Adaptive Page Replacement Based on Memory Reference Behavior

Gideon Glass
Pei Cao

Technical Report #1338

February 1997

Adaptive Page Replacement Based on Memory Reference Behavior

Gideon Glass and Pei Cao
Computer Sciences Department
University of Wisconsin–Madison
{gid,cao}@cs.wisc.edu

Abstract

As disk performance continues to lag behind that of memory systems and processors, virtual memory management becomes increasingly important for overall system performance. In this paper we study the page reference behavior of a collection of memory-intensive applications, and propose a new virtual memory page replacement algorithm, SEQ. SEQ detects long sequences of page faults and applies most-recently-used replacement to those sequences. Simulations show that for a large class of applications, SEQ performs close to the optimal replacement algorithm, and significantly better than Least-Recently-Used (LRU). In addition, SEQ performs similarly to LRU for applications that do not exhibit sequential faulting.

1 Introduction

As the performance gap between memory systems and disks increases, the impact of memory management on system performance increases. Although buying more memory would always alleviate the poor performance of current virtual memory (VM) systems, operating system designers should attempt to improve VM design and policies so that users receive the best attainable performance, regardless of system configuration and budget.

In this study we collected sixteen memory-intensive applications and studied their page reference behavior. Seven applications are from the SPEC95 suite; the rest are "big-memory" applications including integer-intensive programs (e.g. databases) and scientific computations. We found that the applications have very different page reference patterns: some are truly memory intensive, referencing many pages in short time intervals, while others have clear reference patterns that can be exploited for better replacement decisions.

We simulated the Least-Recently Used (LRU) page replacement algorithm and the optimal offline algorithm (Belady's OPT algorithm [2]) for these applications under varying main memory sizes. For the applications that has no visible, large-scale access patterns, both LRU and OPT show gradual, continuous reduction in page fault rate as memory size increases. LRU appears to be a good replacement policy for such programs. For applications that have clear access patterns, however, LRU often performs poorly: it frequently exhibits plateau behavior, where increasing memory sizes does not reduce fault rate until the whole program fits into memory. For these programs OPT obtains at least linear reduction in fault rate as memory size increases.

Based on LRU's observed poor behavior, we propose a new replacement algorithm, SEQ. SEQ normally performs LRU replacement; in addition, it monitors page faults as they occur, detecting long sequences of faults to contiguous virtual addresses. When such sequences are found, SEQ performs a pseudo most-recently-used (MRU) replacement on the sequences, attempting to imitate what OPT would do. SEQ often corrects the poor performance (plateau behavior) of LRU for applications that have sequential behavior, yet it performs the same as LRU for other types of applications.

We also conducted a preliminary study of two global page replacement algorithms: global LRU replacement, and SEQ extended to be a global replacement algorithm. We found that SEQ performs similar to or better than global LRU on mixes of various application types. Our results suggest that SEQ may be a good algorithm suitable for implementation in a real OS kernel VM system.

2 Applications and Traces

The applications we studied are described in Table 1. Shown for each program is the number of instructions executed by the traced program and the amount of total memory used by the program. (Other columns in the table will be described further below.)

2.1 Trace Methodology

We collected memory reference traces using Shade [8], an instruction-level trace generator for the SPARC architecture. All programs ran on machines running the Solaris 2.4 operating system. Because of the length of our traces, recording all memory references individually would result in unmanageably large trace files. Instead, we record "IN" and "OUT" records. We divide program instruction time into fixed-length *intervals* (usually 1,000,000 instructions). At the end of every interval, for every page that was referenced in the current interval but was *not referenced* in the previous interval, an IN record is generated and timestamped with the actual time (in terms of instructions executed) of the first reference to that page. Similarly, for every page that was accessed in the previous interval but was *not*

A shorter version of this paper appeared in the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, June 1997, Seattle, Washington.

Program	Description	Length (millions of instructions)	Memory used (KB)	Executable size (KB)	Min. simulatable memory size (KB)	
					LRU	OPT
applu	Solve 5 coupled parabolic/elliptic PDEs	1068	14524	136	2432	972
blizzard	Binary rewriting tool for software DSM	2122	15632	1153	5332	4772
coral*	Deductive database evaluating query	4327	20284	940	7084	6780
es*	microstructure electrostatics	71003	104488	56	696	316
fgm*	finite growth model	35210	121508	112	10052	2136
gcc	Optimizing C compiler	1371	3936	1599	1900	1052
gnuplot	PostScript graph generation	4940	62516	602	1552	476
ijpeg	image conversion into JPEG format	42951	8260	152	1112	748
m88ksim*	Microprocessor cycle-level simulator	10020	19352	165	1964	328
murphi	Protocol verifier	1019	9380	238	2132	1472
perl*	Interpreted scripting language	18980	39344	569	9636	8428
swim	Shallow water simulation	438	15016	56	6932	6216
trygtsl	Tridiagonal matrix calculation	377	69688	26	2444	1400
turb3d	Turbulence simulation	17989	26052	71	7720	6360
vortex	Main memory database	2507	9676	600	3024	2028
wave5	Plasma simulation	3774	28700	511	3652	1708

Table 1: Benchmark programs measured, with execution duration and memory address space size. * Indicates runs which were terminated before they completed. Also shown are minimum simulatable memory sizes (discussed in section 2.1) and the size of the program binary.

accessed in the current interval, an OUT record is generated with the timestamp of the instruction making the last reference to the page. IN and OUT records in a trace are written out sorted by their timestamps. We used a uniform page size of 4KB throughout this study.

The IN and OUT records associated with a page mark the beginning and end of a period when the page is referenced. The page is accessed at least once during each interval in this period; exactly how many times and exactly when each reference occurs is unknown. However, a page is definitely not accessed in the time between an OUT record until the next IN record for that page.

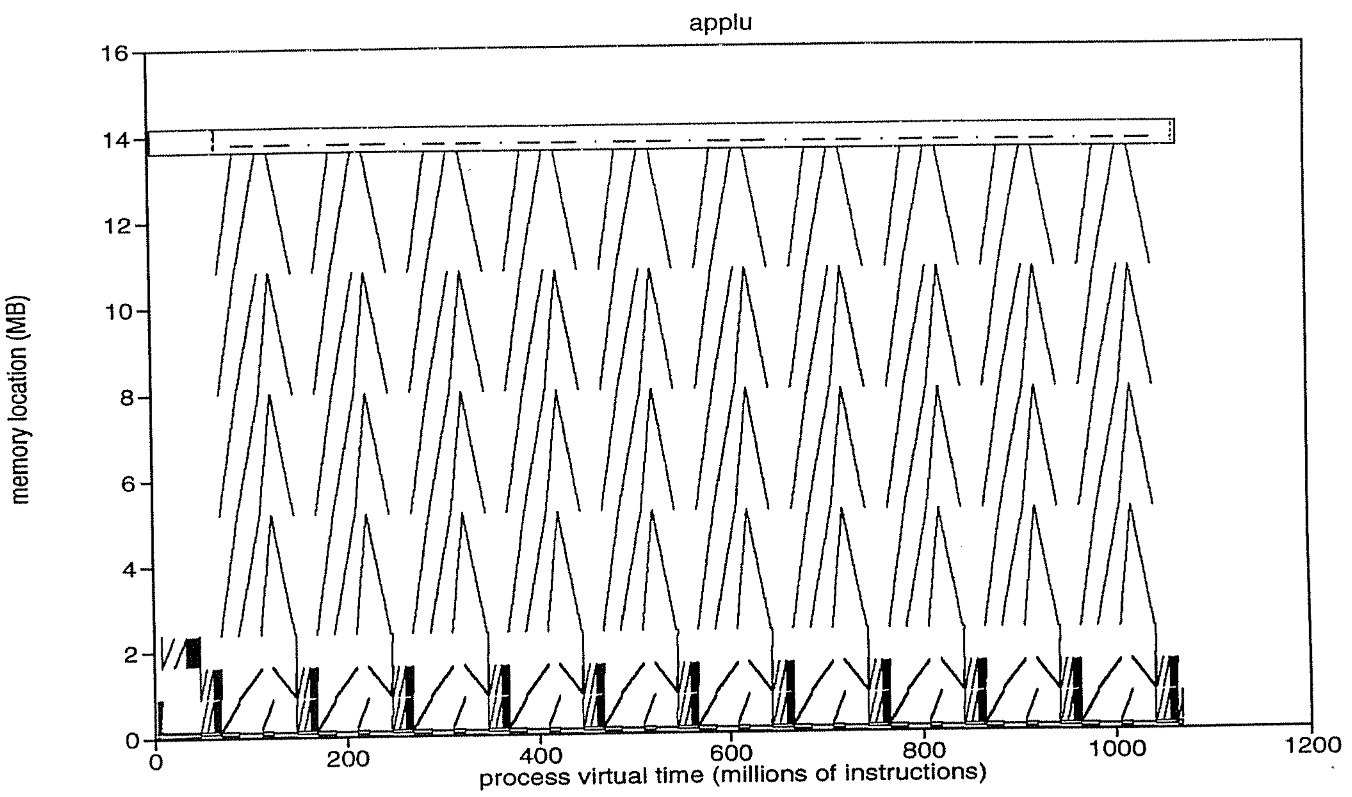
This trace format not only is compact but also allows *accurate* simulation of several replacement algorithms for sufficiently large memory sizes. At any point in a trace, define pages that are between an IN record and an OUT record as being "ACTIVE", and the pages that are between an OUT record and an IN record as "IDLE". Then the OPT algorithm, which replaces the page that is referenced furthest in the future, can be simulated by replacing the IDLE page whose next IN record is both furthest in the future and at least two intervals *ahead of* the current interval. Such a page is indeed the furthest referenced page because any ACTIVE page will be accessed again either in the current interval or in the next interval. By similar reasoning, LRU can be simulated by replacing the IDLE page whose previous OUT record is both the earliest among all IDLE pages, and whose previous OUT record is either two intervals *before* the current interval, or is before the IN records of all ACTIVE pages. These constraints ensure that the page is indeed the least-recently-used page (since any ACTIVE page must have been accessed in the current interval or in the last interval).

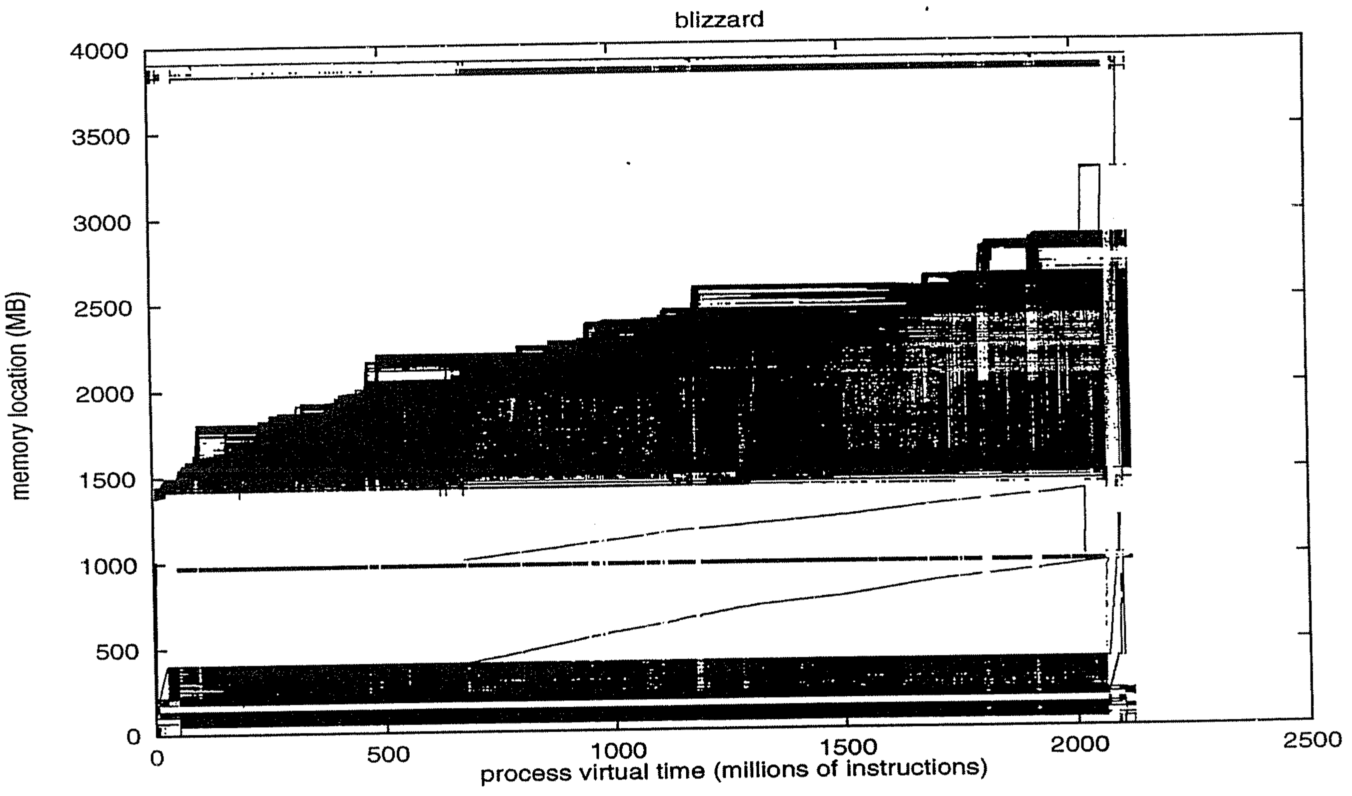
A limitation of our method is that it can only simulate memory sizes above a certain threshold. If the memory size is too small, the simulation will not be able to find an IDLE page satisfying the above criteria. The minimum simulatable memory sizes for each application are listed in Table 1. (For SEQ we used the same minimum as LRU since SEQ defaults to LRU replacement.)

