Memory Reference Behavior and Page Replacement for Modern Applications

Abstract

We present a study of memory reference behaviors of modern applications and performance of page replacement policies in virtual memory systems. We begin with memory reference traces for fourteen non-trivial programs. These traces, when plotted in the form of space-time graphs—address space references as a function of process execution time—reveal strikingly different patterns for different applications.

We then examine how the dominant current page replacement policy (LRU) compares to the optimal policy (OPT). We find that LRU performs acceptably for many programs, but for a significant class of applications, LRU performs quite poorly. To remedy LRU's deficiencies, we have synthesized a new replacement algorithm, SEQ. SEQ is designed to handle programs that suffer under LRU, while at the same time not penalize programs which do well under LRU. Trace driven simulation reveals that SEQ does in fact perform well in practice, often coming within five to ten percent more page faults than OPT.

We then generalize SEQ to handle multi-process global memory replacements. Our simulation of concurrent applications show that global SEQ performs as well as or significantly better than global LRU for this workload.

1 Introduction

As the performance gap between memory systems and disks increases, memory management becomes an increasingly important issue for system performance. Although buying more memory always alleviates the poor performance of virtual memory, system designers need to investigate improvements to virtual memory system design and policies so that the users receive the best attainable performance on a limited budget. In addition, for new architecture alternatives such as single chip computer (or Intelligent RAM), obtaining the best performance from on-chip DRAM is crucial to high performance. Unfortunately, studies on the memory reference behavior of modern applications, or improvements to memory system policies, have been scant for the past twenty years.

In this study we collected fourteen memory-intensive applications and studied their memory reference behavior. Seven of the applications are from the SPEC95 suite; the rest are "big-memory" applications including both integer-intensive applications (e.g. database), and scientific computations, whose poor performance under the current virtual memory systems frustrates users. By collecting page references for each execution interval of one million instructions and then plotting memory references as a function of time, we found that applications have quite different memory reference behavior: some are truly memory intensive, referencing many pages in each one-million instruction interval, while others have clear memory reference patterns that can be exploited for better caching and prefetching.

From the traces of these applications we simulated the performance of the least-recently used (LRU) replacement algorithm and the optimal algorithm (OPT) for varying memory sizes. Surprisingly, LRU performs much poorer for applications that have clear access patterns than for those that have no visible patterns. LRU frequently exhibits plateau behavior, where no reduction in page fault rate is found for an extended range of memory sizes, resulting in what looks like a "working-set" behavior whereas in fact it is just LRU performing poorly. The rest of the applications show continuous benefit (i.e. page fault rate reduction) as memory size increases under both LRU and OPT, and there is no clear point where a "working-set" can be found.

The poor performance of LRU prompted us to search for a new replacement algorithm, SEQ. SEQ normally performs LRU replacement, and monitors page faults as they occur, searching for sequences of faults to contiguous virtual addresses. If sequences of page faults are found, SEQ performs a pseudo-MRU (most recently used) replacement on the sequences, attempting to imitate what OPT would do. SEQ corrects the poor performance of LRU and eliminates the plateau behavior for applications that have sequential behavior, yet performs the same as LRU for other types of applications. Thus, SEQ is able to extract improvement in a manner similar to OPT—continuous improvement as memory size increases—for all applications we investigated.

The need for different replacement algorithms raised the question of what the allocation strategy should be for a multiple process environment. In this paper we also conduct a preliminary study of two global allocation 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. In addition, SEQ effectively overlaps I/O and computation for concurrent applications. Thus, SEQ has the potential to perform well as a global replacement algorithm in a real operating system.

In what follows, we first introduce the applications and their memory space-time graphs. We then show the performance of LRU, OPT, and SEQ for these applications. Section 4 describes in detail the new algorithm, SEQ, which addresses the problems of LRU. Section 5 gives preliminary results on the multiple process allocation question. Finally, Section 6 discusses related work and Section 7 concludes.

2 Applications and Traces

The fourteen applications we studied and their sources are as follows.

- applu (SPECfp95): floating point intensive; solves five coupled parabolic/elliptic Partial Differential Equations; written in Fortran, run with *train* input set;
- coral: a deductive database system [28], written in C++, evaluating a nested query;
- es: microstructure electrostatics simulation used in fluid simulation [15], written in C, computing boundary integral equations for an N-body problem;
- fgm: finite growth model test [14] (compares greyscale images), written in C;
- gcc: GNU CC version 2.6.3, compiling the preprocessed file `cccp.i` from the SPEC95 distribution for that version of the GCC benchmark;
- gnuplot: graph plotting software, processing an address space trace to generate a PostScript image like those shown in Section 2.2 below.
- m88ksim (SPECint95): cycle-level simulator for the Motorola 88100 microprocessor, written in C, using the *ref* input set;
- murphi: protocol verifier tool Murphi [13] version 2.9.1, verifying a locking protocol (sample input file `mcslock1.m`); the verifier compiles the specification into a C++ program; we tested the resulting C++ program, not the compiler itself;
- perl: interpreted scripting language, version 5.001, written in C, executing the *jumble* script from SPECint95 version of the perl benchmark;
- swim (SPECfp95): shallow water simulation, written in Fortran, running the *train* input set (*ref* input was observed to perform quite similarly on all measurements below);
- trygtsl: a tridiagonal matrix solver, from the Linpack linear algebra library package, written in Fortran, solving a 3.5 million element tridiagonal matrix;
- vortex (SPECint95): a single-user object-oriented database transaction exercising a kernel coded in C; in-memory derivative of a full OODBMS, running the *train* input;
- wave5 (SPECfp95):, 2-D relativistic electromagnetic particle-in-cell simulation code for various plasma phenomena, written in Fortran, running the *train* input from SPEC;

A brief summary of these applications, including their total data set size and instruction counts, appear in Table 1.

2.1 Trace Collection

We collected memory reference traces using Shade [9]. Shade is an instruction-level simulator for SPARC that allows one to write a processing routine to examine each application instruction before it executes. We observe PC values and virtual addresses for load and store instructions. Storing every address accessed would result in trace files of immense size for non-trivial run lengths. To remedy this problem, we record "intervals" of references. This requires moderate amount of storage space for log files and reduces processing overhead, yet allows accurate simulations for a wide range of memory sizes.

We recode intervals of memory references in the form of "IN" and "OUT" records. The IN and OUT

Program	Description	Memory used (4KB pages)	Instructions executed (in millions)
applu	Solves five coupled parabolic/elliptic PDEs	938	556
coral	Deductive database evaluating query	7384	8184
ES*	microstructure electrostatics	26122	71003
FGM*	finite growth model	30377	35210
GCC	Optimizing C compiler	984	1370
gnuplot	PostScript graph generation	15629	4949
m88ksim*	Microprocessor cycle-level simulator	4838	10020
murphi	Protocol verifier	2345	1018
perl*	Interpreted scripting language	9836	18979
swim	Shallow water simulation	3754	443
trygtsl	Tridiagonal matrix calculation	17422	377
turb3d	Turbulence simulation	6513	17988
vortex	Main memory database	2419	2507
wave5	Plasma simulation	7175	3773

Table 1: Benchmark programs measured, with memory address space size and execution duration. * Indicates runs which were terminated before they completed.

records associated with a page mark the beginning and end of a period when the page is referenced; exactly how often and when the page is referenced in this period is unknown. However, between an OUT record of a page till the next IN record of the same page, the page is guaranteed not to be accessed.

The IN and OUT records are generated in the following way. We divided program instruction time into fixed-length *intervals*, each one million instructions long. At the end of every interval, we check whether there are pages that are referenced in the present interval but not in the previous interval; for these pages we generate an IN record timestamped with the actual time (instruction number) of reference. We also check for pages that were accessed in the previous interval but are not accessed in the current interval; for these we generate an OUT record with the timestamp of the instruction making the last reference. The records are then sorted based on their instruction number timestamps. In what follows, we measure time in terms of instructions executed. This implies that our per-program execution times are somewhat dependent on local environment (GCC 2.6.3 compiling under Solaris 2.4), but we don't consider this a serious restriction.

We can use the intervals of references to simulate *accurately* a number of replacement algorithms. The details are explained in Section 3.1.

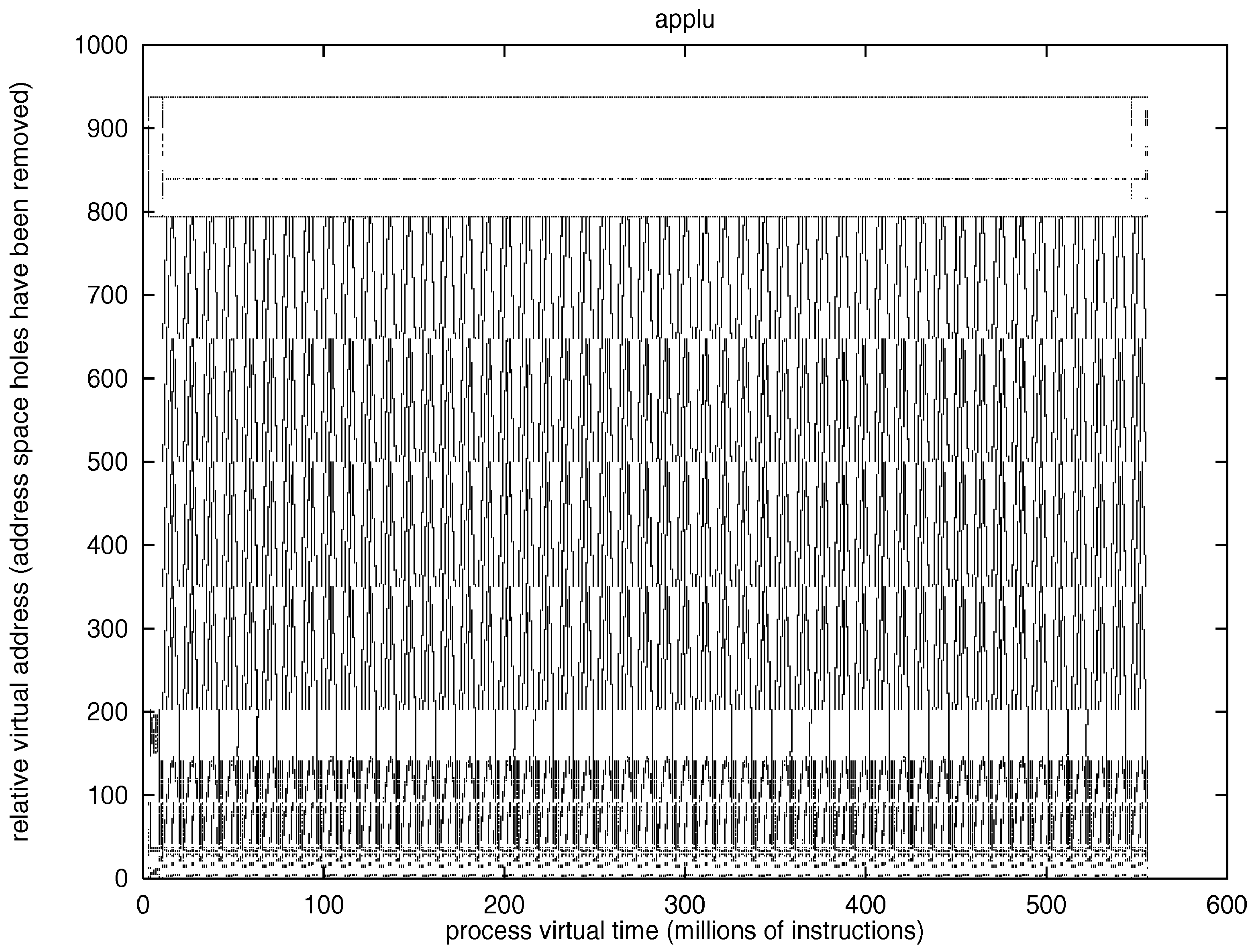
2.2 Space-Time Graphs

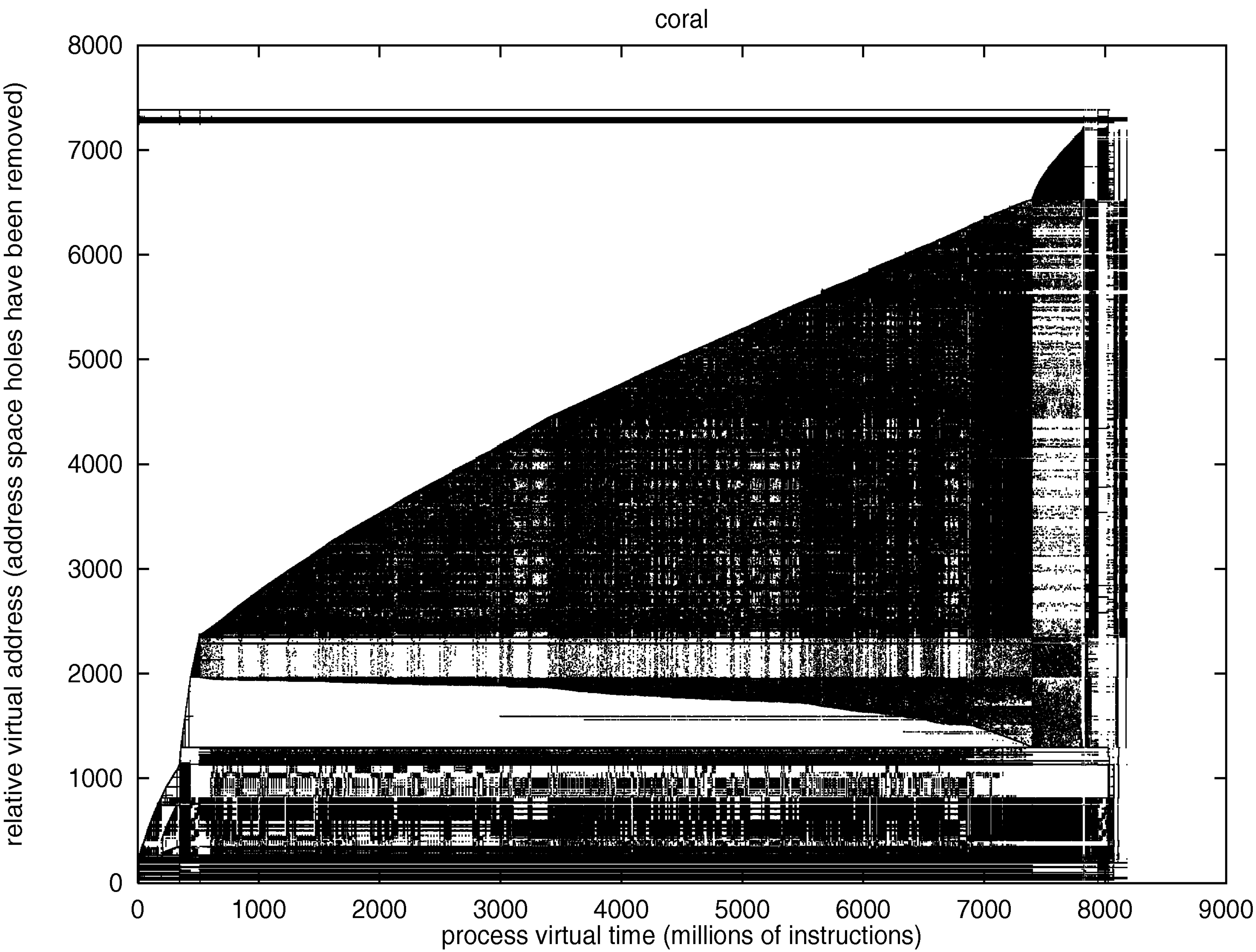
The graphs below show in a graphical form the memory reference behavior for the fourteen applications. The resolution is 300dpi, and the width of each figure is 9.5in, height 7in. [1] From the traces we know what pages are referenced in every 1M instruction interval. For each execution interval (x axis point) we plot a point for each page referenced in that interval [2]. The y-axis are page numbers, with pages numbered from the low end of the address space to the high end, with "holes" (i.e., pages that are never referenced) skipped. That is, y-axis values are merely not virtual addresses. Since we have to fit 500-78000 1M instruction intervals on the x-axis, and generally thousands of pages on the y-axis, the figures can only show access patterns at a coarse level. However, some observations do stand out from the figures.

