Rough Draft

Nov. 18, 1983

Lynn Wheeler

K83/281
San Jose Research
276-1783
SJRLVM1/WHEELER

ABSTRACT

The CP component of VM has always attracted a great deal of attention in the area of performance. A great deal of effort is currently being spent in benchmarks to accurately identify bottlenecks and other performance problems. A much better perspective of current performance bottlenecks can be obtain by comparing the current VM environment with CP/67 at its best in 1973. The similarities and contrasts between the current situation and that of ten years ago can be very valuable for placing current performance problems in perspective.

EARLIEST OS PERFORMANCE

When I first became acquainted with CP/67 in late January of 1968, it worked but relatively slowly. It could be considered a second pass of working on a prototype and the small group working on it were still spending most of their time implementing function and debugging.

The original performance tests I did running O/S under CP were not promising. The elapsed time to run O/S jobs under CP was three to four times what it took them to run stand alone. The O/S system was an MFT system that had been highly optimized though. By hand re-ordering of almost every card in the stage-two sysgen, I hand increased the through-put of the MFT by a factor of two to three (for the particular class of jobs that we were running) over a vanilla MFT system.

Investigating the CP performance problems closer, it became apparent that almost all of the increased overhead was because of CP path-lengths to simulate the 360 machine environment. Attempting to address this, over the next year I worked on reducing the CP implementation path-lengths. It was during this period that I also invented the implementation technique that is currently commonly referred to as Fastpath. I also implemented BALR linkages for the most commonly used subroutines (prior to that all linkages were done via the SVC interface).

The aggregate results of the work at the end of the year is that instead of jobs running three to four times as long as stand-alone, they were running 1.13 to 1.3 three times as long. With optimized path lengths, my invention of Fastpath, and BALR linkages, I had reduced the CP path-length to run an OS virtual machine by over 95%. In most cases, the performance optimization changes were also directly applicable to CMS virtual machines.

CP/67 2.3

During this period, CP/67 was being installed at a number of other locations (my location had been the third after Cambridge and Lincoln laboratory). Path-length is one type of performance problem, but CP/67 was also prone to page thrashing if a large number of virtual machines were active. One of the people at Lincoln came up with a modified scheduler which limited the dispatch list to a number proportional to real storage. This went a long way towards eliminated the page thrashing problem at the time. Unfortunately, it was not very load sensitive. The limit values chosen happen to correspond to the CMS intensive (primarily FORTRAN compiles) that was representative of the workload at Lincoln Labs.

By this time, Denning had published his working set paper. I had already been thinking about the problem for some time. I reviewed Denning's work but dismissed it as not entirely appropriate for the CP environment. Instead, I invented the current moving cursor page replacement algorithm. And then to handle the working set estimation problem, I invented both the algorithm which uses as a basis of working set a function of the average resident pages. The page replacement algorithm was an approximation to a global LRU. Average resident pages result in a working set that is proportional to the overall system activity rather than individual activity.

In order to implement this sort of function, I also had to invent the concept of using feedback algorithms to control system resources and the specific algorithms that implemented the controls. From the feedback algorithm work, I branched off into extending feedback control algorithms into managing dispatching priority (at the time, CP used a very gross round-robin scheme). Feedback control algorithms, path-length optimization, and fastpath are now pretty well accepted concepts.

A separate issue was also starting to show up. As CP/67 was maturing and the number of installations was growing, more and more code was being added to the nucleus. By today's standards, relatively small increases in code size was having a detrimental effect on the total available pageable memory (real storage was typically 512k or 768k). To address this problem, I came up with the implementation for pageable CP nucleus modules.

I was also continuing to work on path length performance. After the initial round at looking at OS performance under CP and the generalized path length problem, I turned my attention to CMS. After a careful analysis of how CMS worked, I identified CMS DASD I/O as a significant contributor to overall path length. To address this problem, I came up with a synchronized I/O interface with a stylized CCW chain. This change cut the path length overhead involved in CCW translation and eliminated the privilege instructions and interrupt simulation required for the asynchronous I/O interface. This change resulted in a 40%-80% cut in the path-length for much of the CMS activity.

1970

At the beginning of 1970, I joined IBM. By that time I had an extremely large set of updates to both CP and CMS which significantly improved performance of the overall system. For the first six months or so, I worked on incorporating some of my changes into the CP product. This work was released as part of release 3.

I then started addressing other problems. Up until this time, my work had been primarily concerned with management (&/or optimization) of the CPU and real memory resources.

After all the attention given to CPU and real memory, other areas of the system were beginning to represent primary bottlenecks. I then worked out the design and implementation for ordered seek queueing and chained page I/O. The chained page I/O had its most significant impact on 2301 I/O performance. The maximum throughput was increased from 80 page requests a second to 300 page requests a second. Ordered seek queuing had a major impact on both page I/O activity to 2314s and packs with high-contention CMS mini-disks (typically the CMS S-disk).

CP/67 3.1L

The last internal release of CP/67 was called 3.1L. There was an external 3.2 release but since 3.2 contained only a subset of what was available in 3.1L, it was not used extensively within the company. At that time, Cambridge Scientific Center had a 768k model 67, with three 2301 drums and 45 2314 drives. Heavy load would peak at 66-80 users with approximately 105 pageable pages and sub-second response time.

VM/370 Release 1 started out on a 512K 145. It had drastically modified I/O subsystem compared to CP/67 and all of the feedback algorithms were missing. It was also missing the majority of the fastpath logic from CP/67. Release 1 of VM/370 had severe performance deficiencies as compared to the then current CP/67.

I worked with the development group to improve several of the items in the VM/370 product. I was able to get in several simple algorithm and fastpath changes in release 1.9 and release 2.1. At the same time, I was converting most of the function and performance work I had done from a CP/67 base to a VM/370 base.

A different piece of work that occurred during this period that had a significant effect on VM performance was VMA. The Virtual Machine Assist also significantly reduced the CP overhead involved with simulating virtual.

VM/370 2.15

By the time of VM/370 2.15, I had the majority of my work converted to VM/370 base. I had also created several additional enhancements to both CP and CMS which had significant effect on performance. I had created relocating shared segments in both CP and CMS. I had rewritten EXEC and EDIT so that they would reside in a shared segment (I had previously rewritten the CP/67 CMS editor to be completely re-entrant, I had dropped it, TWX and 2741 support into HASP for local ryo CRJE support). I had modified the APL generation procedure so that a shared APL segments could be loaded, rather than requiring a special CMS/APL system to be IPL'ed. CP also had swaptable migration (to minimize real storage requirements) and page migration.

A separate item that was part of the 2.15 system, was Paging Access Method (PAM) support. PAM involves changes to both CP and CMS. When I initially created the stylized, synchronous I/O interface it was to primarily minimize the path lengths. PAM took that a step further. PAM reduced the path length involved in performing an I/O, by eliminating the page fixed, followed by the I/O scheduling. Rather than fixing all pages prior to scheduling the I/O, the request could be broken into its individual page components and optimally scheduled. It would also take advantage of chained page I/O that I had introduced, to the extent that requests from different users could be scheduled with the same SIO. Synchronous I/O has the advantage of minimizing the overhead involved in an I/O request, but asynchronous I/O allows more latitude in performance optimization. PAM has the additional attribute that CP can do global optimization on all PAM I/O requests (which is not possibly with my standard synchronous I/O interface), even to the extent of allowing asynchronous I/O activity transparent to the implementation in CMS.

I invented the algorithms, designed the implementations, implemented and debugged the code, shipped and supported the resulting production system, just as I had previously with CP/67.

VM/370 RELEASE 3

A subset of the CP relocating shared segment support and all of the CMS changes were picked up as part of release 3 of VM/370. This had a major impact on the number and uses of shared pages. The increase in the number of shared pages, reduces both the real storage requirements and the page I/O rate.

Most of the rest of the CP code that was part of the 2.15 system, was packaged together into the Resource Manager PRPQ and initially released against a VM/370 3.4 base.

AP AND ECPS

Starting in January of 1975, I began designing modification to CP for multiprocessor environment. Specifically, there was a project called VAMPS which allowed for up to a 1+4 CPU configuration (one I/O processor and up to four additional processors without I/O capability). The machine had extensive microcode capability also, and I started looking at how much CP function could be migrated to microcode to improve performance.

In May of 1975, a group from Endicott visited Cambridge looking for suggestions on operating system functions that could be migrated to microcode. They had already completed the implementation of a piece of VS/1 in microcode. I got together some of the preliminary work, I had been doing as part of VAMPS and suggested some additional work that could be done to further identify possible functions that could be migrated to microcode. The outgrowth of the Endicott group was ECPS which cut the overhead of the CP supervisor at that time by 50%.

Late in 1975, the VAMPS machine was cancelled, but Lexington appeared which was a 1+1 168. I began work on converting my VAMPS design to a straight 370 design that wouldn't have any of the additional microcode capability for supporting VM function. Eventually the development group picked up on it, and started a project to release it. There were some number of difficulties. First my detailed design was built on the Resource Manager PRPQ base and second the development group wanted to make some compromises in the design because of marketing considerations.

The resolution to the first problem was that over 60% of the lines of code in the PRPQ was absorbed into the release 4 base. The resolution to the second problem was that the development group built that part of the system their way, and I built it may way. Of course they shipped theirs to customers. I only ran my release 3 based version production on the HONE machines for a year.

