Palo Alto Research Center

# Virtual Memory Replacement Using Historical Information on Virtual Objects

XEROX

# Virtual Memory Replacement Using Historical Information on Virtual Objects

Robert B. Hagmann
Xerox Palo Alto Research Center

CSL-91-7 September 1991 [P91-00094]

© Copyright 1991 Xerox Corporation. All rights reserved.

Abstract: Most current virtual memory systems retain very little information about page usage. Typically, only a single bit of history is used. The history is usually only kept on pages currently resident in physical memory. With larger main memories, changes in applications, and the increases in file system caching in memory, these design decisions should be reviewed. If multiple bits of history are kept on virtual objects, whether they are memory resident or not, then there is the potential to improve memory replacement performance.

This paper describes a measurement technique that is used to monitor memory usage. It then proposes a new virtual memory replacement algorithm that is partially based on periodic, sequential, and transient behaviors. It also describes an approximation to LRU, called cluster LRU, that performs better on the programs measured than the usual clock algorithm. The performance of various algorithms is compared by trace driven simulation. Finally, this paper shows how a Translation Lookaside Buffer can be designed for a RISC machine to support both cluster LRU and detection of periodic, sequential, and transient behaviors.

CR Categories and Subject Descriptors: D.4.2 [Memory Structures]: Design Styles — Virtual memory; D.4.2 [Operating Systems]: Storage Management — Virtual memory

Additional Keywords and Phrases: Large Main Memories

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

# 1. Introduction

Much has changed since most current virtual memory replacement algorithms were designed over a decade ago. The amount of memory and the usage of memory was different then. The characteristics of program behavior when using hundreds of pages is different from using tens of thousands of pages of memory. Many machines are now single user workstations rather than time–shared machines. On a single user machine, the time for page fault is now directly visible to the interactive user, particularly when pages are faulted over a network. In addition, some systems now do not make major distinctions between pages for file system buffers and pages for program execution (see [Nels88] for Sprite and [Gingell] for SunOS 4.0). Mapped files further confuse the distinction between file system buffers and pages for program execution. Excessive demand by one part of the system may have a bad performance impact on the other.

Systems with vast amounts of memory are being constructed. Algorithms that work well with hundreds of pages may not scale very well to a million pages.

Below are examples of real situations that may cause problems. The author regularly experiences seven of these on his 48 MByte workstation.

transient access to a large amount of file data (e.g., grep through several megabytes of source)

applications that are used every few minutes, but are inactive between uses (e.g., spell checkers, compilers, loaders, debuggers, typesetters, calendars)

garbage collection

interactive and background use of a workstation, where the background program will consume lots of memory so that the interactive pages tend to get paged out if not frequently referenced; sometimes called the "cold window problem"

multi–media data that is transient (e.g., a single screen image is 4 MB [24 bits per pixel, 1024 x 1280])

periodic use of large applications, where the total size is larger than memory

algorithms that cycle through phases, as in some scientific programs

sequential access to very large files

copy of lots of file data using NFS from a client workstation

All of these examples show memory demands that are hard for most current virtual memory replacement algorithms.

Many users never do anything similar to these examples and do not experience any problems. If all you do is read your (text) mail, edit with vi, and run the C compiler, then there won't be a problem. Others buy so much memory that the problem is manageable. Memory is still the dominant cost in many systems. Poor memory management can cause users to buy more memory than they really need.

Operating systems papers are about abstractions and implementations. This paper is primarily about the abstraction of virtual memory replacement. It proposes a small paradigm shift. Virtual memory replacement should be based on virtual object histories. Most popular systems use a single bit of reference data, and the bit is only kept for pages that are in physical memory. With virtual object histories, periodic, sequential, and transient reference behavior (discussed more below) can be detected or estimated. Based on histories and algorithms, substantial improvement in the page-in behavior of a system can be made. This paper presents algorithms that have significant performance improvements for a small set of traces. One algorithm is an approximation of LRU that does substantially better than "clock" for the traces. A second algorithm divides pages into two pools: those pages that have a periodic, sequential, and transient reference behavior and those pages that have an LRU reference behavior, and then selects pages for replacement from these two pools.

The rest of this paper is organized as follows. Section 2 has a discussion of the issues and history of memory replacement; Section 3 investigates the reference patterns that are typical for problem applications; Section 4 describes the tracing technique, the simulator that was built, and the algorithms that were simulated; Section 5 reports on the measurement and simulation; Section 6 gives a straw man design of hardware that can implement one of the proposed algorithms; Section 7 is the conclusions and proposals for future work.

## 2. Discussion

Currently, it is typical that virtual memory systems are built for short term demands. Until fairly recently, paging was a major bottleneck in system performance. Although paging is now not normally a bottleneck, systems are still plagued with occasional, substantial delays due to poor paging decisions. Part of this can be easily traced to the system making reasonable short-term decisions that have a bad medium-term effect. Some of the cause of this is the algorithms in use today, as they are just tuned versions of older algorithms. The older algorithms were concerned with short-term memory demand because memory was too expensive to have enough to

allow medium – term effects to be important. Even though they have been re – tuned (hopefully), the basic assumptions of algorithms partially determine the results.

This is not a new observation. The book on the 4.3BSD UNIX Operating System [Leff89] says:. "4.3BSD is not perfect. In particular, the virtual – memory system needs to be completely replaced. The new virtual – memory system needs to provide algorithms that are better suited to the large memories and slow disks currently available, and needs to be less VAX architecture dependent."

For example, consider the global clock algorithm that was used in 4.3 BSD ([Baba81] and see discussion in [Leff89] Section 5.12). This algorithm is used in similar systems such as SunOS and USL System V Release 4. Global clock is a commonly used approximation of LRU. A variant of clock is also used in IBM's VM system [Tetz87]. Sprite uses clock for VM pages, but exact LRU for file system pages [Nels88]. Mach also has a LRU approximation. It uses two queues of pages: active and inactive lists. FIFO replacement is done on the inactive queue. Pages are moved from the active queue to the inactive queue when the inactive queue is too small. Faults on inactive pages move them back to the active queue [Youn89]. This is similar to clock in that it makes pages vulnerable to page – out in a window, with a reference in the window preventing page – out.

Basically as implemented in SunOS, the clock algorithm has two hands that move together but are separated by some fixed number of pages. The anticipated need for memory causes the hands to turn. The hands point to physical memory pages that are conceptually arranged as a clock. Each physical page has a reference bit and a modified bit. The front hand turns off the reference bit. A reference to a page causes the hardware to turn on the reference bit. The back hand samples the reference bit. If the reference bit has not been set, then the page is added to the "cached free" list. If modified bit is set, the page must first must be cleaned by writing it to disk. Once the page is added to the cached free list, its identity and contents are not lost. If the page is needed (e.g., faulted) before it is used as a free page, it is removed from the cached free list and used.

Note that this is an algorithm that is physical memory based: the only reference data kept is for pages in physical memory. In addition, the amount of data is one bit or less. When the page is not between the two hands, the value of the reference bit is meaningless.

When everything fits, the clock does not turn. When the clock is not turning, no sampling of memory usage is being done at all (almost). If there is sudden memory

demand, the system reacts by turning the clock. Extremely long – time – to – last – reference pages are not replaced if they are not near the clock hands. At high clock turning rates, it takes only two seconds between the front and back hand examining a page (using the SunOS 4.0 tuning parameters). If in this narrow window the page is not touched, it is taken and made into a free page. At worst, this degenerates into random replacement. In practice random replacement would not happen for all pages, but it could happen for a good number of pages. In any event, the reference history during the time when everything fit is lost.

Systems are tuned to make the clock turn slowly. For the DEC VAX (or at least for the VAX – 11/780) this was important. There are no reference bits on the VAX. vmunix (and the BSD UNIXes) simulates them by invalidating the page, taking a trap if the page is referenced, and recording the reference [Baba81]. Using hardware with reference bits, the cost of sampling pages is greatly reduced. On a SparcStation – 1, turning the clock a few times faster than normal using SunOS 4.0 had a barely noticeable change in system CPU time (as measured by vmstat). The default maximum clock rate was tuned in SunOS 4.1 to be a factor of five faster than in SunOS 4.0. Turning the clock "too fast" makes the system do page cleans (write of dirty pages to the swap file) so it is not easy to get a good cost figure. The point is that turning the clock quickly does not and should not cost too much CPU time.