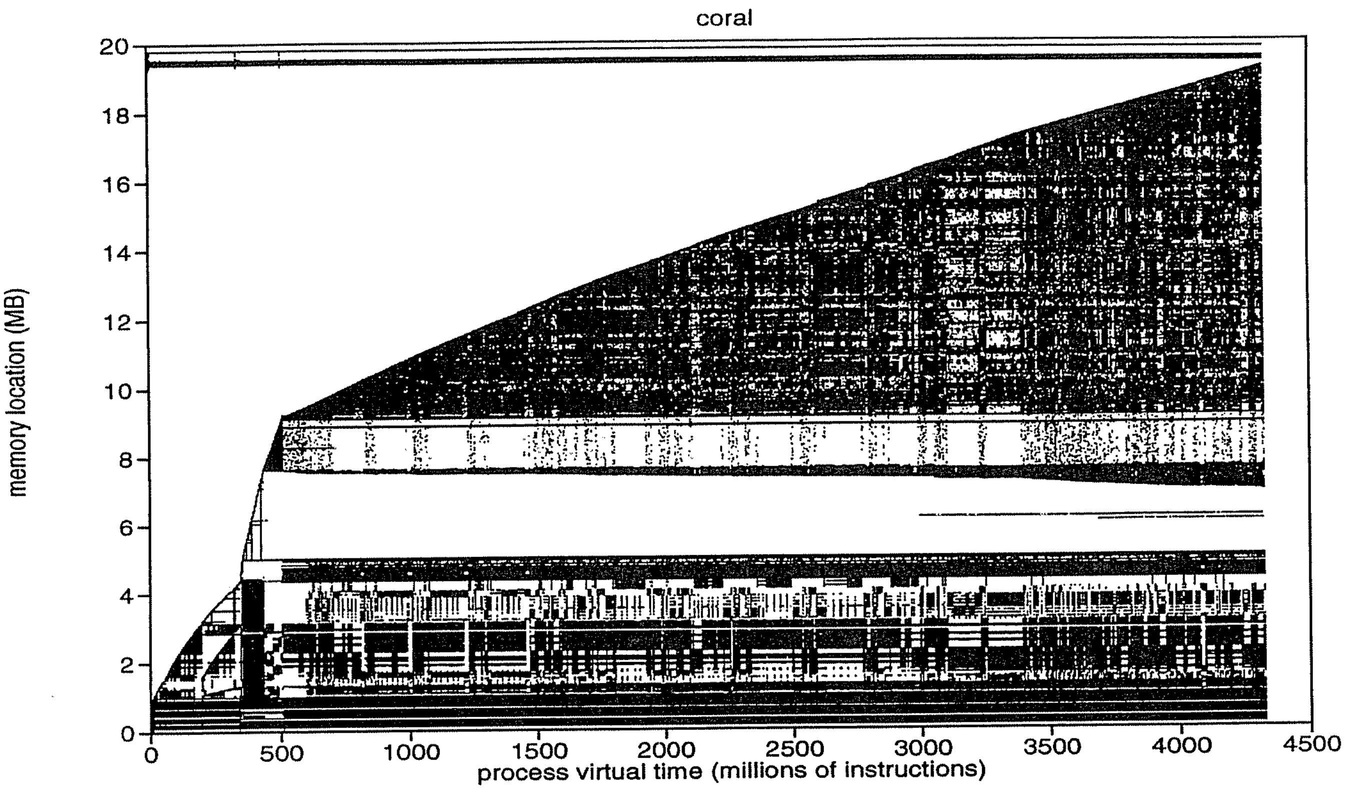
2.2 Application Page Reference Behavior

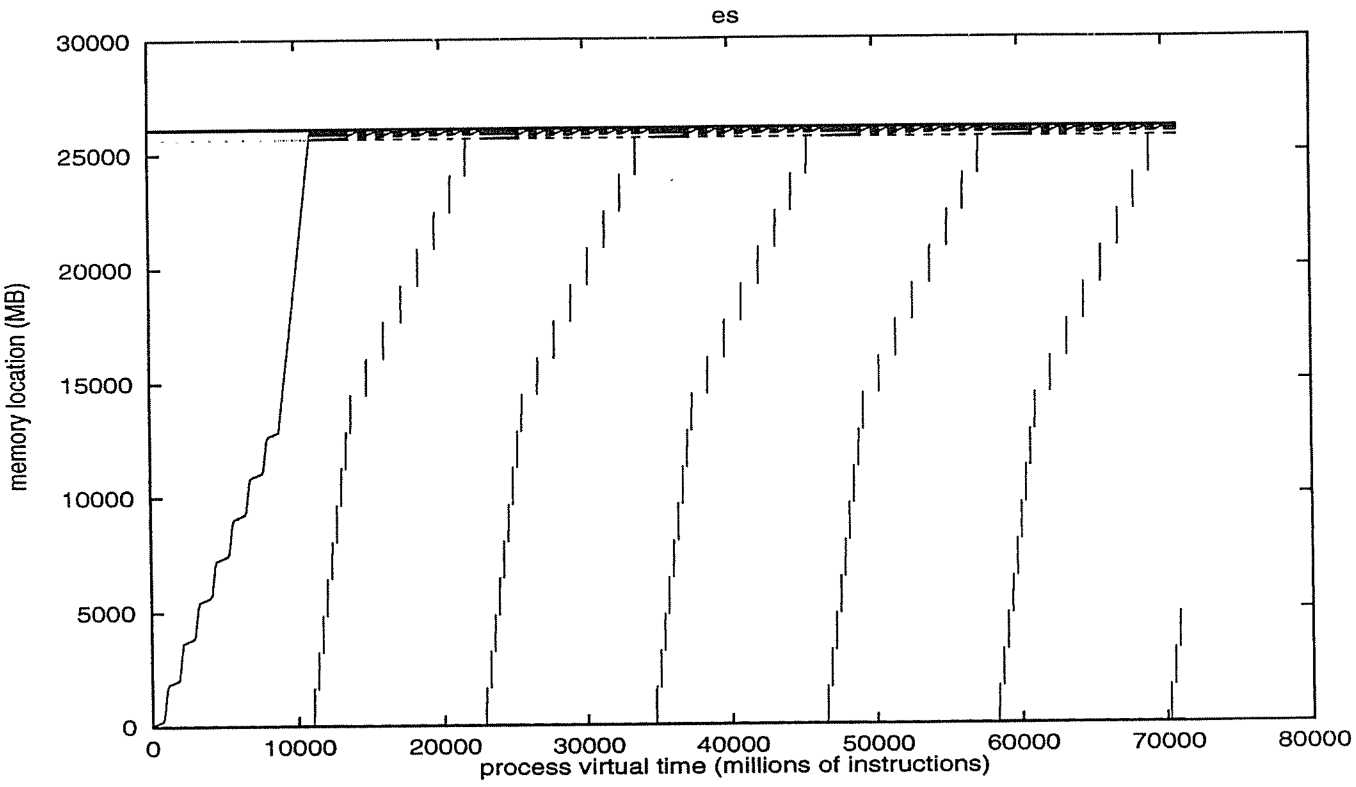
We can plot space-time graphs of references from the traces described above. For each execution interval (a point on the x axis) we plot a point for each page referenced in that interval. The y axis values are relative page locations within the program's address space (since the application's address space is usually sparse and contains many unused regions, we leave out the address space holes and number the used pages from low addresses to high addresses on the y axis). On the following pages are space-time plots for each our of applications.

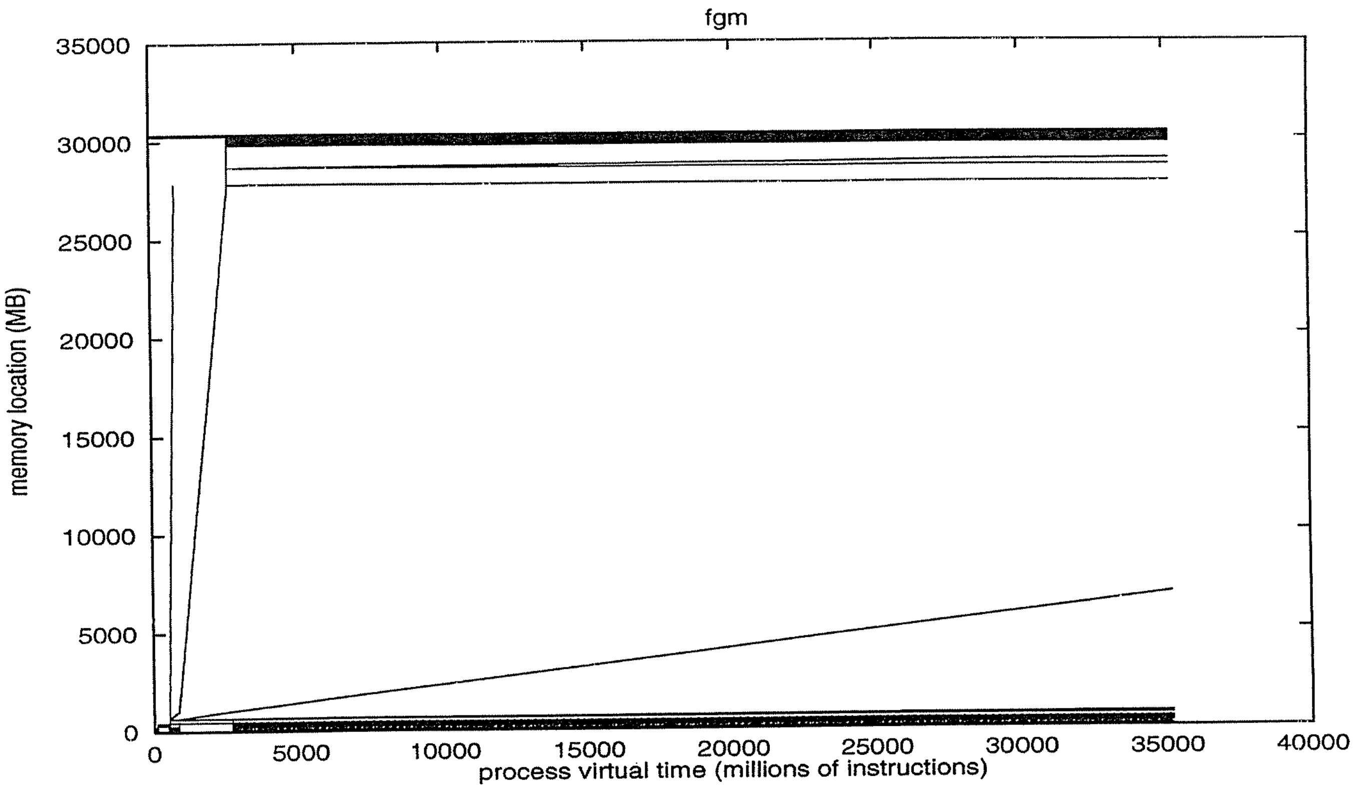
Observing the space-time graphs, we found that the applications fall roughly into three categories. The first, which includes coral, murphi, m88ksim and vortex, are truly memory intensive—large numbers of pages are accessed during each execution interval. There are no clearly visible patterns within the vast dark areas. The second category, which includes blizzard, gcc, and perl, are also memory intensive, but have patterns at a small scale (for example, in gcc, the traversal of pages in the 0.5MB–2.25MB range follows a certain pattern). (These kind of small-scale patterns might be exploited for techniques such as prefetching, but we have not investigated prefetching in this paper.) The third application category, consisting of the rest of the applications, show clearly-exploitable, large-scale reference patterns. Ranges of address space are traversed in the same pattern repeatedly. The applications seem to be array-based, though some of

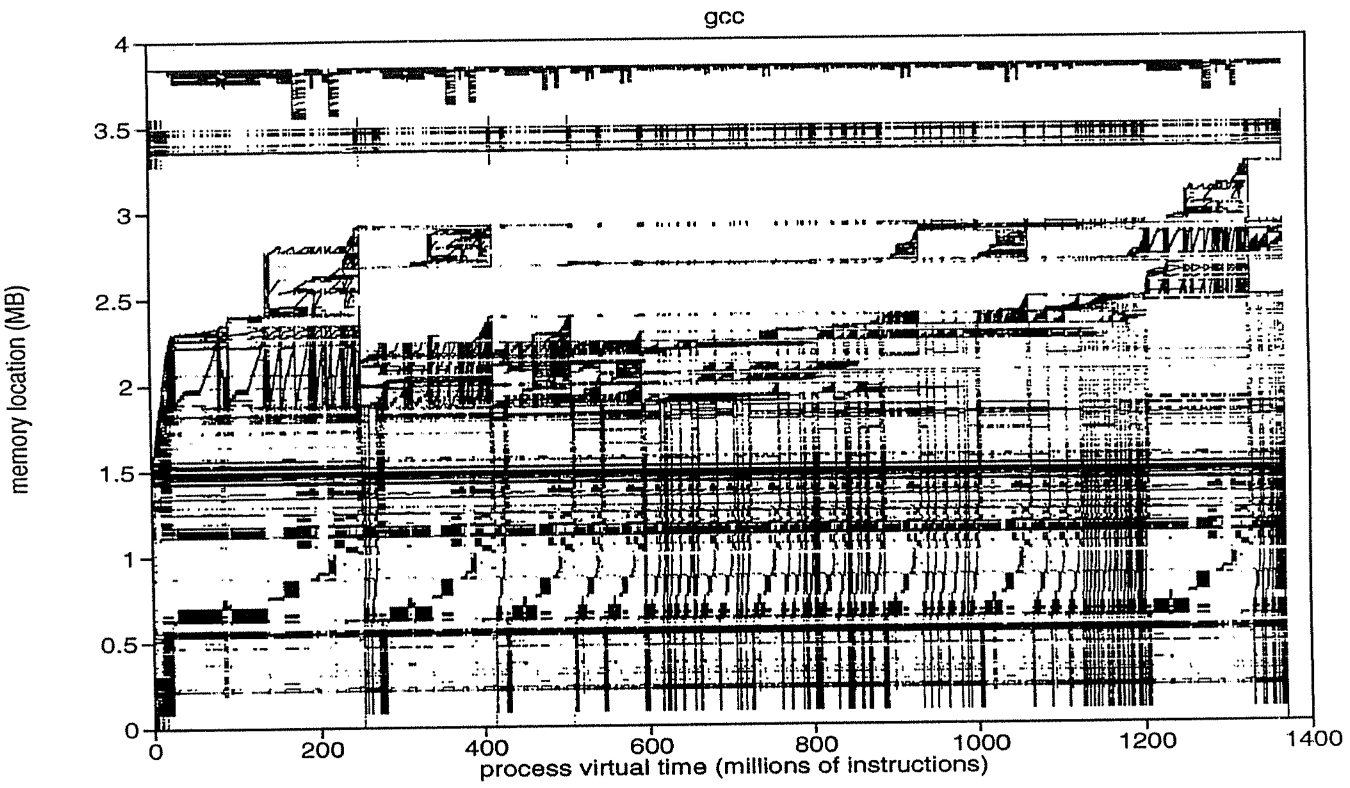

applu
memory location (MB)
process virtual time (millions of instructions)
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200

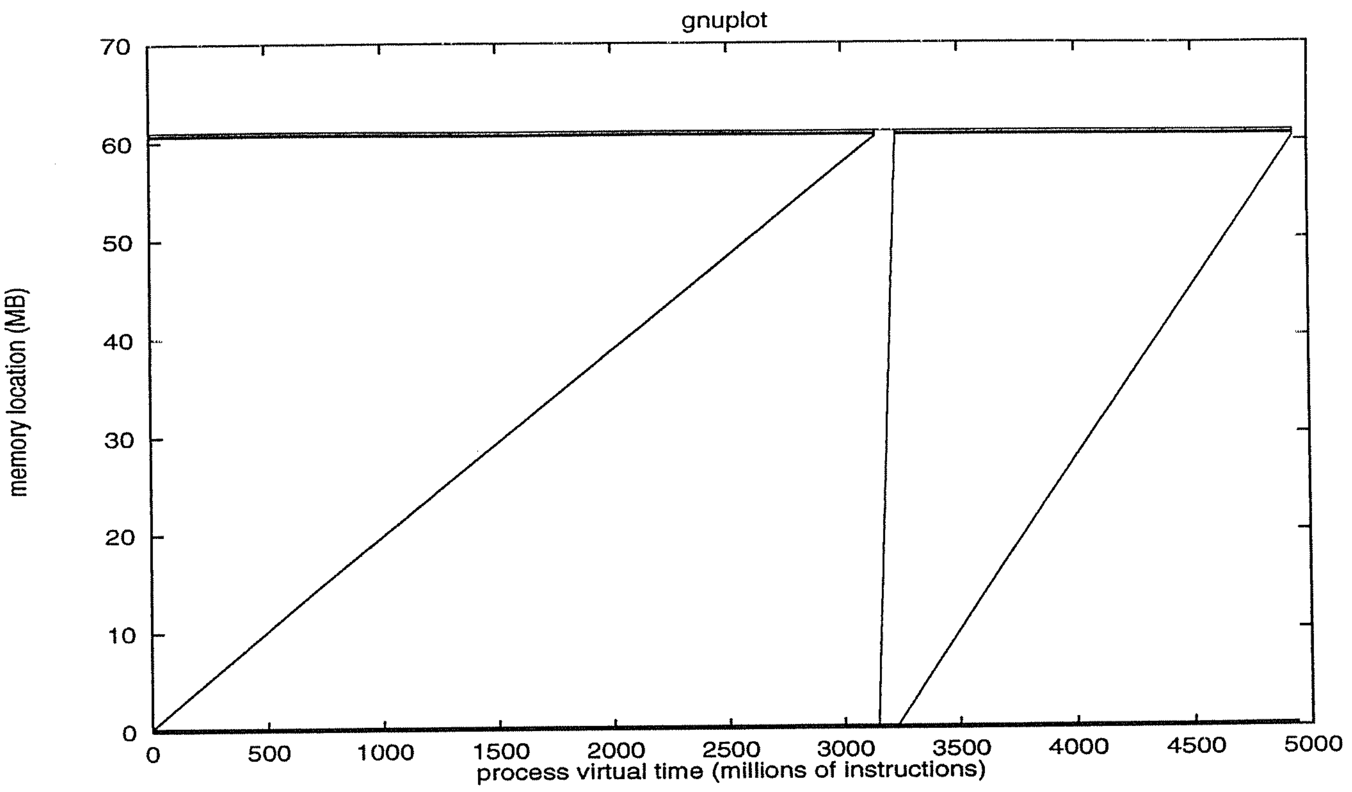

blizzard
memory location (MB)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
3000
3500
4000
0
500
1000
1500
2000
2500