For a real look at current performance and where the problems may be, it is helpful to place a current environment side by side with the 3.1L system.

system	3.1L	SP	change
machine	360/67	3081K	
mips	.3	14	*46
pageable pages	105	7000	*66
users	80	320	*4
channels	6	24	*4
drums	12megs	72megs	*6
page I/O	150	600	*4
user I/O	100	300	*3
drives	45	32	*4?perform.
drive capacity	27meg	630meg	*23
drive access	32mills	16mills	*2
drive data rate	.4meg?	3meg	*9
total data	1.2gig	20.1gig	*20

If we compare the resources that are traditional considered critical, CPU and memory, we see an increase between 45 to 65 times between the 67 and the 3081. However we only see an increase by roughly a factor of four in the number of users supported. Even at that we see performance problems in supporting that many users. There is ten times as much resources per user on the 3081 compared to the 67 and there are still performance problems. Why? An even more interesting table is to show the same information as a function of raw MIPS.

system	3.1L	SP3
machine	360/67	3081K
mips	.3	14
resources as function of mip rate		
pageable pages	350	500
users	266	22.8
channels	20	1.5
drums	40megs	5megs
page I/O	500	43
user I/O	333	21
drives	150	1.1

Major problems can be easily seen in the data at the bottom of the tables, I/O of all kinds are now the primary bottlenecks in the system. The page I/O capacity (distinct from the page capacity, i.e. real storage) has increased by a factor of four. In addition, the user I/O capacity in terms of accesses per second has increased by a factor of four to eight. However, the rest hardware in the system (CPU, real storage) has increased by factors of 40 to 60.

There are identifiable problems in usage of CPU and real memory which can be fixed to improve performance but the overall 3081 system is still restricted to only doing 4-8 times as many I/Os as the 67.

Lets look at the traditional resources first. Based on raw MIPS rate, the 3081 is basically about 45 times faster than the 67. It turns out that things don't directly scale up. Because of added function and lack of attention to path lengths, the effective path length in VM/SP is at least two times longer than that of CP/67 (even taking into account the benefit of VMA). There are specific places where that can be improved. One is in the MP support. My original AP design had a single lock, but with a novel twist that I called lock bounce to avoid the traditional lock-spin problem. The result was very short path lengths to implement and little or no lock contention. It had the disadvantage that is was dependant on a large number simultaneous tasks to achieve its through-put. Over the years, work has been done to enhance the performance of a single large guest environment, attempting to get both processors operating on work for the same virtual machine. That with other work has drastically increase the overhead in managing a multiprocessor environment. The situation now is that typically 9-10% of total elapsed time of both processors is spent in just lock contention and that is only one aspect of the MP management overhead.

Real storage management has other problems. To get to 7000 pages requires going to 32meg which involves the >16meg support. 3200 pages is a more typically value for a 16meg machine. Because of implementation problems, the effectiveness of pages in the >16meg area is drastically reduced. The effectiveness of the additional 4000 pages is closer to only 1000 or 2000 pages in terms of standard storage.

A more representative comparison of a VM/SP system on a 3081K compared to 3.1L on a 67 is only an effective factor of 20 increase in CPU resources and 45 increase in real storage resources. Adjusted raw numbers would indicate that there is about twice as many real pages per workload MIP on the 3081 as compared to the 67. The effective increase in real pages per workload MIP is actually much larger because of the enhancements I did for segment sharing. The result should be to eliminate a paging problem by creating a much lower paging rate per workload MIP.

Assuming a factor of four reduction in paging rate per workload MIP because of the increased storage size per MIP and the increased usage of shared pages, the a 3081 could be expected to have a paging rate of 1750 page I/Os per second. Unfortunately, we have a problem. The typical I/O configurations on a 3081 aren't capable of achieving 1750 page I/Os per second.

The resulting paging performance related problems can cause people to believe that 3081s have page thrashing problems. The traditional page thrashing problem, contention for real storage, no longer exists. A new class of problems exist because there isn't enough I/O capacity to move all the required pages back and forth between real storage and backing store.

Even if we could create a 3081 which had a page I/O capacity of 1750 page I/Os per second, there remains the problem of user I/Os per second. The 67 was churning out 100 user I/Os per second (over 300 per workload MIP). That represents over 4500 user I/Os per second on a 3081. Although a typical installations with 3380s has the capacity to hold enough data for 20 times as many users as the 67 with 2314s, 3380s don't have the performance capacity to execute 20 times as many I/Os (30*20 = 600 I/Os per second per 3380 actuator).

Looking at it slightly differently, the 67 system could perform ten to twenty times as many I/Os per instruction executed as can a 3081.

There are changes between the 3.1L system and the current SP system which helps the user I/O performance. EDF (and PAM) have increased the physical block size from 800 to 4096. Unfortunately that doesn't directly translate into a factor of five-fold increase. First, CMS all along would generate chained CCWs for multiple block requests where the blocks were contiguous (negligible difference between 5 adjacent 800 byte blocks and one 4096 byte block). Second, some percentage of the requests are for files that are 800 bytes or less in size. Even if the full factor of five could be achieved (which would better take advantage of the large increase in channel transfer rate), the I/O capacity is still short by a factor of three. In addition to the blocking factor (which is the same for vanilla 4k EDF and PAM), PAM has numerous performance advantages over the straight stylized, synchronous I/O interface that is currently standard in CMS.

OPTIMIZATION AND CONTROL

My work on CP/67 consisted of two distinct activities, one was optimization of resource utilization and the second was feedback control of those resources. I only extended the feedback control to those resources which where primary bottlenecks at that time, namely CPU and real storage. During the intervening years there has been additional optimization work but no extensions of the control algorithms. The current level of resource optimization is basically at the VM/370 release 3 or 4 level (slightly worse for CPU, increased lock contention; slightly better for real storage, i.e. more real storage per MIP, greater use of shared pages).

The current level of the resource control algorithms are basically at 3.1L (the algorithms date from 1969, the first full implementation date from 3.1L, and the current incarnation date from the Resource Manager PRPQ). They only measure, account for, and control CPU and real storage resources which where the primary bottlenecks which I face in 1969. During the last 15 years, the primary bottlenecks on the system have shifted from CPU and real memory to the various parts of the I/O system. The current effective I/O through-put per workload MIP is 1/5th to 1/20th what it was on the 67. Prior to the last five to eight years, there was little point to creating sophisticated control algorithms for the I/O resources because they didn't represent a major through-put bottleneck. The instructions executed by such algorithms would have represented a "cost" that resulted in little or no net "benefit".

Obviously now, optimization of the I/O system will increase the performance of the system. A more important problem is that the resource control algorithms currently don't control the primary bottleneck in the system, I/O.

In the interim we are stuck with attempting to manipulate the resource control algorithms which have little or no control of the major bottleneck in the systems. As a result, changes in control algorithm and/or control algorithm parameters appear to have little or random affects on overall system performance. It is something like trying to use the gas pedal to steer an automobile. Although the gas pedal is a resource control, it unfortunately has no direct effect on the critical resource that we are interested in. It is not quite that bad since control of CPU consumption can indirectly affect the amount of I/O a virtual machine can perform. On the other hand, attempting to fair share the CPU resource without regard to I/O resource consumption will tend to favor the I/O intensive virtual machine that you would prefer to bias against.

A secondary outcome, is that any sort of optimization (manual or otherwise) that is performed on the I/O system can carry with it significant effects. This is a result of the corollary that optimization of the primary system bottleneck always carries the most leverage.

For most installations at the moment, this implies careful ordering of minidisks across physical drives and within drives. It also requires careful review of issues like the price/performance trade-offs of several I/O optimization techniques. For some installations the price/performance of doubling the number of 3380 drives, but filling each only half full will be better than spending the same amount of money on paging drums.

APPENDIX A. >16MEG SUPPORT

Lynn Wheeler
SJRLVM1/WHEELER
276-1783
Aug. 27, 1983

The CP/67 3.1 and VM/370 PRPQ (and incorporated into the base as part of the VM/370 Rel. 4 AP support) page replacement algorithm went to a great deal of trouble in an attempt to treat all virtual pages the same. The replacement algorithm would search all storage locations in a cyclic manner looking for a page without its reference bit on. The replacement algorithm would start at the top of real storage and examine the reference bit of a page. If the reference bit was on, it would be reset, and the algorithm would bypass the page, proceeding to the next lower real storage page. If the reference bit was off, the virtual page would be selected for replacement. The algorithm would then stop, check-pointing the address of the next lower real page address. This check-pointed address would be used to resume the search when the algorithm was invoked the next time.

The PRPQ algorithm had significant advantages over the VM/370 algorithm. In the PRPQ, all virtual pages were essentially assigned random real storage locations. Since all real storage pages are examined in one pass of the algorithm, the time between examinations of a particular page is predictable. The interval for one pass is dependent on the number of real storage pages available and the contention for those real storage pages (or dependent on the reference patterns of all virtual pages in the system). This interval is independent of any characteristic of a virtual page excepting the setting of the reference bit for that page. The most important factor in the algorithm is the examination interval for pages. If the examination interval is 200 seconds, then for a page to remain in real storage, the virtual page must be referenced at least once every 200 seconds (the reference bit must be set between the time it is turned off by the algorithm and the next time the algorithm examines the page.)

The base VM/370's replacement algorithm was superficially similar to that of the PRPQ's, in that it also looped around examining and resetting reference bits. The base system's replacement algorithm, however, followed a CORTABLE chain, rather than looping via real storage address. With proper design of the CORTABLE chain, it would have been possible for the base system to exactly simulate the PRPQ's algorithm. The base system's implementation of the CORTABLE chain, however, resulted in CORTABLE entries being unchained continually from one position in the chain and rechained at another location. One of the results was that there was no predictable examination interval for pages. The average interval between examining pages might be 200 seconds, but the actual interval for some pages could be 50 seconds and for other pages, 500 seconds (depending on how they were removed and restored to the CORTABLE chain).

HPO 2.5 PAGE REPLACEMENT

The HPO2.5 implementation for selecting pages to be replaced has similar problems to those encountered by the original VM/370 implementation. In the HPO2.5, greater than 16-meg support, real storage is partitioned into two areas: those real pages below 16-meg and those above 16-meg. Because of implementation restrictions in HPO2.5, certain virtual pages must be located in the <16-meg area. As a result, the page replacement function must support two types of calls. The standard invocation results in all real storage being treated as one logical area and page replacement can select a page from anywhere. The other case is when a page is required to be located in the <16-meg area.