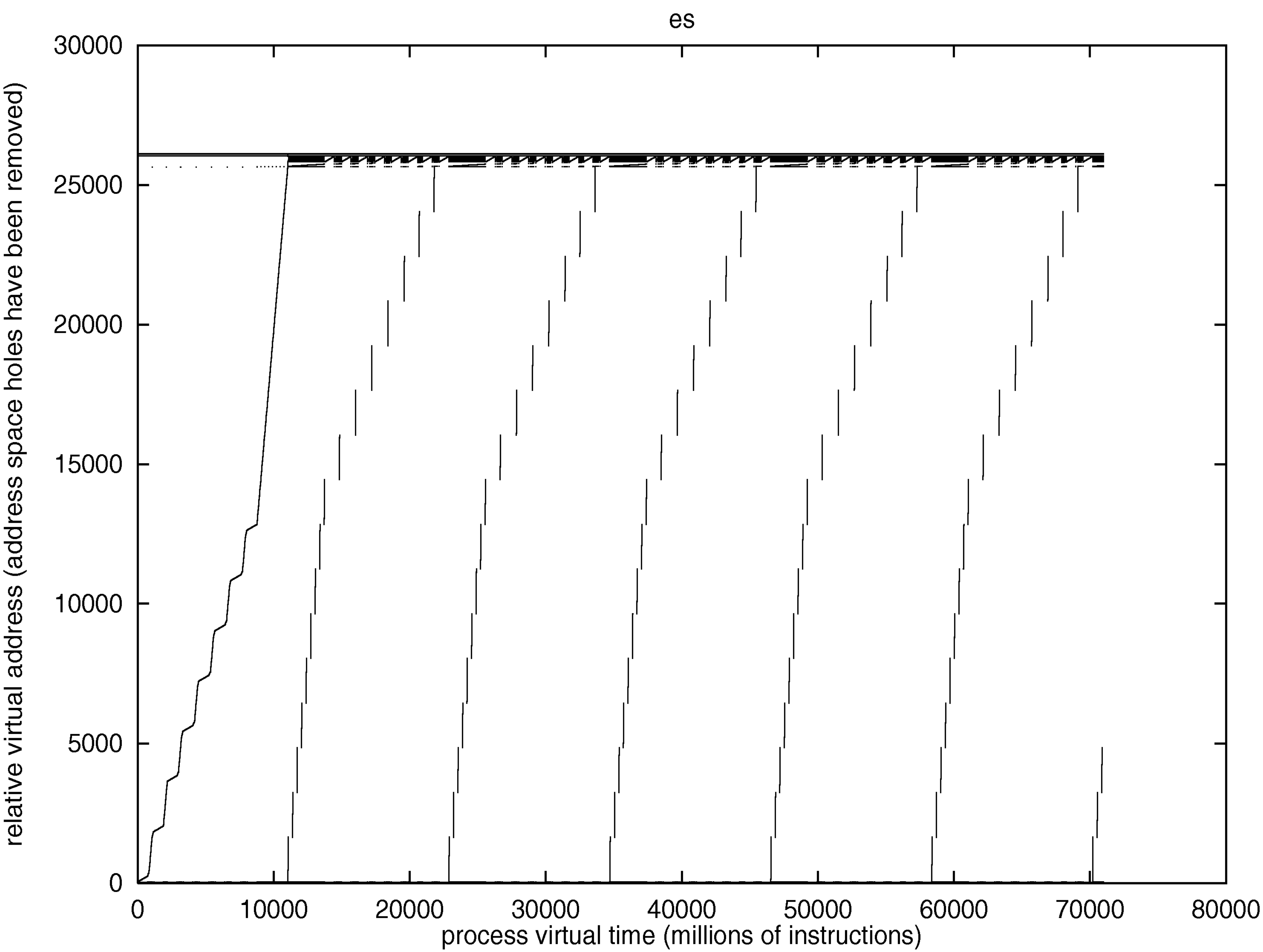
The applications seem to fall in three categories. The first, including coral, murphi, m88ksim and vortex, are truly memory intensive—large amounts of pages are accessed during per execution interval. There are no clearly visible patterns within these vast dark areas. The second one category, include applu, gcc, and perl, are also memory intensive, but have some patterns at small scale (for example, in gcc, the traversal of pages numbered $\tilde{2}0$ to $\tilde{2}80$ follows a certain

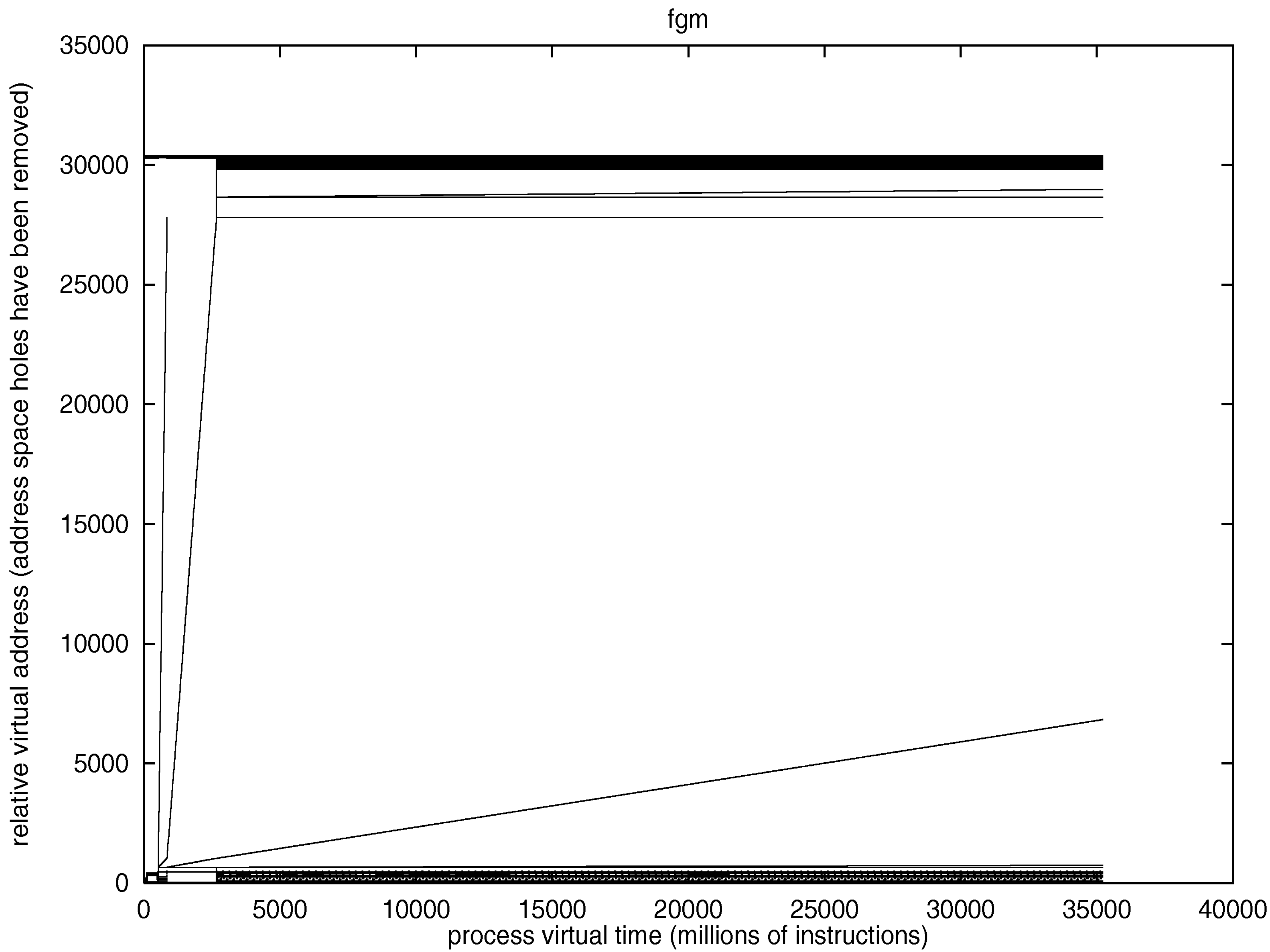
[1] The resolution of the figures is unnecessarily low due to our concerns about the postscript file size; we expect to be able to provide higher resolution figures for the final version.

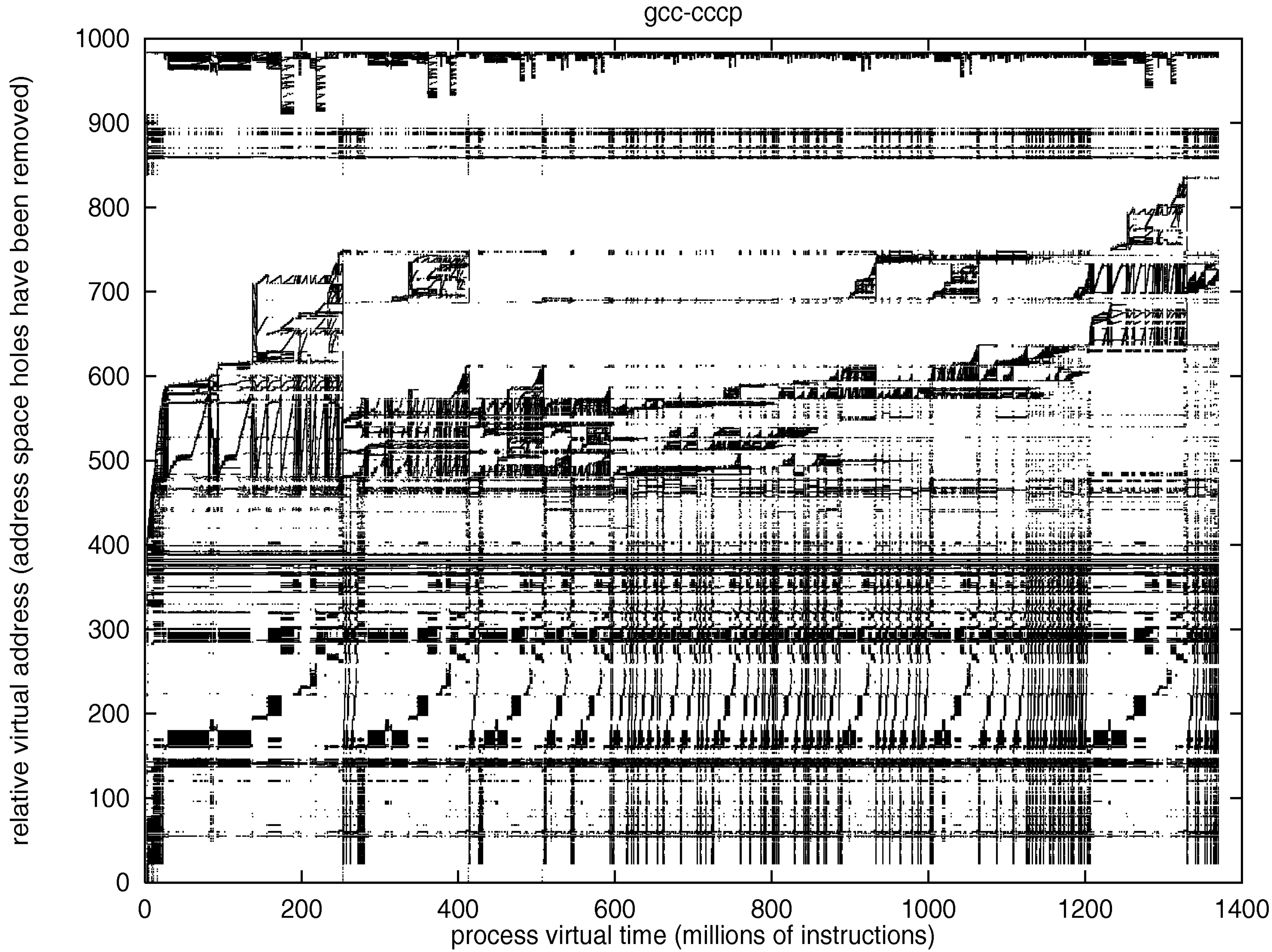
[2] In the plots below, long running applications coral, es, fgm, perl, and turb3d have x-axis granularities of 5M instructions for reasons of space.