With larger memories and single user workstations, faults are by comparison much worse now that 20 years ago. For a PDP – 10 (late 1960's), it took about 30,000 instruction times to answer a fault. For a SparcStation – 2, a fault takes twenty times as long, when measured in instruction times. Systems are getting more and more I/O bound [Oust89]. On a time – shared PDP – 10, the time of a page fault was not lost: other users would get the CPU. On a workstation, there are often multiple activities running concurrently, but the interactive use of the machine determines how happy the user is with it. A fault in the interactive software stops the machine as far as the user is concerned. As our machines do not fault very much, a storm of faults is very noticeable and very obnoxious. We tend to remember the storms and forget the normally good performance. The variance is high and people do not like high variance.

Systems are merging uses of physical memory [Abro89]. Before SunOS 4.0, the SunOS virtual memory system and the file systems had separate, fixed – size pools of physical memory to administer. For virtual memory bound or for file system bound tasks, this was unfortunate since physical memory was not being used where it was

needed. SunOS 4.0 changes this by making one pool of physical memory from which both the virtual memory system and the file system(s) compete [Moran]. (Some data structures are allocated from kernel memory, but all buffer pool pages and virtual memory pages are allocated from the same pool.) However, the clock algorithm is still used to determine what page to evict when memory is needed. This means that transient file system behavior that used to destroy the buffer pool but leave virtual memory intact, now destroys buffering and caching of all physical memory. Typical transient behavior that does this is by "greping" (searching for a pattern) a file that is bigger than memory. SunOS 4.1 had some performance tuning so that some types of clustering and sequential behavior can be told to the system as hints (madvise system call) [McVo91].

Other systems have used large physical memories to improve file system performance. In Sprite, very large client memories are used. There is a soft boundary between the file system and virtual memory pages. File system pages are artificially aged to make better VM performance.

## 3. Reference patterns

Individual, isolated page faults are usually not a problem. Except for single threaded, real-time applications, they influence the latency or throughput of a system in a minor way. But, page fault storms do effect the system. Why do page fault storms occur? By looking at the reference patterns, three types of non-LRU reference behaviors emerged. The patterns are:

Periodic: An object or a page is used infrequently, but is re-referenced. The period may or may not be fixed. Examples: calendars (the author just had a 15 second page fault storm opening his calendar), garbage collection, and receiving mail.

Sequential: When an object is used, it is read from the beginning to the end. This is typical for files. Examples: grep of a large file, loading an image file to a workstation, and playing a video segment.

Transient: An object is referenced for a brief time, then not again for a very long time. Examples: grep of many files in a directory and listening to voice mail.

Periodic behavior, where the period is longer than the sampling time of the virtual memory replacement algorithm, is not handled well by algorithms that approximate LRU. References are so far apart that they are not noticed by the algorithms. For sequential references, the page referenced previously is the least likely to be referenced next. Transient references also are not good predictors of future

references.

. For all of these cases, the LRU assumption is violated: all of these types of references are not LRU. Pages that have references that are periodic, sequential, or transient appear to be better served by algorithms that replace pages in a MRU – like fashion, while other pages are replaced LRU. The challenge is to discover these types of reference patterns and integrate this information into the resource management for memory.

Of these three patterns, the one that is most commonly recognized in current systems is sequential. With sequential access, pre – paging or file system read – ahead can be used to stream data in more efficiently by overlapping processing with reading, and by accessing secondary or network file storage in larger pieces. Also, post – flushing of pages once they are used can get high bandwidth access to data in a fixed buffer size (as opposed to polluting memory with transient data).

If LRU or Global Clock replacement is done for transient pages, then the transient pages will evict other pages and will be retained in memory for a long time (as long as the clock or LRU replacement takes (e.g., minutes)).

Other access is periodic. In fact, the first virtual memory system, Atlas [Kilb62], was built to principally handle periodic behavior. Array references in small memory systems often are periodic (e.g., matrix multiplication).

For a synthetic example of periodic behavior, suppose that there are four equal sized tasks that the user does periodically, yet there is only enough memory for three. The user might edit, compile, load, then execute/debug cyclically. LRU, or its approximation Clock, carefully does the optimally worst. For example, in the execute/debug phase, it takes pages from the editor. The editor then must page in its pages when execute/debug is done. All phases start with nothing in memory. The system runs the same as if it had only one third the memory.

In some cases, MRU replacement is optimal. This has been shown in databases for certain types of access (e.g., sequential) [Chou85]. Post – flushing pages in sequential access is another example of MRU behavior. In the example above, if MRU between phases is done for the edit, compile, load, and execute/debug example, out of every three phases two start pre – paged. (Startup by loading the edit, compile, and load phases. Execute/debug steals pages from load, so edit and compile start pre – paged. Load steals pages from compile. Execute/debug and edit start pre – paged, and compile steals pages from edit.) Note that true MRU replacement is not proposed: it is the phases of the example that are handled MRU, not the pages inside of a phase.

Notice that there is an essential conflict in behavior. There are the LRU pages and the MRU – like pages. Pages that are being used periodically, sequentially, and transiently are MRU – like, while LRU works well for all other pages. While it is reasonable to compare MRU – like pages with other MRU – like pages and LRU pages with other LRU pages, it is unclear how to compare MRU – like pages with LRU pages.

## 4. How to proceed

One standard way to proceed it to measure existing systems, propose different algorithms and build models, and simulate using trace driven simulation. Algorithms should be proposed without regard for the execution cost. It is the abstraction that is important, not implementation. Either hardware changes or clever implementations might be found for algorithms of merit. Finally, proposals for good, efficient implementations should be made.

### 4.1 Measurement

Briefly, the method of measurement is very simple. First, detune the operating system so that it is forced to sample page references and modify data more often. For SunOS, this was done by speeding up the clock and forcing it to turn (e.g., use adb to patch the kernel's constants). Second, periodically read the kernel data structures for physical pages (or virtual pages if that is easier) from a user – level process. Record the reference and modify bits to a disk file for all pages for all virtual objects. Write trace data bitmaps for all active objects. For SunOS, the kernel was also recompiled with the "TRACE" option. This allows the monitoring code to track the use of the swap area so that faults can be distinguished from zero filled pages.

The period between samples was set at five seconds. Each interval is called an epoch. Combined with the clock turning rate, this leads to a granularity of 5 – 10 seconds in the trace data. But it takes minutes of inactivity to evict a page using LRU for memory sizes that are common today using the traces obtained. Hence, the coarseness of the 5 second sample time is not very important.

### 4.2 Simulation

A simulator for multiple virtual memory algorithms and for this trace data has been constructed. Global Clock, LRU, Atlas, Random, and MIN were all programmed, as were two new algorithms that will be described in section 4.3

For each epoch, the normal way to simulate was to pretend that the references

occurred in the order that they showed up on the trace. To verify that this ordering did not have a great impact, randomization of the ordering was also done. This did not produce any significantly different values from the simulation.

The Global Clock algorithm was discussed before. The simulation for it collects all the references for a 5 second epoch. It then proceeds in 1/10 of a second subintervals. In each subinterval it applies 1/50 of the references of the interval. Based on the references, it may decide to turn the clock. The rate of turning is computed by the clock algorithm.

Atlas is the algorithm used in the Atlas computer [Kilb62]. This was the first virtual memory machine. It is primarily concerned with periodic behavior.

LRU is Least Recently Used. The granularity of LRU is the epoch size.

Random selects a page to replace at "random."

MIN is minimum number of faults possible [Bela66]. It requires knowledge of the future. Thus it is not a practical algorithm, but useful for reference. It is optimal.