One performance measure of this class of page replacement implementations is the reset interval. The HPO2.5 implementation maintains separate reset interval statistics for the two areas (<16-meg and >16-meg). Ideally, with a careful implementation, the two intervals should be equal. Measurement data actually shows that the interval for the >16-meg area can be 5-15 times the interval for the <16-meg area. For instance, the <16-meg interval can be three minutes, while the >16-meg interval can be 15-45 minutes. That means a page which resides in the <16-meg area must be referenced at least once every three minutes to remain in real storage, while a page in the >16-meg area needs to be referenced only once every 45 minutes. Virtual pages which have the unfortunate fate of residing below the 16-meg line will be thrown out of real storage unless they are used at least once every three minutes. Other pages which are more fortunate can lay around in real storage unused for up to 45 minutes before they are selected for replacement.

From the stand-point of the page replacement algorithm (as opposed to the HPO2.5 page replacement implementation), when

1. there is a reset interval of 200 seconds,

2. a page must be replaced in the <16-meg area because its last reference was greater than 200 seconds,

3. there are pages in the >16-meg area which haven't been referenced for much greater than the page being replaced in the <16-meg area,

then those pages in the >16-meg area might as well not exist.

In other words, an implementation of the page replacement algorithm (as opposed to whatever the current HPO2.5 implements) could give approximately the same level of performance with much smaller real storage. What type of performance could be expected if the page replacement algorithm was implemented in HPO2.5? First, if the reset interval between the two areas were equalized, what would happen? There are two ways of looking at the answer. First, what would the reset interval be if the real storage size was to remain the same? Second, how much less real storage would be required to give the same level of performance as the current HPO2.5 implementation?

The first question is the easiest. In a 32 megabyte real machine, there will be 4096 pages in the both the <16-meg area and the >16-meg area. Since all of the CP nucleus and working storage must be located in the <16-meg area, the actual number of pageable pages in the <16-meg area will typically be in the 3000-3500 range (dependent on the size and number of virtual machines logged on and how active they are). Having a consistent reset interval in the two areas would increase the interval in the <16-meg area and decrease the interval in the >16-meg area (as compared to the HPO2.5 implementation). The resulting reset interval will be dependent on numerous complex interacting factors. Given reset intervals of three minutes and 45 minutes with the current HPO2.5 implementation, then a composite interval could be in the 10 minute range.

An analogous view of the situation is to imagine what happens when a computing center performs load balancing on two separate 16-meg machines. One of the 16-meg machines is performing 450 page I/Os per second and has a reset interval of three minutes. The other 16-meg machine is performing 10 page I/Os per second and has a reset interval of 45 minutes. Equalizing the load (and the reset interval) on the two machines won't result in an aggregate paging rate of 460 page I/Os (230 page I/Os per machine). Instead, the overall paging rate is decreased and the total productive work is increased. Using this analogy, it is easy to see that the current HPO2.5 implementation exhibits two completely different performance characteristics on the same machine. Such actual comparison in the real world is difficult since almost immediately after the load balancing, the users would increase the work load because of the improvement in service (and as a result all types of activity in the system would increase).

The second question can also be viewed from the two separate machine analogy. Assume that the scenario is the same, except that after balancing the load on the two machines, the real storage on both machines is reduced until the paging rate on each machine is 230 page I/Os per second. Again the actual numbers are dependent on numerous complex factors, but it might be possible to reduce each machine to 12-meg (for a total of 24-meg instead of 32-meg) before the aggregate paging rate again reaches 460 per second.

The current HPO2.5 replacement page implementation can "waste" 8-12 megabytes on a 32-meg. machine as compared to the performance possible with true implementation of the PRPQ page replacement algorithm. Another way of looking at it would be to say that a correct implementation of the algorithm would need much less real storage to give the same level of performance as the HPO2.5 implementation.

PAGE REPLACEMENT ALTERNATIVE ONE - MULTI-LEVEL STORE

There are numerous possible solutions to this HPO2.5 problem. One scenario is to actually treat the areas as two distinct separate storage types. In this scheme, the initial fetch of a page would always be to the <16-meg area. The standard (<16-meg) algorithm would be invoked if there was a requirement to replace a page. The difference would be that the replaced page instead of being removed from storage, is simply moved to the >16-meg area. A similar replacement algorithm would be invoked for the >16-meg area, any time a page had to be moved to the >16-meg area (and there were no available page slots). Only pages being replaced in the >16-meg area would be moved to secondary storage.

Using this scheme the >16-meg area simulates part of a multi-level paging store. In such a situation the three minute and 45 minute reset intervals, might be changed to three minutes and 20 minutes. The advantage would be that pages which failed the three minute test in the <16-meg area would be moved to the >16-meg area. If the pages were referenced within the 20 minute interval, they would still be available in real storage - eliminating a physical page I/O. The reason that the interval for the >16-meg area is reduced is that the replacement routine for >16-meg area would be invoked more frequently (as compared to the current implementation; i.e, essentially every time a page is replaced from the <16-meg area).

Would the performance actually be better? Consider a simplistic scenario. One hypothetical page is brought into the <16-meg area and is used once every 10 minutes, requiring a page I/O at every use. Another page residing in the >16-meg area is used once every 45 minutes (and remains in storage). The result is a page I/O rate of six/hour for these two pages. Under the revised implementation, the page being referenced once every 10 minutes will remain in real storage (decrease of six page I/Os per hour), but it will occupy the space formerly held by a page being referenced once every 45 minutes (increase of 1.25 page I/Os per hour). The overall paging activity has decreased by 4.75 page I/Os per hour. Some might view this as a negligible decrease in the overall paging rate.

ALTERNATIVE ONE - MODIFIED (OPTIMIZED)

There is a temptation to "optimize" the alternative one implementation. Theoretically a virtual page can be used directly while it resides in the >16-meg area. The optimized alternative, instead of just treating the >16-meg area as an extended paging device, will also allow its use as real storage. Rather than move a page from the >16-meg area into the >16-meg area when a page fault occurs, the page table is just updated to show the current (>16-meg) real storage location. The effect on the reset interval will be to increase the (three minute?) interval for the <16-meg area and decrease the (20 minute?) interval for the >16-meg area. The cause for the increase in the <16-meg reset interval is because the demand for <16-meg pages is decreased (pages are used directly in the >16-meg area, instead of replacing pages in the <16-meg area). The cause for the decrease in the >16-meg reset interval is that there will be fewer pages available for replacement on each pass (more pages will have to be examined on each invocation, decreasing the elapsed time it takes to search all pages). It might even be possible for the example, that the intervals would be equalized, i.e., possibly ten minutes for both reset intervals (instead of three and 20).

Such an "optimization" will introduce an imbalance similar to the current HPO2.5 implementation, but not nearly as bad. In the current situation, pages which must be located in the <16-meg region are at a severe disadvantage compared to pages in the >16-meg region (replaced if they haven't been used in three minutes as compared to being replaced after 45 minutes of inactivity). In the "optimized" version, pages which must be located in the <16-meg region will be at an advantage. All pages will be initially brought into the <16-meg region. Any page after ten minutes of inactivity would then migrate to the >16-meg region. Now enters the difference. If some pages can be used in place (>16-meg region), they will not be removed if they are used within an ten additional minutes. Effectively each page initially has a 20 minute death sentence. Once the first ten minutes have elapsed, pages relocated to the >16-meg region have a ten minute death sentence. As long as the page is used at least once every ten minutes from then on it will continually be reprieved from replacement. Things are almost fair and consistent? The exception involves pages which must be brought down to the <16-meg region. Every time they are brought down, their death sentence is reset to the combined interval (2*10 minutes).

Although such an "optimized" implementation is better than what is currently implemented, it is still not fair. The improvement results in pages which are forced below the 16-meg line having a distinct advantage (because their clocks are reset to zero each time) instead of drastic disadvantage (three minute limit compared to a 45 minute limit). Furthermore the discrepancy between the two intervals is much smaller so that there should be better overall performance (differing by possibly only a factor of two - with the bias in a "desirable" direction).

PAGE REPLACEMENT ALTERNATIVE TWO - BALANCED USE

Why does the implementation have to be unfair? One of the objectives of my original design and algorithm (circa 1969) is to establish a uniform reset interval for all pages. As long as a page is used at least once within that interval, then it will remain in storage. Let's go back and examine the problems in the current HPO2.5 implementation. The basic problem has to do with more requests occurring for pages in the <16-meg area. This causes the replacement algorithm to be invoked more frequently, resulting in a decrease in the elapsed time for examining all pages.

The current HPO2.5 implementation is somewhat confused; sometimes it views storage as one large homogeneous area and at other times it restricts its attention solely to the <16-meg area. A solution can be found by explicitly viewing storage at all times as two distinct areas. When pages are explicitly requested from the <16-meg area, the <16-meg area can be searched. Requests for pages that can be satisfied from either area can be handled differently. For these requests, an explicit decision about which real storage region to search should be based on the current reset intervals for the two areas.

A simple implementation involves two constants, two counters and some simple monitoring code. Every time the <16-meg area is searched, counter1 is incremented by constant1. Every time the >16-meg area is searched, counter2 is incremented by constant2. When a page is requested that can come from either area, the decision on which area to search is based on which counter has the smallest value.

Remember that the reset interval is dependant on the frequency which the area is searched and the total number of pages in the area. The objective is to equalize the reset interval, not necessarily the search frequency. In the above example, the search frequency will be equalized if both increment constants are the same. Unfortunately, a equal search frequency is desired only if the number of pages in the two areas are equal. To solve the more general objective, the magnitude of the two constants must be adjusted to take into account differences in the number of available pages in the two areas. Adjusting the relative magnitude of the two constants in proportion to the sizes of the two areas will result in a corresponding adjustment in the search frequency. Adjusting the search frequency proportional to the number of pages in each area has the objective of equalizing the total reset interval. The objective is for the reset interval (time to examine all pages) for both areas be equal.