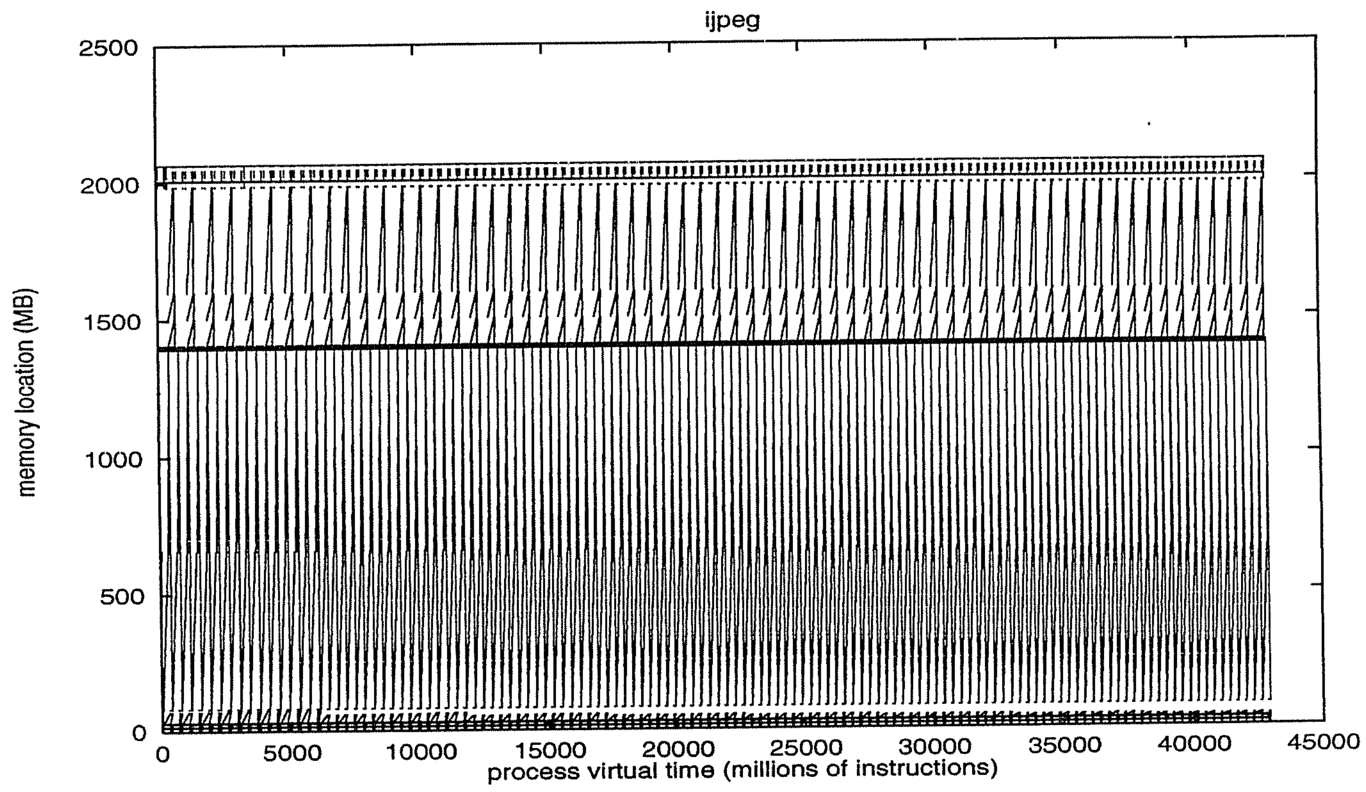

coral
memory location (MB)
process virtual time (millions of instructions)
0
2
4
6
8
10
12
14
16
18
20
0
500
1000
1500
2000
2500
3000
3500
4000
4500

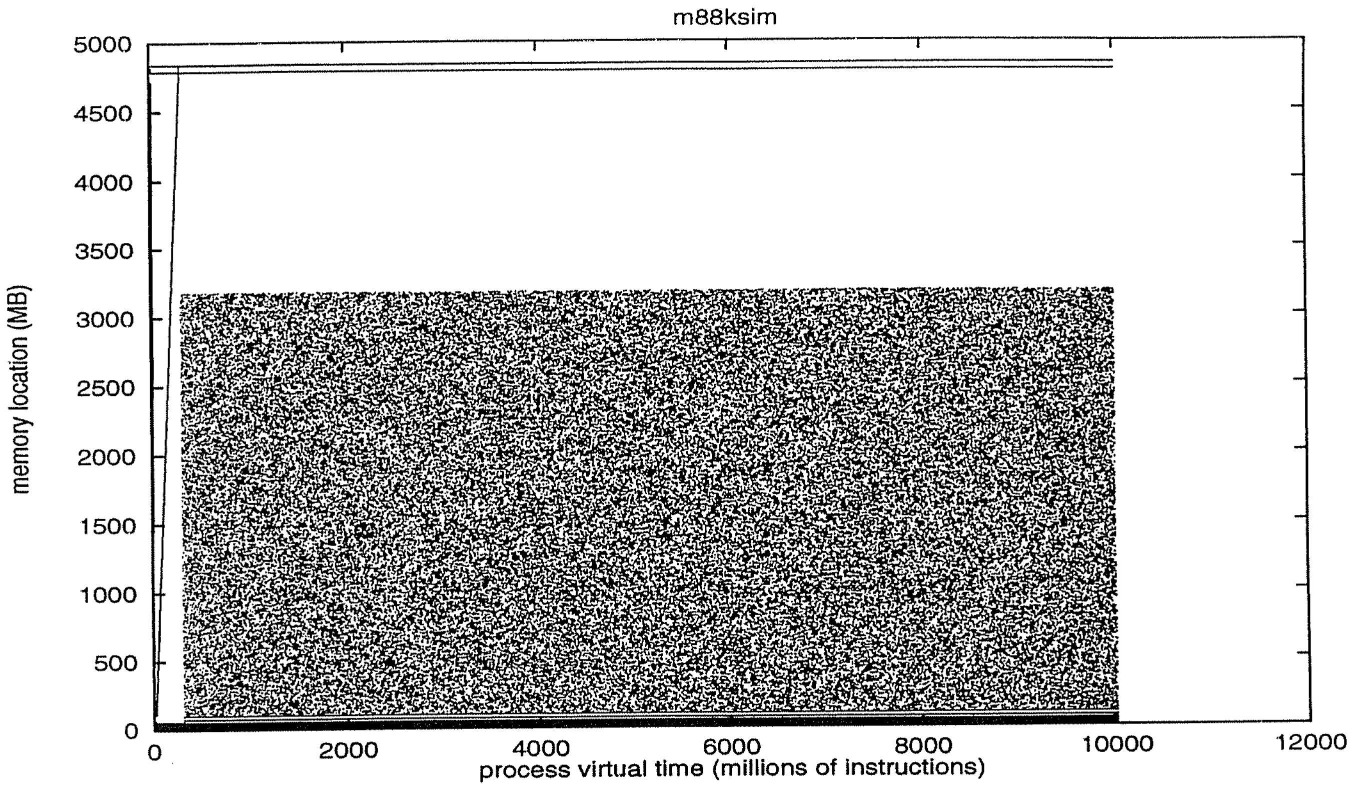

es
memory location (MB)
process virtual time (millions of instructions)
0
5000
10000
15000
20000
25000
30000
0
10000
20000
30000
40000
50000
60000
70000
80000

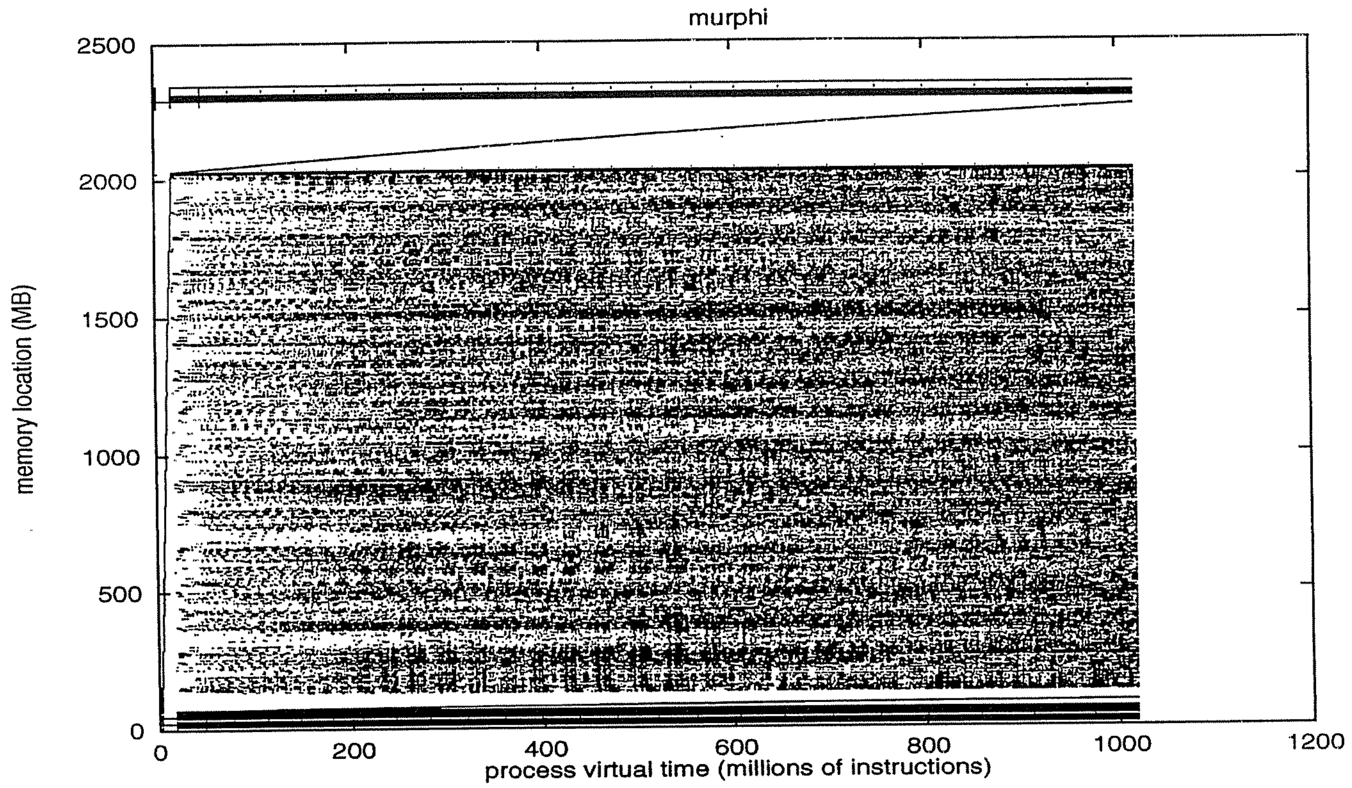

fgm
memory location (MB)
35000
30000
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000
30000
35000
40000
process virtual time (millions of instructions)

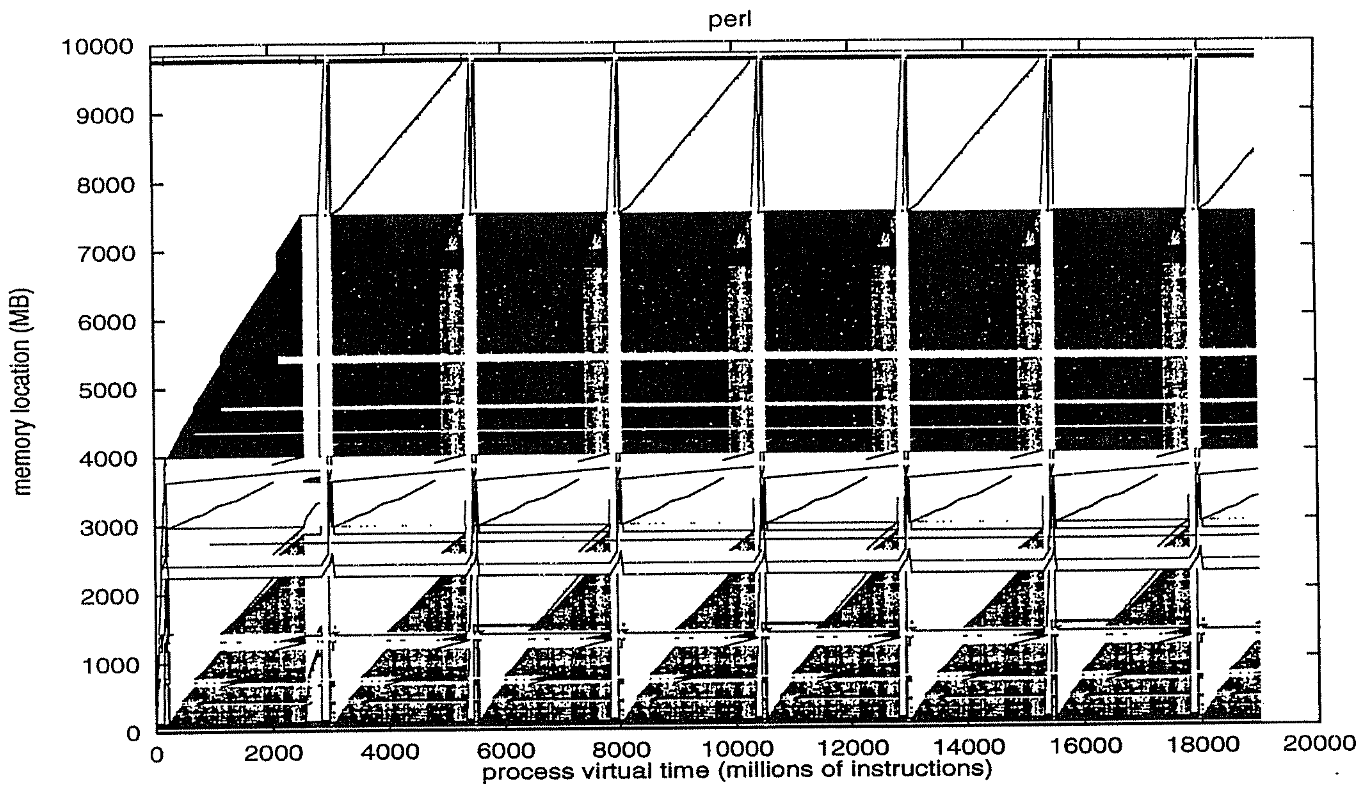

gcc
memory location (MB)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
200
400
600
800
1000
1200
1400
process virtual time (millions of instructions)

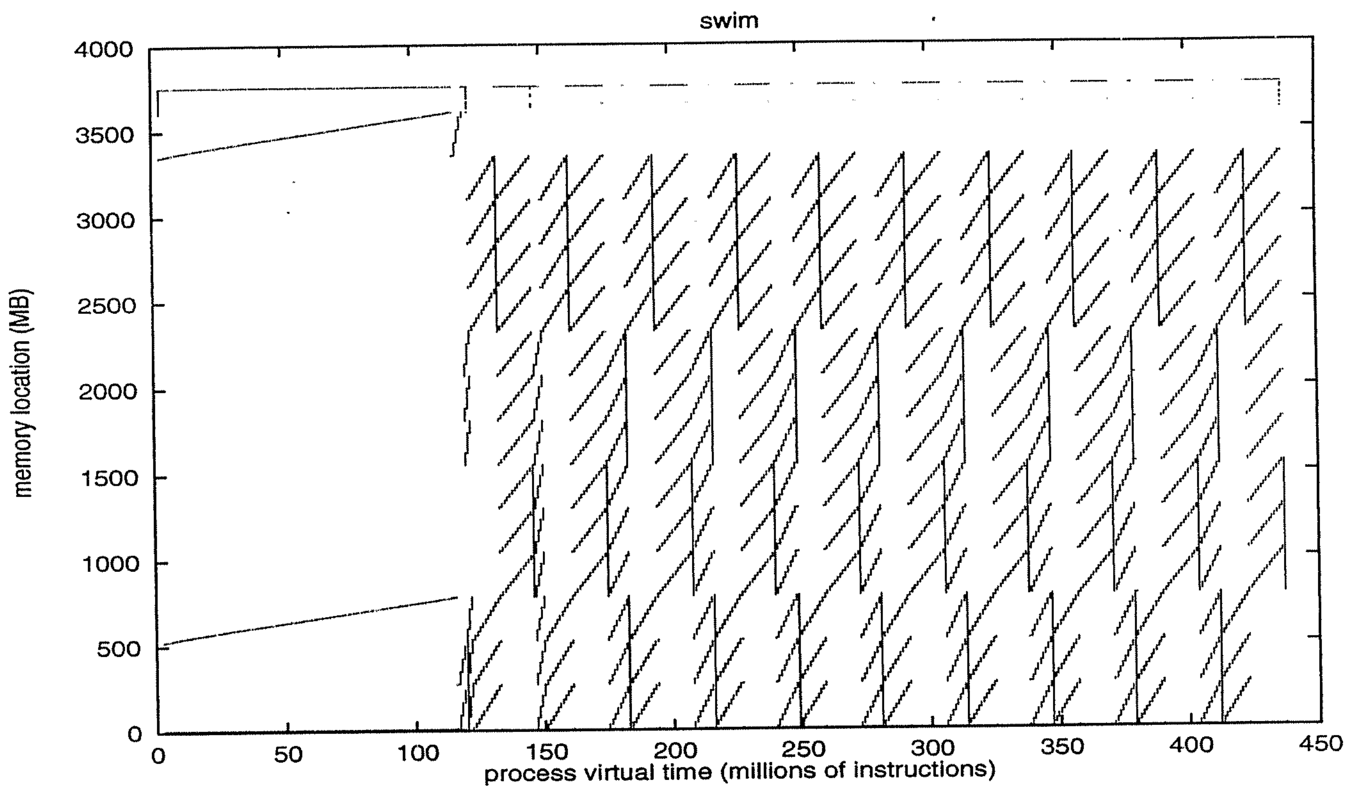

gnuplot
memory location (MB)
process virtual time (millions of instructions)
0
10
20
30
40
50
60
70
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