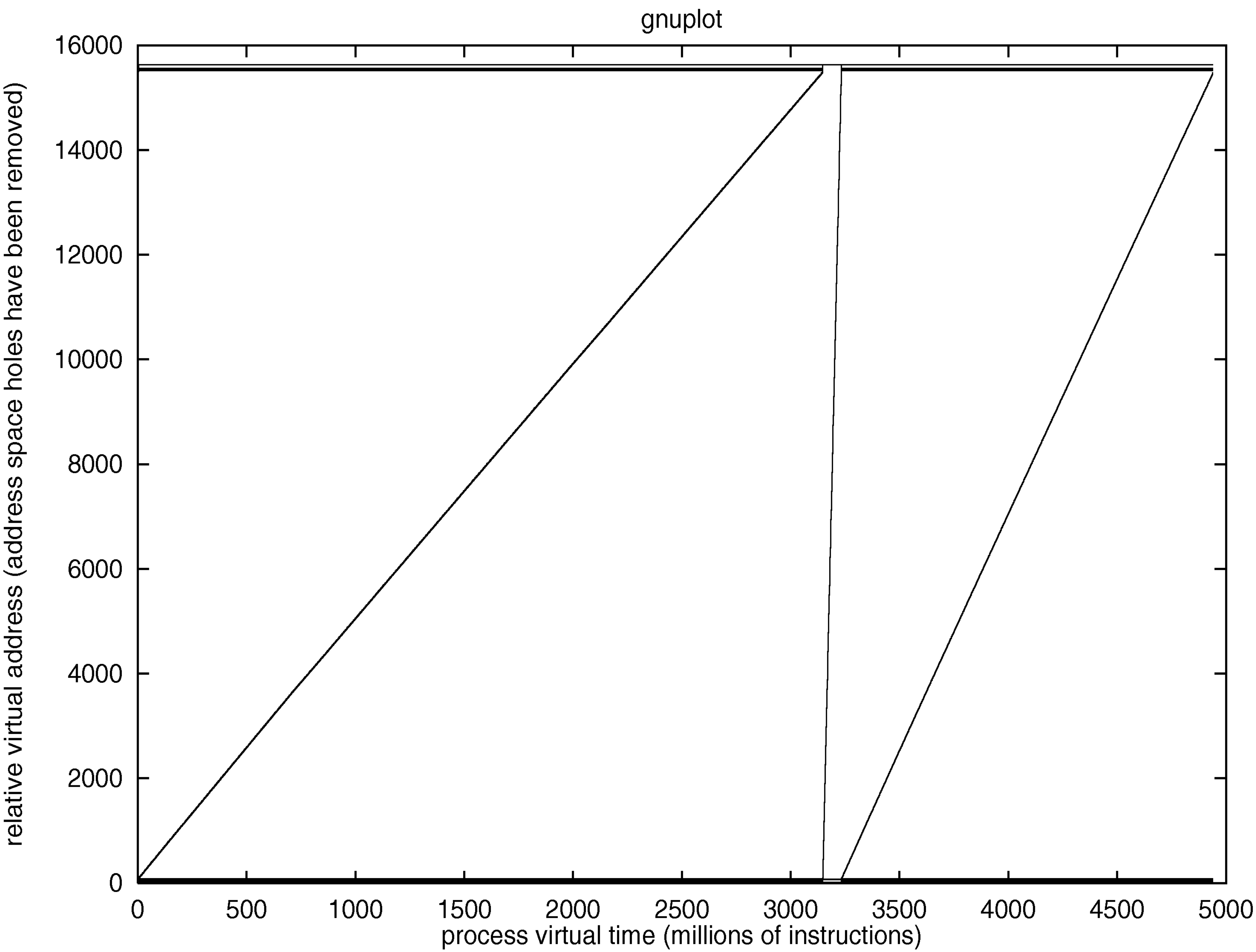

applu
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
100
200
300
400
500
600
700
800
900
1000

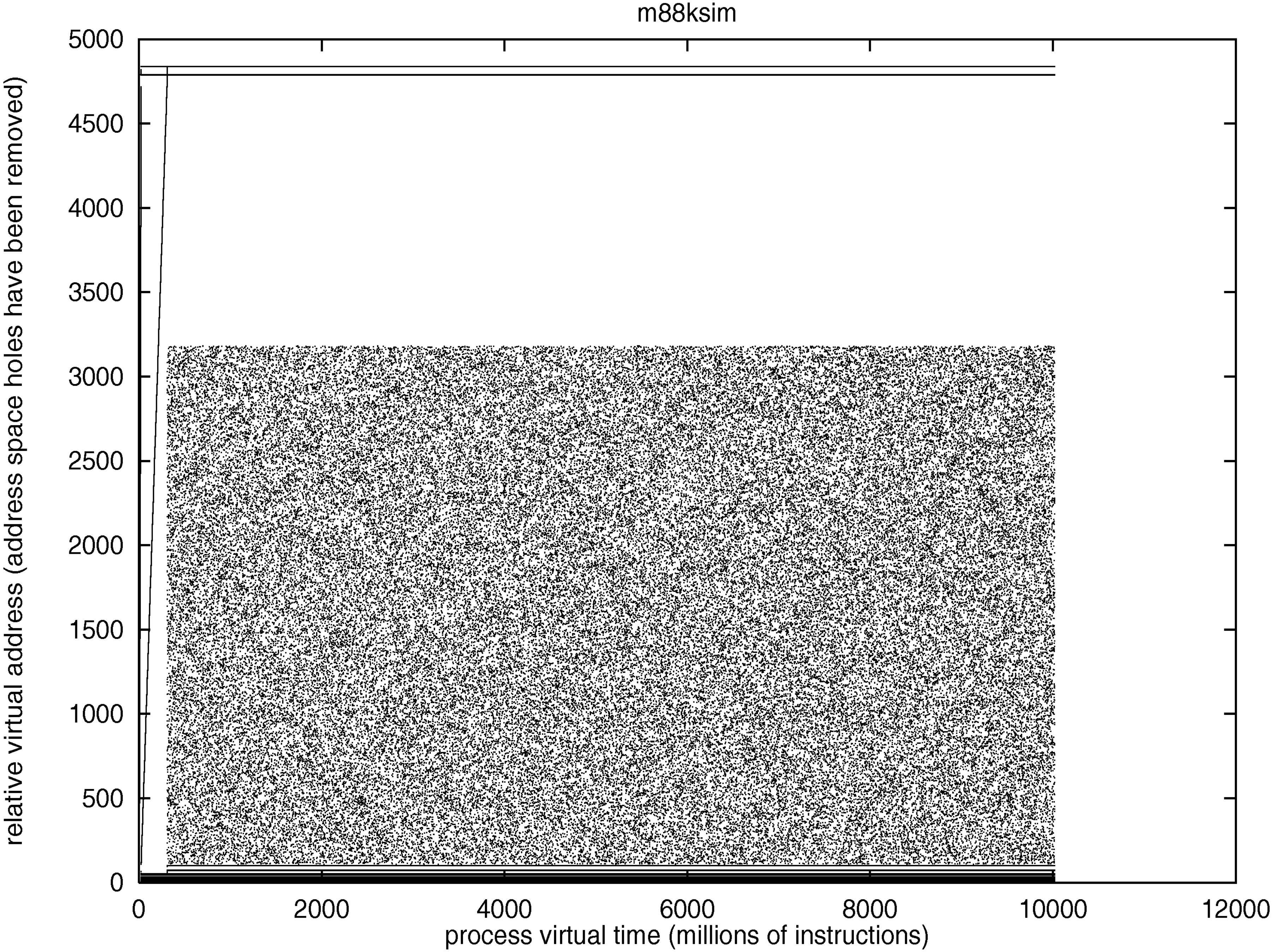

coral
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

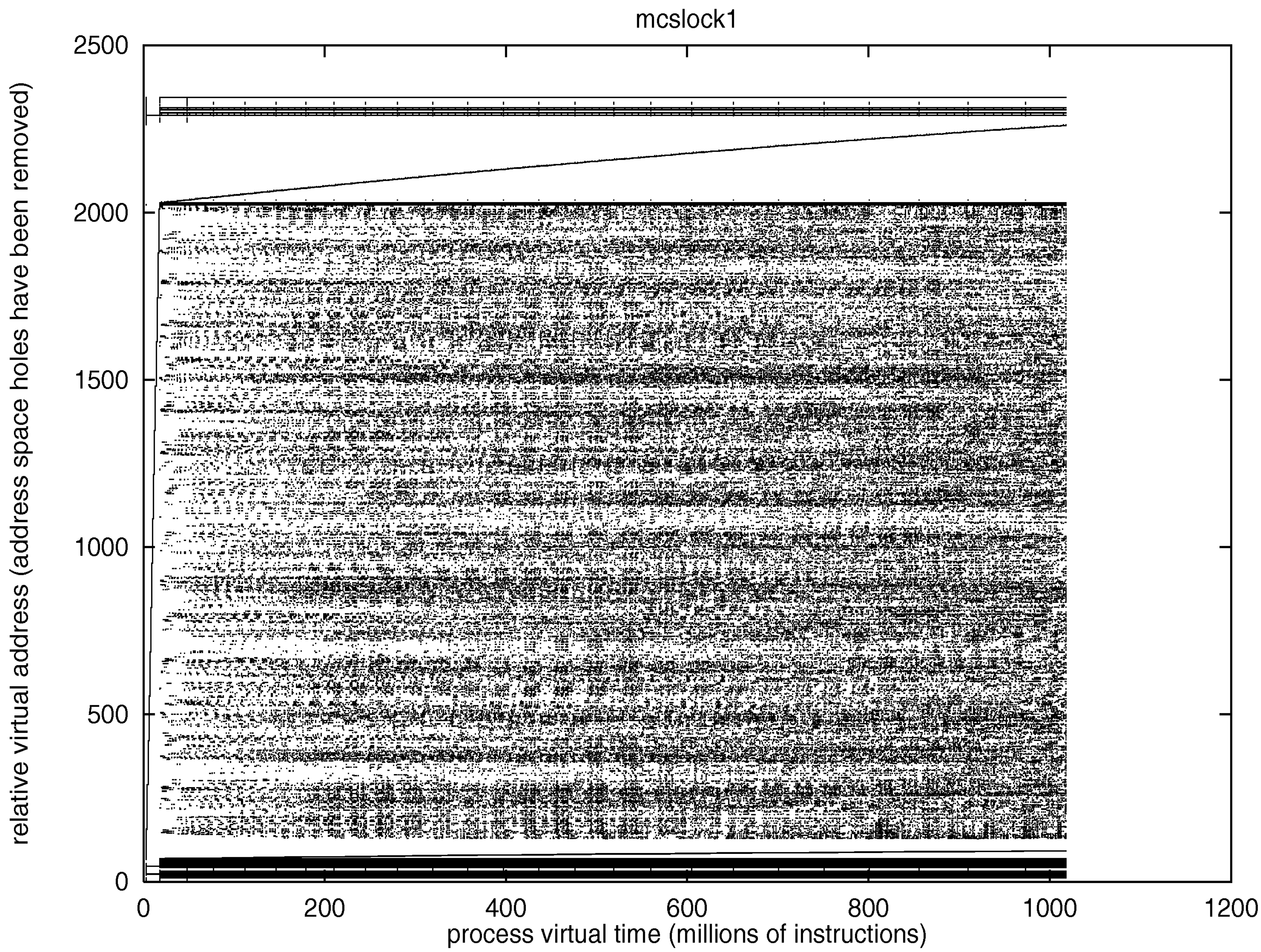

es
30000
25000
20000
15000
10000
5000
0
relative virtual address (address space holes have been removed)
0
10000
20000
30000
40000
50000
60000
70000
80000
process virtual time (millions of instructions)

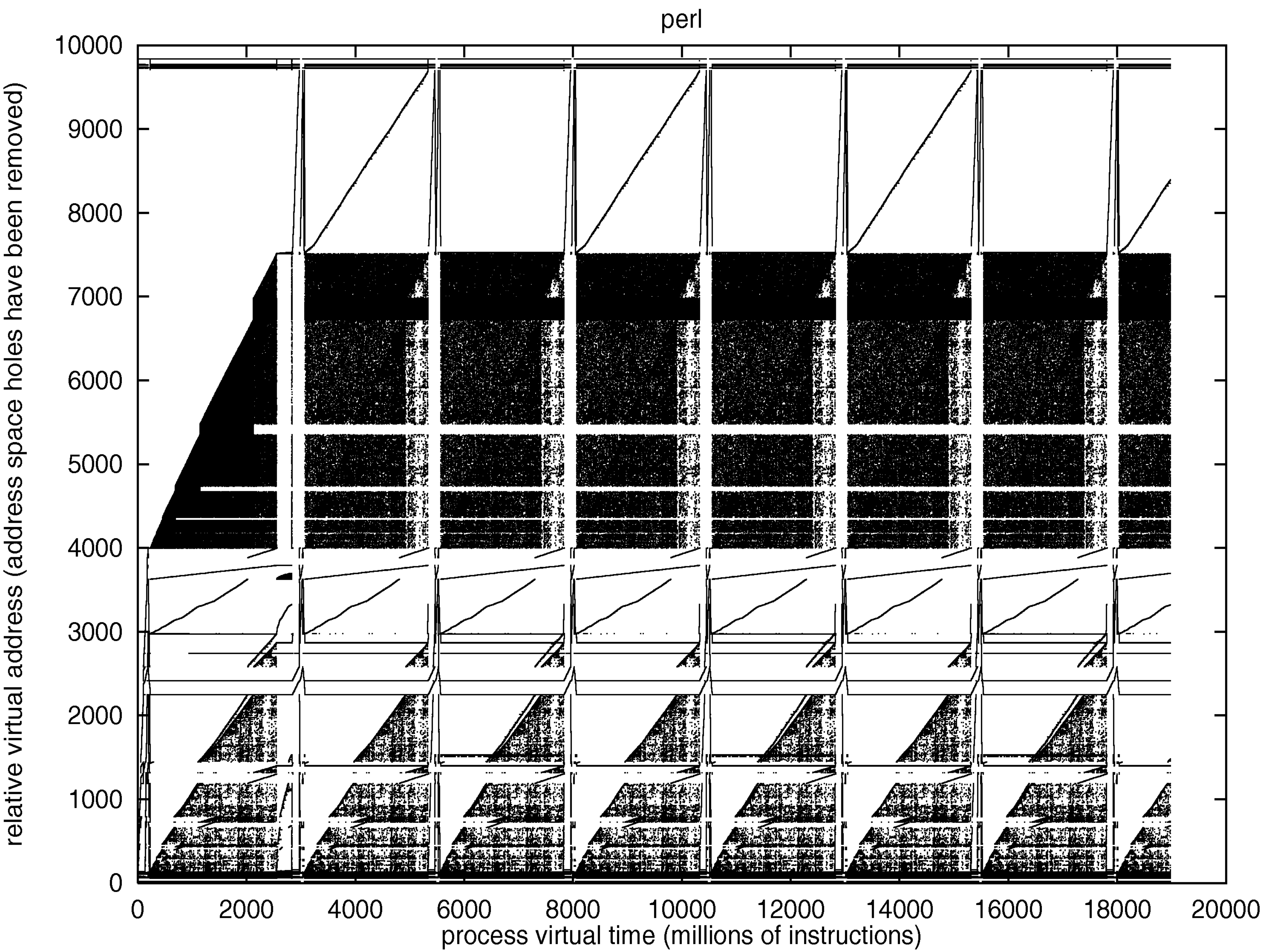

fgm
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
5000
10000
15000
20000
25000
30000
35000
40000