Assuming constant1 to be some fixed value (say 16), the monitoring code would adjust constant2 such that the reset intervals for the two areas remain approximately equal. The initial approximation for constant2 would be:

constant1 * (size of <16-meg area) / (size of >16-meg area)
i.e. the ratio of constant1 to constant2 would be equal to the ratio of available pages in the <16-meg area to the number in the >16-meg area.

For a 24-meg machine, there would be approximately 3400 available pages in the <16-meg area and 2048 available pages in the >16-meg area. Assuming constant1 set to 16, then constant2 would have a value of 27. For a 32-meg machine, there would be twice as many pages in the >16-meg area and therefore constant2 would have a value of 13.

The monitoring code would periodically adjust constant2 based on changes in the size of the <16-meg area and the measured reset intervals. It would also be the responsibility of the monitoring code to periodically adjust counter2 to minimize large discrepancies (and the long term effects of any temporary aberrations in activity).

COMBINED ALTERNATIVES

I have proposed three modifications to to my basic page replacement algorithm to allow for the problems introduced by the current CP use of real storage above the 16-meg boundary. The first alternative preserves the correctness of the algorithm, but significantly under-utilizes the real storage available above the 16-meg boundary. The optimized version of the first alternative makes better use of the resources available but significantly perturbs the algorithm implementation. Finally, the second alternative appears to both fully utilize the resources available and implement the algorithm correctly.

Before ending the discussion, a drawback must be pointed out in the second proposal. Inherit in the design of the second alternative is an assumption that by forcing page replacement, when possible, to select pages in the >16-meg area, the reset intervals for the two areas can be equalized.

The drawback involves the phrase when possible. In the CP design for >16-meg support, page requests are broken into two types: a) <16-meg and b) don't-care. The <16-meg page are those which must be accessed in some way by executable CP code (and therefore must reside in the first 16 megabytes of storage). The don't-care pages are those which are only accessed by virtual machine execution. One way to characterize the demand for pages is to calculate the request ratio of <16-meg pages to the don't-care pages.

The implied assumption in the second alternative is that the ratio of the demand for pages doesn't exceed the page availability ratio (the ratio of available pages below the 16-meg boundary to the pages above the boundary).

When the demand ratio exceeds the availability ratio, no amount of adjustments can maintain the algorithm design point. In a 24-meg system, the availability ratio of approximately 3500:2048 should easily be larger than the demand ratio. On a 32-meg system, the availability ratio of 3500:4096 should still be adequate. But what about a 48-meg system, or a system that has a large V=R region defined in the <16-meg area? Defining an 8-meg V=R area on a 32-meg system would drop the ratio to 1450:4096 or less.

It is entirely conceivable for environments to exist which will violate the basic design assumptions inherent in alternative two. What can be done? One solution might be to implement both proposed modifications. As long as the demand ratio stayed within the design limits, the second alternative would be active. When the demand ratio exceeded the design limit, a switch would be thrown and the optimized first alternative would be activated. This solves the problem of equalizing the demand, but has some drawbacks. In principle, the optimized first alternative should be avoided because it violates the algorithm. Switching completely over to the first alternative, even when the demand ratio exceeds the design limit by a small margin, appears a little drastic.

A better compromise would be to gradually phase in the first alternative. What indicators can be used to recognize the condition? One indication of the presence of the condition is a large difference between the two counters (defined in alternative two) The <16-meg counter is incremented every time there is page replaced in the <16-meg area. The >16-meg counter is incremented every time there is a page replaced in the >16-meg area. When a demand is made for a don't-care page, the algorithm will always select a page from the >16-meg area (if the >16-meg counter is less than the <16-meg counter). It is still possible that this course of action will not be sufficient to allow counter2 to overtake counter1.

If the <16-meg counter remains larger than the >16-meg counter, and the difference increases over time, then the design point of the algorithm has been exceeded. This characteristic holds the solution for phasing in the first alternative implementation. First, a "difference" threshold can be defined to be some value that the <16-meg counter can exceed the >16-meg counter. When the gap threshold is exceeded, the replacement of <16-meg pages is changed. Instead of removing the <16-meg page from storage, it is migrated to the >16-meg area. In this case, it will not be necessary to increment the <16-meg counter (since the page was never actually removed from storage). However, the >16-meg counter will increment because a >16-meg page will have to be removed to make room for the migration (decreasing the difference between the two counters).

This modification has the effect of dynamically adapting the real storage management algorithm across diverse configurations and loads. It isn't a perfect solution since it doesn't really create <16-meg storage. It effectively partitions storage into three logical areas. The <16-meg area, the >16-meg area, and the <16-meg staging area (located above the 16-meg line).

The solution can also be viewed from the standpoint of the <16-meg area being a critical system bottleneck (rather than the more general case that real storage is a bottleneck). There are additional resources in the form of the >16-meg area, which don't directly solve the demand for <16-meg pages. Another way of viewing the problem is that the resources available in the form of pages above the 16-meg line exceed the demand for those resources (as compared to the resources available for servicing the <16-meg demand). Logically, in such a situation, there are excess pages above the 16-meg line. Effectively these pages are idle (similar to the argument concerning the current HPO2.5 implementation). The question becomes: how to use the idle resources to address the bottleneck problem for <16-meg pages? Using the >16-meg area as a staging area for <16-meg pages doesn't increase the size of the <16-meg area. It does improve performance because the overhead associated with retrieving a page from the >16-meg area is less than from backing store.

APPENDIX B. -- CP PAGING

Lynn Wheeler
SJRLVM1/WHEELER
276-1783
Oct. 1982

Several questions have been asked about the current CP paging "algorithm". The current implementation is an algorithm design I did before I joined IBM plus several misc. changes from the 73-74 era as part of the conversion to VM/370. There are two main items of the paging algorithm. The first is the page replacement algorithm. The second is the page thrashing or multiprogramming control algorithm.

The original CP had no page thrashing control and a very primitive page replacement algorithm. The original implementation allowed all runnable tasks into queue. The original page replacement algorithm consisted of looping the CORTABLE searching for a page that didn't belong to an IN-Q virtual machine. If the complete CORTABLE was looped without finding a page to replace, a second search was made to find the first available page.

As soon as any sort of heavy load happened, the system very quickly got into a page thrashing situation. A very primitive page thrashing control was implemented by Lincoln Labs. and distributed with CP/67 2.3. Page thrashing was controlled by limiting the multiprogramming level (number of virtual machines allowed to execute &/or in queue). The MPL was limited to a number that was proportional to the amount of real storage available in the system (something like four virtual machines per 256K of real storage). This represented a significant performance improvement because it minimized the page thrashing conditions.

I was working on CP/67 at this time & it was also about the same time that Denning published his Working Set Paper (1968). The problem with Denning's design was that it required a large number of CPU cycles to implement (scan complete virtual memory tables every 10,000 to 30,000 virtual instructions) and made no provisions for being able to adapt to the robustness of the paging I/O subsystem (although IBM Science Center in France did implement a version against CP/67 release 3). I established goals for a different algorithm which would 1) accomplish essentially the same objectives as Denning's working set algorithm, 2) have an implementation with much shorter path lengths, 3) have path lengths that were proportional to the activity, and 4) would dynamically adapt to the load and situation.

The first area I changed was the page replacement algorithm. Instead of looping the CORTABLE looking for a page that didn't belong to an in-queue virtual machine, I replaced that with the (current) reset reference bit algorithm. I invented &/or investigated several different classes of algorithms. I was looking for an optimal algorithm. It was not necessary that the algorithm make the optimal page replacement choice ... it had to provide that overall optimal system performance. The implemented algorithm had to provide the best page replacement in the shortest possible path-length resulting in the optimally net performance throughput.

The looping algorithm that I invented had several major advantages over Denning's design. Denning required that each task's (virtual machine) pages be reset every time the task consumed a fixed number of CPU cycles. Pages that didn't have their reference bits set where removed from the active working set and placed in the available pool. A task's real storage requirements (or working set) was determined from the average number of pages that had their reference bit on in each interval. Several problems occur with Denning's algorithm. One of the major ones is that the overhead to manage the resource (real storage) is not proportional to the demand for that resource (note: the current implementation with very low real storage demand & resetting of bits at queue drop effectively approximates Denning's algorithm ... one of the very things that I was trying to avoid).

Instead of resetting the reference bits at fixed intervals, I designed an algorithm which would reset the reference bits at a rate proportional to the demand for real storage. This represents a natural adaptive algorithm. The cut-off point for acceptable page life in real storage will automatically change to be whatever value is appropriate for that installation and load. As the load increases, the acceptable page life decreases; when the load drops off, the acceptable page life increases. This is a very significant point. At a high level design overview, both Denning's algorithm and mine periodically reset the reference bits in real storage. Major differences between his algorithm and mine are 1) my overhead is proportional to contention for real storage rather than proportional to number of instructions executed by a task and 2) my algorithm naturally, implicitly, dynamically adapts to the contention for real storage.

OK, my algorithm significantly shortens the path-length to manage real storage and at the same time was able to dynamically adapt to its environment. Unfortunately it also did away with the mechanism for measuring and predicting working sets. No longer were a count of the average number of reference pages kept. As a result I had to come up with a substitute which would approximate the same effect. If you logically analyse what is happening in the looping through core & resetting the reference bits, that process is very similar to what Denning was doing on a fixed interval. My algorithm would in fact be resetting the reference bits. I conjectured that there would be some correlation between the number of pages in real storage for a virtual machine and its demand for real storage (seems somewhat obvious now). The number of pages in real storage for a virtual machine somewhat trails the "exact" definition of working set. Pages that have their reference bit reset (and are no longer in the working set) would be included in the in-core, real storage page count until the replacement algorithm has completely looped the CORTABLE once. After a complete loop of the CORTABLE, the pages would be removed. I decided to attempt to calculate an approximation to a real working set value by using the count of resident pages for a task.