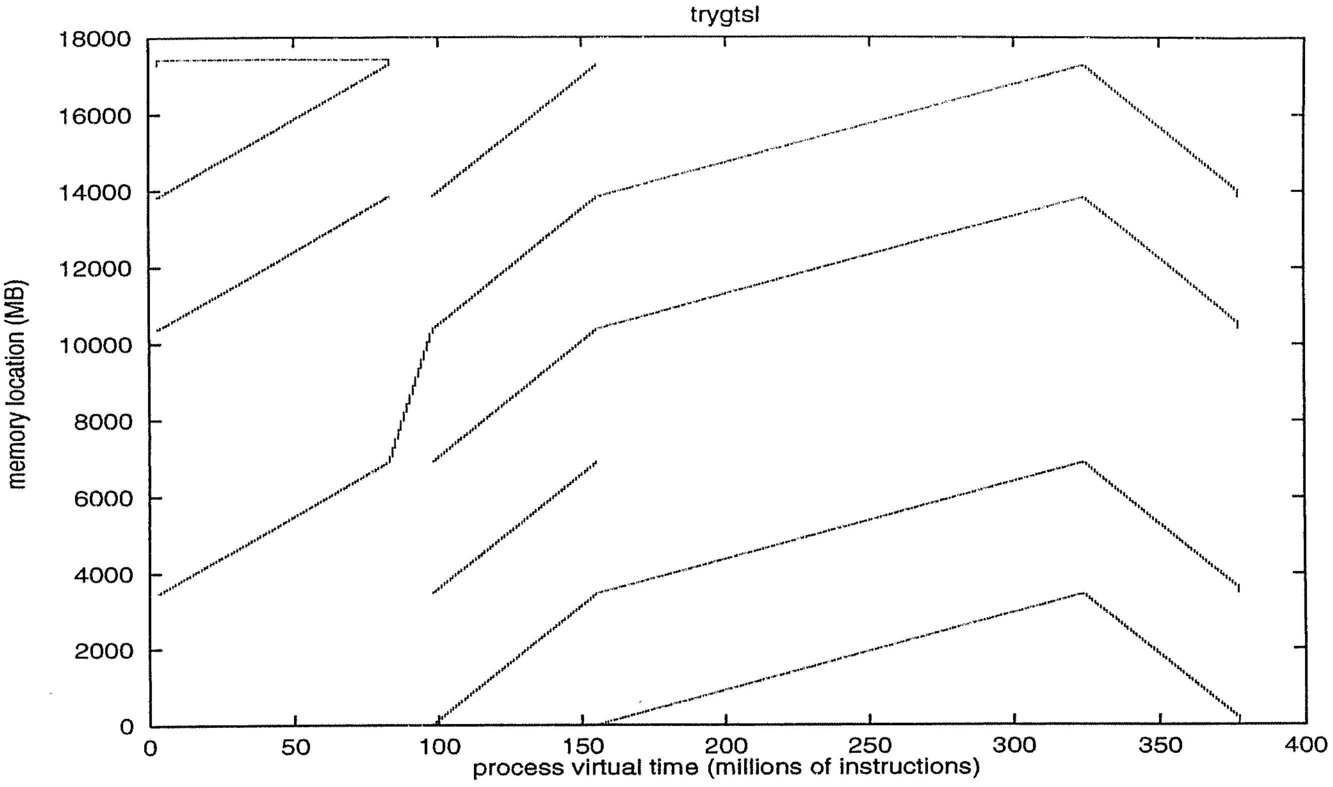

ijpeg
memory location (MB)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
5000
10000
15000
20000
25000
30000
35000
40000
45000

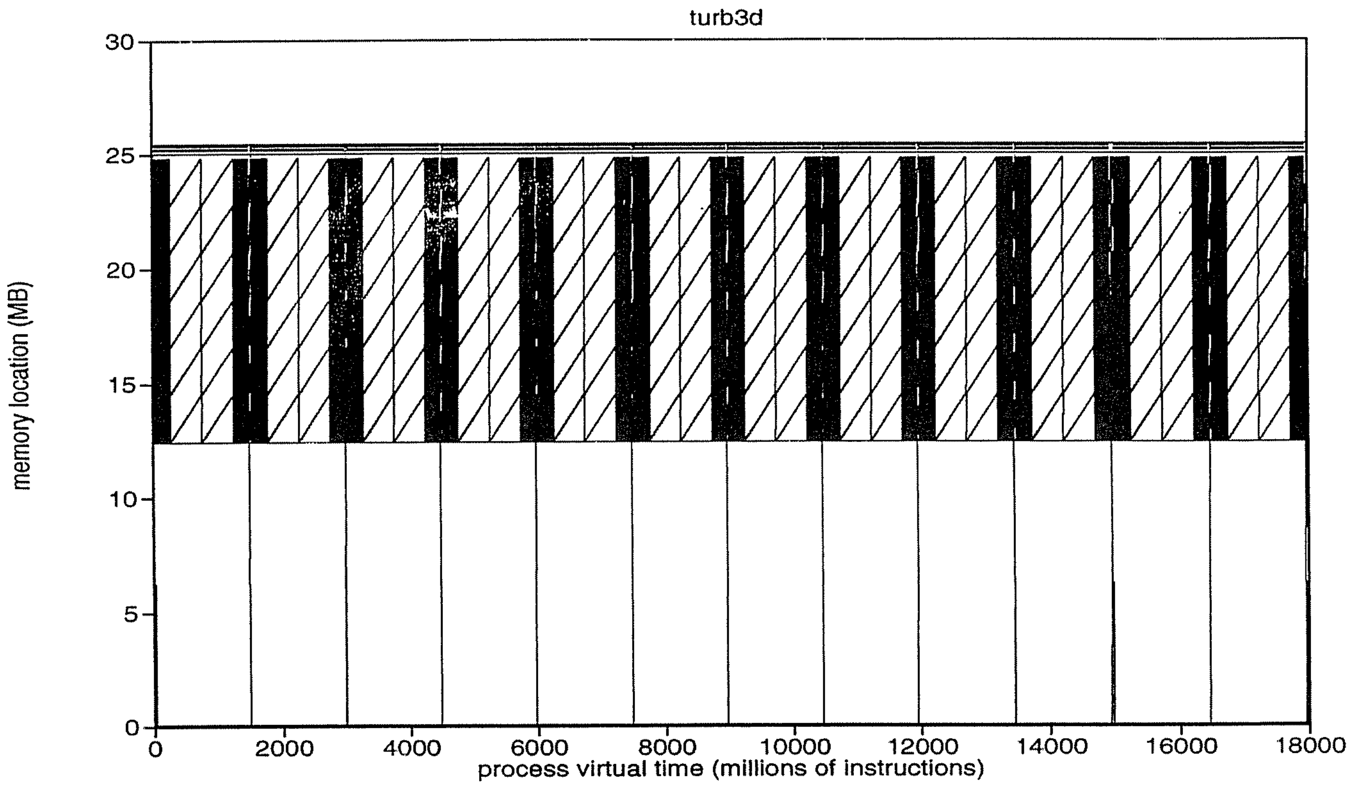

m88ksim
memory location (MB)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
0
2000
4000
6000
8000
10000
12000

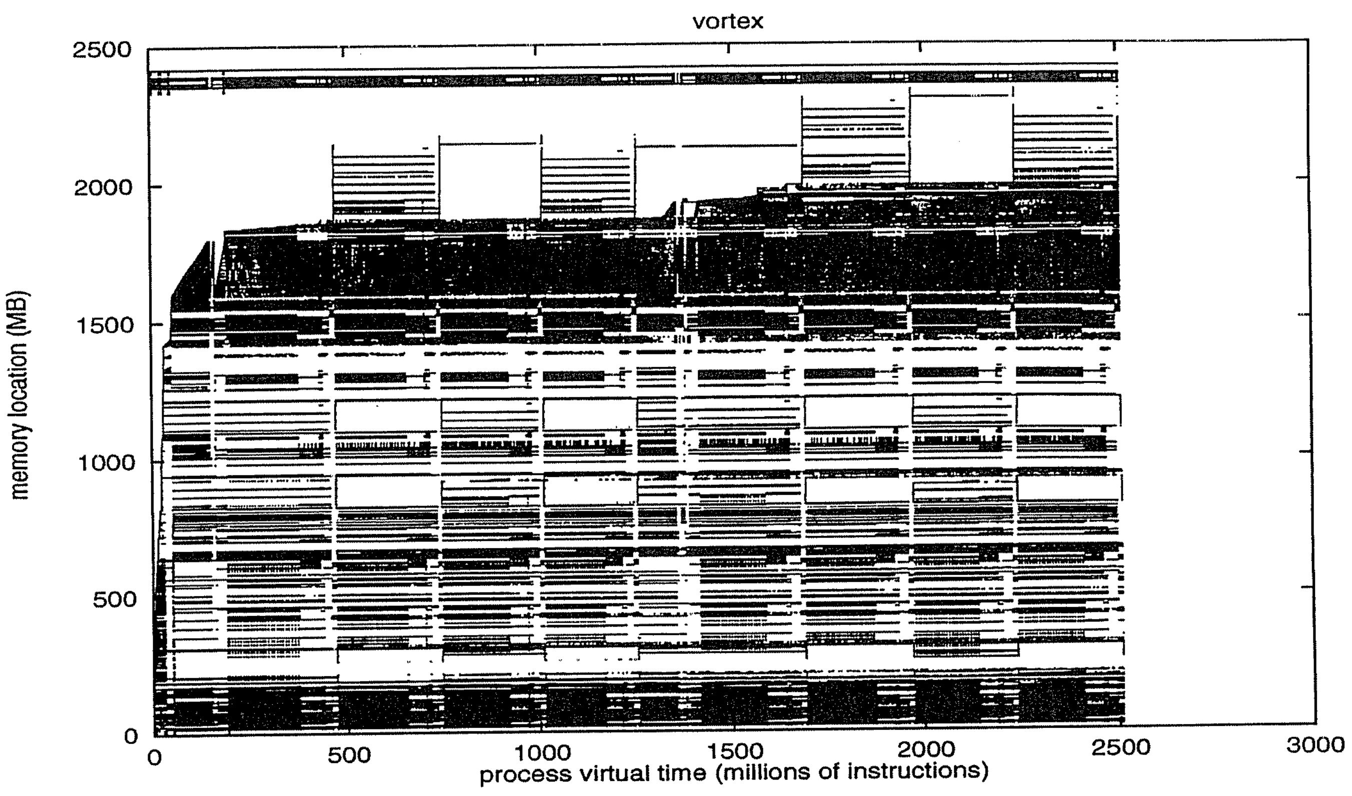

murphi
memory location (MB)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
0
200
400
600
800
1000
1200

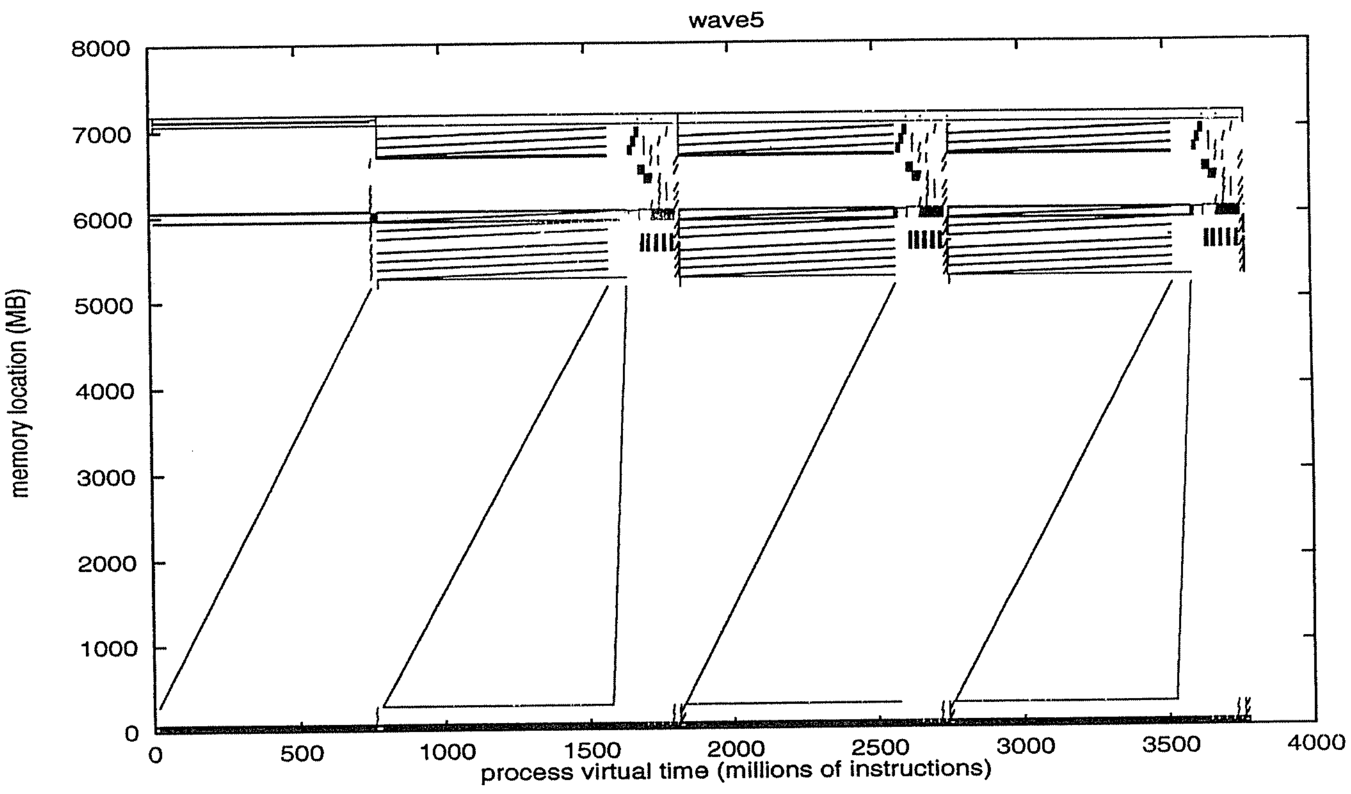

perl
memory location (MB)
process virtual time (millions of instructions)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000


swim
memory location (MB)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
3000
3500
4000
0
50
100
150
200
250
300
350
400
450


trygtsl
memory location (MB)
process virtual time (millions of instructions)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50
100
150
200
250
300
350
400


turb3d
memory location (MB)
process virtual time (millions of instructions)
30
25
20
15
10
5
0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000


vortex
2500
2000
1500
1000
500
0
memory location (MB)
0
500
1000
1500
2000
2500
3000
process virtual time (millions of instructions)


wave5
8000
7000
6000
5000
4000
3000
2000
1000
0
memory location (MB)
0
500
1000
1500
2000
2500
3000
3500
4000
process virtual time (millions of instructions)

them are written in C (fgm and gnuplot). Some programs (ijpeg, applu, and trygtsl) traverse ranges of memory in one direction and then change direction, but most programs simply go in one direction. The number of sequentially-traversed regions also varies, with swim doing about sixteen and other programs (es, gnuplot) covering only one large region.