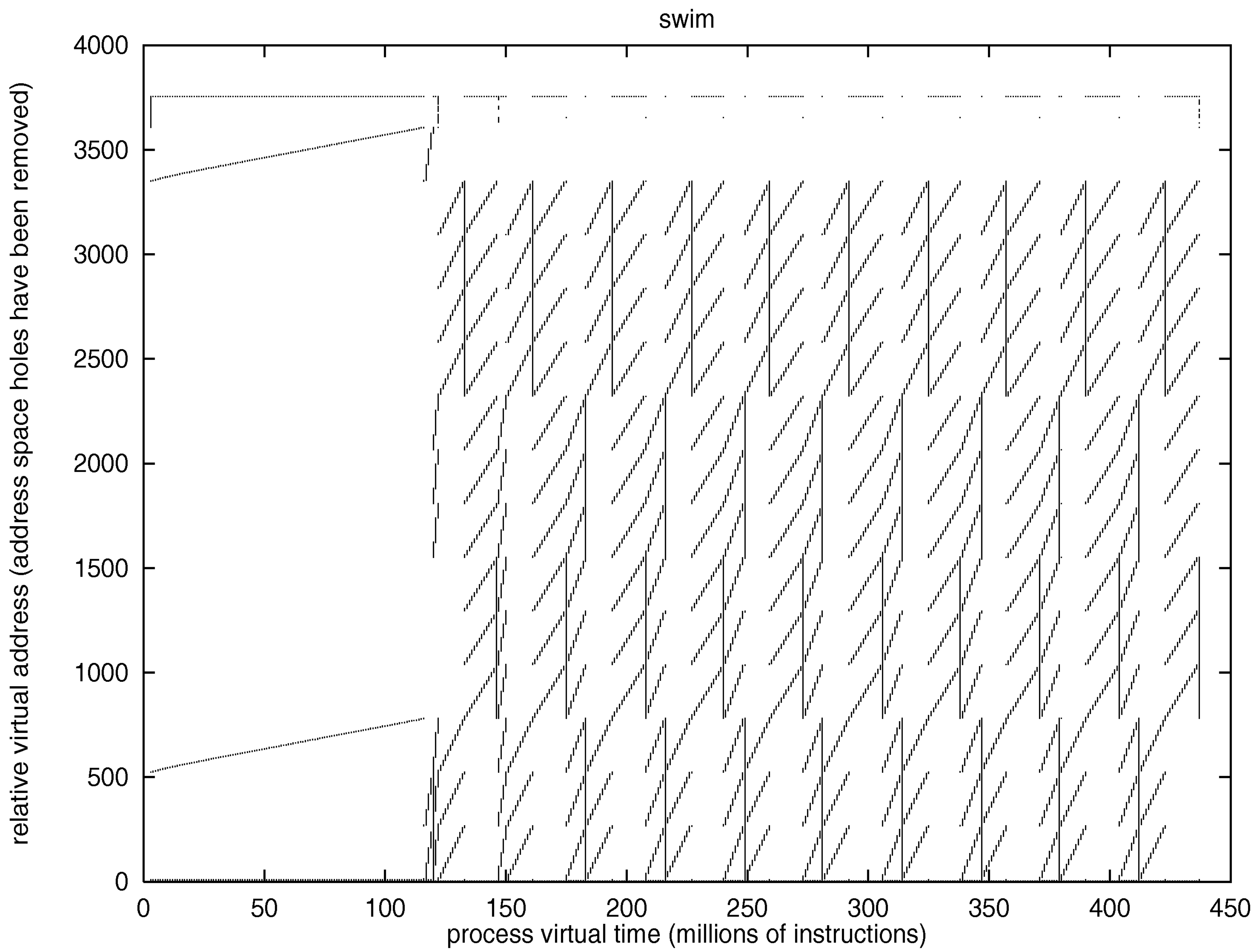

gcc-cccp
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
100
200
300
400
500
600
700
800
900
1000
0
200
400
600
800
1000
1200
1400

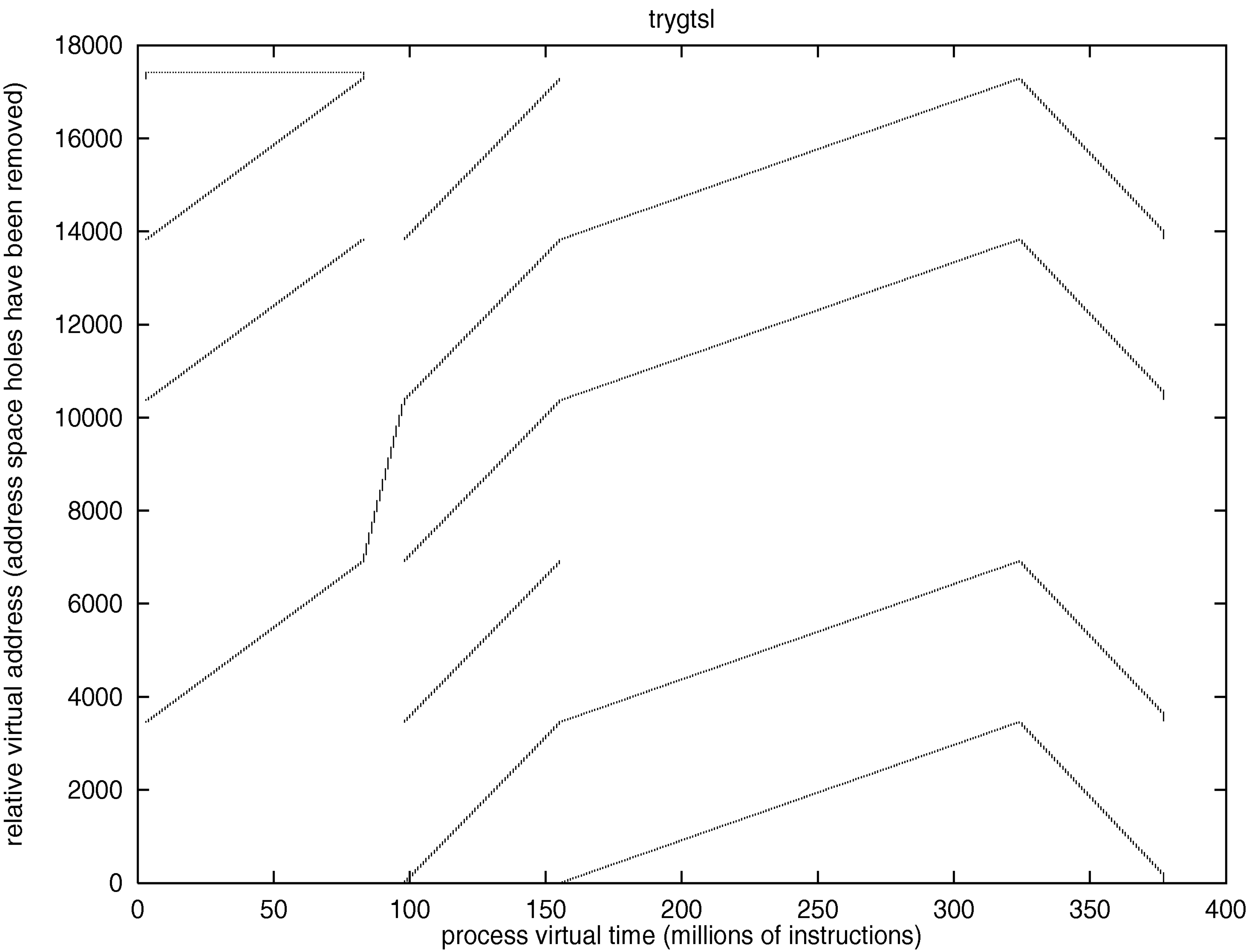

gnuplot
relative virtual address (address space holes have been removed)
0
2000
4000
6000
8000
10000
12000
14000
16000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
process virtual time (millions of instructions)

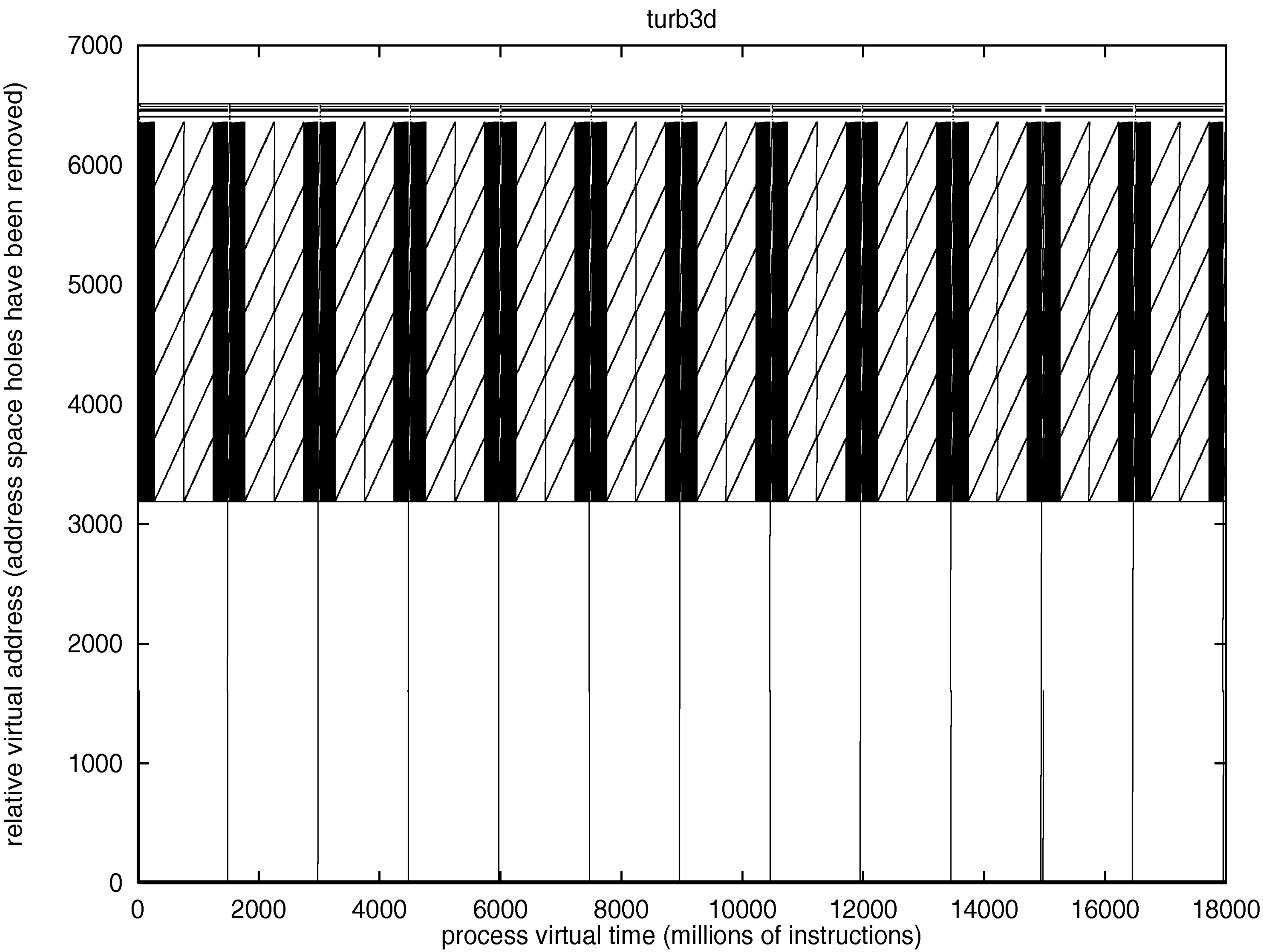

m88ksim
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
0
2000
4000
6000
8000
10000
12000

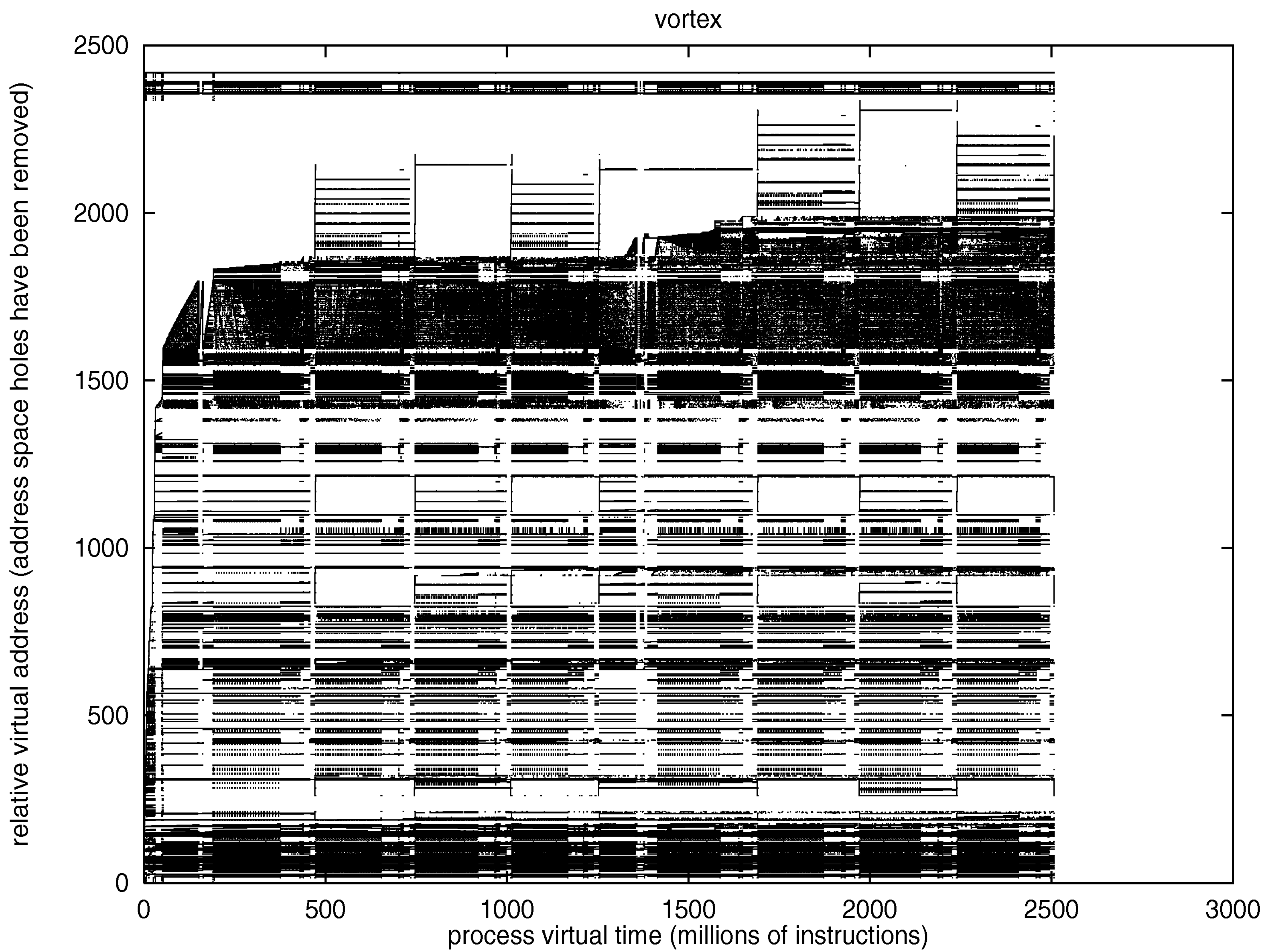

mcslock1
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
200
400
600
800
1000
1200