Using the real storage page count has all sorts of pit falls. First, if the looping time was extremely long, a page would be included in the real storage count for a period much longer than its life time. This could grossly enlarge the approximate working set value. Enlarged working set values would lead to decreased number of virtual machines executing simultaneously, further reducing the real storage contention. Reduced real storage contention will contribute to increased elapsed time for the reset algorithm to wrap the CORTABLE, further increasing the error in the working set calculations.

The reverse is also true. If there is a sudden spike in demand for real storage, leading to decreased reset algorithm wrap time, then it is possible to underestimate the working set value. Estimating the working set too small will allow a too large MPL and excessive real storage contention ... further reducing the working set estimate. To handle both situations, there had to be other factors than the number of resident pages belonging to a virtual machine that were used to estimate working set. Otherwise the situation is non-stable.

Another problem was where to get the count of real storage pages. An working set estimate was being recalculated at queue drop. Normally there could be large fluctuations in the number of pages in real storage during a queue stay. One approach might be to create a timer driven task that periodically samples the number of real storage pages. The number of real storage pages would be accumulated at each observation. At queue drop the accumulated total would be divided by the number of observations to arrive at the average number of resident pages.

Here again was something I disliked doing ... creating frequent timer driven events ... when it might not even be necessary. As a compromise, I decided that page read events appeared to occur relatively frequently. I made the assumption that the page read events could be used to approximate the intervals (although not necessarily evenly). Instead of taking the sample at fixed intervals, I would take the sample at the page read. Then at queue drop, the sum of the count of resident pages at each sample would be divided by the number of samples (in this case the number of page reads).

The basic number used in calculating the working set prediction is the average number of resident pages. Unfortunately it has several external factors contributing to errors in the calculations. One is real storage contention which affects how fast pages that are no longer in the working set are removed from real storage (and the resident page count decreased) &/or pages being removed that are in fact still in the working set. The other is the virtual machine's page read distribution which can cause the observations & average to be significantly skewed from a true time average. Both of these deviations have to be corrected for in the calculations to come up with a useable prediction.

So much for theory. There are several implementation additions to the basic algorithm which has restricted the adaptive latitude capabilities of the code. First, a point about the average resident page sum. The average resident page sum in addition to being corrected in other ways, has some code which implicitly assumes that the number of pages in real storage at queue add time is zero. The observation was made that at queue add time, the number of resident pages will be zero and that the virtual machine must initialize the working set by page reads. Each page read will create a sample value. This leads to the sum of the average resident pages being increased by 1+2+3+....+n-1+n, where n is the working set size, just to establish the initial working set. If no other page reads occur, then the calculations would result in a value about 50 percent of the true value. Code is in place to use as the first approximation to 'n', the maximum number of real storage pages that was reached during the queue stay. This value (VMMXPG) is subtracted from the number of page reads and 1+2+....+VMMXPG is subtracted from the sum of resident page count. There are places where the assumption in this code can be invalid. First the number of resident pages at queue add time might not be zero (this could be corrected by saving the initial resident page value). The second is that the maximum resident page count might be achieved w/o performing any page reads.

There is a CP feature that allows a virtual machine to accumulate real storage pages w/o performing a page read. Originally all real page allocation included a page read. Part of the system CP IPL disk format included a page that was all zeros. A virtual machine at virtual IPL time would have all of its SWPTABLE entries initialized to the CCPD of this page. Sometime after developing the above algorithms, I invented the "zeros page". Not only was clearing a page by instructions faster than reading the page in, but it also required a shorter total path length.

Either "zeros page" activity and/or non-zero initial page resident count can lead to situations where the maximum number of real page level was reached w/o performing page reads. This can lead to a significant under calculation of the average resident pages.

Another implementation addition which can significantly affect the paging system is the reset code at queue drop. There is currently code which at queue drop time will reset all the reference bits belonging to a virtual machine and/or place all the pages on the flush list. The code was originally conditional, both in the resetting of the reference bits and placing the pages on the flush list. Also until the last 4-5 years it represented a relatively small proportion of the page manipulation that went on. With the current large storage environment, the majority of virtual page manipulation may be the queue drop code.

If the reference bits were to be reset at queue drop time and then only those pages that didn't have their reference bits on, were to be placed on the flush list ... then effectively we have Denning's 1968 working set algorithm ... and just the thing I was trying to originally avoid. My intent in the CP/67 queue drop design was to only invoke it if there was no possibility of the pages being reclaimed and it didn't significantly affect the overall page replacement algorithm. The current situation is a result of a incremental, partial merge of my CP/67 code into the VM/370 rewrite (small excuse).

At very low real storage contention (large real memory, low MPL) the majority of the paging may occur at queue add and queue drop time. Previously this had only been an MVS design point since its CPU overhead to perform page operation would cause CPU saturation in a page thrashing situation. VM/370 is now finding itself in a similar situation where real storage contention is at a minimal loading level. Two changes have occurred to bring this about. The first is the number of available pages per MIP has significantly increased cutting down on real storage contention (per MIP). The other is the decrease in I/O capacity per MIP. With the current I/O configurations it is not possible to support large multiprogramming levels. High multiprogramming levels encounter severe bottlenecks in the I/O system (high page wait time used to be an attribute of page thrashing ... i.e. contention for real storage was too high ... currently we can have high page wait time just moving pages into & out of storage at queue add/drop time because of the severe bottlenecks in the I/O system). Thus a smaller multiprogramming level per MIP also contributes to reduced real storage contention.

There are a couple of things that can be done to clean-up the current algorithm implementation ... but the "new" major problem is not directly with managing real storage (although there still are configurations with MIPS/real-storage and MIPS/IO ratios which require real storage contention control ... and the current algorithms to adaptively operate as they do). The new problem is to control MPL level operating in the I/O system (i.e. those tasks which can simultaneously compete for I/O resources) similar to the controls on MPL level for real storage. There is also some trade-offs that are theoretically possible with utilizing the excessive real storage as a logical paging device (and alleviating the load on the page I/O system)

NUMBER OF BITS

Without any activity at queue drop, the code implementation is algorithmic correct ... it may not give optimal/desired performance in all circumstances ... but it gives predictable, understandable performance. The code implementation is basically a good approximation to an LRU replacement algorithm. It turns out that an LRU replacement algorithm does not necessarily result in optimal performance in all circumstances. I designed and implemented a new variation on the one bit replacement algorithm which is better than the current algorithm and at times better than a true LRU replacement algorithm.

Most page replacement algorithms can be classified as either locally LRU or globally LRU, the distinction being whether the resetting of the reference bits occur against just a particular task's pages periodically or against all real pages. Project MAC did some studying of the general class of replacement algorithms using one bits, two bits, three bits, and four bits. The published results were that multiple bits were normally marginally better than one bit but the path-length increase usually didn't justify the amount of improvement over two bits.

N.5 BIT REPLACEMENT ALGORITHMS

I invented a different class of global page replacement algorithm ... the simplist case I refer to as the 1.5 bit algorithm. It requires two physical bits of implementation but the way they are manipulated results in less than two full bits of history information for each page. In fact the amount of history information is variable between one full bit to two full bits of information. On the average, each page in memory will have 1.5 bits of history information (ref: to unclassified research report, VM/370 Modifications). After detailed simulation studies it was found that the 1.5 bit algorithm would perform better than either a one bit or a two bit algorithm. In fact, there was normally a particular member of the class of 1.5 bit algorithms which would perform better than true LRU.

The general class of N.5 bit algorithms have even more interesting characteristic in a multiprocessing environment. The current reset and test reference bit code requires very few instruction in an UP environment, but the path-length drastically balloons when going to an MP environment because of the synchronization problem with multiple processors. The N.5 bit class of algorithms drastically reduces that overhead because all reference bits are reset at one time. During the search sequence bits are only testing (with ISK instruction ... not requiring synchronization) and not reset.

My version of AP support that I did for release 3 at the Palo Alto HONE system implemented the 1.5 bit algorithm. At the point where all keys must be reset, both processors were synchronized and began executing the same instruction loop in a DMKPTR subroutine (one of the processors was assigned half of the pages to reset and the other processor was assigned the the other half of the pages). The function was implemented with a special SIGNAL function which specified that the signaled processor was to branch to a section of code pointed to by address in low core.

The other area where the current AP implementation has long path lengths has to do with the invalidating of the page once the reference bit is reset because the other processor must be placed in STOP state (a DMKEXT subroutine) prior to the invalidation. That problem is eliminated with the IPTE instruction. For my release 3 implementation (without the IPTE instruction), I had the signaled processor turn on the invalid bit in the PTE (while the signaling processor waited for the invalid bit to appear).

The N.5 class of replacement algorithm with the new SIGNAL function allowed me to drastically reduce the path-length in the paging supervisor and while eliminating the requirement for STOP synchronization functions (path-length for the STOP function is deceptive since it requires a branch to a subroutine which issues a SIGP and then spins until it receives an indication that the other processor has stopped).

FLUSH LIST MANIPULATION

There have been several instances recently of flush list manipulation which results in unpredictable performance changes. The problem is that the flush list manipulation is perverting the basic algorithm in strange ways ... sometimes completely overriding any base algorithm execution at all. The exact effects happen to load and/or configuration dependent. Normally performance improvements of this type occur when there is only one continuously running guest machine along with periodic other types of usages.

The basic replacement algorithm attempts to approximate LRU as closely as possible. In several instances that may not be the best strategy. One of those situations is where it is desired to bias in favor of a large guest machine. One of the tendencies of most of the flush list manipulations is to biases the replacement algorithm against virtual machines that drop from queue and enter idle state, i.e. idle machines have their pages placed on the flush list. Because of the current implementation using all pages on the flush list prior to executing the basic replacement algorithm may be a large number of very low usage pages which get replaced infrequently or not at all. The result is that the paging rate will be higher than if the implementation conformed to the LRU approximation algorithm.