These classes of behavior remind us of the following comment by Rob Pike: "The following data structures are a complete list for almost all practical programs: array, linked list, hash table, binary tree." [23] The statement clearly has some truth to it: most applications exhibiting regular reference patterns are array-based; vortex, m88ksim, murphi, coral, and perl are apparently either making heavy use of hash tables or are traversing tree structures; gcc and perl (to some extent) seem to use linked lists heavily. From the virtual memory system's point of view, array-based application would be the easiest to handle, while hash tables are the hardest.

One interesting observation from the space-time graphs is that for the programs we investigated, any given program does not change its memory behavior radically—there are not many distinct phases of behavior. (Some of the programs *repeat* various patterns—turb3d for example—but there is not a clear start-to-finish progression of different phases.) Program behavior generally varies much more between different programs than it does between any two phases of a single program.

2.3 Performance of LRU and OPT

Figures 1 and 2 show page faults per one million instructions executed for each application as its memory spans the range from the minimum simulatable size to the total number of pages the application uses. [1] The three curves in the graph are LRU, OPT, and the new algorithm SEQ that we will describe in the next section. We do not include startup faults in the figures, because most of these faults are due to initialization of processes' address space, and are usually serviced by zero-filling a page, not by invoking a disk I/O. (The number of pages that must be demand-paged from disk can be estimated by dividing the "program size" column in Table 1 by the 4KB page size.)

The results show that for the first and second categories of applications, which are memory intensive and do not have strong patterns, LRU performs similarly to OPT, though LRU suffers about twice as many page faults on average. For these application classes, the fault rate under LRU drops continuously when more memory is available; the rate of improvement is similar to that under OPT. The improvement appears to be super-linear for memory sizes less than half of the total memory needed by the program (i.e. doubling the amount of memory more than halves the number of page faults), and the improvement slows down after that point.

The situation is completely different for the applications in the third category (programs with highly regular sequential access patterns). LRU performs much poorer than OPT, generating up to five to ten times more page faults. LRU frequently gives no improvement till memory size reaches a certain threshold, and results in "staircase" graphs. This gives the appearance that the applications have certain working-sets that, once in memory, will reduce the fault rate significantly. In fact, OPT is always able to reduce the fault rate continuously, and LRU simply fails to reduce the fault rate until it reaches certain memory sizes.

The problem is that these applications (gnuplot, for example) are looping over large address space ranges; LRU replaces pages starting at the beginning of the address range (since those are oldest), replacing pages a constant distance behind the location where the program is accessing memory. When the program begins another iteration at the bottom of the range, LRU pages out the top. All pages in the range must be paged in on every iteration, resulting in the worst possible performance. This "LRU flooding" phenomenon is the primary motivation for our SEQ algorithm, described in Section 4.

3 Inter-fault Times

In addition to observing *fault rate* for varying memory sizes, we also observed *mean inter-fault times* for varying memory sizes. Although mean inter-fault time is simply the inverse the fault rate, we found it instructive to examine both types of graphs. The mean inter-fault time graphs help illuminate the right end of the fault rate curves, where the curves approach zero.

Figures 3 and 4 graph mean inter-fault times for varying memory sizes for the three replacement policies OPT, LRU, and SEQ. The y-axis is scaled from zero to ten million instructions between faults. (On modern computer systems, a program taking page faults less frequently than one per ten million instructions will suffer very little slowdown from paging.)

Denning illustrates Working-Set page replacement and its rationale by means of mean inter-fault time plots for a hypothetical program [10]. His plots contain some "knee" (a global maximum of $f(x)/x$, i.e., mean inter-fault time divided by memory size). Our plots show little evidence of knees. The programs whose curves do contain knees perform poorly under LRU (e.g. fgm and turb3d). Since the OPT curves all slope gracefully upwards, as one would expect from the fault-rate OPT curves, we conclude that knee behavior in inter-fault-time curves are more likely LRU replacement relics than signs of inherent program memory demand.

4 SEQ Replacement Algorithm

The intuition behind the SEQ replacement algorithm is to detect long sequences of page faults and apply MRU replacement to such sequences. The goal is to avoid LRU flooding, which occurs when a program accesses a large address space range sequentially. If a program accesses an address range once, LRU would page out useful pages that would be accessed again; if the program accesses the address range multiple times and the range is larger than physical memory,

[1] We plot page fault *rates* rather than fault *counts* because it allows us to compare fault rates for different programs more easily. To obtain fault counts, simply multiple the fault rate (at a given memory size) and the trace length from table 1.

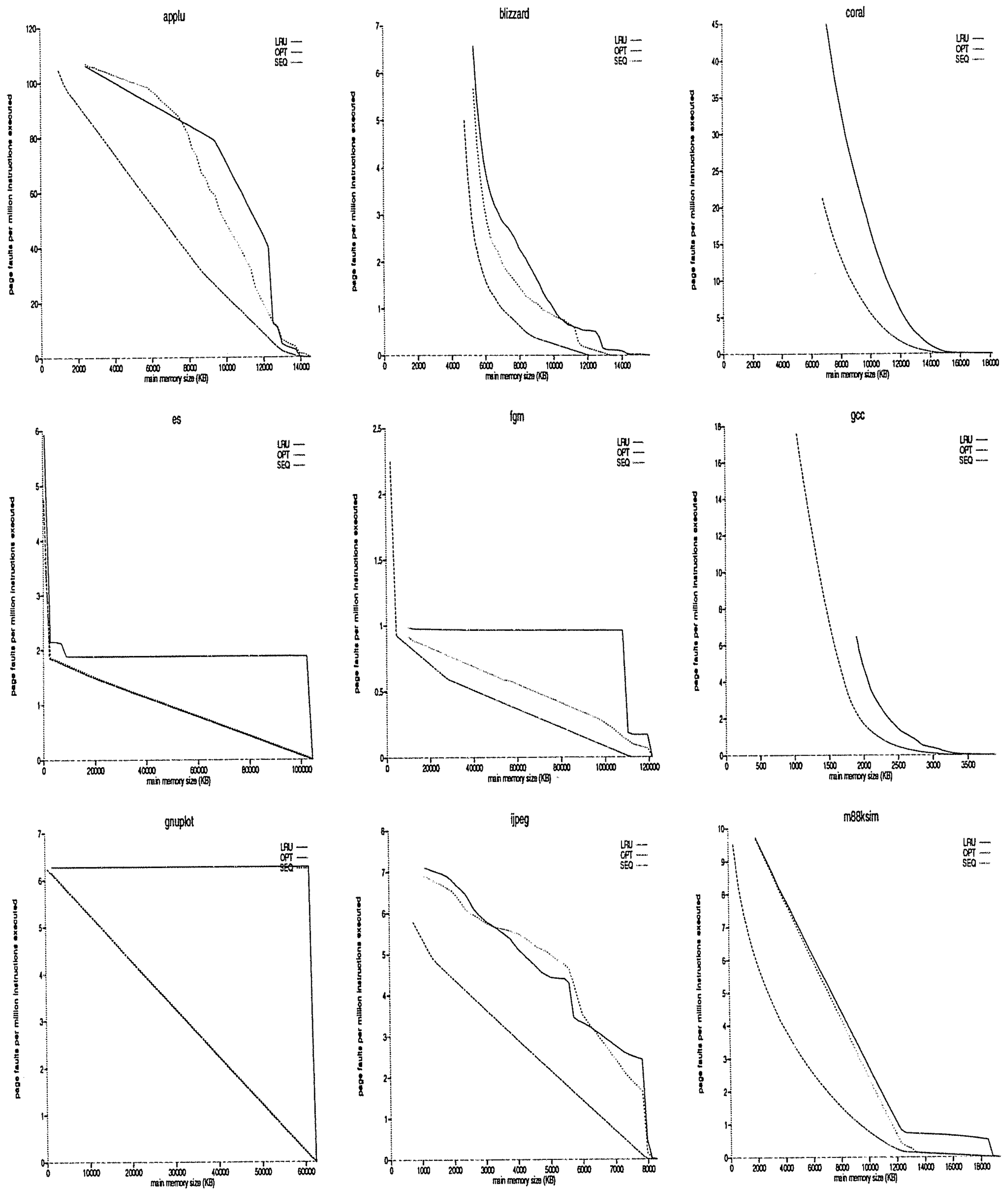


Figure 1: Performance of OPT, SEQ and LRU. For `es` and `gnuplot`, the SEQ curve almost overlaps the OPT curve. For `coral` and `gcc`, the SEQ curve overlaps the LRU curve.

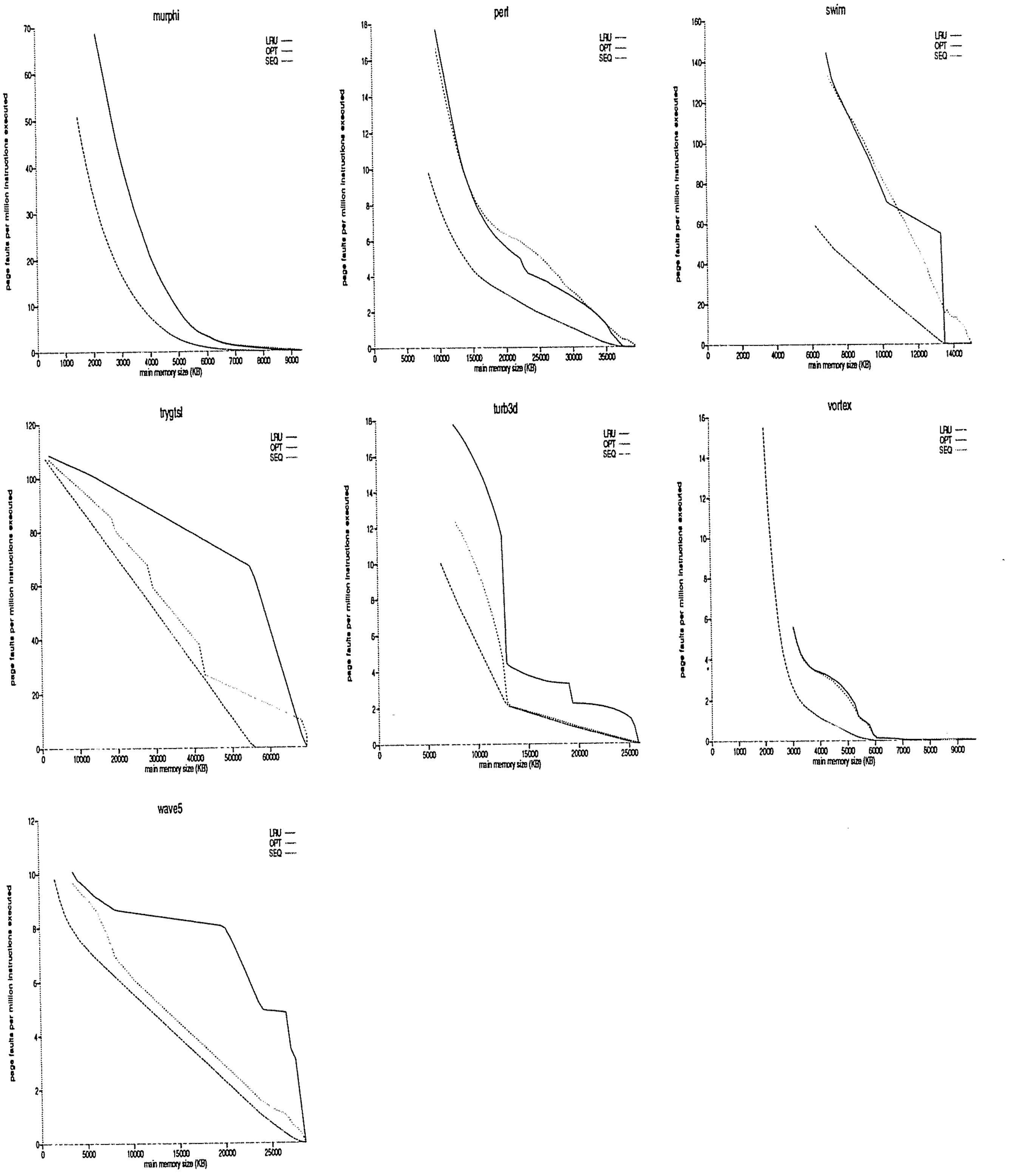


Figure 2: Performance of OPT, SEQ and LRU. For `murphi`, the SEQ curve overlaps the LRU curve. For `vortex`, the SEQ curve mostly overlaps the LRU curve.

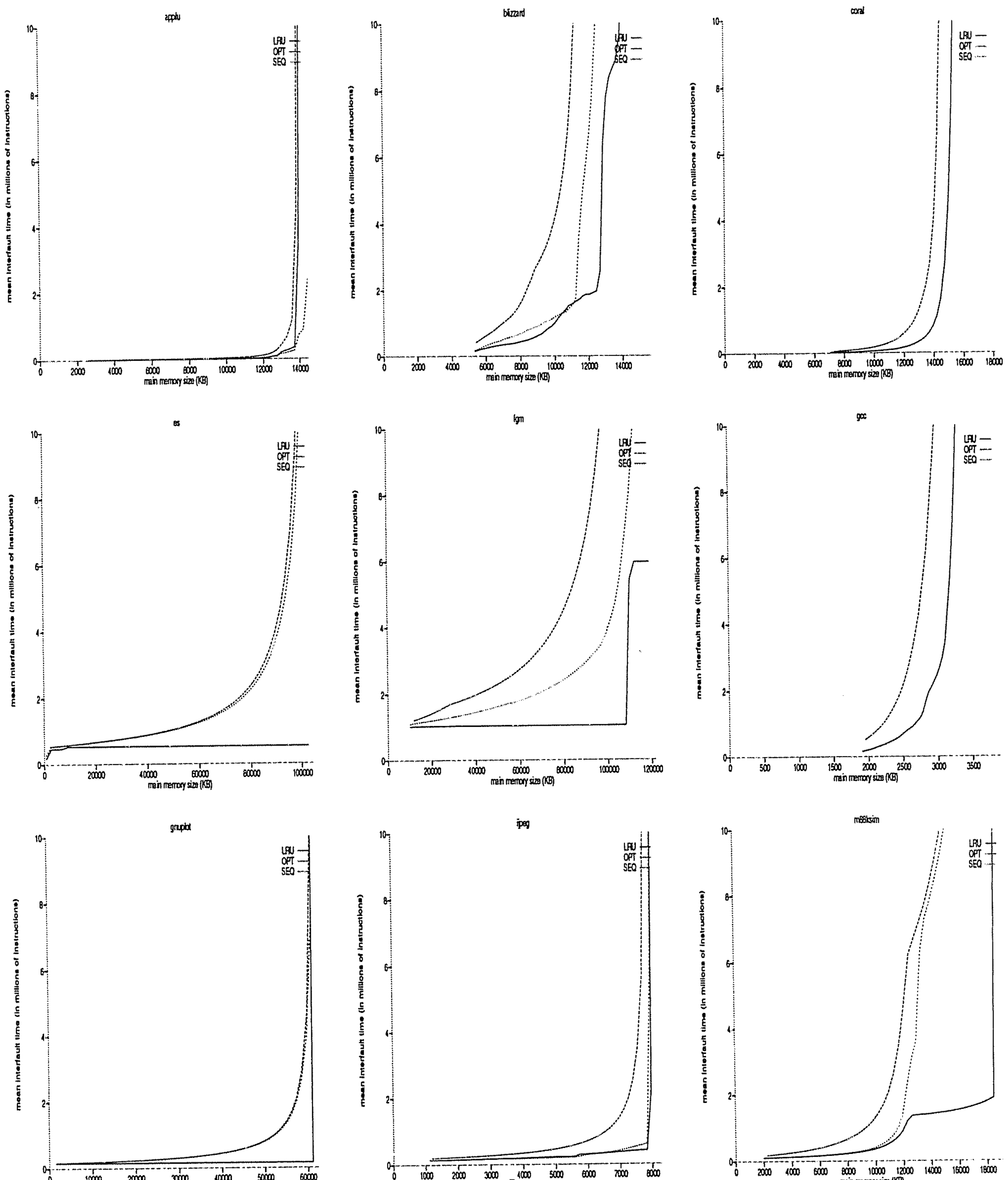


Figure 3: Mean inter-fault times for OPT, SEQ and LRU. For coral and gcc, SEQ and LRU curves overlap. The apparent "knee" in the fgm curve only appears for LRU replacement, under which fgm performs poorly.