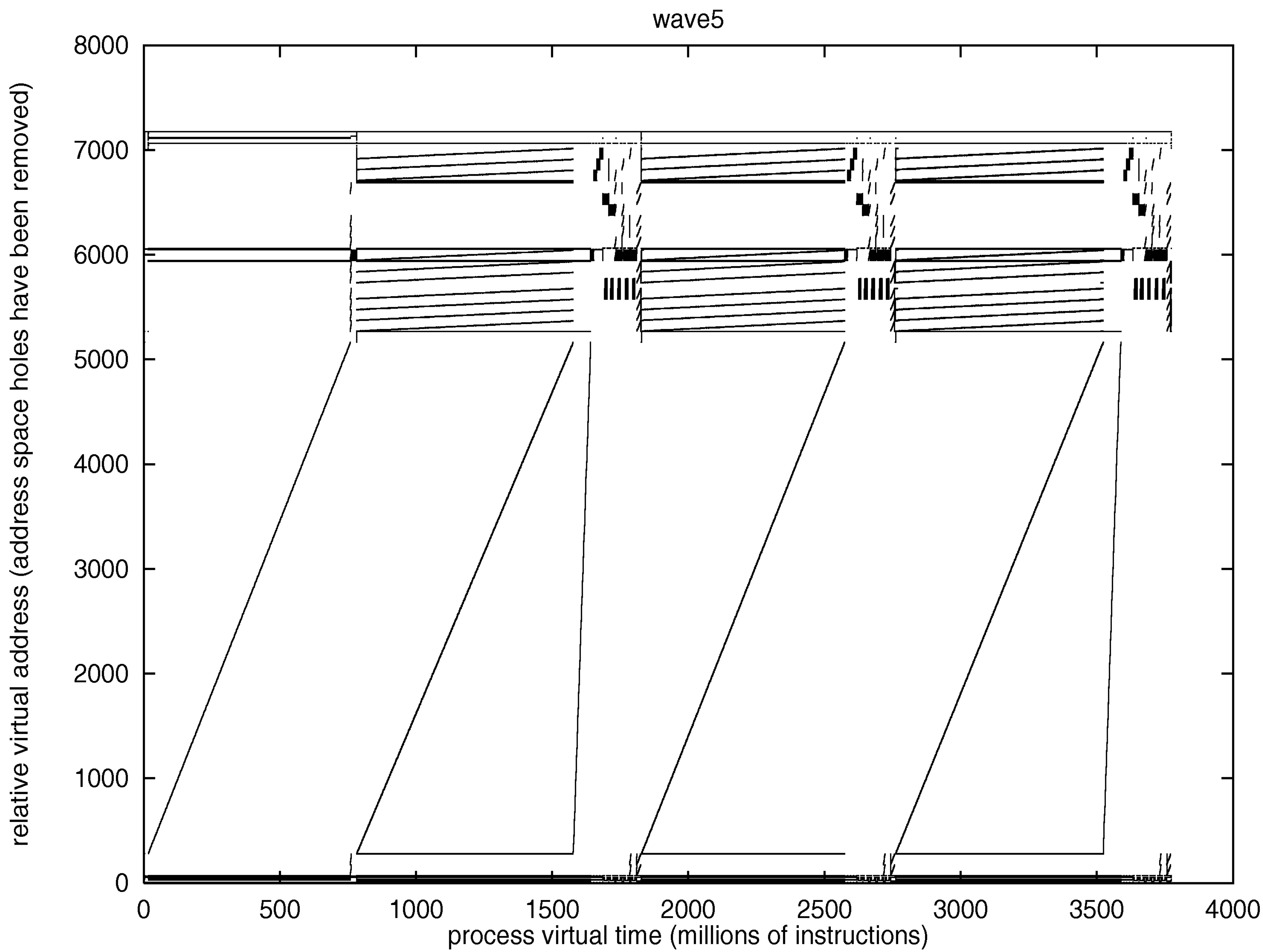

perl
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000


swim
relative virtual address (address space holes have been removed)
0
500
1000
1500
2000
2500
3000
3500
4000
0
50
100
150
200
250
300
350
400
450
process virtual time (millions of instructions)


trygtsl
relative virtual address (address space holes have been removed)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50
100
150
200
250
300
350
400
process virtual time (millions of instructions)


turb3d
relative virtual address (address space holes have been removed)
0
1000
2000
3000
4000
5000
6000
7000
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
process virtual time (millions of instructions)


vortex
relative virtual address (address space holes have been removed)
process virtual time (millions of instructions)
0
500
1000
1500
2000
2500
3000



wave5
relative virtual address (address space holes have been removed)
8000
7000
6000
5000
4000
3000
2000
1000
0
0
500
1000
1500
2000
2500
3000
3500
4000
process virtual time (millions of instructions)

pattern). These patterns might be exploited for techniques such as prefetching, but we are not concerned with that at present. The third application category, consisting of the rest of the applications, show clearly exploitable reference patterns. Ranges of address space are traversed in the same pattern repeatedly. These applications seem to be array-based, though some of them are written in C (fgm and gnuplot).

These behaviors remind us of the comments of Rob Pike: "there are only three data structures: linked lists, hash tables, and arrays." [27] It does seem to us that it is true in practice: the applications exhibiting regular reference patterns are array-based; vortex, perl and coral are either using hash tables or traversing a tree structure (judging from the way the address space grows); gcc seems to use linked lists heavily, and we have no idea how and why m88ksim behaves the way it does. From the memory system's point of view, array-based application would be the easiest to deal with, while hash tables are the hardest to handle.

We have not yet studied in detail the source codes of these applications to see why they exhibit such memory behavior.

Though the graphs show intuitively the application's page reference behavior, it is hard to predict replacement algorithm's performance from them. Especially, it is unclear how LRU performs compared to OPT for the first category of applications, although it does seem that good replacement policies cannot improve performance much because the applications simply need lots of memory. The next section gives the performance of LRU and OPT algorithms on these applications.

3 Performance of Replacement Algorithms

Although the traces we collected are only in intervals of page references, we can still simulate various replacement algorithms accurately for large ranges of memory sizes. Below we describe how our simulation works, and report the results on the performance of LRU and OPT.

3.1 Simulating Replacement Algorithms

We can think of 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, if the memory is big enough such that there is always an *IDLE* page when a replacement is needed, the algorithm would choose among the *IDLE* pages.

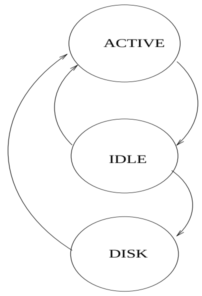


Figure 1: Page states and transitions that can occur.

For example, LRU would choose the *IDLE* page with the earliest OUT record. This is indeed the least-recently-used page since all active pages must have been accessed in the period following that earliest OUT record. Thus, the replacement decision is the same as that would be made if a complete reference trace is available. OPT would also make the same replacement decision as with a complete trace, because it would choose the *IDLE* page with the IN record furthest into the future. Since idle pages would not be referenced in the next interval, whereas active pages will be referenced then, the page chosen would be what OPT would choose given complete reference traces.

The page replacement policy simulator keep each page in one of three possible states. In addition to the states *ACTIVE* and *IDLE* described above, a third state, *DISK*, is introduce to mark pages that are considered to be residing on secondary storage. That is, *ACTIVE* pages are the pages that are between IN and OUT records, and *IDLE* pages are pages that are between OUT and IN records and are still retained in main memory, while *DISK* pages are those that are between OUT and IN records and reside on secondary storage. All pages are considered to be initially in *DISK* state. The primary responsibility of the simulated replacement algorithm is to choose from among the *IDLE* pages a page to replace when the number of *ACTIVE* pages plus the number of *IDLE* pages exceeds the specified memory size. This implies that the simulated memory size cannot be allowed to be less than the maximum obtained value (for a given program's trace) for the number of *ACTIVE* pages. (If that happened then there would be no *IDLE* pages from which to choose a replacement—*ACTIVE* pages *cannot* be replaced. In practice, the minimum memory size is usually quite low, several orders of magnitude less than the number of pages used.)

Pages move among the states as shown by the finite state diagram in Figure 1. Though there are six possible transitions among the three states, only four of them actually occur:

1. *DISK* ⇒ *ACTIVE*: this is triggered by an *IN* record from a trace file; the page in question has either not been previously referenced, or was paged out at some time in the past; it any case it is now marked as being in active use.

2. *IDLE* ⇒ *ACTIVE*: also triggered by an *IN* record; here, the page is in memory but before this transition had not been referenced recently by the application, so had fallen into *IDLE* state.

3. *ACTIVE* ⇒ *IDLE*: triggered by an *OUT* record; a page that had been in active use has not been referenced for at least a million instructions. The page is now a candidate for replacement.

4. *IDLE* ⇒ *DISK*: in contrast to the above three transitions, this transition is *not* triggered by a record read from the trace file. Instead, this transition is initiated by a replacement policy when it has chosen to swap out the page in question. Replacement policies must choose an *IDLE* page to replace when free memory becomes full.

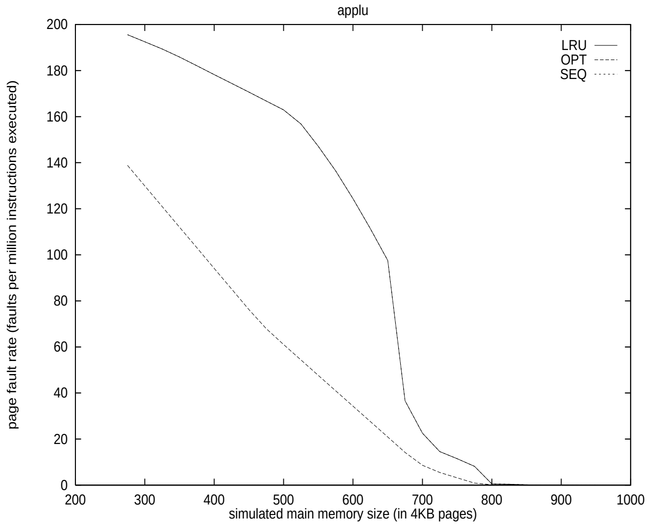
The remaining two transitions between states cannot occur:

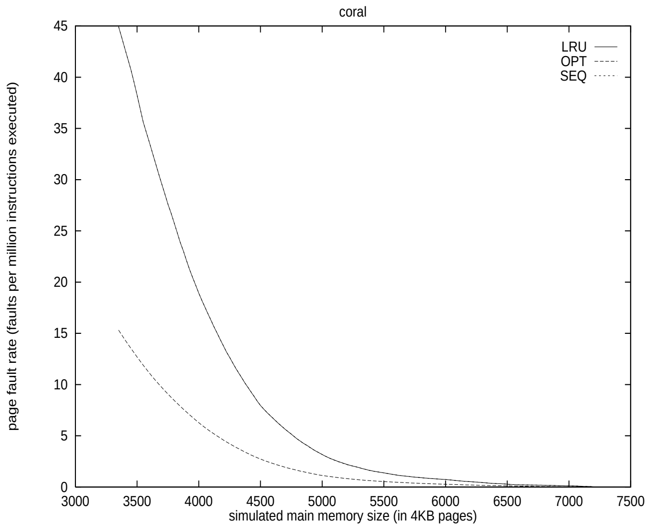
1. *DISK* ⇒ *IDLE*: pages on disk are not brought into memory unless they are needed—in which case they go into *ACTIVE* state, not *IDLE*. This is standard demand paging.

2. *ACTIVE* ⇒ *DISK*: *ACTIVE* pages may not be replaced because they simply *must* be kept in memory since they are being referenced by the application. Only *IDLE* pages may be replaced.

LRU simulation is straightforward. Because only pages in *IDLE* state are replacement candidates, and because every transition into *IDLE* state is logged explicitly in the trace file by an *OUT* record, LRU simulation becomes a simple matter of enqueueing each newly *IDLE* page; when a page is to be freed, the head of the queue is removed and is the replaced page (it enters *DISK* state).

OPT is slightly more complex since it requires future knowledge: the page to be replaced at any given time is the one that will be referenced the farthest into the future. Thus we need to know, for each page, when it will be next referenced. The solution is to preprocess the trace file so that it includes with each *IN* record the time of the *next IN* record (or +inf if the present *IN* record is the final *IN* such record for the page in question). Pages in *IDLE* class are sorted by their *next-IN* time; the replacement candidate is the page with the largest *next-IN* value.