### 4.3 Proposed Algorithms

#### 4.3.1 SPT

The first new algorithm attempts to detect periodic, sequential, and transient behavior (SPT). Every epoch, SPT builds two heaps. SPT detects sequential, periodic, and transient behavior by processing historical reference data for pages of virtual objects. For periodic and transient detection, the reference string of a page is broken in "reference runs" and "unreferenced runs". A reference run starts with a page reference and continues until the page is not reference for six epochs. The first page of the six becomes the first page of the unreferenced run. The period is computed as the number of epochs between the last two reference runs. Pages from this detection are put into a heap that is ordered by the absolute value of the expected time to next reuse (the next – time heap). Pages with the longest time to reuse are taken first. Note that this achieves MIN paging behavior provided that sequential, periodic, and transient behavior are correctly recognized. All other pages are added to a different heap (the LRU heap). These pages are ordered by the last touch time where pages that have not touched longest are taken first (LRU). Various ways to choose between the two heaps have been tried. The current algorithm is:

take very, very old LRU pages first (30 epochs)

take pages alternately from the two heaps while there are somewhat old pages in the LRU heap (15 epochs)

take all the pages from the next – time heap until it is empty

take pages from the LRU heap

No claim is made that this is the best that can be done! Of a dozen or so alternatives, it did the best on the trace data. Various setting for the constants in the algorithm were explored, with the values reported here (6, 30, and 15) working well together. Much more extensive work is needed to find better algorithms.

#### 4.3.2 Cluster LRU

The second algorithm was to approximate LRU over clusters of memory (Cluster LRU). Simulation showed substantial differences between Clock and LRU. Differences of 10 – 40% were common. We speculate that the causes for these differences have always been present between clock and LRU, but were secondary effects. With large memories the secondary effects are becoming noticeable.

Cluster LRU works as follows. A somewhat detailed history of usage is kept per page. It is assumed that acquiring this data is very cheap (e.g., block move of the reference bits from the TLB). When a page is needed, a set of pages (a cluster) is examined. The clusters are fixed in size and are a contiguous run of physical pages. The page that is least recently used in the cluster is the victim.

Rather than building a heap for all memory as in LRU, only a restricted subset of memory (a cluster) is examined. Usage data per cluster is examined only when a page is needed. This avoids the cost of maintaining a global heap for all of memory. Also, only when pages are actually needed does any significant processing need to be done. Good results were obtained with a cluster size of 16.

Simulations indicate that cluster LRU closes the gap between global clock and LRU. While not quite as good as LRU, it is much better than clock. It also has the potential for a reasonably fast implementation.

## 5. Measurement and Simulation

Several traces were taken of workstations where users were trying to use their machines aggressively. That is, the traces were not of people reading their mail, but rather of situations where the user was trying to push the system. All of the traces were taken under SunOS 4.1. The three traces are designated "A," "B," and "C" below. These traces used window systems, editors, compilers, linkage editors, grep, large "ls" and "find" commands, and simulation executions.

Traces were then run through the simulator. Figure 1 shows the number of

simulated faults for Trace A. LRU – 16 is Cluster LRU with a cluster size of 16. Random does random replacement of memory. Atlas is the algorithm from the Atlas system [Kilb62] and is included since it only cares about periodic behavior. All of the traces are on SUN Sparcstation – 1's which have 4K pages.

Of course, MIN is the best. Note that SPT, LRU, and Cluster LRU all perform much better than Clock. Clock is much better than Random. Also note that Atlas has very bad performance, particularly for large memories. Probably the reason for this is that Atlas was built to react to periodic behavior in the reference string, and this did not work well for large memories.

SPT, LRU, and Cluster LRU are all better than Clock, but how much better? The raw number of faults does not tell the story. MIN is the best one can do. How much does SPT or LRU close the gap from Clock to MIN? Figure 2 shows the percentage of faults SPT, LRU, and Clock takes over those of MIN (MIN would have a table entry of all zero's). Small numbers are good. Figure 3 shows the decrease in the number of faults by SPT and LRU from Clock, expressed as a percentage of the number of MIN faults. At first this may seem like a strange statistic, but it is a very fair way to show how much better SPT is. MIN is the best you could do with knowledge of the future, and SPT covers 20% to 50% of the gap from Clock to MIN. Although SPT is consistently better than LRU, the difference is small compared to how much better either is compared to MIN.

One interesting observation is that the most useful improvement occurs at moderate memory sizes. One possible reason for this is that small memories will force the clock to turn quickly. Short – term demand is still very important, and Clock does fine for short term demand. On the other end of the graphs, with very large memories the traces did not stress much of the system. The fault rates are so low that improvement is hard and not too beneficial.

Figures 4, 5, and 6 present the results Trace B. Note that the axis of the graphs are sometimes different. The traces used different total number of pages, so that the interesting parts of the curves are at slightly different values. MIN achieved a stable value over 6,000 pages for Trace B, so it was questionable to simulate much higher memory sizes. Also, the maximum number of referenced pages per interval is different, so different minimum number of pages are used.

Trace B shows the same basic pattern: SPT, LRU, and Cluster LRU are all better than Clock. The results are mixed about whether LRU or SPT does better. It's a tossup.

Trace C (Figures 7, 8, and 9) used even fewer pages. But again, SPT, LRU, and

Cluster LRU are all better than Clock. However, the gap between them is smaller than in the other traces.

Measurements of a user's perception of a system are harder to make than simply counting faults. To approximate the "variance" that a user would see, the five intervals with the highest number of faults were computed for Trace A for MIN, Clock, SPT, and LRU. These are scaled to MIN and presented in Figure 10. The results are about the same: SPT does a little better than MIN, and both do significantly better than Clock.

While it is impossible to make general statements based on trace data, the following conclusions are supported by the traces and simulations:

SPT, LRU, and Cluster LRU are all better than Clock

Cluster LRU with cluster size 16 is close to LRU. The difference between Cluster LRU – 16 and LRU is many times smaller than the difference from LRU to Clock.

SPT sometimes achieves a few extra percent improvement over LRU, but this improvement is not always apparent.

Clock does much better than Random.

Atlas does not perform well.

Given the potential for improvement, how can this be realized? The next section shows how hardware and software can cooperate to implement Cluster LRU.

## 6. Implementation of Cluster LRU

There are two parts to the implementation. First, the system has to be monitored so that historical information about page references is kept. Second, based on this data both SPT and cluster LRU have to be implemented. Of these, cluster LRU seems to be the harder and will discussed here. SPT can use page faults to detect sequential behavior and can do periodic and transient detection based on the historical information about page references.

In this section, the system of choice is loosely based on the SparcStation – 1. This system takes the RISC idea to the extreme with its Translation Lookaside Buffer (TLB) design, which they call Hardware Address Translation (HAT). A TLB is a cache that maps virtual addresses to physical addresses when the processor takes an instruction or data cache miss. Details of the HAT are SUN proprietary, so this discussion is based on what is non – proprietary (for example, from [Irla90].) The features of interest of the

HAT are:

probes that miss the HAT cause a trap to the operating system (in true RISC fashion)

on a trap, the operating system fixes up the HAT, possibly including flushing a line to make room for a new entry, then restarts the instruction

entries in the HAT are lots of pages wide, each with a valid bit

pages in the HAT have reference bits; the hardware sets these bits without trapping

With this design, one thing that is desirable is a fast way to get a copy of all the reference bits and to clear all the reference bits. This possibly can be done by use of video RAM. The HAT is so large (thousands of page entries) that executing even a modest number of instructions per entry every five seconds would have a high overhead. A hardware implementation of a TLB would have to provide similar features.

Suppose that the kernel read and saved a copy of the bitmap of references from the HAT. While a line stayed in the HAT, the only overhead for sampling would be the interrupt every five seconds to get a copy of the reference bits and clear the reference bits. An interrupt every five seconds plus just a bit of processing is nearly free.

Based on HAT address, a history of reference is kept by the above. Note that the kernel can have full knowledge of what is in the HAT since it put it there; the HAT is a software maintained cache used by the hardware. With a properly sized HAT, lines remain in the HAT for extended periods. A steal has to happen when a line is needed and there are none free. If the HAT line size is a simple multiple of the word size (e.g., 32 or 64 entries per line), then the reference data from the HAT has to be copied to a data structure associated with the virtual pages corresponding to the HAT line. If this is word aligned, does not have too deep a history, and is an integral number of words, copying the reference data should present no major performance problem. After some number of 5 second samples, a set of them should be compressed to a single sample (say to make 2 minute samples). Only a few hours of history are needed, so the storage cost of this is reasonable.