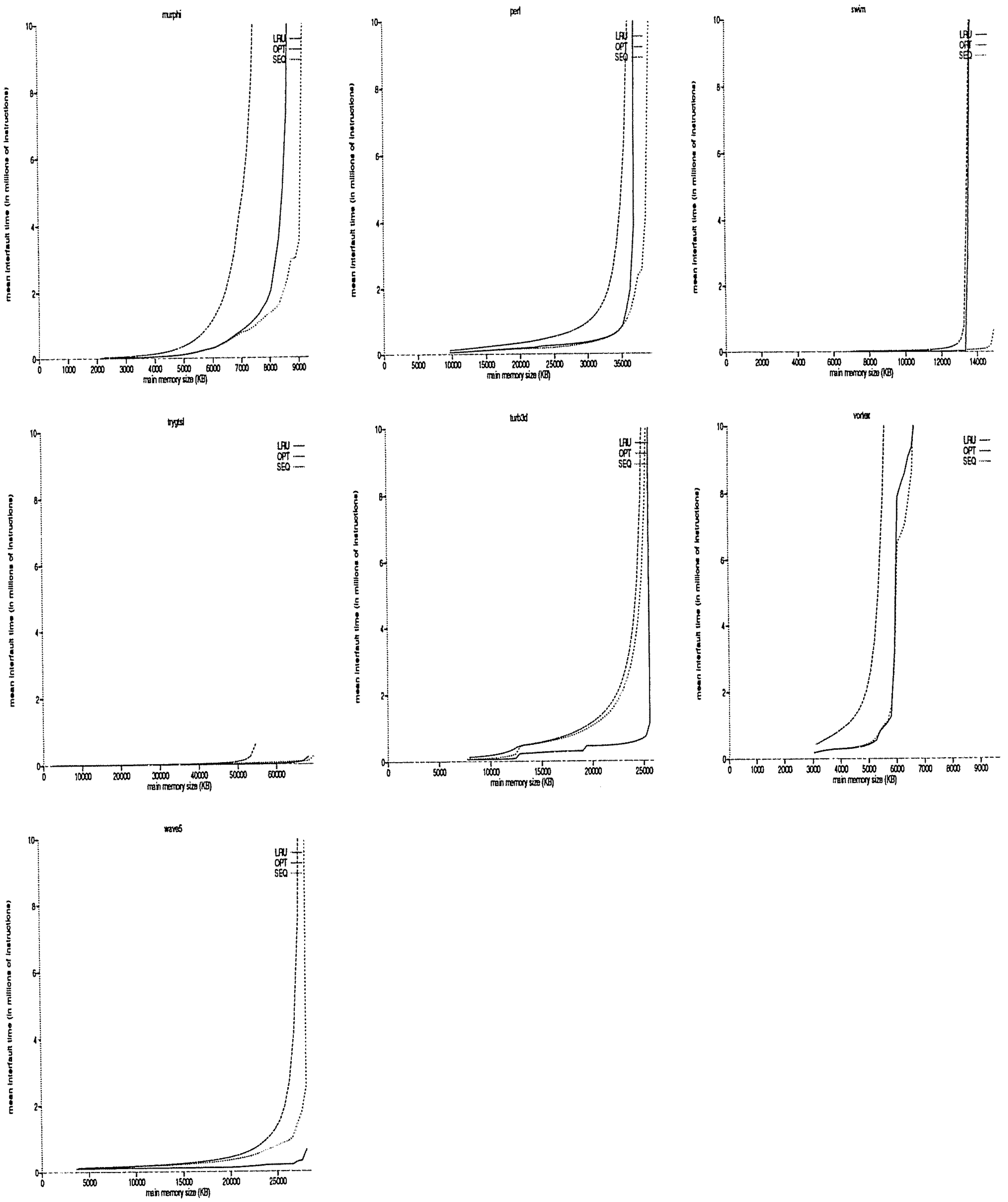


Figure 4: Mean interfault times for OPT, SEQ and LRU. For swim, the SEQ curve appears to terminate early due to the next plot point being above 10 on the y-axis. Note the knees in the LRU curve for turb3d; for SEQ and OPT the knees are not present.

LRU would page out the pages in the order in which they are accessed and thus perform poorly, as described above.

If no sequences are detected, SEQ performs LRU replacement.

4.1 Design

There are four main components in SEQ's design:

1. *What is a "sequence"?* A sequence is a series of page faults to consecutive virtual addresses, growing in one direction (increasing addresses or decreasing addresses) with no other faults to pages in the middle of the series. (We refer to most recently-added page—the page at the end of the sequence in the direction of growth—as the *head* of the sequence.)

2. *When memory is low and a page much be paged out, which sequence is chosen to replace a page from?* SEQ chooses only sequences of length greater than L (currently 20 pages); it examines the time of the Nth (currently $N = 5$) most recent fault in each sequence, and chooses the one whose fault is most recent.

3. *Which page from the chosen sequence is replaced?* SEQ chooses the first in-memory page that is M (currently 20) or more pages from the head of the sequence.

4. *What happens to a sequence if a page fault occurs in the middle of the address range of the sequence?* SEQ splits the sequence into two sequences, one ranging from the beginning of the sequence to the page immediately preceeding the faulted page, and the other consisting of the faulted page alone.

Choices of values for L, N and M is discussed in Section 4.2.

SEQ detects replaceable sequences by observing *page faults* (not page references) and associates them based on adjacent virtual addresses. SEQ maintains a list of sequences, recording (for each sequence) the tuple <*low_end, high_end, dir*>. The tuple indicates a sequence ranging from virtual address *low_end* to virtual address *high_end*, faulting (as time increases) in the direction *dir* (which is either *up* or *down*). When a page fault on page *pf* occurs, SEQ examines sequences adjacent to *pf*. If the new page fault extends the sequence (i.e. *pf* = *high_end* +1 and *dir* = *up*, or *pf* = *low_end* −1 and *dir* = *down*), the sequence's *low_end* or *high_end* is changed to include the current fault.

If *pf* falls in the middle of the sequence (i.e. *low_end* ≤ *pf* ≤ *high_end*), then the sequence is split into two, one being <*low_end, pf* −1,*dir*> if *dir* = *up* or <*pf* +1, *high_end, dir*> if *dir* = *down*, and the other consisting of the new fault only (i.e. <*pf, pf, nil*>, nil meaning the direction cannot be determined for now). If *pf* does not extend any existing sequence nor overlap any sequence, then a new sequence is built, <*pf, pf, nil*>. If *pf* can extend two existing sequences, SEQ deletes the older of the sequences (the one whose last fault is earlier) and extends the newer sequence. In addition, if extending a sequence would lead to overlapping with another sequence, then the sequence that would be overlapped is deleted.

SEQ limits the number of sequences that it tracks. (Currently the limit is 200). When adding a new sequence would exceed the limit, SEQ first deletes the oldest sequence (by time of the most recent fault to that sequence) of length less than L. (If all sequences are longer than L, SEQ would delete the oldest sequence with length $\leq 2 * L$, etc.)

When a replacement page must be chosen, SEQ examines all sequences of length $\geq L$, and tries to pick the sequence that faulted most recently. The heuristic we use is to sort these sequences based on the faulting time of their Nth most recent fault, and choose the one with the more recent fault time. Currently $N = 5$. If no sequence with length $\geq L$ exists, the default LRU replacement is used.

Once a sequence is picked, SEQ is constrained not to replaces pages closer than M pages away from the sequence head. Starting from the Mth page away from the head, SEQ skips any on-disk pages, choosing the first in-memory page it finds. If it cannot find an in-memory pages in this sequence, SEQ examines the next sequence as determined above. For efficiency, SEQ keeps track of the range of on-disk pages in each sequence, so that the search for a replacement page can skip many on-disk pages in one step.

To illustrate how SEQ works in practice we'll consider a simple example. The example corresponds to the simplest case in which SEQ will be effective; the behavior of our benchmark es is similar to the behavior in the example, and graphs of es's faults, and of SEQ's chosen replacement pages, will follow.

Suppose a program makes several sequential passes over a single, large memory region (larger than memory size), going from the low end of the region to the high end of the region as time passes. When the memory region is first accessed, each page will be faulted into the address space in turn. A single, large sequence will be created. Midway through the first pass memory will have filled up (the lower portion of the address range occupies memory). Because there is a sequence from which to replace pages, SEQ will page out the newly-faulted pages behind the head of the sequence, which continues to grow upwards as the program progresses. The result after the program's first pass over the address range is that the bottom half of the sequence remains in memory and the top portion has been paged out. On the next iteration, the region's bottom half will be in memory and hence no faults occur for those pages. However, as soon as the program reaches the point in space where memory filled up on the first iteration (and where SEQ started replacing pages), faulting will again commence. The very first fault will have the following effect: since the fault is somewhere in the middle of the single, long sequence that existed up to this time, that fault will *split* the sequence. The bottom half of the old sequence will remain, and a new sequence beginning at the first faulted page will be created.

As the program continues upwards and more pages fault in, the newer sequence will grow, extending upwards with each new faulted page. In a short time the new sequence will have grown to length L and, because its Nth fault is more recent than the Nth fault of the original (bottom half) sequence, SEQ will start to replace pages from the new (upper) sequence. Again the bottom portion of the address range remains in memory and the upper part is paged out. On successive iterations, no faults will occur until the program references memory midway through the address range;

as faults do occur, a sequence will be built, and SEQ will replace pages from the upper memory region once again. [2]

Figures 5 and 6 show, respectively, the memory locations of page faults taken by ES and the pages chosen by SEQ as replacements. The graphs are for runs simulating 50MB of memory, about half of ES's total demand. Recall from Section 2 that es's behavior consists of essentially an iteration over a single large memory region. Observe how the SEQ replacements closely mirror the faults taken by ES.

As a more complex example, Figures 7 and 8 show the faults and replacements for SEQ operating on applu (10MB simulated memory size). From applu's space-time graph we observe that it iterates over four large areas, first accessing addresses in increasing order and then accessing them in decreasing order. The pattern of sequences that are made can be observed from Figure 8. The dashed lines in the figures are due to the fact that applu does not iterate consistently through its address space. It often touches approximately 30–50 consecutive pages and then skips a few pages, leading to a fairly large number of medium-size sequences to feed SEQ. When no sequences of suitable length (longer than $L = 20$ here) are found, LRU replacement is done; LRU accounted for about 70% of page-outs for this run of applu. The combined result of LRU and SEQ replacement on applu is that the upper and lower ends of the four large memory regions remain in memory and the middle regions are the source of replacement pages.

4.2 Simulation Results

Since our traces contain only IN and OUT records, we cannot simulate SEQ accurately under all circumstances. Instead, we conduct a slightly conservative simulation. That is, if a chosen-for-replacement page is IDLE (i.e. it is not accessed until its next IN record), the page is simply replaced; if the page is ACTIVE (i.e. it is between an IN record and an OUT record, which means it is accessed actively during this interval), we replace the page and then immediately simulate a fault on the page to bring it back into memory. This results in a simulation that slightly under-estimates the actual performance of SEQ, because in reality the page fault would occur sometime later in the current or the next interval.

Simulation results are shown in Figures 1 and 2. Clearly, SEQ performs significantly better than LRU, and quite close to optimal, for the applications with clear access patterns (for example, gnuplot and turb3d). For other applications, SEQ's performance is quite similar to LRU. For only one program (perl) does SEQ perform worse than LRU. (We are investigating why SEQ performs poorly for perl.)

We have varied the three SEQ parameters (L, M, and N) and observed resultant performance changes. Intuitively, the larger the value of L, the more conservative the algorithm will be, because it is less likely that a run of faults will be long enough to be considered a sequence. Reducing L has the opposite effect. Similarly, the parameter M is set to guard against the case when pages in a sequence are re-accessed in a short time period. If the pages in the sequence are accessed only once, then M should be set to 1; however, if there is reuse of pages near the head of the sequence, then M should be larger to avoid replacing in-use pages.

We experimented with three different settings of L and M: ($L = 20$, $M = 20$), which are the defaults, ($L = 50$, $M = 20$), and ($L = 50$, $M = 50$), and found that SEQ's performance is unaffected for most of the applications. The three applications that show visible differences are applu, perl, and swim. Figures 9 and 10 shows their fault curves under the three parameter settings. (These figures shows all programs for which any noticeable change occurred in SEQ performance when any parameter(s) changed. SEQ performance was unchanged for coral, gcc, murphi—where it reflected LRU performance—and for es and gnuplot, which both had essentially the same near-OPT performance as before for all parameter combinations.)

For applu, since it has many short sequences that are disqualified for replacement when $L = 50$, SEQ at $L = 50$ essentially performs LRU replacement most of the time. `Swim` also has many small to medium length sequences, and SEQ at $M = 50$ appears to interact poorly with swim's behavior at small memory sizes. For the rest of the applications, SEQ's performance is essentially unaffected by the parameter changes.

The parameter N affects the choice of sequences in situations when sequences grow at varying rates: as N increases, so does the likelihood that SEQ will choose the sequence that grows fastest. We did not choose $N = 1$ because we want to avoid sequences that grow at sporadic rates. Since the space consumed by SEQ is directly proportional to N (it must store the times at which the last N faults occurred), small N is desirable. We varied N from 5 to 20, and found only negligible differences in SEQ's performance; varying N from 5 to 2 has virtually no effect on SEQ's performance. Thus, we set $N = 5$.

To reduce SEQ's modest runtime space requirements further, we experimented with setting $X = 50$. Compared to the default $X = 200$, performance was unchanged for all but two programs, applu and fgm (where performance changed only slightly). Graphs for ($L = 20$, $M = 20$, $X = 50$) also appear in Figure 9. Applu performance degraded slightly in that SEQ did not drop below LRU until a larger memory size was used. The change in SEQ's FGM performance were very minor, just a slight rise at the very high end of the memory size range. We conclude tentatively from this that the number of sequences maintained per program by a real implementation can likely be lowered well under 200 if

[2] In more detail, at the beginning of each iteration after the second, there will be two sequences, one for the bottom portion of the address range and one for the top. When the program reaches the start of the upper sequence—which is on disk, having been replaced the previous iteration—the first fault will destroy that sequence. The fault will fall within that sequence's <*low_end*, *high_end*> range, so the old upper sequence will split. The result will be that if there are any pages between *low_end* and the new fault, then these pages will be put into a sequence. This sequence will be short, or it may not be created at all, because typically the entire upper sequence on the previous iteration will have been paged out (not including pages at the head of that sequence, but the head is way up at the top of the address region). After the split of the previous upper sequence, the new one-page sequence will grow upward to contain the entire upper portion of the address region. If a small, intermediate sequence had been created, it will languish between the large, lower sequence (which remains in memory) and the large, upper sequence (the source of replacement pages). The intermediate sequence will not grow and may remain in memory, but its size is insignificant.

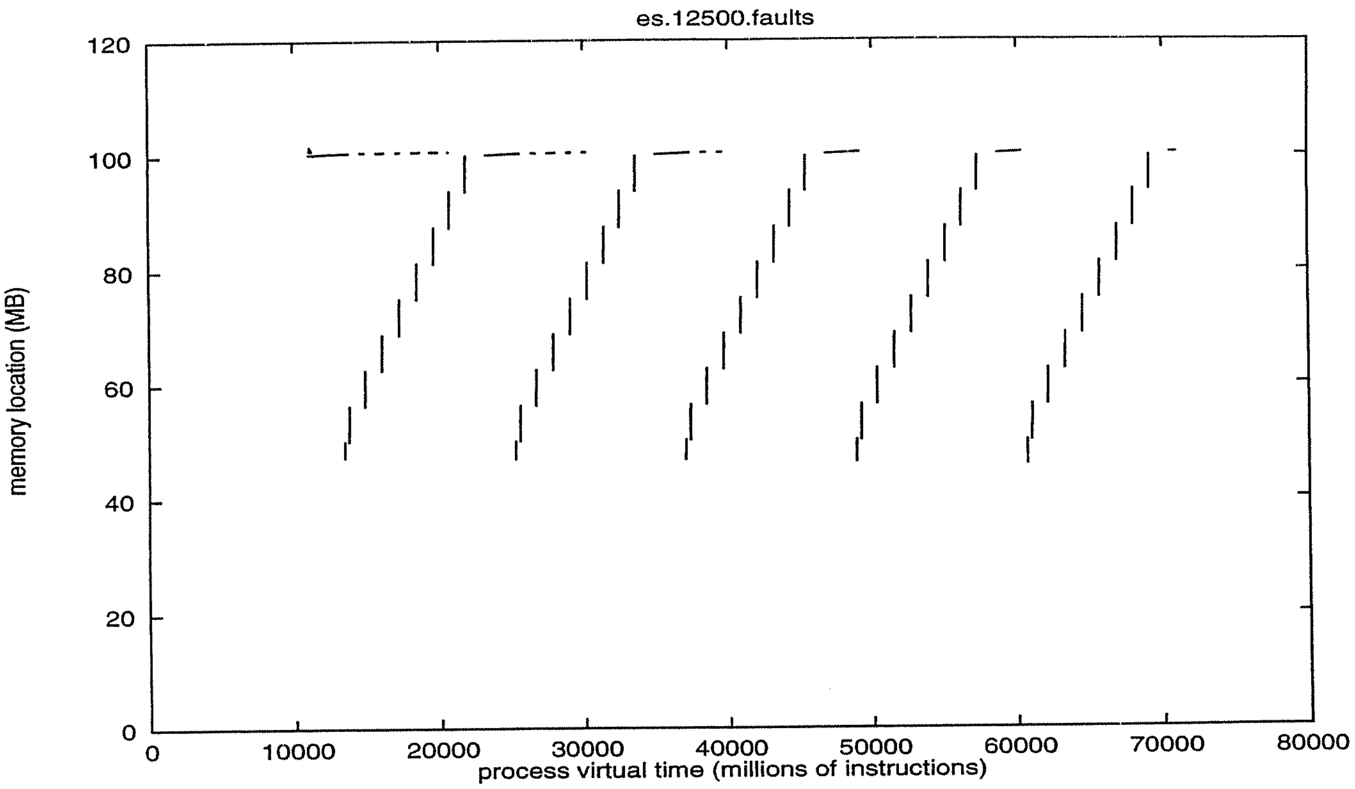


Figure 5: Time and location for page faults taken by ES (50MB main memory) under SEQ (using default parameters). Initial faults (almost all being zero-fill) are not shown.