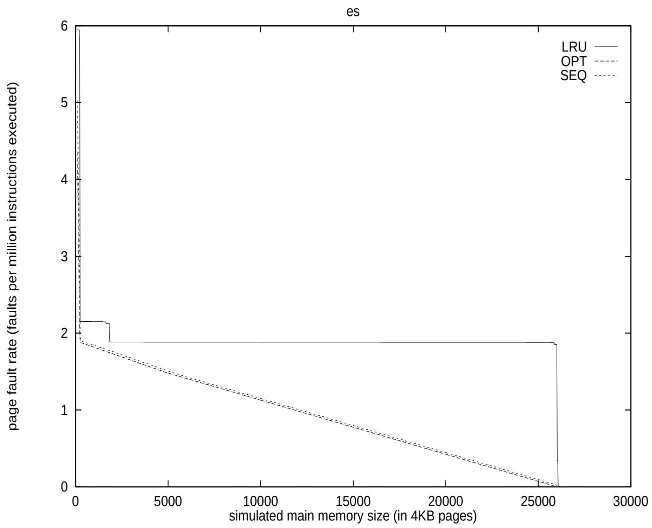


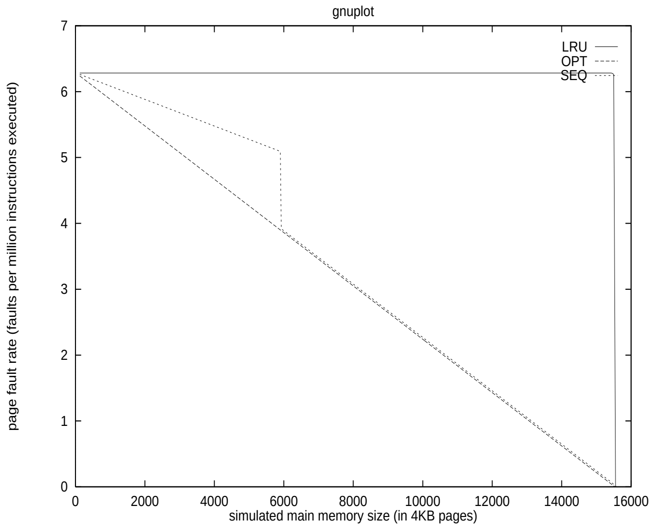


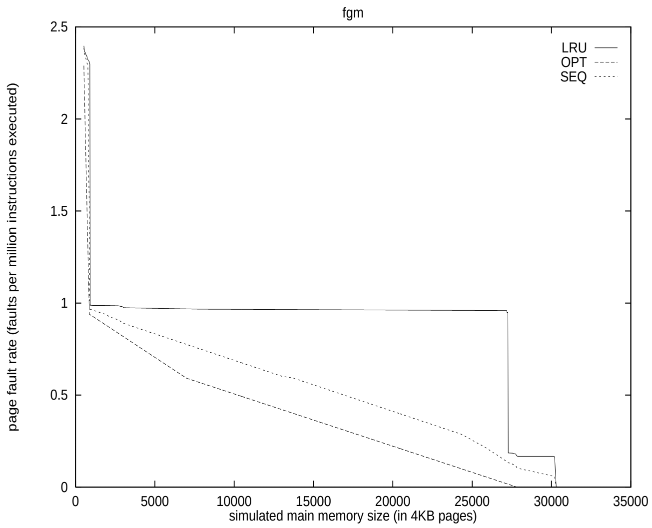
3.2 Fault Curves

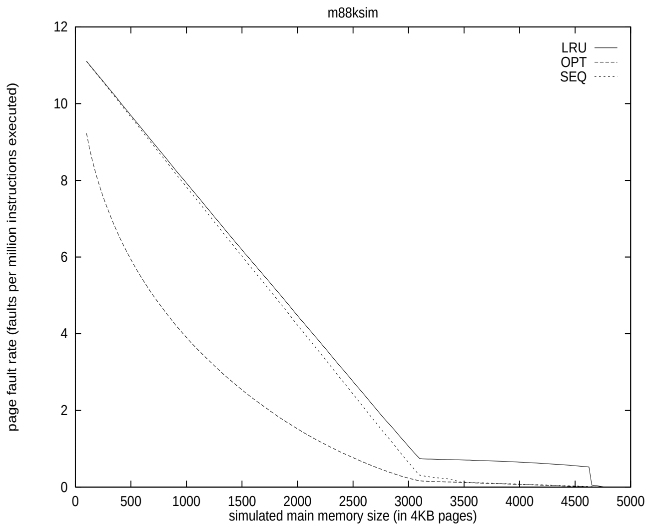
The figures below show the page fault rate for each application as its memory spans the range from the minimum (in simulation) value to the total number of pages the application uses. Rate is expressed as page faults per million application instructions executed. The page faults considered when calculating the fault rate do not include compulsory faults (minor faults), which would be the same for all replacement algorithms. The three curves in the graph are LRU, OPT, and the new algorithm SEQ that we will describe in the next section.

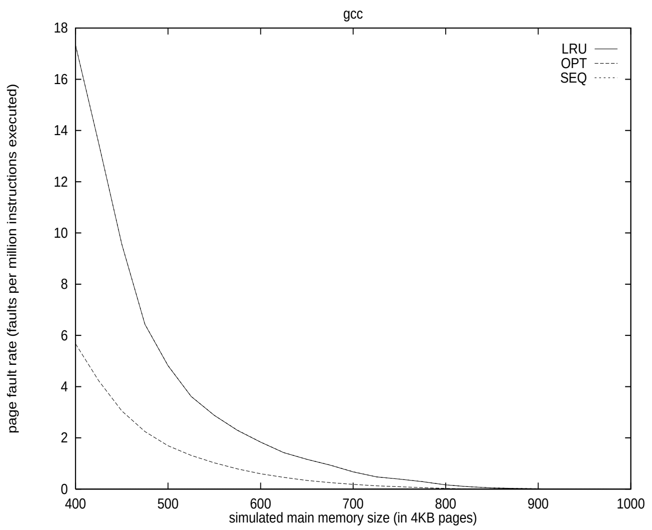
The figures show that for the first category and second category of applications, which are memory intensive and do not have strong patterns, LRU performs similar to as OPT, though LRU suffers about as many page faults on average. For these application classes, the page faults under LRU drops continously when more memory is available; the rate of improve-

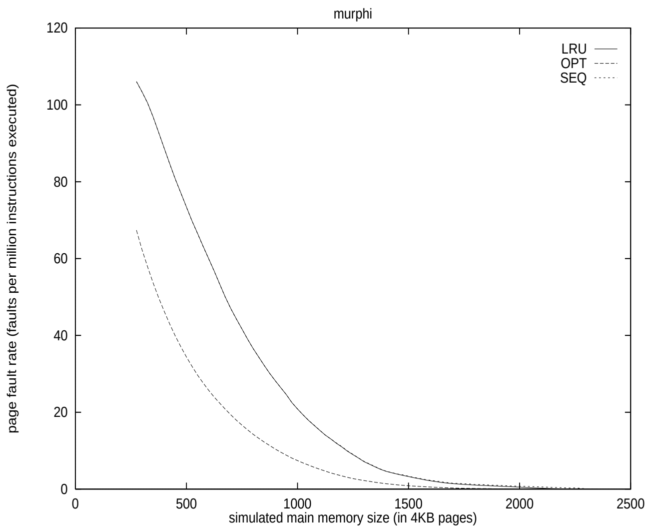

es
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

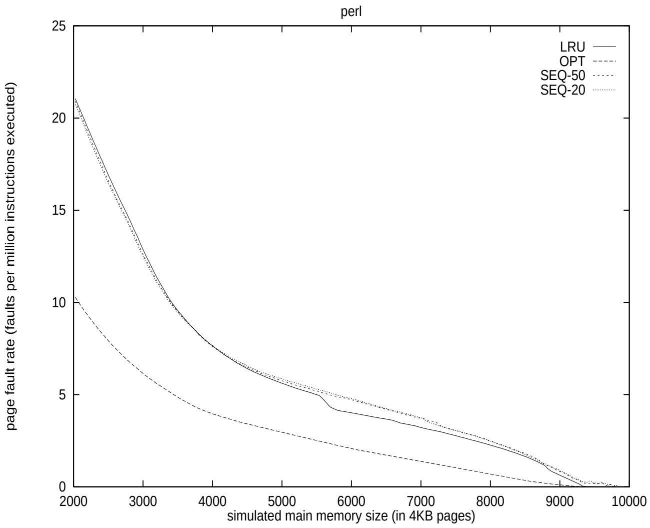

gnuplot
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

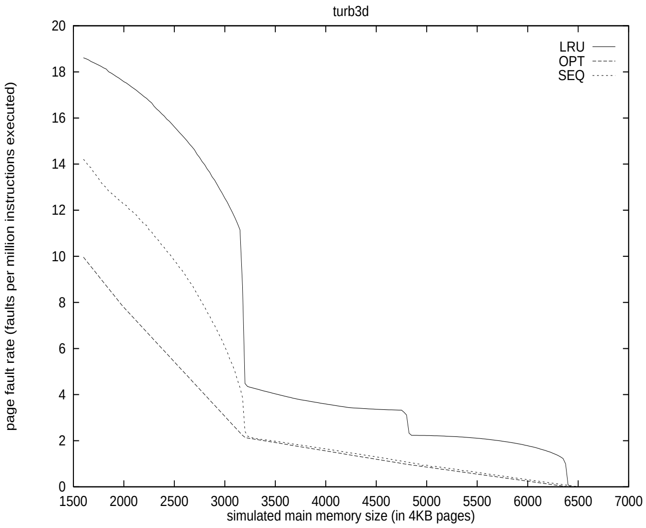

fgm
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

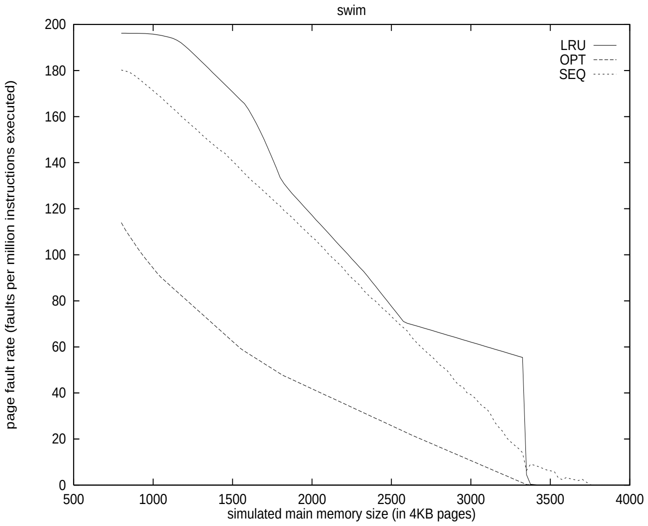

m88ksim
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

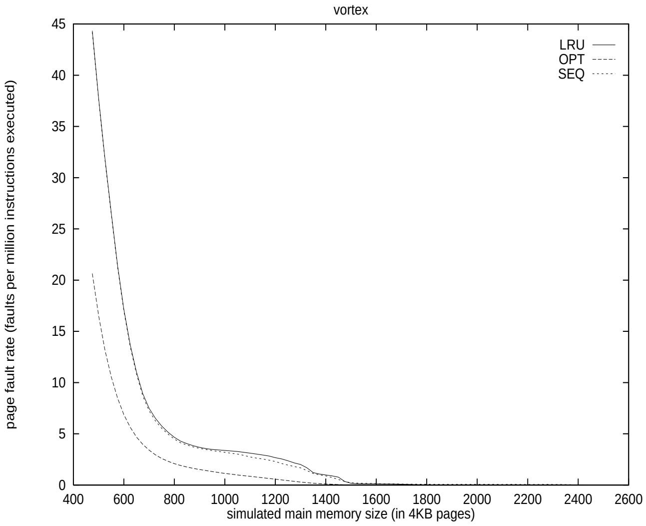

gcc
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

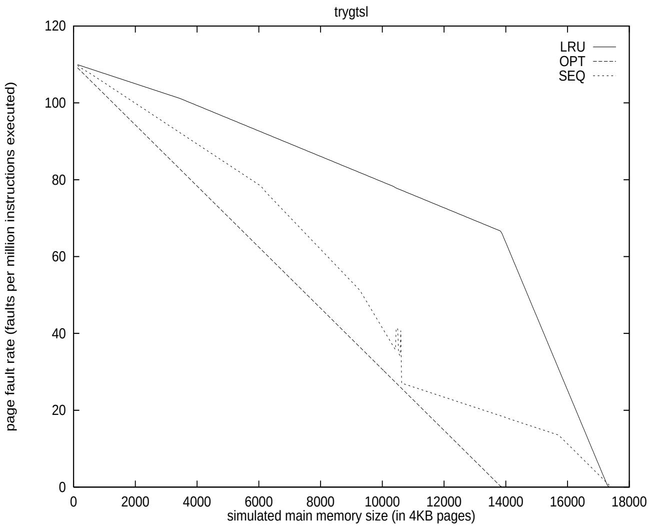

murphi
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

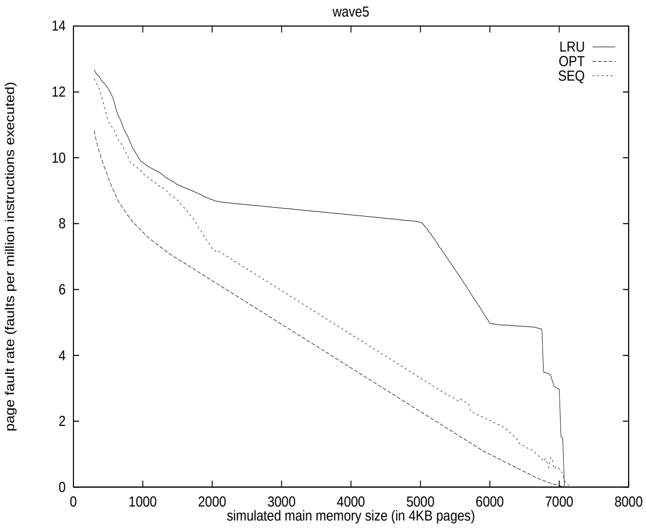

perl
LRU
OPT
SEQ-50
SEQ-20
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)


turb3d
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)


swim
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)


vortex
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)


trygtsl
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)


wave5
LRU
OPT
SEQ
page fault rate (faults per million instructions executed)
simulated main memory size (in 4KB pages)

ment is similar to that under OPT. The improvement tends to be superlinear; that is, doubling the amount of memory more than halves page fault rage, at least for the memory sizes less than roughly half of the total memory needed by the program; the improvement slows down after this point. Note that there is no clear "working-set" for these applications.

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

In the next section, we introduce a new algorithm, SEQ, would correspond to the middle curve in the graphs.

4 New Algorithm: SEQ

The basic idea of the SEQ replacement algorithm is to detect long sequences of page faults and apply MRU replacement on such sequences. If no sequences are detected, SEQ performs LRU replacement. The goal is to avoid the common problem of LRU flooding, which occurs when a program iterates over an address space range larger than available main memory. LRU pages out such regions in the order in which the pages are accessed; iterations following the first then must page in the entire sequence, clearly an undesirable situation. By performing MRU on such ranges, as much as possible will remain memory resident; the remainder will have to be paged in and out as necessary.

SEQ has two components. The first (*seq_handlefault*) executes whenever a page fault occurs; its job is to detect sequentially-accessed memory regions. The second component (*seq_findreplace*) is invoked to find a page to replace.

SEQ detects sequences by associating page faults of spatially adjacent virtual addresses. A list of sequences of past page faults is maintained, in the form of <*low_end, high_end, dir*>, indicating that a sequence of faults from *low_end* virtual address to *high_end* virtual address, going in the direction of *dir*, has been observed. The sequence can be just one page fault, with *low_end* = *high_end*. When a new page fault occurs, faults sequences neighboring the new fault are examined. If the new page fault extends an existing sequence and the direction of the extension is the direction (*dir*) the sequence was growing, then the sequence's *low_end* or *high_end* is modified. (The end of the sequence in the direction of growth *dir* is referred to as the *head* of the sequence.) If the new page fault falls in the middle of some existing sequence, then the sequence is split into two sequences, one going from the original *low_end* (when *dir* = *up*) or *high_end* (when *dir* = *down*) to the current fault, and one sequence consisting of simply the new fault. If the new page fault does not extend any existing sequence nor overlap the address range of any sequence, then a new sequence consisting of just the single page fault is built.