Now in what circumstances would the system perform better with a slightly higher paging rate. One of the major circumstances is when the paging "burden" is shifted from a batch guest machine (which the installation desires to consume the majority of the resources) to trivial CMS machines. This shift in paging burden must be such that the guest machine is the only recipient of the benefit and the overall increase in paging rate doesn't saturate the I/O system. Problems occur when there is significant, multiple virtual machine activity which executes for long periods of time. The paging burden shift is not selective by virtual machine priority or favoring, just by virtual machine execution characteristics.

The major problem is that some flushlist manipulation perverts the implementation code so that it no longer conforms to the algorithm. Typically the code "tuning/changing" is being performed by individuals who don't understand the intimate relationship involved between a statement of an algorithm and what is required in code to implement that algorithm. Small code changes can frequently modify the dynamic execution characteristics of a system so that drastically different algorithmic implementations are the result.

Anyway, back to flush list. The "shifting" burden explanation turns out to have second order effects that can skew the resulting performance data, greatly confusing people doing performance analysis. If the guest virtual machine happened to be favored at 90-95 percent, and it was only executing at 80 percent because of page faults, then random modification to the flushlist implementation might be able to cut the page fault rate for that specific virtual machine in half ... possibly doubling the page fault rate for other virtual machines. Turns out page faults occur when virtual machines execute instructions. Also different types of programs and/or virtual machines may have a different ratio of instructions per page fault. Favored guest machines tend to have rather high ratios of instructions per page fault. (in general, most of the IBM O/S derivatives tend to have large, weak working sets, i.e. large number of pages ... and large number of different pages ... accessed per instruction interval ... which ordinarily would create large page fault rates but a favored guest is set-up by the installation to have a large number of pages resident all the time).

Lets say the favored guest is only executing at 80 percent of the CPU without any unusual flushlist manipulation. Flushlist manipulation could cut the page fault rate for the guest in half (say by 10 page faults) allowing it to execute at 90-95 percent of CPU. The paging "burden" has been shifted to the trivial interactive users so that they will now experience an increase of in the number of pages faults. But since the page replacement algorithm is making less than the optimal choice in order to allow the guest machine to execute better, the trivial CMS users may experience an increase of 20 page faults (verses a decline in 10 for the guest). But that increase of 20 page faults is only if the CMS machines continue to consume 20 percent of the CPU, i.e. one page fault for each percent of the CPU. With the guest executing at 90-95 percent of the CPU, the CMS virtual machines will only consume one half to one fourth of the total CPU that they had previously. As a result their page fault contribution per unit time will decrease by a factor of 1/2 to 1/4. The net result for the overall system is a total decrease in paging rate ... primarily because of shift in the type of instructions executed (&/or which virtual machine is executing those instructions). The guest virtual machine ratio of CPU executed per page fault was increased from 40mills/fault to around 80-90mills/fault. The CMS virtual machines had their CPU/fault ratio decreased by the increase of one page fault per 10 milliseconds of CPU.

That shift may be desirable, but it is purely a side affect of the code change. On other systems with multiple, continuously executing virtual machine, the effect can be a drastically higher page I/O rate. The changes in flushlist manipulation provide beneficial paging performance to virtual machines with specific execution characteristics at the expense of poorer overall choice in the pages selected (although an accompanying shift in load characteristics may in fact improve overall system throughput). The change occurs without consideration for any external installation objectives. At some installations the virtual machine(s) that are desired to execute faster may have the specific internal execution characteristics (which specific code changes may bias towards) -- on other systems they may not.

BIASED PAGE REPLACEMENT ALGORITHMS

At several installations there is a requirement for biased page replacement algorithms. The problem is normally one of a primitive guest machine that has not been modified to execute in a virtual machine environment. The installation has a performance goal for a particular guest to consume a large percentage of the total system resources. Unfortunately any page fault for that virtual machine can make it totally unable to execute for the duration of the time it takes to service the page fault request. The objective in such a situation is not necessarily to choose the least recently used page in the whole system ... but to modify the algorithm to make the best page replacement (LRU) choice in the whole system ... while at the same time minimizing the page fault rate for a specific virtual machine(s). If the best page replacement choice happens to belong to such a favored virtual machine, the algorithm may have to skip choosing that page and choose a page belonging to some other virtual machine.

One example is the CP reserved page option. I did a rough design of a reserved page extension to support multiple virtual machines for BNR (& is a modification of some extended CP/67 page algorithm work I had done). The reserved page options basically attempt to avoid selecting pages for specific virtual machines if their number of in-core pages falls below a minimal threshold. The reserved page options are an attempt to indirectly minimize the virtual machine page fault rate for a specific virtual machine. They attempt to achieve that goal by maintaining a minimum number of resident virtual pages for the virtual machine.

A more direct way of affecting a virtual machine's page fault rate has been implemented at Cornell University. In Cornell's modification, pages that are selected for replacement may be "skipped" in a manner very similar to the multiple reserved page option. In the multiple reserved page option, pages are "skipped" if the selected page will cause the number of reserved pages for a specific virtual machine to drop below a specified threshold. The Cornell modification allows that every Nth page chosen for selection from a specific virtual machine to be "skipped". The value for N is dynamically adjusted based on the total number of resident pages belonging to the specific virtual machine and that virtual machine's recent page fault rate. The Cornell modification is thus able to control directly the page fault rate for a specific virtual machine.

REVIEW

The previous sections discussed the management of real storage. They dealt with the concept of "working set" which has been roughly defined to be those set of pages required to do productive work. The concept was developed to deal with the problems of uncontrolled page thrashing (real storage contention) in operating systems using page replacement algorithms. The general objective was to limit the number of simultaneously executing tasks whose working sets can be contained in real storage. The second problem is how to choose a page for replacement from all the pages in storage. Most algorithms tend to approximate a least recently used criteria on the assumption that those pages have the smallest probability of being used again in the near future.

During the early development of paging systems, the first concept (working sets) was of prime importance because of the relative scarcity of real storage. Current hardware configurations are radically different. Typically real storage now, is at least two to three times larger than the sum of the working sets for contending tasks. As a result, fine tuning of the working set control algorithms has little or no affect on overall system performance (all contending tasks will always be able to execute regardless of the algorithm).

As a result the primary focus for algorithm concern has shifted from the working set control algorithms to the page replacement algorithms. In addition, a different resource bottleneck has emerged. While the number of real storage pages per MIP has been increasing, the amount of I/O capacity per MIP has been decreasing. A requirement for a concept similar to working set control is now required for the page I/O subsystem, since it has replaced real storage as the important bottleneck in the paging system.

For page replacement algorithms, a logical enhancement is to minimize the number of page faults per CPU consumed (or maximize the CPU/fault ratio). Such an algorithm will effectively be equivalent to increasing the size of the working set for specific virtual machines. Since there is normally large amount of excess real storage, it should not be detrimental to overall system performance. There will have to be some sort of reasonable upper bound on this technique, since it is still possible to exceed real storage capacity and effectively cause page thrashing for a subset of the tasks.

The new area for implementing algorithm control is the area of contention for the resource represented by the page I/O subsystem. Working set controls effectively minimized the contention for real storage, allowing the scheduling algorithm to distribute the computer resources in a controlled manner. A page I/O contention situation has no similar controls. Excesses demands on the page I/O system will saturate the resource. This lead to excessive queueing delays over which there are no algorithmic controls. Requests are essentially satisfied in a FIFO manner with no consideration given to scheduling and/or throughput objectives.

APPENDIX C. (GROUP AND USER) FAIR SHARE SCHEDULER.

Lynn Wheeler
SJRLVM1/WHEELER
276-1783
May, 1982

This scheduler was released as part of the VM Resource Manager PRPQ (which was later renamed VM/SEPP and eventually became VM/SP). The scheduler got its "nickname" from the fact that it kept track of all virtual machines' CPU consumption (on a recent history basis) and included the CPU consumption value in the dispatching priority calculations. The dispatching priority controls the ordering of the VMBLOKs in the dispatch list, thereby controlling their dispatch frequency and indirectly controlling the rate at which virtual machines are allowed to consume resources.

The CPU consumption value is a primary component in the dispatch priority calculations. The computation involving the virtual machine's CPU consumption value is the calculation of the ratio of the individual consumption value to a target consumption value. If the measured consumption rate is larger than the target value, then the ratio is greater than one. On the other hand if the consumption rate is lower than the target value, the the ratio is less than one. The ratio is multiplied times a value which represents the interval between periods when a virtual machine is allowed to execute (and thereby consume resources). The larger the ratio, the less frequently a virtual machine is allowed to execute (and the fewer resources which will be consumed).

It turns out that the calculations are self-correcting, in that as a virtual machine increases its CPU consumption, the calculated ratio increases, leading to a slow-down in execution. On the other hand, low CPU consumption results in a small calculated ratio and faster CPU consumption. The calculations are also automatically biased towards trivial interactive tasks. By definition, trivial interactive tasks require a small amount of CPU consumption, and as long as their CPU consumption remains small, their calculated ratio will be close to zero.

The term "fair share" comes into play because the value chosen to be the "target cpu consumption" in the calculations is the total available CPU divided by the number of virtual machines. The target CPU consumption is the allowed fair share CPU consumption. Virtual machines consuming more than their fair share will run slower, while virtual machine consuming less than their fair share will run faster.

A recent modification which I've made to the fair share scheduler is the addition of an optional "group fair share" algorithm. In this case rather than calculating the target CPU consumption by taking the total CPU available and dividing by the total logged on users, the target CPU consumption is calculated by dividing a predefined allocated group CPU percentage by the number of members currently logged on in that group.