Consider how cluster LRU would work. A cluster size would be chosen, say 16 pages. When a page is needed, the current cluster is examined. For each page, the reference data is kept with the HAT reference data (if the page is currently in a mapped line) and/or with the virtual page data. Physical pages with multiple virtual mappings would have to interrogate all mappings to find the least recent one. While

this does take some time, the time is only when a page is needed. Normal system operation when everything fits is nearly unaffected. When a page is needed, examining bitmaps for 16 pages does take some time, but it is not all that long. A mask has to be constructed and probed against a bitmap. Only a few instructions are needed per probe and only a few tens of probes are needed per page. Suppose that the averages were 3 instructions per probe and 15 probes, then 45 instructions would be needed for the loop. Adding a few instructions for the mask generation gives 50 instructions per page. With 16 pages per cluster, that is 800 instructions. On a 12 MIP machine, that is 60 microseconds. Note that this is nearly three orders of magnitude faster than a page fault time. 800 instructions is also comparable with the number of instructions to issue an I/O on many systems (and a small fraction of the number of instructions to issue an I/O on some systems). 800 instructions is an acceptable overhead.

This section has shown that straightforward TLB design can lead to low cost reference data collection. With this data, cluster LRU is straightforward to implement at reasonable performance.

## 7. Conclusions

Large memories require new strategies for page replacement. Both large memory personal workstations and very large memory server and time-shared machines should not run Global Clock. Better memory resource management is possible based on histories of reference data.

Our measurements show that existing memory management does not perform acceptably on sequential, periodic, and transient references. These types of references are MRU-like in nature, not LRU. They can cause page fault storms. This leads to high variance in the performance of the system or to consistently poor performance.

This paper presented two new algorithms for memory management. The algorithms seem to provide significant improvement over existing approaches. Faults decrease approximately 10-40% in the range of interest for the memory traces. About one third of the gap between Global Clock and MIN is closed by the new algorithms.

We believe that our designs are practical and feasible. With small hardware changes from an existing product, the hardware and operating system can cooperate to obtain historical reference information. The overhead for obtaining the reference data is low. The cost of victim selection is reasonable.

This work has focused on a single operating system and hardware base. To validate this work, other operating systems, other benchmarks, and other hardware must be studied. The algorithms proposed in this paper are only two of a large class of algorithms. Other algorithms should be studied.

Nevertheless, we feel that our results are broadly relevant across architectures. With the continued, projected growth of memory sizes, the problems addressed in this paper will likely become the dominant problems in memory management.

## References

[Abro89] V. Abrossimov, M. Rozier, and M. Shapiro, Generic Virtual Memory Management for Operating System Kernels. Proceedings of the Twelfth Symposium on Operating Systems Principles. Litchfield Park, Arizona. Dec. 1989 (Operating Systems Review, Vol. 23, No. 5).

[Baba81] O. Babaoglu and W. N. Joy, Converting a Swap – Based System to do Paging in an Architecture Lacking Page – Reference Bits. Proceedings of the Eighth Symposium on Operating Systems Principles. Pacific Grove, California. Dec. 1981 (Operating Systems Review, Vol. 15, No. 5).

[Bela66] L. Belady, A study of replacement algorithms for a virtual – storage computer. IBM Systems Journal. Vol. 5, No. 2, 1966.

[Chou85] H. Chou and D. DeWitt, An Evaluation of Buffer Management Strategies For Relational Database Systems. Proceedings of the Eleventh International Conference on Very Large Data Bases. Stockholm. August, 1985.

[Gingell] R. Gingell, J. Moran, and W. Shannon, Virtual Memory Architecture in SunOS. SUN White paper.

[Irla90] G. Irlam, A Guide to Sun – 4 Virtual Memory Performance. SUN – Spots Digest (an electronic mail digest). Vol. 9, Issue 257 (July 12, 1990)

[Kilb62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, One – Level Storage System. IRE Transactions on Electronic Computers. Vol. EC – 11, No. 2. (April, 1962)

[Leff89] S. J. Leffler, M. K. McKusick, M. Karels, and J. Quarterman, The Design and Implementation of the 4.3BSD UNIX Operating System. Addision – Wesley, May, 1989.

[McVo91] L. McVoy and S. Kleiman, Extent – like Performance from a UNIX File System. Proceedings of the Winter 1991 USENIX Conference. Dallas, Jan. 1991.

[Moran] J. Moran, SunOS Virtual Memory Implementation. SUN White paper.

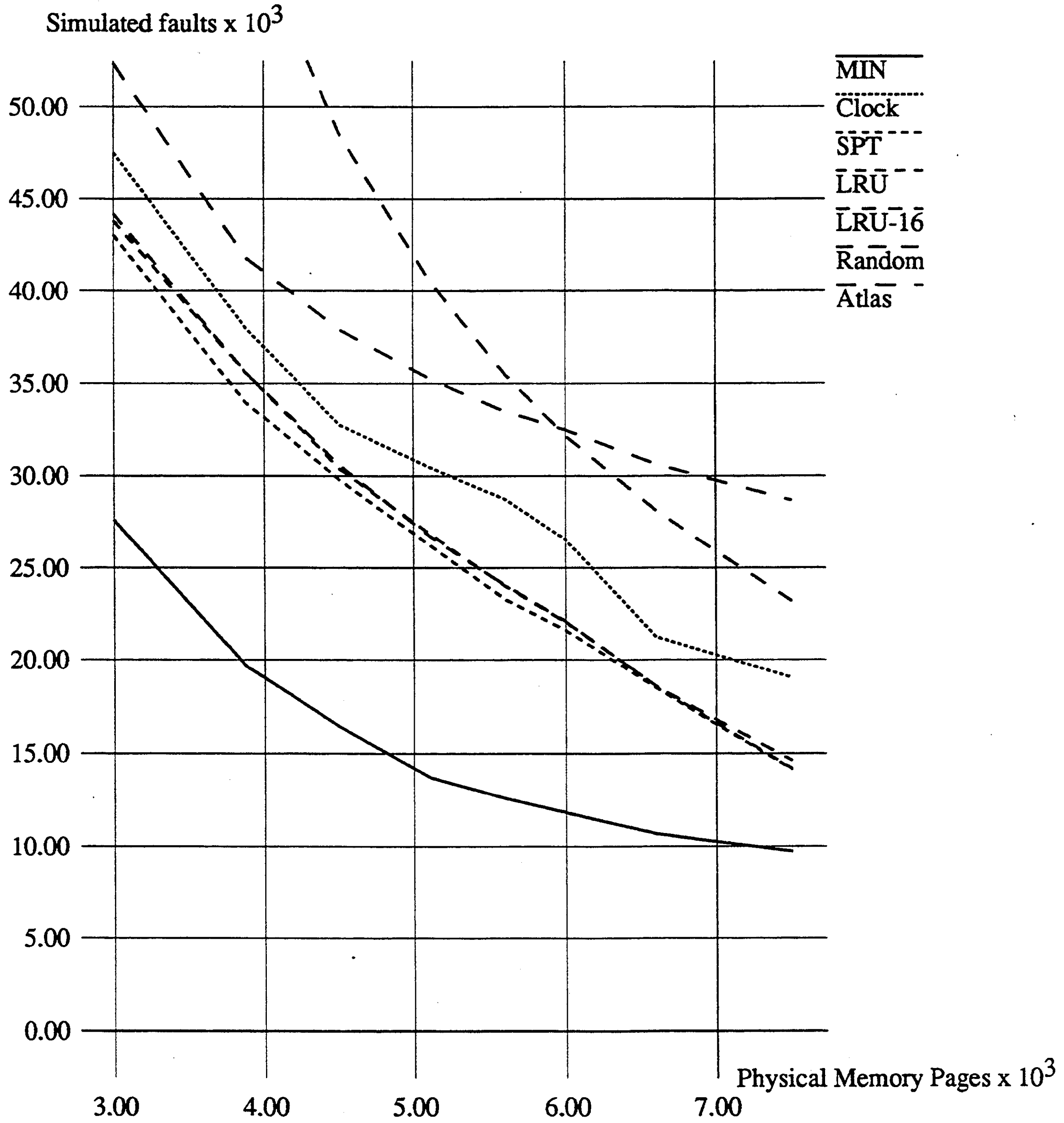
[Nels88] M. Nelson, Physical Memory Management in a Network Operating System. PhD thesis, University of California, Berkeley, Report Number UCB/CSD 88/471, Nov. 1988.