To choose a page for replacement, SEQ examines all sequences of length greater than or equal to L (currently 20.) If no sequences of this length exist, the default LRU replacement is used. If suitable sequences are present, then SEQ examines the candidate sequences based on the order of last fault activity. SEQ tries to replace a page from the sequence that faulted recently. The particular heuristic we use is to sort sequences based on the time of the twentieth from most recent fault (for each sequence). For each sequence considered, SEQ examines each page starting twenty pages behind the sequence head, to avoid replacing pages that are still in use. Furthermore, a page is only replaceable if it is in *IDLE* state, due to the constraints of our interval-based traces. Though the OUT record information would not be available in practice, a real implementation can effectively estimate whether a page is still in use by using reference bits and observing the usage time of past pages in the sequence.

The order in which the sequences are examined, i.e. most recent fault activity, was chosen for the following reason. Consider a program making multiple passes over a region of memory larger than physical memory. On the first pass, a single long sequence will be constructed, as pages are demand paged into the address space (from heap or uninitialized data segment). When the region reaches main memory size, page replacement will be initiated. SEQ will observe the single large sequence and replace backwards from its head. The entire address range will eventually be accessed, with the pages near the head being replaced soon after they are faulted into memory (the "bottom" part of the sequence stays in memory). At the beginning of the second iteration over the address range, the program will find the bottom portion of the sequence entirely memory resident, and will not incur faults until it reaches the midway point at which replacement had earlier commenced. A fault will occur at the first page that had been replaced, and the existing single, large sequence will be split. As program execution continues, the upper sequence will grow, eventually reaching length L. At this point there are two sequences long enough to supply pages for replacement. Which should be chosen? We want to avoid choosing the bottom sequence since that covers the address space region that we want to keep memory resident; we want instead to continue doing MRU replacement on the upper portion. Therefore, we choose the sequence with recent fault activity.

We choose the twentieth most recent fault for the reason that we want to be sure that sequences are the result of genuine sequential behavior. A sequence that happens to be long enough, but whose constituent faults occurred long in the past, is less likely to have resulted from actual sequential program behavior. (Such a sequence may have resulted, for example, from a dynamically extending heap.)

SEQ's sequence detection algorithm is designed to avoid identifying sequences that are not always accessed sequentially. For example, if an application first scans through an address range then later accesses it randomly, the algorithm would split the sequences repeatedly till they would no longer be large enough for SEQ replacement. Thus, for application that do not have sequential behavior but scans through an address range occasionally, SEQ uses LRU replacement almost all of the time and thus performs virtually identically to LRU. The restraint on splitting sequences with faults in the middle is important because non-sequential programs whose heaps grow dynamically have sequential fault patterns (a page at a time in the direction of heap growth) but should not have MRU replacement done on them.

For the applications that are memory intensive and do not have visible patterns (i.e. the first and second category of applications described in Section 2.2), SEQ performs the same as LRU. For most of the applications in the third category, SEQ improves upon LRU significantly and approaches, quite closely in some cases, OPT's performance, reducing page faults continuously as memory size increases. Thus, SEQ performs much better than LRU for applications with regular patterns, and almost never performs worse for other applications. We believe that SEQ is a promising algorithm to be used in virtual memory systems.

The current heuristics in building sequences and choosing replacements from them are by no means the best. We are still working on refining them.

The general technique we are applying here is detecting patterns through observing page faults, and using replacement algorithms tailored to the patterns. SEQ is a simple algorithm that applies this technique—it merely detects repeated sequential patterns. More sophisticated algorithms can be designed and will improve the memory system performance for applications like applu and wave5 where SEQ currently falls back to LRU. This remains part of our future work.

5 Comparison of Global Replacement Algorithms

Our discussion so far has only looked at the performance of various replacement policies for single processes. In practice, multiple processes run at the same time and compete for memory. One approach to handle the multi-process memory management issue is to use global replacement algorithms, i.e. 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. In this section, we compare the performance of LRU and SEQ as global replacement algorithm, and compare the performance of *concurrent* execution of applications versus *serial* execution.

To investigate these issues, we built a simplified simulator of a multiprocess time-sharing system. Our simulator reads multiple application traces, and issues their records in a multiplexed way. A scheduler chooses which trace to issue the next record from. The scheduler uses round-robin time-slice 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 crude estimate of CPU speed—in our experiments the quantum is 2 million instructions (or 40ms on a 50M instruction per second machine), and page fault service time is 400,000 instructions (or 8ms on a 50M instruction per second machine).

In our simulation, the SEQ algorithm is modified to detect sequences only among pages belonging to the same process. That is, a process ID is attached to each page faults, and only page faults with the same process ID are associated for sequence detection.

We picked five combinations each of two applications. The combinations are chosen to have a variety of mixes of application behavior and relative memory needs. They are: applu and gcc, gnuplot and vortex, swim and trygtsl, es and fgm, coral and wave5. For each combination, we measure the page faults and the simulated execution time of the concurrent execution of the applications, under both global LRU and global SEQ, for a range of memory sizes. Again, the page faults reported here do not include initial (compulsory) faults. Our simulated execution time assume a fixed 400K instruction cost for page faults.

In general, concurrent executions of applications can either perform better or worse than serial executions of applications. Serial execution (i.e. one applications runs after another one finishes) let each application take advantage of the total amount of memory, and thus reduce their page faults. However, serial execution permits no overlap of computation and page faults I/O (unless prefetching is done; we ignore that in the present study). Concurrent execution has the advantage of overlapping one application's computation with another application's page fault I/O. However, in concurrent execution all applications compete for memory, so each has less than the total memory size, possibly leading to an increase in total page faults. This is essentially the load control problem describing in Denning's Working Set paper [12].

In our study, we are curious as to how concurrent executions of these applications compare to serial execution, that is, whether multiprocessing is a win or loss under the different replacement policies. Thus, in each figure, we also plot the total page faults and total execution time of the serial execution of the two applications, and compare them with those of the concurrent execution.

Figures 2 to 6 show the results.

The first combination, applu+gcc, shows no difference between the LRU algorithm and the SEQ algorithm (as in the case for individual applications). The serial execution is slightly better as a result of fewer page faults. However, the difference in execution time is not as big as that in page faults, showing some amount of I/O and compute overlap. Thus, concurrent execution in this case does not hurt performance much, but did not improve it either.

The next pair are coral and wave5, a mix of one program with complex memory reference patterns and one with a more regular pattern. In this run, SEQ performs better than LRU in both the concurrent execution and the serial execution case. The concurrent execution reduces the execution time significantly for low memory sizes, though the improvement disappear as the memory increases. This is mainly because for a range of memory ($\tilde{4}$500 pages to $\tilde{7}$000 pages), concurrent execution results in more page faults than does giving the total memory to the applications one at a time.

The third pair are es and fgm, both sequential programs for which SEQ performs well in the single process case. The plateau behavior for LRU is evident in both the concurrent and serial executions. Thus, global SEQ performs significantly better than global LRU for most of the memory sizes. In addition, concurrent execution performs significantly better than serial execution by overlapping the computation of one sequential program with the page faults of another nearly perfectly.

The next multiprocess pair, gnuplot and vortex, is dominated by the gnuplot run. Although vortex uses over 2400 pages, its memory demand drops off after about 800 pages, and from its space-time plot, it looks like those 800 pages are used actively enough to remain essentially permanently resident during vortex's execution. With vortex out of the picture, gnuplot's LRU flooding is obvious. More peculiar are the bumps in the multiprocess SEQ graph. Presently we are investigating the SEQ interactions which cause this. Nevertheless, SEQ outperforms both LRU versions by a heavy margin.

Finally we have swim and trygtsl. Both programs exhibit sequential behavior, with LRU performing similarly for concurrent and serial cases (similar to gnuplot/vortex above). Global SEQ performs slight worse than global LRU for the high end memory sizes, indicating that our SEQ algorithm still need improvement.

To summarize, in multiprocess systems, global SEQ performs significantly better than global LRU whenever sequential applications are run, and it performs similar to global LRU when no sequential applications are run, similar to the situation for single processes. Thus, our preliminary simulation results show that SEQ is also a promising algorithm for global replacement. In addition, under global SEQ, concurrent execution of applications do not perform significantly worse than serial executions most of the time, and can perform significantly better for some application mixes. Thus, the algorithms control the thrashing in concurrent execution well.

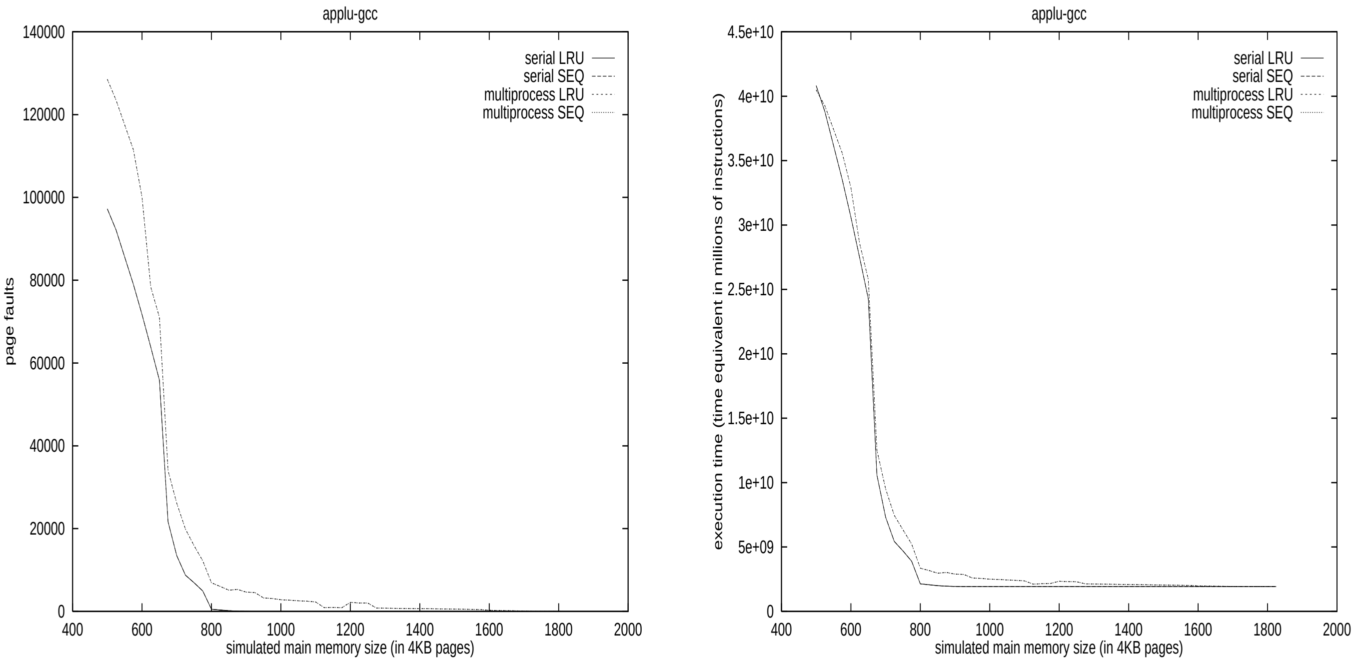


Figure 2: Total page faults (left figure) and simulated execution time (right figure) for multi-application run mix applu and gcc. Curves are serial and concurrent executions under LRU and SEQ policies.

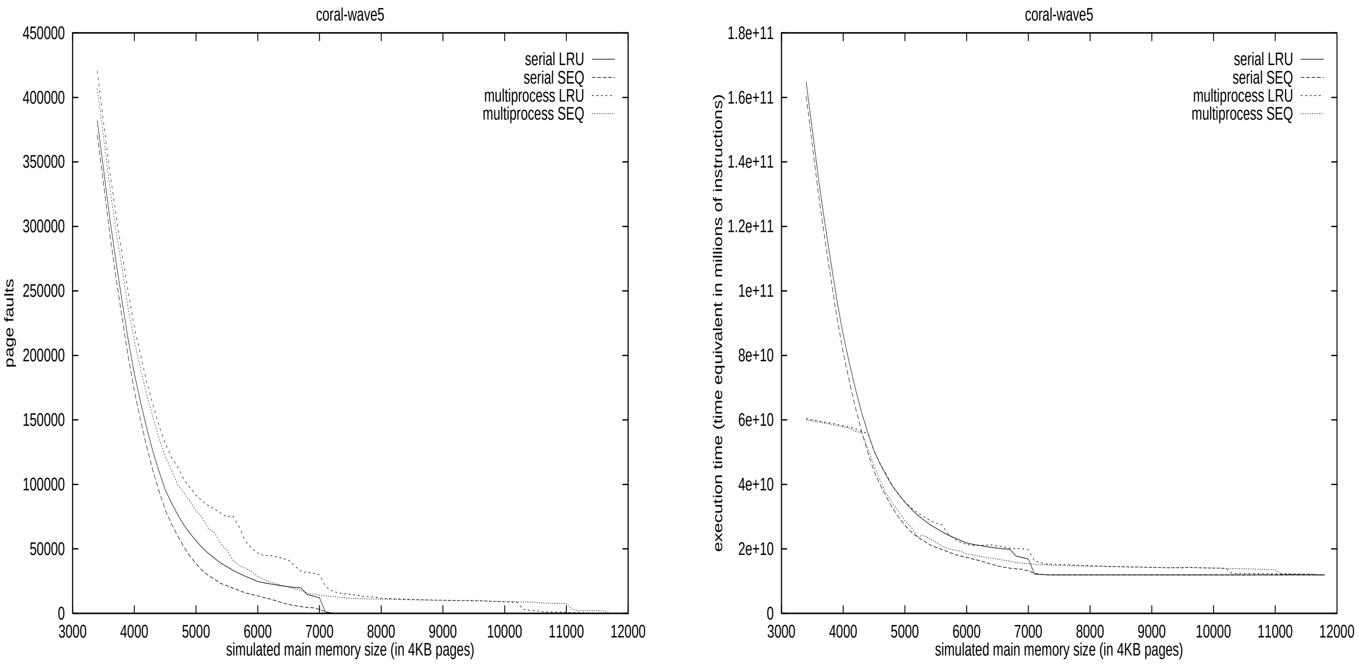


Figure 3: Total page faults (left figure) and simulated execution time (right figure) for multi-application run mix coral and wave5. Curves are serial and concurrent executions under LRU and SEQ policies.