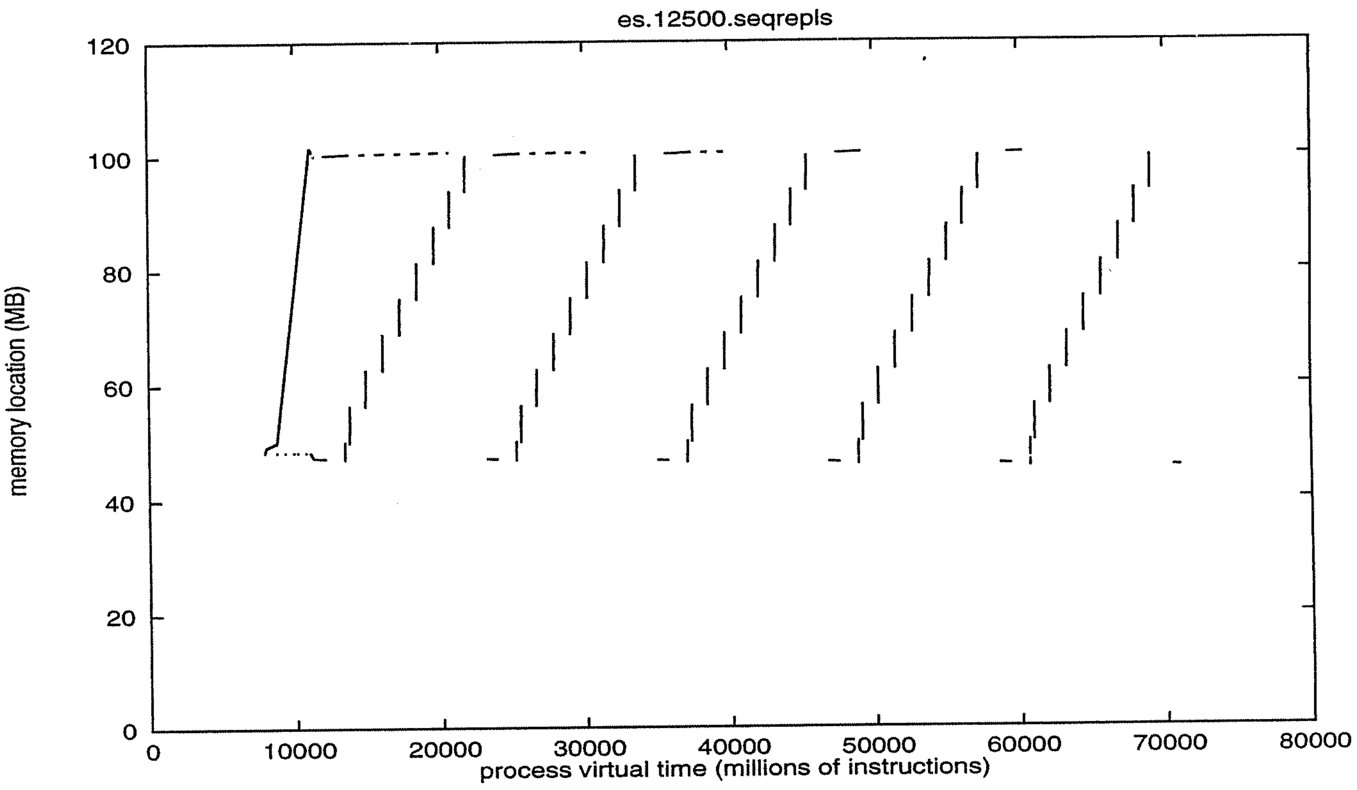


Figure 6: Time and location of page replacement decisions made by SEQ for ES. (LRU decisions are not shown—for ES, in fact, every time a page-out was necessary, SEQ found a page in a sequence, so *no* pages were replaced by the fallback LRU policy.) 50MB main memory was simulated under default SEQ parameters.

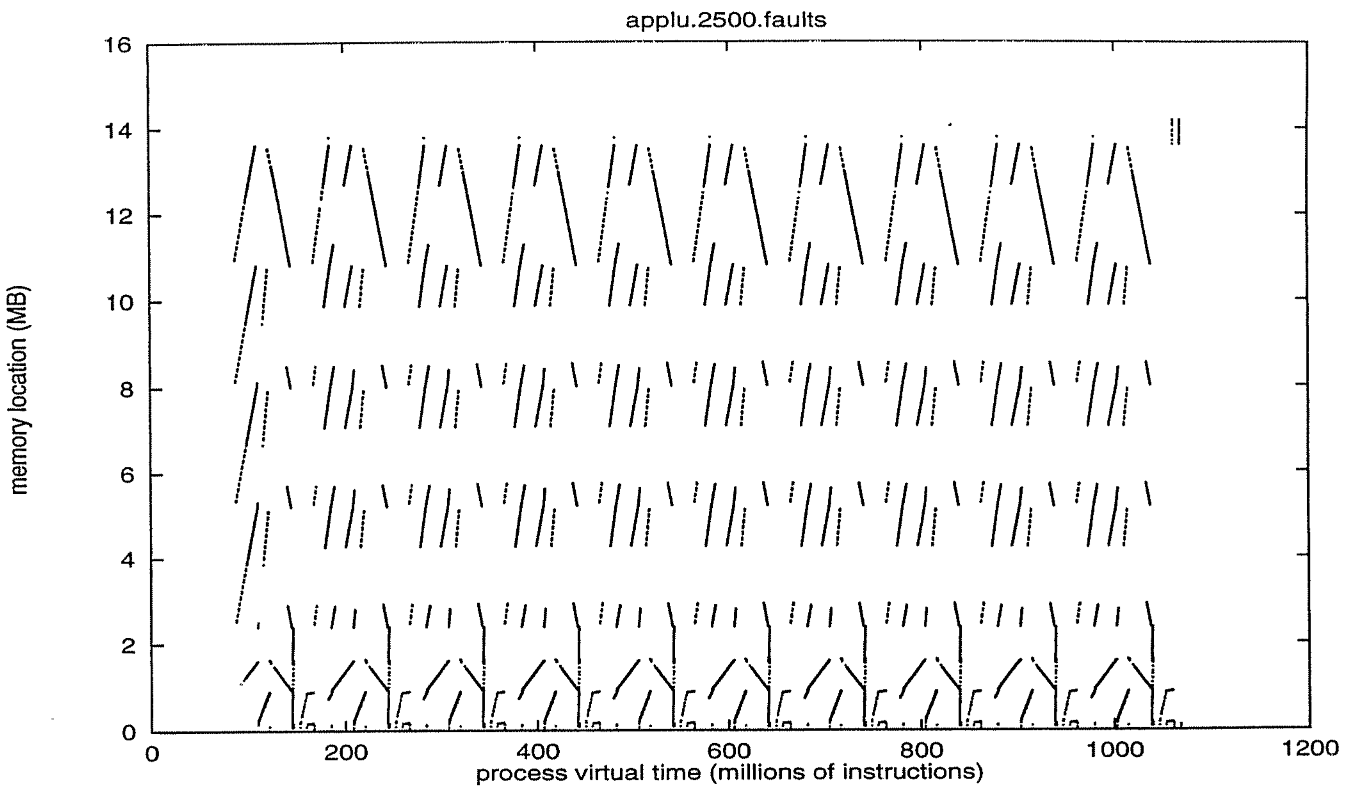


Figure 7: Time and location for page faults taken by applu (10MB main memory) under SEQ (using default parameters).

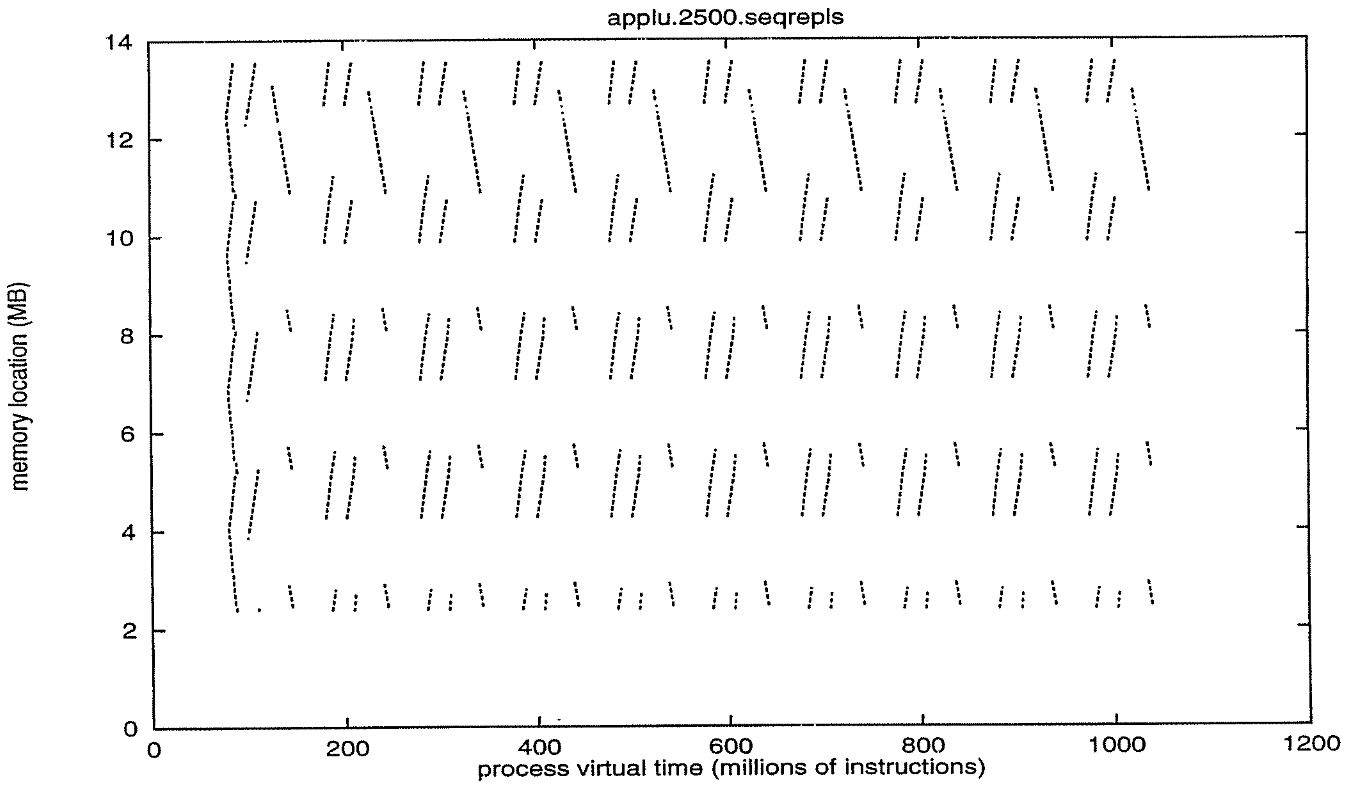


Figure 8: Time and location of page replacement decisions made by SEQ for applu. 10MB main memory was simulated under default SEQ parameters. For this memory size, SEQ made about 15,000 page-outs from sequence pages; default LRU replacement (when in-sequence pages could not be found) accounted for about 35,000 page-outs. Note the dashed curves, which are due to the fact that applu does not access its memory completely sequentially.

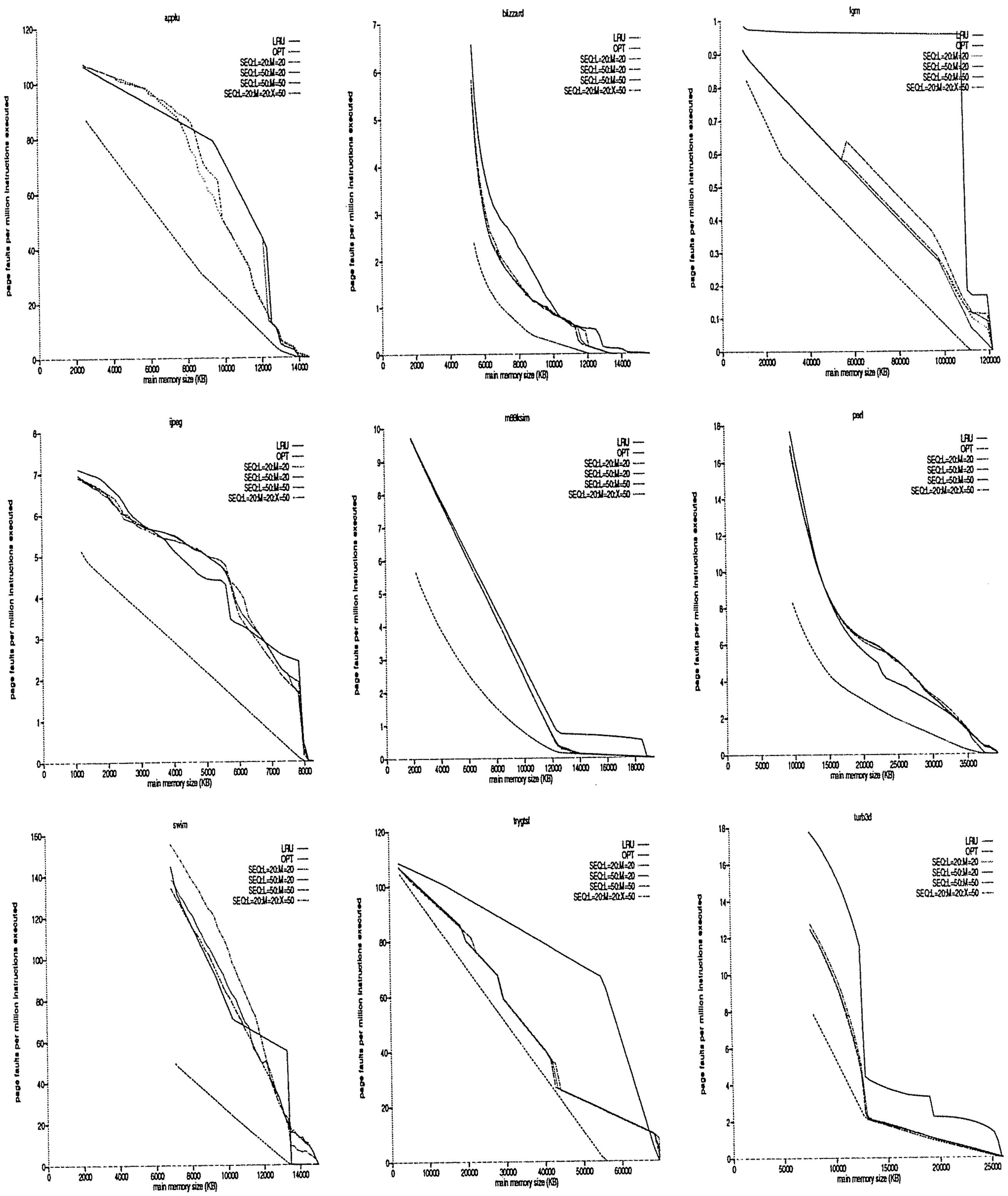


Figure 9: Performance of SEQ under varying parameters. For applu, the curve for SEQ:L=50:M=50 completely overlaps the LRU curve, and SEQ:L=50:M=20 overlaps LRU most of the time. For fgm, SEQ:L=50:M=50 performs worse than other SEQ settings. For swim, SEQ:L=50:M=50 performs more poorly than other SEQ settings. For other programs, differences in SEQ performance are negligible.