[Oust89] J. Ousterhout and F. Douglis, Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Operating Systems Review. Vol. 23, No. 1. (January, 1989)

[Tetz87] W. Tetzlaff, T. Beretvas, W. Buco, J. Greenberg, D. Patterson, and G. Spivak, A page-swapping prototype for VM/HPO. IBM Systems Journal. Vol. 26, No. 2, 1987.

[Youn89] M. Young, Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. PhD thesis, Carnegie Mellon University, Report Number CMU-CS-89-202, Nov. 1989

# Figure 1: Trace A: Simulated Number of Faults



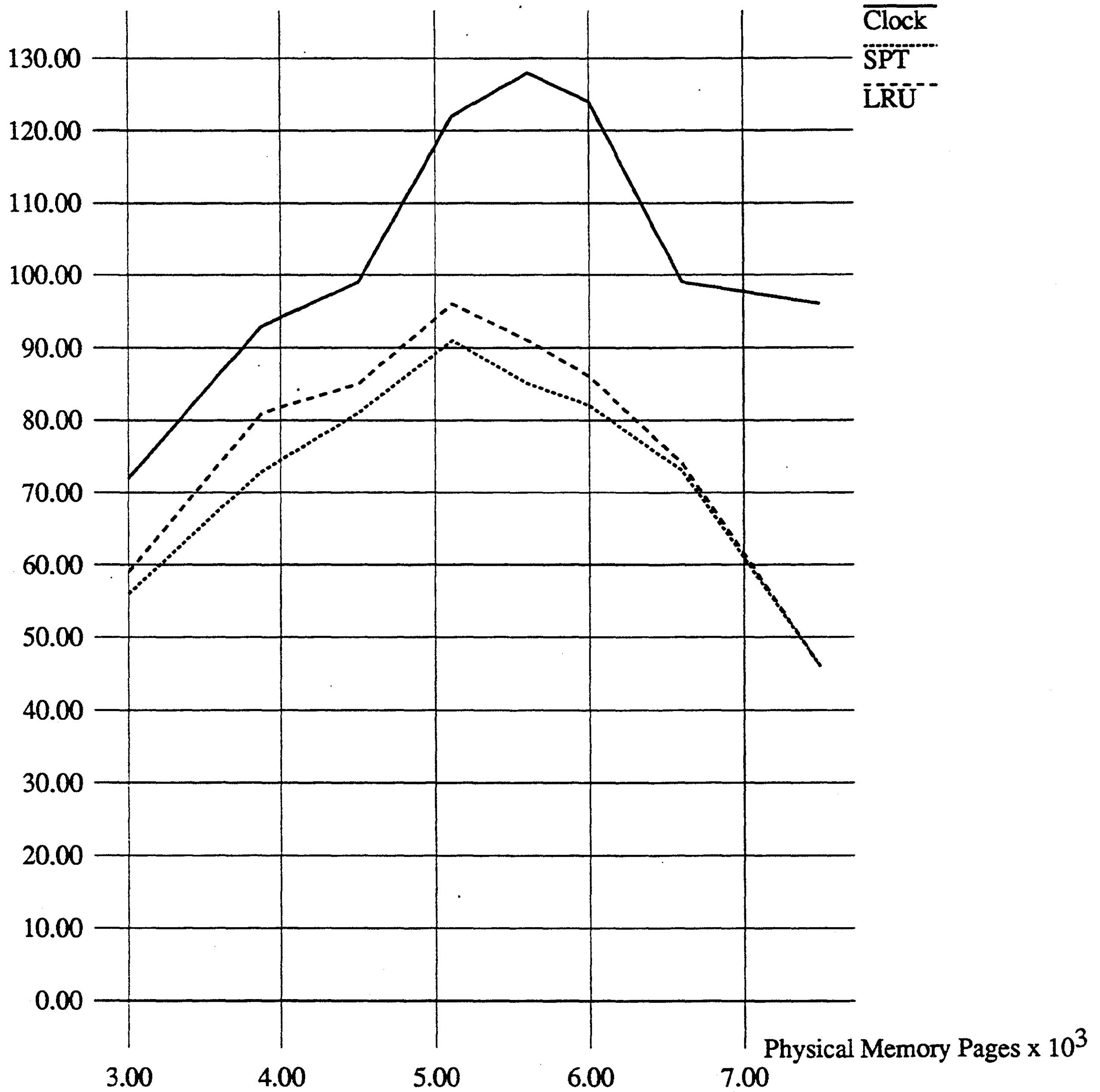
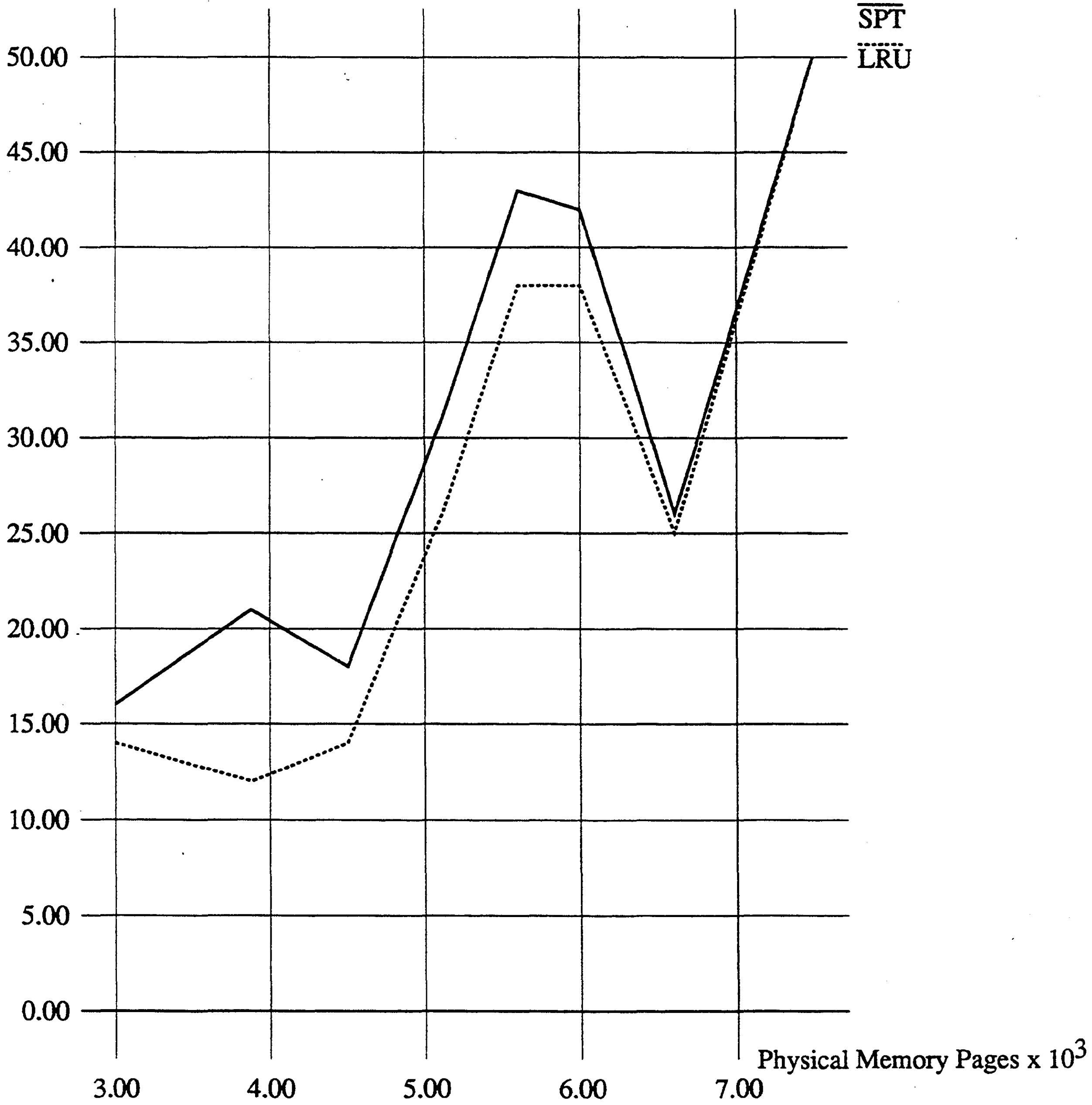