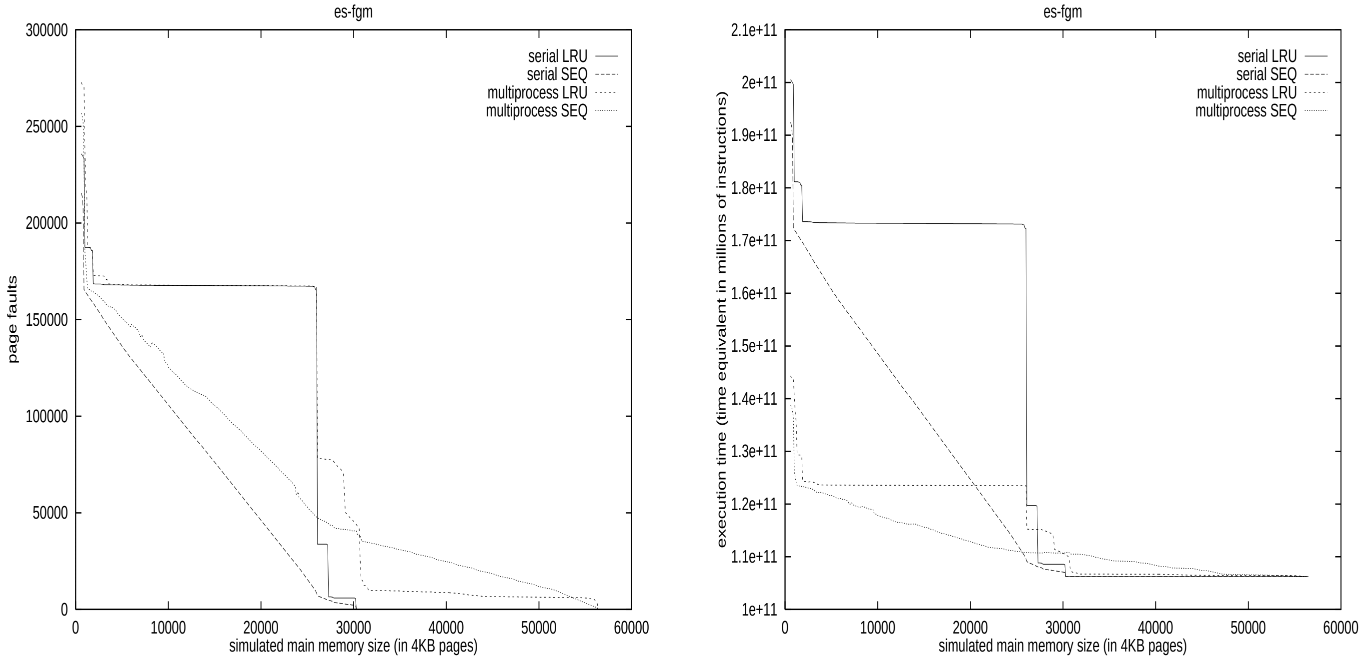


Figure 4: Total page faults (left figure) and simulated execution time (right figure) for multi-application run mix es and fgm. Curves are serial and concurrent executions under LRU and SEQ policies.

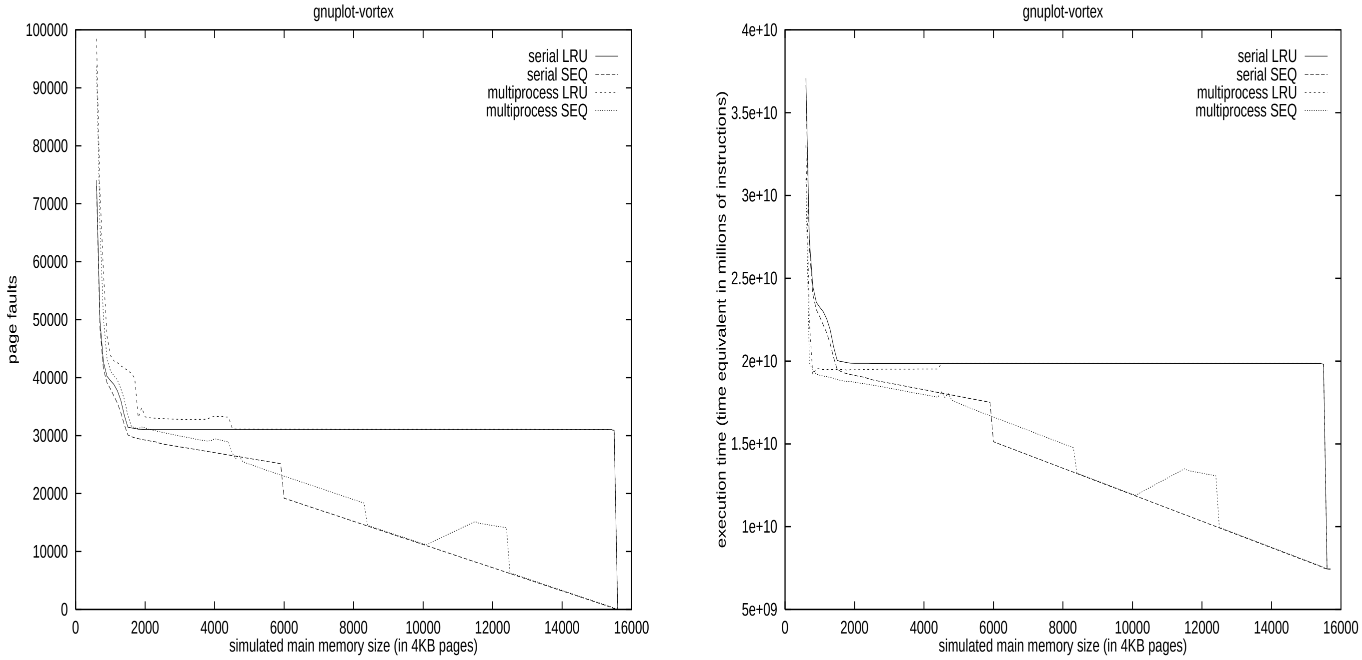


Figure 5: Total page faults (left figure) and simulated execution time (right figure) for multi-application run mix gnuplot and vortex. Curves are serial and concurrent executions under LRU and SEQ policies.

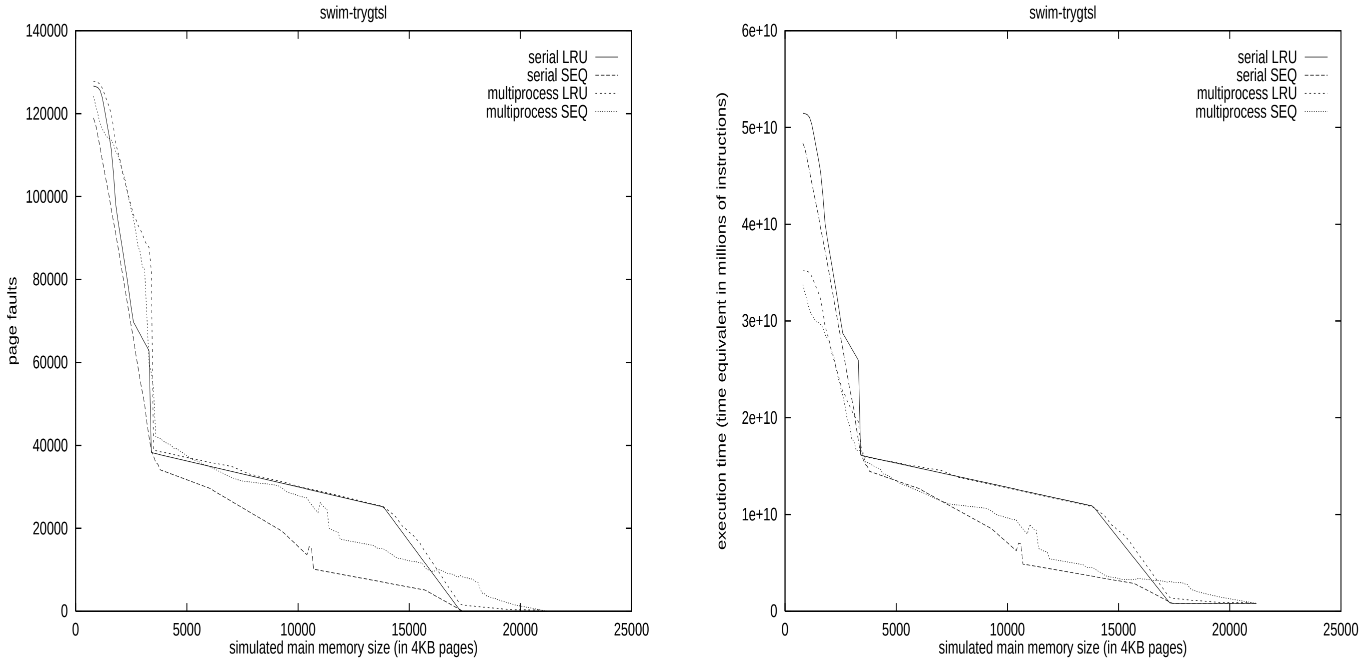


Figure 6: Total page faults (left figure) and simulated execution time (right figure) for multi-application run mix swim and trygtsl. Curves are serial and concurrent executions under LRU and SEQ policies.

6 Related Work

Operating systems researchers have investigated the memory management problem for over thirty years. Belady's paper in 1966 [3] articulated the primary purpose of virtual memory paging: manage physical memory automatically so that programmers do not have to divide a program into separate portions (overlays) and bring them into memory under program control. The paper [3] also introduced the optimal offline replacement algorithm, which is the OPT algorithm used in this study. Since then there have been extensive theoretical studies on the page replacement algorithms; a good survey can be found in [16].

Among these theoretical studies, of particular interests to our study are the ones on program behavior modelling and optimal online algorithms for each model. The models include independent reference [1], LRU stack [30], working set [11], access graphs [5], Markov model [21]. For each of these models optimal online algorithms are found, for example, LRU for LRU-stack model [16], access-graph algorithm [18], and Markov paging algorithm [21]. Our study suggests that the access-graph model might be the appropriate model for the regular-pattern applications. Indeed, the SEQ algorithm can be viewed as an attempt to apply the access-graph algorithms for a simple graph: sequences of pages. Part of our future work is to investigate applying the general access-graph algorithms for more complicated patterns in applications.

Along with the theoretical studies there have been extensive experimental studies. A good survey can be found in [12, 32]. Except a few early studies [22], most experimental studies focused on efficient implementation of LRU page replacement algorithms in practice. A number of approximation algorithms, FIFO with second chance [4], CLOCK algorithms [3], etc. have been proposed and implemented in various systems [8, 24, 31]. The studies show that the performance of the approximation algorithms is similar to that of LRU. Most operating systems today uses CLOCK or a variant of it [10, 19, 2]. Though not reported in this paper, we performed simulations of FIFO and a simple version of CLOCK algorithm on our traces. Our findings are similar to the early studies: FIFO generally performs worse than LRU, and CLOCK performs similar to LRU across the applications.

Our observations of program memory behavior arrive at different conclusions from the early research results in this area, such as those described in Denning's excellent survey on memory management studies up to the 80's [12]. The biggest difference is that the applications we investigated do not have significant "phase-transition" behavior, and their reference patterns tend to be the same throughout execution (i.e. no phases). Also, there is also no clear "working-sets", and no clear "knees" in the fault-curve, contrary to what is observed in [12].

Recently there have been a number of studies on applications' memory reference behavior in the context of cache management in computer architecture. One study [6] (on out-of-chip bandwidth requirement) confirmed that there is a significant difference in cache miss ratios under LRU and under OPT replacement policies. Another study on the similar topic [26] plotted a space-time graph for some SPEC95 benchmarks at cache line level. Though their graphs are for a much shorter duration of the application's execution (e.g. 50K instructions), the graphs are similar to our graphs for the SPEC95 benchmarks. Finally, one study on large-scale multiprocessor architectures investigated the "working-set" and cache size issues for parallel scientific applications [29]. 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 [29] 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.

There are also sequence detection in hardware cache management. Mostly the detection is for prefetching, as in the case of stream buffers [33, 20]. Though we only talked about replacement policy in this paper, it is obvious that sequence detection can be used for prefetching as well. How to balance the prefetching and caching, though, is a complicated issue that needs further study [7].

Recent research projects on application-controlled kernels show the potentials of application-specific replacement policies [34, 17, 25, 23]. However, most of those studies focus on letting applications inform kernel of their access patterns, instead of having operating systems detect them at run-time (as in the SEQ algorithm). Our study shows that in fact, runtime automatic detection by kernel is feasible and can perform quite well.

7 Conclusion

Our study on application reference behavior and space-time graphs show that applications' memory reference behaviors vary significantly. There are at least three categories: very memory-intensive and no visible pattern, memory-intensive and small amount of pattern, and very regular patterns. Different applications should be treated differently. We found little evidence of "working-sets" in the applications we examined. Furthermore, applications' behavior tends to stay the same, instead of going through different phases of different patterns.

We found that though LRU performs similarly to OPT for the memory-intensive and pattern-less applications (except consistently about twice as bad), it performs much more poorly for regular-pattern applications. Thus, we propose a new replacement algorithm, SEQ, for the latter application class. SEQ works by detecting simple linear access patterns (sequential behavior) and performing 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 regular-pattern applications. Overall, SEQ performs much closer to OPT.

We also found that for multi-process systems, SEQ is also a good algorithm for global replacement. Comparison of global LRU and global SEQ show that global SEQ can effectively improve multi-application performances just as it improves single applications, and corrects the poor performance of global LRU as well. In addition, global SEQ effectively overlap computation and page fault I/Os for concurrent applications.

There are a number of limitations in our work. We have not investigated the application's source codes to explain all of the behavior shown in the space-time graphs. Our SEQ algorithm still needs refinements. Better pattern-based algorithm needs to be designed for applications like applu and wave5. Our simulation of multi-process case is quite crude. We have not considered prefetching, though it is an obvious technique to use with sequence detection as well. Finally, only a real implementation can show the performance benefits of SEQ in practice.

Our future work include addressing the above limitations. In addition, we will look at how the space-time behavior at page level correlates to that at cache line level, and how to apply techniques similar to SEQ to cache management.

References

[1] A.V. Aho, P.J. Denning, and J.D. Ullman. Principles of optimal page replacement. In *Journal of ACM, vol. 18*, pages 80–93, January 1971.

[2] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall Software Series, 1986.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Helen Custer. *Inside Windows NT.* Microsoft Press, 1992.

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

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

[13] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proc. 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[14] Eric Ristad et. al. *untitled working paper.* October 1996.

[15] F. Fraenkle, M.D. Hill, and S. Kim. Solving microstructure electrostatics on a proposed parallel computer. *Computers chem. Engng*, 19(6/7):743–757, 1995.

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

[17] 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.

[18] 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.

[19] Michael K. Johnson. *The Kernel Hacker's Guide 0.7*. http://www.redhat.com:8080/HyperNews/get/khg.html, 1996.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

[28] Raghu Ramakrishnan, Divesh Srivastava, Praveen Seshadri, and S. Sudarshan. Implementation of the CORAL deductive database system. In *19 ACM SIGMOD Conf. on the Management of Data, Washington,DC*, May 1993.

[29] 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.

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

[31] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, Massachusetts, 4th edition, 1994.

[32] Alan J. Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

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

[34] 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.