APPENDIX D. VM DESIGN WORKSHOP

Call for Papers
December 1981

VM Design Workshop

to be held at IBM San Jose Research

Lynn Wheeler
276-1783
SJRLVM1/WHEELER

A restricted attendance VM Design Workshop is tentatively being scheduled around the time of the VM Internal Technical Exchange in San Jose (late Feb. or early March of 1982). Objectives are to cover several related subjects which could lead to a rewritten CP system.

TOPICS

- High level system programming language
- Software development tools
- Distributed software development
- Migration of CP functions to virtual address spaces
- Migration to non-370 architectures
- 370 simulators
- Dedicated, end-user system

A possible project which would utilize extensions in all the before mention areas is a relatively inexpensive, relatively fast non-370 CPU. A VM kernel (many CP functions having been migrated to virtual address spaces) is coded in an high level system programming language. The kernel will initially be compiled into 370 code and executed using the 370 simulator. Eventually the kernel (and possibly some of the virtual address space code) will be recompiled into the native machine language and execute along side the 370 simulator (providing both native mode and 370 virtual machines).

Although a definite pilot project is envision, nearly all work will be beneficial to all current VM/370 environments.

APPENDIX E. VM/370 ENHANCEMENTS FOR DYNAMIC SCHEDULING, PERFORMANCE EVALUATION AND CAPACITY PLANNING.

R.E. Braine
Lynn Wheeler

IBM
San Jose, California 95193
12/3/80

ABSTRACT: This report describes proposed VM/370 scheduling algorithm enhancements.

DISCLAIMER: The work described in this paper was done in support of internal IBM VM/370 installations only. It should not be confused in any way with support associated with the VM/370 product.

Introduction

This is a proposal for a restructuring and enhancement of the current VM/370 scheduler. The enhancement will have a major impact on the way that the capacity and performance of a VM system is measured, planned and managed on a time scale extending from the immediate through to days or months.

The following outlines the scope of this proposed project:

1) Move the code that performs the analysis of system performance and makes scheduling decisions from the nucleus and place it in a virtual machine. The code that will remain in the nucleus is that which is necessary to carry out the scheduling decisions, and items appropriate for control on a millisecond basis.

2) Expand the data collected by CP to provide improved input for performance analysis and scheduling decisions.

3) Develop improved scheduling algorithms, controls, and feedback techniques.

4) Extend "3" to include policy decisions at the group and user level. Such an extension would provide:
   a) A set of policy rules which management could selectively apply to groups and/or users.
   b) Analysis of system performance and decisions based on the selected policy rules as to which groups and/or users will be affected.
   c) Feedback for:
      - management giving the effects of the policy decisions.
      - the virtual machine to dynamically determine how effective the particular decisions were and permitting additional adjustments as needed.

1.0 Separate Performance Analysis from CP Nucleus

The first item of the enhancement will be to move significant portions of the CP scheduler out of the CP resident nucleus and into a virtual machine. Those items that are appropriate to control on a millisecond basis will be left in the nucleus. Those parts of the scheduler algorithm which involve longer term decisions and more complex logic will be implemented in the virtual machine. This change will have a great impact on the design and testing of new algorithms and will permit the use of efficient programming techniques through the use of high level languages, the implementation of more sophisticated and complex algorithms and the protection of the system from programming errors in the scheduler.

1.1 Current Implementation

The current CP scheduling code gathers only a limited amount of immediate statistics in an attempt to do local and short term performance optimization. CP can use only a small area of fixed storage for data gathering. CP is implemented exclusively in assembler language.

1.2 Limitations

i) The current CP implementation structure imposes severe limitations on the development and testing of new algorithms.
ii) Software failures result in the whole CP system abending. Relatively common errors, like divide overflow, lead to CP system failure.
iii) System shutdown and re-IPL is required following even the most trivial algorithm changes.

1.3 VMPT - Potential Home for Scheduler Implementation.

The current VM/370 Performance Measurement Package (VMPT) is a virtual machine sub-system for the collection and analysis of VM performance data.

VMPT is implemented in a higher level language (FORTRAN) with all the usual advantages over Assembler in programming ease and efficiency. It already has extensive and sophisticated CP performance data gathering and reduction facilities. Changes and enhancements to the VMPT code running in the virtual machine do not require a CP system IPL. Any failures in the VMPT code do not bring down the whole CP system; many software (or data) errors are routinely handled by FORTRAN support routines.

VMPT has large proportion of a virtual machine scheduler already implemented and it would be relatively easy, working in a high level language and in the isolation of a virtual machine subsystem, to develop the proposed scheduler enhancements.

2.0 Enhance Performance Data Collected by CP

Changes to the Control Program (CP) are proposed to gather more performance information, particularly involving I/O activity and various VM service times. This will allow new forms of performance evaluation based on new models of system performance and the control of potential bottlenecks in a manner never before possible.

2.1 Current Implementation and Limitations

Currently, in the standard VM system, the only information that is available is gross I/O counts by real device and virtual machines. This information is not sufficient to make any decisions about I/O resource contention, bottleneck identification, control of virtual machines contending for a scarce I/O resource, etc. What is required is detailed information about the duration of I/O operations, virtual machine compute bound ratios, length of queues on specific real devices and many other pieces of information. Without this type of information it is all but impossible for the scheduler to make any sort of intelligent decision about how virtual machines should be dispatched to minimize I/O contention.

2.2 Needed Information

Status for each user & device that includes:
- device used (CPU, Dasd, Page, Terminal, etc.)
- operation (Process, Read, Write)
- count of operations
- total user wait time
- total service time

In addition the device info should also include:
- total time device is detected as being unavailable for service to this system.

2.3 Changes to Date

Currently several changes have already been implemented to start gathering data to address problems in this area (<2> VM/370 Modifications).

1) VMBLOKs are time-stamped on entry and exit from wait state
   a. Individual VMBLOK compute bound ratios are calculated
   b. Total VMBLOK runnable time is accumulated
   c. Average number of runnable VMBLOKs is calculated (CPU queue size)

2) IOTASKs are time-stamped when queued on a RDEVBLOK
   a. Total IOTASK queued time is accumulated for each RDEVBLOK

3) IOTASKs are time-stamped when they are initiated

4) All paging I/O requests are time-stamped when they arrive
   a. Total page I/O service time is accumulated

Using this information, there have already been several minor changes in the scheduling code.

2.4 Modifications needed to VMPT and VMAP for new data.

VMPT Enhancements

VMPT should be enhanced to gather the additional performance data produced by the proposed CP changes. This includes information about IOTASK queued time by device, virtual machine total runnable and non-runnable time, and other queue delay service times.

VMAP Enhancements

Enhancements will be required to the VM/370 Performance Analysis Package (VMAP) to allow this information to be analysed and displayed. From this, reports can be generated to detect CPU bottlenecks (if any) and to highlight particular devices which have large queueing time (probably representing major throughput bottlenecks).

Further development will proceed in an interactive manner where additions to the data gathering and reduction capability will provide information on how to recognize particular bottlenecks.

2.5 Enhancements to VM Accounting Data

A side-benefit may be in the evolution of more detailed VM accounting records.

The accounting data presently accumulated by VM/370 is crude and incomplete by MVS standards. The proposed CP changes will greatly increase the amount of detailed information on system utilization at a user level. This, along with the information that is now collected by VMPT, will permit the production of the sort of 'accounting' data that is taken for granted in SMF.

Ultimately, it may prove desirable to modify CP to put this data directly into a new form of VM accounting data record. In the short time-frame of this project, it would be possible to evaluate the practicality of such a modification.

3.0 Scheduler Decisions: Analysis, Controls, and Feedback

3.1 Analysis

In general, specific implementation designs have not been laid out yet for the VMPT changes other than overall goals. This is partly because the information is not yet being gathered on which the new algorithms are supposed to base their decisions. Until we see what the data looks like, it is somewhat difficult to design code which will make decisions based on the data being gathered.

There are several clearly defined goals for the VMPT implementation.

1) The process will be iterative
   a. New data gathering capability leads to new algorithms
   b. A specific implementation will be refined as it is used

2) The CP nucleus implementation will not require VMPT
   a. CP will work as well as it does today if VMPT is not present

3) Algorithms will be self correcting
   a. Code will compare predicted results against measured results
   b. Comparison information will be available online for developer

3.2 Potential Implementation of Controls

Immediate Dispatch Pool

The next stage will develop the concept of the "immediate dispatch pool". All virtual machines that are past their time of day to start execution should be in the immediate dispatch pool. In addition, an attempt will be made to "round out" the pool with an intelligent choice of virtual machines from those remaining in the dispatch list, in order to minimize resource contention. This is an attempt to minimize resource thrashing in other areas beside real storage contention. The current eligible list and dispatch list structure do an adequate job of eliminating real storage contention by restricting the members of the dispatch list to those virtual machines who's working set sizes can fit into real storage simultaneously. On a large number of present day configurations, real storage is not a critical resource. The limitation of the dispatch list set based on real storage demand is optimizing a non-critical resource. There is sufficient real storage such that the number of virtual machine simultaneously allowed into the dispatch list (multiprogramming level) is saturating other resources (<1> Paging/Spooling Enhancement II).

The objective of the immediate dispatch pool implementation is to be more selective about which virtual machines are allowed to execute simultaneously in order to minimize contention for resources other than real storage. Part of the information that will be used in determining the size of the immediate dispatch pool is derived from the compute bound ratios. One of the objectives of the scheduling algorithm, in addition to minimizing contention, is to maximize critical resource utilization (like the CPU). Summing the compute bound ratio values of all virtual machines in the dispatch pool will give a good indication about whether or not the CPU can be 100 percent utilized. Other information to be obtained will be a combination of immediate information about virtual machine I/O activity and feedback information from the VMPT part of the scheduler identifying devices that represent major system bottlenecks (all devices will be assigned default numbers so that the algorithm will be at least partially effective even if the virtual machine scheduler isn't present or currently running on the system).