Figure 2: Trace A: Percent Over MIN
Percent Over MIN
Clock
SPT
LRU
130.00
120.00
110.00
100.00
90.00
80.00
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
3.00
4.00
5.00
6.00
7.00
Physical Memory Pages x 10^3

**Figure 3: Trace A: Percent Better Than Clock Scaled To MIN**



# Figure 4: Trace B: Simulated Number of Faults

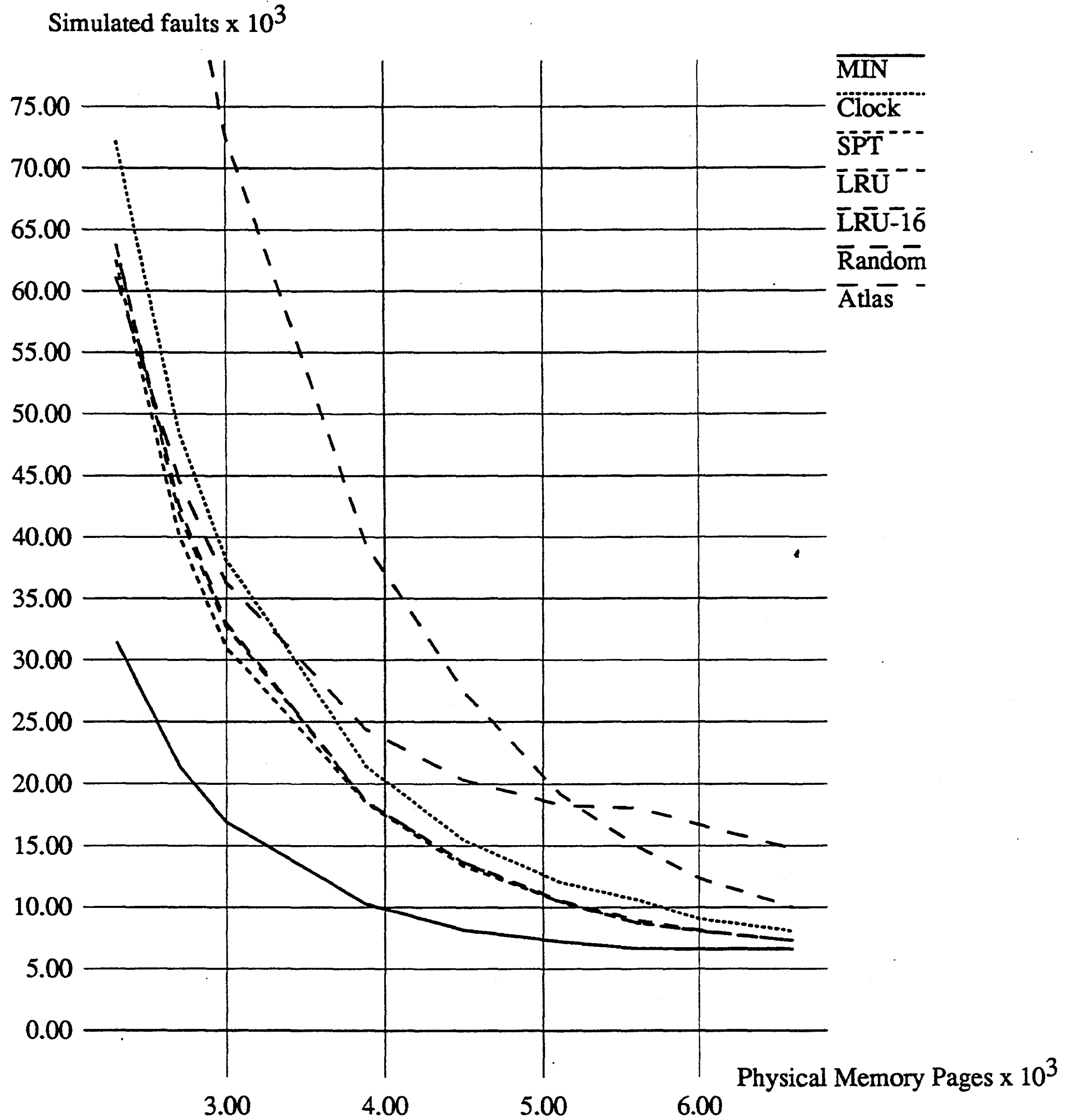
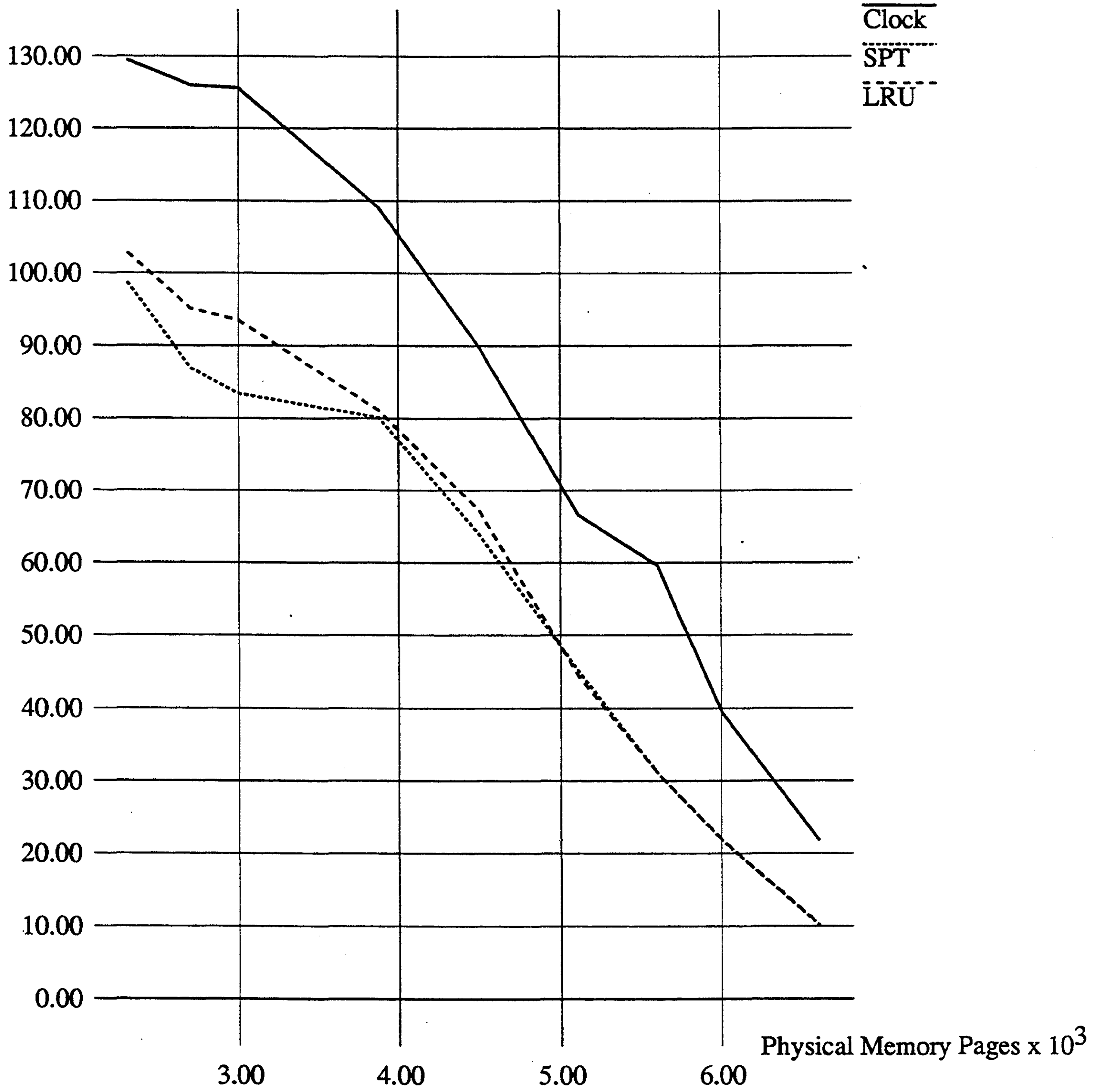
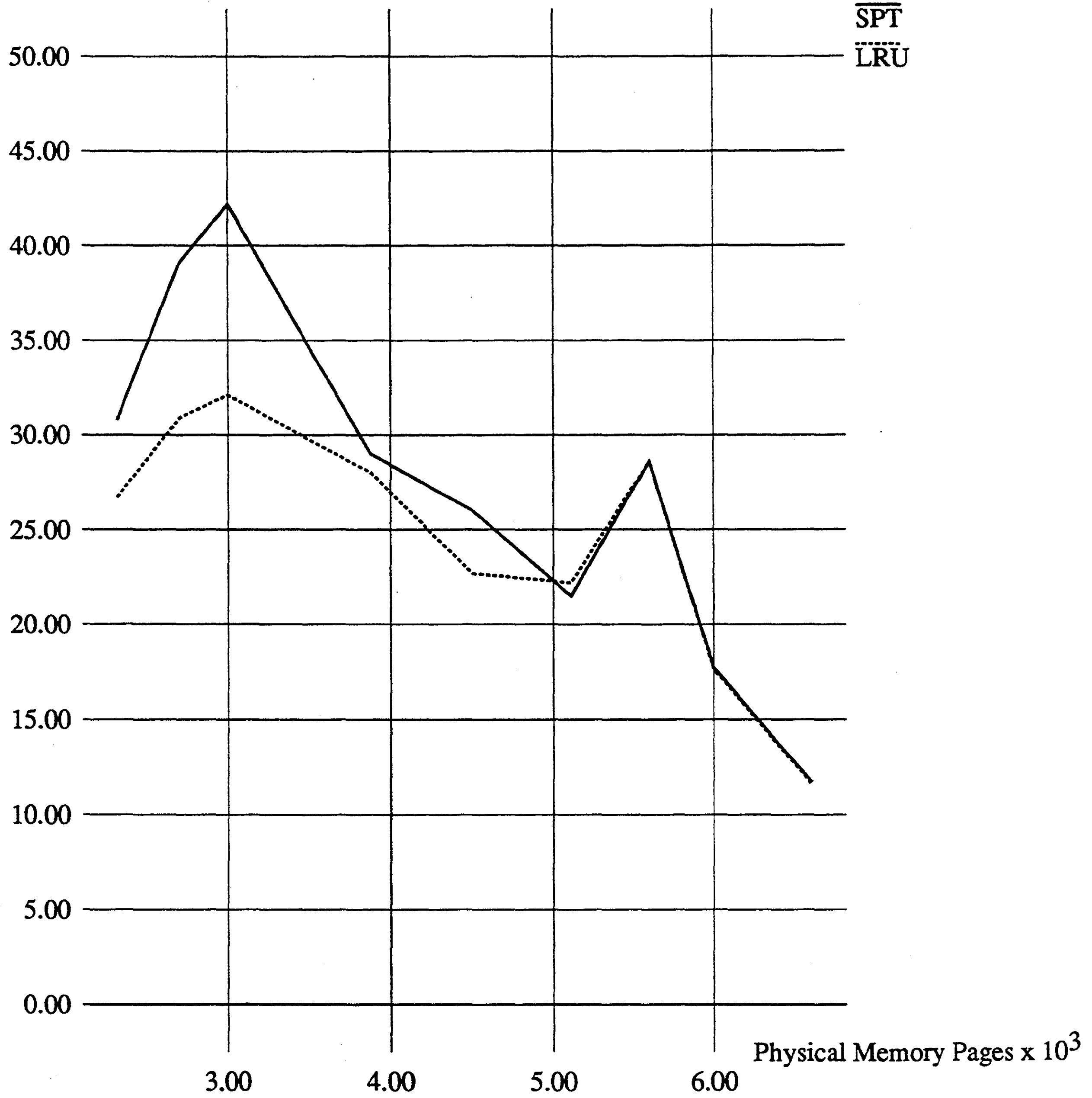


