazio, "Locality of Reference in Hierarchical Database Systems," *IEEE Transactions on Software Engineering* **SE-9**, No. 2, 128–134 (March 1983).
13. M. C. Easton, "Model for Database Reference Strings Based on Behavior of Reference Clusters," *IBM Journal of Research and Development* **22**, No. 2, 197–202 (March 1978).
14. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1984), pp. 179–221.

Accepted for publication December 15, 1991.

Thad Jennings *IBM Networking Systems, 3039 Cornwallis Road, P.O. Box 12195, Research Triangle Park, North Carolina 27709.* Mr. Jennings is a staff programmer in the NetView* product performance department at the Networking Laboratory. He received a B.A. in mathematics from Winthrop College, Rock Hill, South Carolina, in 1983 and an M.A. in computer science from Duke University in 1985. Since joining IBM in 1985, he has worked on modeling, analyzing, and measuring the performance of NetView on several operating systems.

Reprint Order No. G321-5475.