3.3 Feedback

The availability of information about how well the scheduler is doing its job is another area that is strongly missed today. Currently all that is available is information about how well the overall system ran. There is no detailed information about whether or not that is how the scheduler attempted to run the system (because it couldn't do any better) or if it was actually trying to do something else and was unable.

4.0 Specifying Resource Service Objectives

Another objective of the scheduling enhancements is the ability for a system administrator to easily define complex resource service objectives. At this time, it is not even possible to accurately (or inaccurately) discover what the capacity of the system is or to predict how it will perform under a given user workload or workload forecast.

The proposed CP and VMPT enhancements will make these tasks much easier and more accurate. The new CP queuing data will allow the use of more sophisticated performance evaluation and capacity planning models.

4.1 A Potential Set of Policy Rules for Management

Based on a rational assessment of system capacity, service objectives can be specified to the modified VMPT scheduler. Controls and objectives can be specified by individual virtual machines or groups of virtual machines. Activation of specific service policies can be triggered by various events (including time of day, system performance thresholds, forecast versus actual, etc.). Modification of the service objectives by negotiation, priorities, etc. will be easy to put into effect.

References

1) L. Wheeler, VM/370 Paging/Spooling Performance Enhancement II, IBM Research Report to be published

2) L. Wheeler, VM/370 Modifications, IBM Research Report RJ2906, Aug., 1980, 46 pp. PAM I/O, performance measurements.

Following numbers were taken with MC instructions inserted thru-out CP on an early release 3 CP system during late May and early June of 1975. To compensate for overhead involved in MC data gathering several thousand MC class were executed in a loop at Monitor Start. The average overhead to process each one was subtracted from the path timings. Path times for IOS associated functions will be greater because of the addition of alternate path code starting in release 4. Times to perform specific 'global' functions requires that the specific paths be identified and their timings added together. Following is about 1/2 of the data path entries that I have out of all paths accounting for .5% or greater of CP time Run was made with one VS1 virtual machine running MS02 job stream. Time values are microseconds to execute path on 145.

This work was the original that Bob Creasy & I did in support of the ECPS project -- Lynn Wheeler

path	count	time	percent cp
dsp+8d2 to dsp+c84	67488	374.	9.75
from 'unstio' end to enter problem state			
prg+56 to prv+46	69848	232	6.27
from prog. interrupt to priv. simulation			
ccw+33e to ccw+33e	64868	215	5.38
loop in ccw calling page lock			
fre+5a8	73628	132	3.77
'FRET'			
ccw + f4 to ccw = 33e	45297	213	3.73
from initial 'FREE' call to page lock call			
dsp+4 to dsp+214	84674	110	3.61
main entry to start of 'unstio'			
ptr+a30	124502	75	3.59
unlock page			
ccw + 33e to '3'	44839	207	3.58
from lock page to ticscan return			
ios+20	19399	474	3.55
dmkiosqv (before alternate path finding)			
fre+8	73699	122	3.47
FREE			
IOS+1c2 to DSP+4	27806	208	2.23
call SCN(real) until DSP entry (after I/O int)			
dsp+4 to dsp+c84	15105	374	2.18
asysvm entry until enter prob state			
sch+4	23445	221	2.00
ios+108 to ios+1c2	27952	165	1.78
i/o interrupt to call scn(real)			
scn+84	84359	54	1.76

```
dsp+93a to dsp+c84           11170   374     1.62
         sch call to entry problem mode
prv+46 to dsp+b8             20976   199     1.61
         non-i/o priv. instruction to new psw DSP entry
ccw+1252 to EXIT             26212   156     1.58
         ticscan return to exit
vio+13a to ccw+0             19405   191     1.43
         v.sio, ioblok free call until ccwtran call
vio+1d0 to ios+20            19399   181     1.36
         ccwtran return to DMKIOSQV call
ios+0                        8423    416     1.35
         DMKIOSQR
vio+3e to VIO+13a            19405   169     1.27
         vio entry(for sio) to 'FREE' call
dsp+214 to dsp+8d2           70058   45.     1.21
         'unstio' with no calls
vio+992 to unt+5a            19410   157.    1.17

ccw+28a to fa (via FREE)     26140   107     1.08
         ticscan return till loop back for next block
unt+9e to 116 (FRET)         44694   60.     1.03

unt+9e to 9e (PTR+A30)       65092   38      .97       (79.55 cumm.)

unt+116 to exit              19407   118     .89
         from FRET call to EXIT
vio+4 to 3e (SCN+84)         45240   49      .86
         vio entry until scan call for vdevblok
vio+3e to dsp+4              25504   86.     .685
         from SCN call to DSP (non-SIO)
SCN+4                        27979   69      .75
         real i/o scan (most IOS+1c2)
dsp+214 to 4ce (SCN+84)      14637   126.    .72
         'unstio' until scn call
-----
```

APPENDIX G. VM/370 2.15 DISTRIBUTION LIST

BILLIE BOVIE
IBM Corporation
Neighborhood Road
Kingston,New York 12401
Telephone: 373-2254

BILL BUCO
IBM Research center
P.O. Box 218
Yorktown Hts., N. Y. 10598
Telephone: 862-1611

TOM DOUGHERTY
IBM Corporation
Bodle Hill Road
Owego, New York 13827 --Dept. 106 Bldg 101a--
New York, 13827
Telephone: 662-3108

RON HUBB
IBM Corporation
3424 Wilshire Blvd.
Los Angeles, California 90010
HONE Phone: 285-1694

KENT LUTHER
IBM Corporation
1133 Westchester Ave
White Plains, N. Y. 10604
HONE Phone: 254-2267

JOE ROOSEVELT
IBM Corporation
673 Morris Ave.
Springfield,New Jersey 07081
Telephone: (201)-463-2266

HUBERT WROBEL
IBM Deutschland
Datenverabeitung
69 Heidelberg
Tiergartenstrasse 15, Germany

ALFRED SCHATEN
IBM Lab Boeblingen
Comp. Center Dept. 318U
Entwicklung und Forschung
Schoenaicher Strasse 220
703 Boeblingen, Germany

CLAUDE HANS
Cie IBM France
92102 Boulogne Billancourt
France

BOB ABRAHAM
IBM SCD
2651 Strang Blvd.
Yorktown Hts., N. Y. 10598
telephone: 8-721-2481

BOB DIXON
IBM Dept 997-H589, Bldg 622
P.O. BOX 12195
Research Triangle Park
North Carolina, 27709
Telephone:

R. G. van WEL
IBM Corp. SCD Uithoorn
P.O. Box 24
Uithoorn, Netherlands
Telephone:

DON CLARK
IBM Palo Alto Sci. Cntr.
1530 Page Mill Road
Palo Alto, Cal. 94304
Telephone: 624-3100

BOB PRINTIS
IBM L.A. Sci. Cntr.
1930 Century Park West
Los Angles, Cal. 90067
Telephone: 545-6375

ED HAHN
IBM Dept 107
Sterling Forest
Telephone:

M. Brown, et al, VM/370 in the GPD Engineering Laboratories, IBM Research Report (to be published).

L. Wheeler, VM/370 Paging/Spooling Performance Enhancement I, IBM Research Report (to be published), Control for page and spool record allocation.

L. Wheeler, VM/370 Paging/Spooling Performance Enhancement II, IBM Research Report (to be published), Control for page and spool record allocation.

L. Wheeler, VM/370 Modifications, IBM Research Report RJ2906, Aug., 1980, 46 pp. PAM I/O, performance measurements.

L. Wheeler, CSC VM/370 Extended II: Virtual Memory Management, IBM Science Center Report ZZ20-6002, July 1974, 19 pp., shared segments, migration.

B. Margolin, et al, Analysis of Free-Storage Algorithms, IBM System Journal 10, 283-304, (1971).

Y. Bard, Performance Criteria and Measurement for a Time-sharing System, IBM Systems Journal 10, 193-216, (1971).

P. Denning, Working sets past and present, IEEE Trans. Softw. Engrg., SE-6, 64-84, (Jan. 1980).

Y. Bard, Application of the page survival index (PSI) to virtual memory system performance, IBM J. of R&D 19, 212-220, (1976).

P. Denning, The working set model for program behavior, Comm. ACM 11, 323-333, (May 1968).

J. Rodriquez-Rosell and J. Dupuy, The design, implementation, and evaluation of a working set dispatcher, Comm. ACM 16, (April 1973).

VM/370 Resource Management Programming RPQ-PO-9006, Programmer and System Logic Guide, IBM LY20-1996.

J. Rodriques-Rosell, et al, Brown Univ. proposal for working set hardware on the System/360 Model 67, internal memo, (Nov. 1972).

James Morris, Demand Paging Through Utilization of Working Sets on the MANIAC II, Comm. ACM 15, (Oct. 1972).

Richard Cogger and Robert Cowles, SHARE VM/370 Scheduler White Paper, SHARE XLVI, (Feb. 1976).

L. Wheeler, CSC VM/370 Extended I: Dispatching/Scheduling, IBM Science Center Report ZZ20-6001, (July 1974), dispatching and scheduling.

T. Rosato, CMS 3.1 updated to run under VM/370., IBM Installation Newsletter Article, (1973).

B. Creasy, Psuedo Machines, IBM Cambridge Scientific Center internal memo, (1965).

L. Wheeler, VM/370 I/O Reliability Enhancement, IBM Research Report RJ3013, (Dec. 1980).

VM/SP COMMON System Programmer's Guide, IBM Internal document.

G. Zadow, CMS File System in Release 2 of VM/370 Basic System Extensions, IBM World Trade Systems Center report ZZ10-9892, (April 1979).

R. Carr, Virtual Memory Management, Stanford University STAN-CS-81-873, (1981)

G. Bozman, MDREORG - Mini-disk Reorganization Interal IBM document and program

Bob Braine Requirements for Computer Measurements