**Figure 5: Trace B: Percent Over MIN**



## Figure 6: Trace B: Percent Better Than Clock Scaled To MIN



# Figure 7: Trace C: Simulated Number of Faults

Simulated faults x $10^3$

10.00
9.50
9.00
8.50
8.00
7.50
7.00
6.50
6.00
5.50
5.00
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00

MIN
Clock
SPT
LRU
LRU-16
Random
Atlas

2.50 3.00 3.50 4.00 4.50 5.00

Physical Memory Pages x $10^3$

**Figure 8: Trace C: Percent Over MIN**

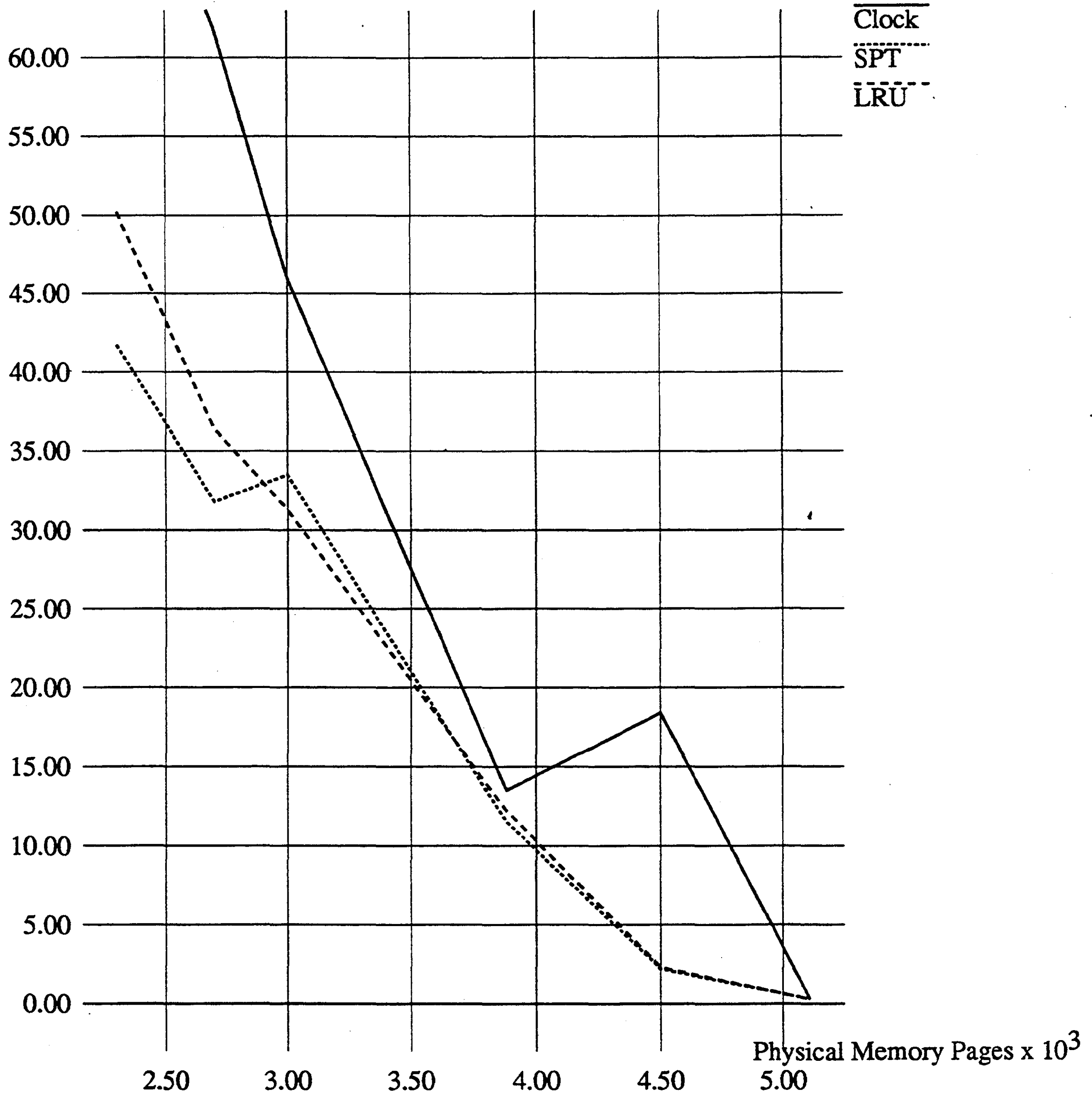
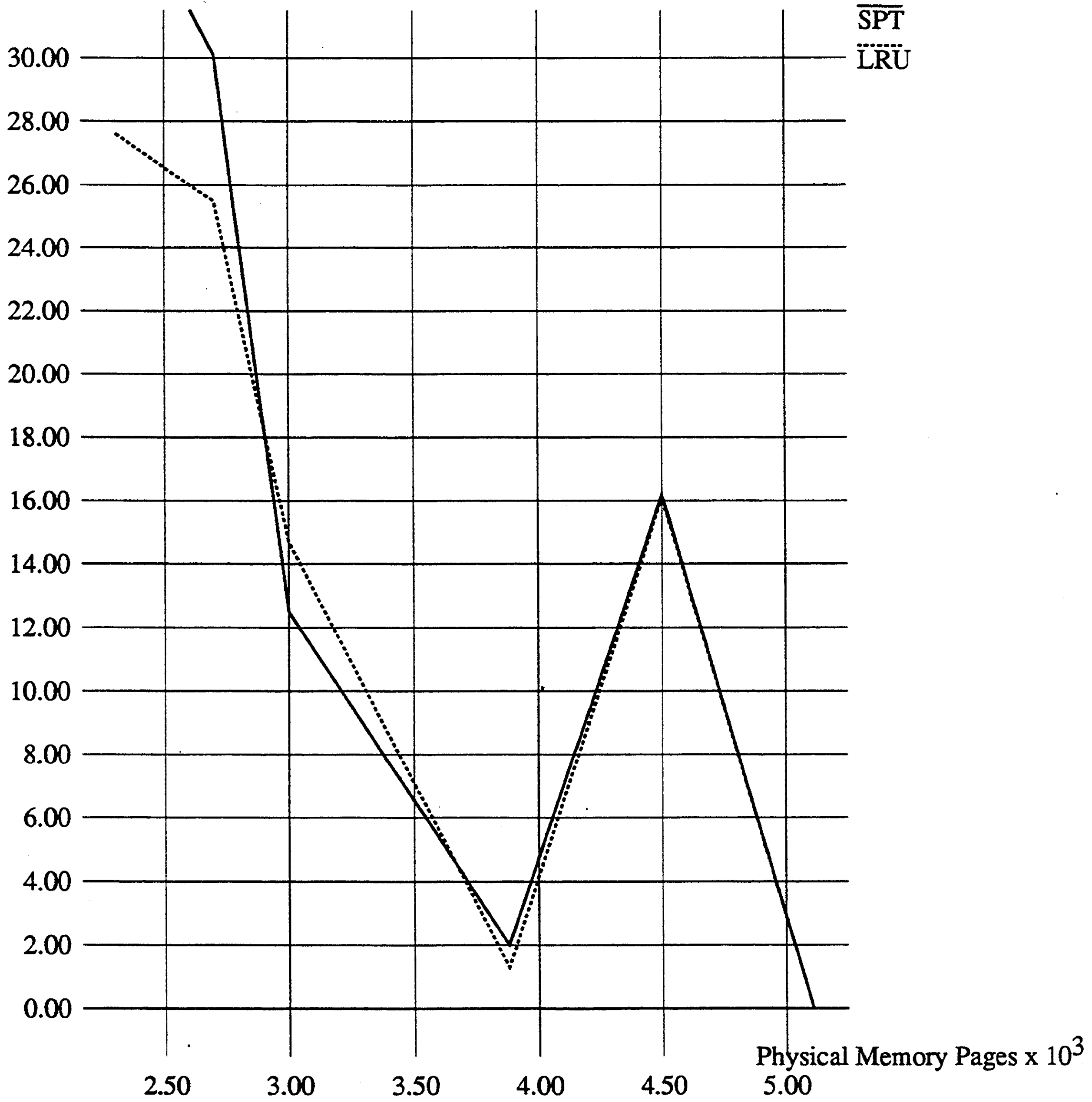
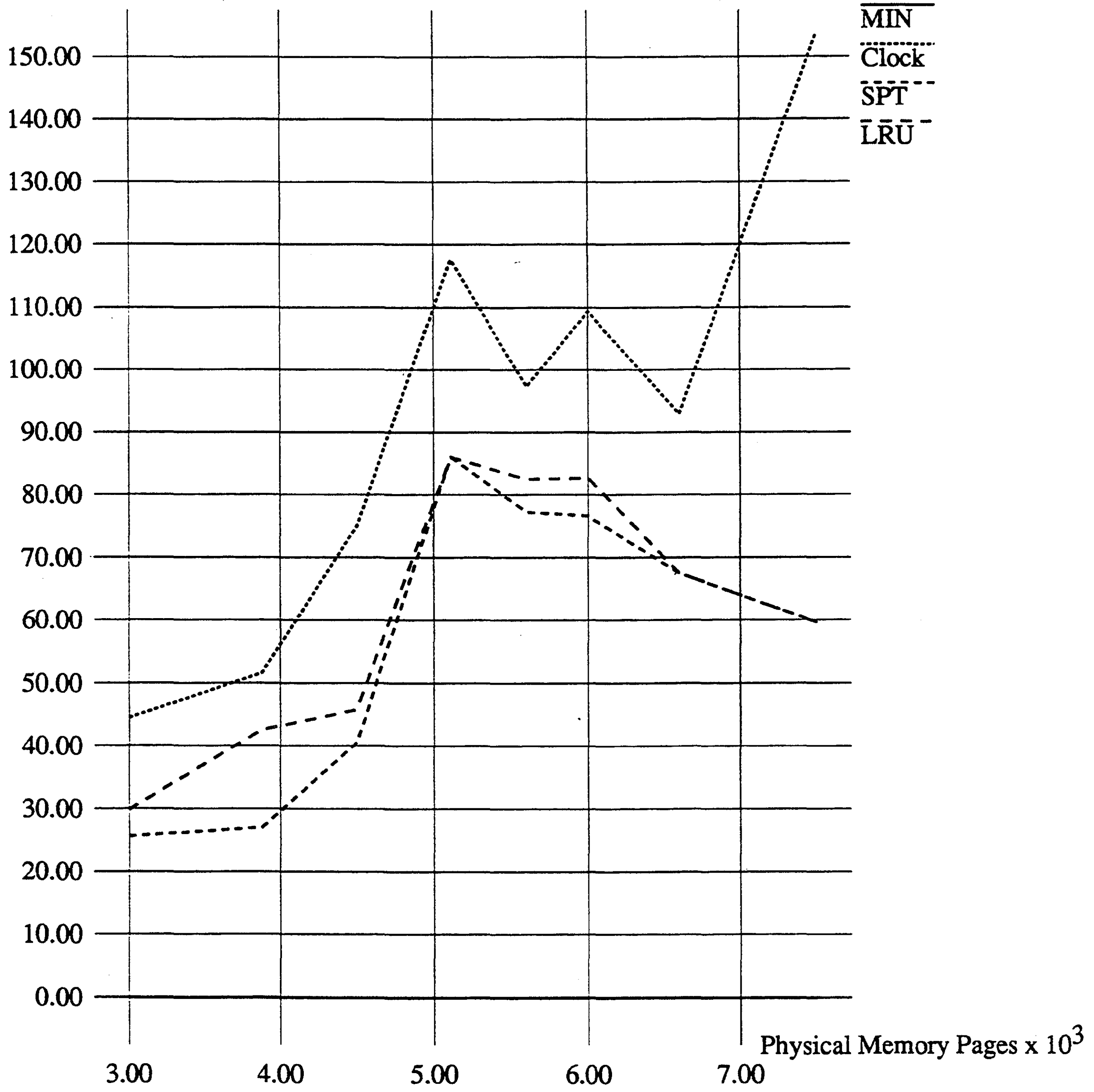


**Figure 9: Trace C: Percent Better Than Clock Scaled To MIN**



## Figure 10: Trace A Average For Worst 5 Intervals Percent Over MIN



XEROX Virtual Memory Replacement Using Historical Information on Virtual Objects Robert B. Hagmann CSL-91-7