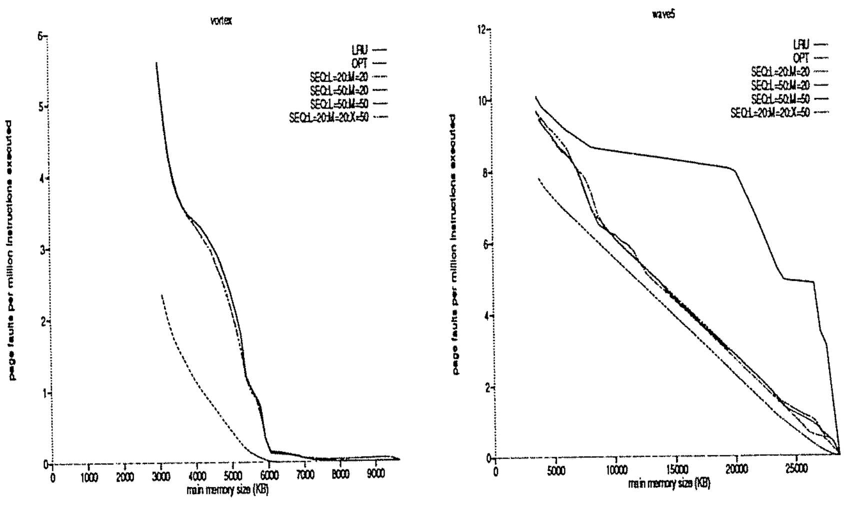


Figure 10: Performance of SEQ under varying parameters for vortex and wave5. Performance is basically constant across parameter settings.

SEQ's runtime resource consumption becomes a problem.

To summarize, we found that the performance of the SEQ algorithm is fairly insensitive to the parameter values, and our current settings appear appropriate, though we plan continued testing in this regard. In our current implementation, SEQ takes roughly 10K bytes to keep track of 200 sequences (each taking roughly 48 bytes). Depending on applications, SEQ also takes slightly more CPU time than LRU for each replacement. We are still working on reducing the SEQ overhead.

5 SEQ as a Global Replacement Algorithm

So far our discussion has focused on the performance of various replacement policies for single applications. In real systems, multiple processes run at the same time and compete for memory. There are two general approaches to page replacement in multi-process environments [12]. One approach involves a memory allocation policy that allocates memory to different processes, and a page replacement policy that chooses replacements among each process' pages when processes exceed their memory allotments. Another approach uses a "global" replacement algorithm, where a replacement page is chosen regardless of which process it belongs to. For example, global LRU replaces the page whose last reference was earliest among all memory pages. Currently, most time-sharing operating systems use some approximation of global LRU replacement.

SEQ can be extended fairly easily to function as a global replacement algorithm. The only modification necessary is that the sequences must be grouped explicitly on a per-process basis, i.e. only page faults with the same process ID are associated for sequence detection.

An obvious question is whether global SEQ would perform well in a time-sharing multi-process environment. To provide a preliminary answer to this question, we constructed a very simplified simulator of a multi-process system that captures the dynamic interleaving of process execution. We use a simple round-robin time slicing policy (simulating execution of each program for a certain length of time) and a time delay to model the service time for a page fault to disk. We then compared the performance of global LRU and global SEQ under concurrent executions of the applications.

Our simulator reads multiple application traces, taking a record each time from the trace corresponding to the program that is currently executing. We schedule processes according to round-robin time-sliced scheduling with context switch at page faults. That is, each trace (process) is run for a quantum, and when the quantum expires, the scheduler puts the trace on the wait queue and picks a different trace (process). When a page fault happens, the current process is suspended for the duration of the service time of the page fault, and the scheduler picks another process to run. The two parameters, quantum time and page fault service time, are determined by a simple estimate of CPU speed—in our experiments the quantum is 1 million instructions (corresponding to 10ms on a machine capable of executing our programs at the uniform rate of 100 MIPS). Page fault service time is a uniform 400,000 instructions (4ms on the same 100MIPS machine). This is obviously a simplistic model, but it suffices for the purpose of creating a reasonable interleaving of multiple program traces.

We picked four combinations, each of two applications, and one combination of three applications. The combinations are chosen to have a variety of mixes of application behavior and relative memory needs. They are: `es+fgm`, `gcc+vortex`, `swim+trygtsl`, `vortex+gnuplot`, and `coral+wave5+trygtsl`. For each combination, we measure the fault rate for the concurrent execution of the applications, under both global LRU and global SEQ, for a range of memory sizes. Again, since most of the initial faults are zero-filled pages rather than disk-read pages, we do not include them in the figures. The results are shown in Figure 11.

The results show that in simple multi-process environments, global SEQ tends to outperform global LRU when sequential applications are run, and it performs similarly to global LRU when no sequential application is run. For example, global SEQ's improvements over LRU in the cases of `vortex+gnuplot` and `coral+wave5+trygtsl` are similar to those in gnuplot and wave5, and global SEQ performs similarly to global LRU in gcc+votex. Thus, our preliminary simulation results show that SEQ is also a promising algorithm for global replacement.

6 Related Work

Operating systems researchers have investigated the memory management problem for over thirty years, originally to determine if automatic management of memory (i.e. virtual memory) could perform as well as programmer-controlled physical memory allocation. Belady's paper in 1966 [2] introduced the optimal offline replacement algorithm (the OPT algorithm). A good survey on early research results on paging policies can be found in [12]. There have also been many studies on program behavior modelling and optimal online algorithms for each model. The models include independent reference [1], LRU stack [25], working set [9], access graphs [4], and the Markov model [16]. For each of these models, optimal online algorithms are found [12, 14, 16].

The SEQ algorithm is similar to the access-graph algorithms [4] in that it tries to take advantage of patterns found in reference streams. However, most theoretical studies on access-graph algorithms assume that the graph is known ahead of time, rather than being constructed at run-time. A recent study [11] investigated constructing the graph at run-time; however the study only looked at references to program text, not data. Also, the algorithm proposed in [11] is more complex and more expensive than SEQ.

Although most early experimental studies focused on efficient approximation of LRU page replacement [3, 2, 7, 19], one scheme, the Atlas Loop Detector, investigated loop detection and MRU replacement on scientific programs [17]. SEQ differs from the loop detector in that it tries hard to work well on applications where LRU is appropriate. The Atlas scheme apparently performed poorly for non-scientific programs [9].

Recent research projects on application-controlled kernels show the potential of application-specific replacement policies [27, 13, 20, 18]. These studies focus on mechanisms by which applications inform the kernel about what pages would be good candidates for replacements. Our SEQ algorithm is basically the antithesis of such schemes. It will be interesting to see over time which philosophy prevails. Our study shows that run-time automatic sequence detection by the kernel may be a promising way to increase performance, at essentially no cost to the programmer.

Recently there have been a number of studies of applications' memory reference behavior in the context of cache management. One study regarding processor pin bandwidth requirements [5] confirmed that there is a significant difference in cache miss ratios under LRU and under OPT replacement policies. Another study [22] included space-time graphs for some SPEC95 benchmarks. Though their graphs are for a much shorter duration of execution execution (on the order of one second), the graphs are similar to our graphs for the SPEC95 benchmarks. Finally, one study of large-scale multiprocessor architectures investigated the "working-set" and cache size issues for parallel scientific applications [24]. The study investigated a number of parallel applications, measuring their "working-sets" by simulating the number of cache misses versus cache sizes under the LRU replacement. The cache misses versus cache size curves in [24] are quite similar to our LRU page fault curves for scientific applications. These studies suggest that the reference behavior at page level might be similar to that at cache line level. We plan to investigate this correlation.

Sequence detection can be used for prefetching purposes as well. Indeed there are sequence detectors for prefetching in hardware cache management [26, 15, 21]. However, prefetching does not reduce bandwidth consumption; it merely reduces latency by overlapping I/O with computation. Good replacement policies, on the other hand, reduce both bandwidth consumption and latency. In this paper we focused on replacement algorithms only; how to balance prefetching and cache management (page replacement) is a complicated issue that needs further study [6].

7 Conclusions and Future Work

Our study of application reference behavior and space-time graphs shows that applications' memory reference behavior varies significantly. There are at least three categories: no visible access pattern, minor observable patterns, and regular patterns. We found that LRU performs similarly to OPT, though incurring roughly twice as many page faults, for the memory-intensive and pattern-less applications. However, LRU performs poorly for regular-pattern applications.

We proposed a new replacement algorithm, SEQ. SEQ detects linear access patterns (sequential behavior) and performs semi-MRU replacement on sequences associated with such patterns. SEQ performs similarly to LRU for memory-intensive applications, and corrects the LRU flooding problem for many regular-pattern applications. Indeed SEQ's performance approaches that of OPT for a number of regular-pattern applications.

We also found that for multi-process systems, SEQ appears to be a good algorithm for global replacement. Comparison of global LRU and global SEQ show that global SEQ can effectively improve multi-application performance just as it improves single application performance.

There are a number of limitations in our work. We need to experiment SEQ on a wider variety of applications. Kernel implementation of SEQ is underway to test its performance in real systems. Finally, we plan to incorporate prefetching in SEQ.

Acknowledgements

We would like to thank our referees for their detailed comments and for pointing out a simulation error (which we have fixed) in the earlier version of the paper. Mark Hill, Mary Vernon and Doug Burger provided helpful feedback on an early draft of this paper. The research is partially supported by a generous grant from Intel Corporation.

References

[1] A.V. Aho, P.J. Denning, and J.D. Ullman. Principles of optimal page replacement. In *Journal of ACM, vol.*

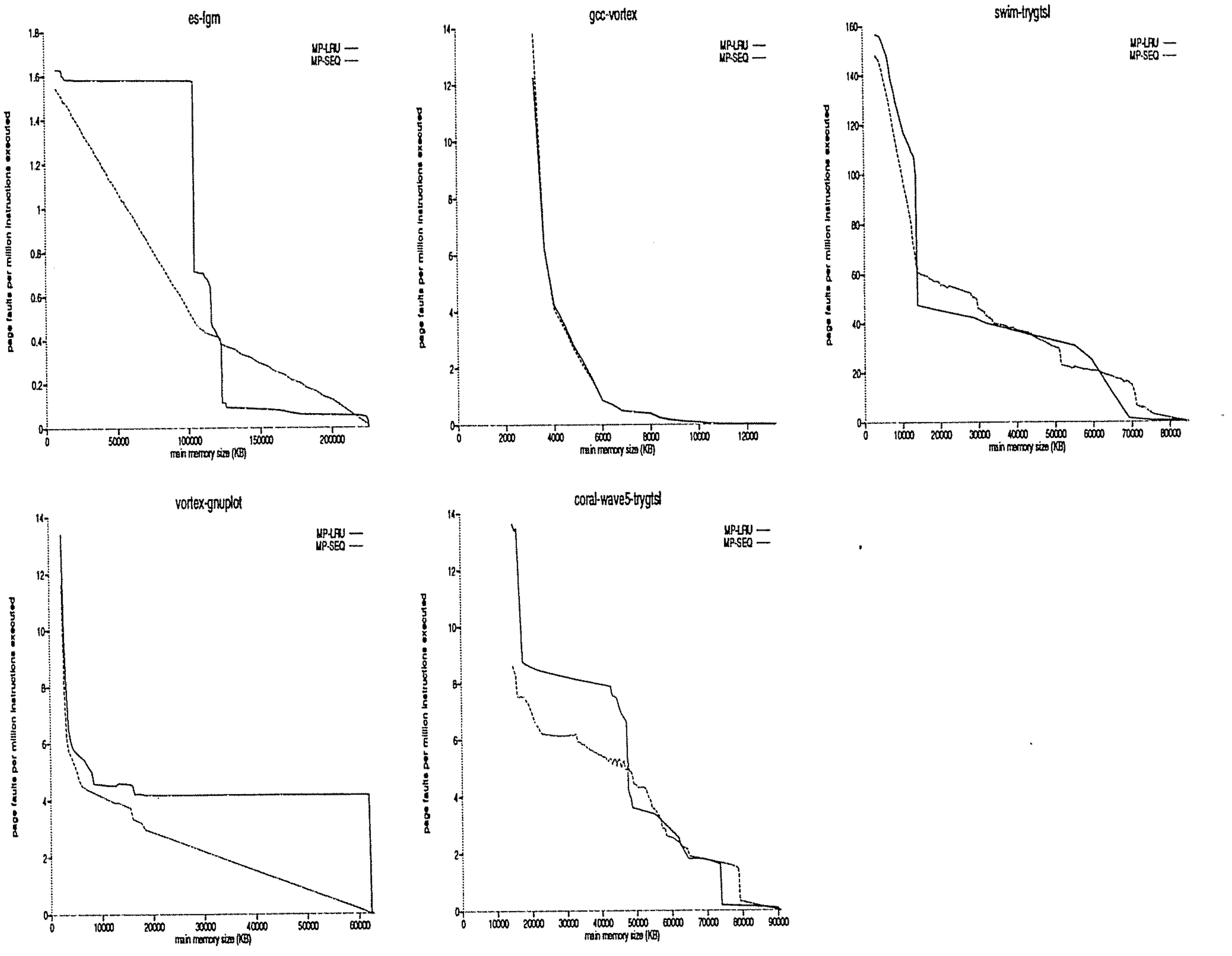


Figure 11: Performance of Global LRU and Global SEQ for concurrent execution of applications.

18, pages 80–93, January 1971.

[2] L. A. Belady. A study of replacement algorithms for virtural storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[3] A. Bensoussan, C.T. Clingen, and R.C. Daley. The multics virtual memory: Concepts and design. In *Communications of the ACM, 15(5)*, pages 308–318, May 1972.

[4] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. In *Proc. 23rd ACM Symposium on Theory of Computing*, pages 249–259, 1991.

[5] D.C. Burger, A. Kagi, and J.R. Goodman. Memory bandwidth limitations of future microprocessors. In *The 23rd Annual International Symposium on Computer Architecture*, May 1996.

[6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proc. 1995 ACM SIGMETRICS*, pages 188–197, May 1995.

[7] R. W. Carr and J. L. Hennessy. WSCLOCK — A simple and effective algorithm for virtual memory management. In *Proc. 8th SOSP, Operating Sys. Review*, page 87, December 1981. Published as Proc. 8th SOSP, Operating Sys. Review, volume 15, number 5.

[8] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems Labs, 1993.

[9] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[10] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[11] Amos Fait and Ziv Rosen. Experimental studies of access graph based heuristics: Beating the LRU standard? In *1997 SIAM Symposium on Discrete Algorithms*, January 1997.

[12] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., 1973.

[13] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, SIGOPS Operating Systems Review Special Issue*, volume 26, page 187, Boston, MA, October 12-15 1992.

[14] S. Irani, A. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 228–236, 1992.

[15] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] Anna R. Karlin, Steven J. Phillips, and Prabhakar Raghavan. Markov paging. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 208–217, June 1992.

[17] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. In *IEEE Transaction on Electronic Communications*, pages 223–235, April 1962.

[18] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 153–164, November 1994.

[19] H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 15(3):35–41, March 1982.

[20] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of USENIX Mach Symposiumi '91*, pages 17–29, 1990.

[21] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. 21st Annual International Symposium on Computer Architecture*, pages 24–33, May 1994.

[22] Sharon E. Perl and Richard L. Sites. Studies of windows NT performance using dynamic execution traces. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Seattle, October 1996.

[23] Rob Pike. Notes on programming in C, February 1989. Available at `http://www.lysator.liu.se/c/pikestyle.html`.

[24] E. Rothberg, J.P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *The 20th Annual International Symposium on Computer Architecture*, pages 14–26, May 1993.

[25] G. S. Shedler and C. Tung. Locality in page reference string. In *SIAM J. Computer, vol. 1*, pages 218–241, September 1972.

[26] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[27] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the implementation of a Multiprocessor Operating System. In

The Proceedings of the 11th Symposium on Operating System Principles, November 1987.