ADAPTIVE PAGE REPLACEMENT

By

Gideon W. Glass

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

(Computer Science)

at the

UNIVERSITY OF WISCONSIN – MADISON

1998

ACCEPTED

Pei Cao (Assistant Professor of Computer Sciences) Date

Mary K. Vernon (Professor of Computer Sciences) Date

Abstract

Virtual memory is a ubiquitous abstraction in general purpose computers. Typical virtual memory implementations employ LRU or FIFO page replacement, which works satisfactorily for most programs but performs extremely poorly for applications that access sequentially large portions of their address spaces.

This thesis presents SEQ, new page replacement algorithm. SEQ augments traditional LRU replacement by adapting its behavior at runtime to perform MRU replacement on regions of memory it detects as being accessed sequentially. SEQ suppresses its MRU replacement if it discovers non-sequential references to previously detected sequential regions.

Eighteen benchmark programs from numerous application domains are introduced and studied in detail. Using trace-driven simulation, I demonstrate that SEQ performs significantly better than LRU—often within a few percent of the theoretically optimal offline algorithm—for many of these benchmarks. Furthermore, SEQ rarely performs poorer than LRU.

I then describe a kernel implementation of SEQ and present results in terms of application execution times and observed paging behavior. These results reinforce the simulation results: SEQ significantly improves—often by speedups of 50 to 100 percent or more—application performance for programs that LRU handles poorly; moreover, SEQ infrequently performs worse than LRU.

Acknowledgements

This thesis is the result of a yearlong collaboration with my advisor, Pei Cao. This thesis would not exist without her ideas, enthusiasm, financial support, and occasional nagging.

My thanks go to Pei, Mary Vernon, and Sridhar Gopal for their feedback on drafts of this thesis. I benefitted from discussions with Doug Burger, Andy Glew, Jim Goodman, Sridhar Gopal, Milo Martin, and Mary Vernon. Alan Cox, Steve Dirkse, Andy Glew, and Yoonho Park kindly suggested benchmarks.

I must thank my parents and grandparents for their high standards and constant encouragement over the years. This thesis would not be possible without them.

Finally, thanks to Premika for her patience.

Therefore we are recommending mechanisms that monitor the behavior of a computation, basing allocation decisions on currently observed characteristics and not on advisories from programmers or compilers.

Peter Denning, 1968

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Empirical Program Behavior 5

2.1 Applications 5

2.2 Trace methodology 9

2.3 Memory reference graphs 13

2.4 LRU Stack Graphs 25

2.5 Cumulative fault rates 33

2.6 Summary 37

3 SEQ: An Adaptive Replacement Algorithm 39

3.1 SEQ in Action 40

3.2 The SEQ Algorithm 45

3.2.1 Observing page faults 46

3.2.2 Choosing pages for replacement 49

3.3 Simulation results 51

3.4 Varying the SEQ Parameters 56

3.5 Throttling SEQ 64

3.5.1 The Reference Race 67

3.6 Summary 70

4 An Implementation of the SEQ Algorithm 71

4.1 Linux Memory Management 72

4.2 Adding SEQ to Linux 75

4.3 Additional modifications 76

4.4 Swap Prefetching 79

4.5 Results 80

4.5.1 Single-process Results 83

4.5.2 Multi-process Results 93

4.6 Quantifying SEQ Overhead 98

4.7 Summary 99

5 Related Work 100

6 Summary 104

Bibliography 106

Chapter 1

Introduction

Almost all general purpose computers over the last thirty years have provided the abstraction of virtual memory: an essentially unlimited supply of memory available to application programs.

Central to any virtual memory system are the policies that determine the contents of the virtual memory system's high-speed storage component (main memory). Under the control of these policies, data are admitted and ejected from main memory in fixed-size units—pages. In most operating systems, the policy regarding when pages may *enter* memory is that pages enter main memory when they are first accessed—this detection is simple and efficient with common memory management hardware. On the other hand, the *page replacement* policy—the policy that chooses pages for eviction—is typically implemented in software.

Operating systems typically implement a least-recently-used (LRU) page replacement policy, whereby the page chosen for eviction is the page not accessed for the longest period of time. In practice, the exact per-page time of last reference is usually not available, so most policies only approximate LRU. LRU replacement works well in practice, and has a reasonable theoretical underpinning (the Working-Set model [12]), and is simple to implement. However, when a single large program requires more main memory that is present, LRU often suffers poor performance.

A well-known case for which LRU page replacement performs poorly is so-called "sequential flooding," in which a program accesses memory in an iterative fashion, looping over a large region several times. When memory becomes full, LRU chooses the oldest pages for replacement—the pages at the beginning of the sequentially-accessed range. On subsequent iterations, then, the whole range must be brought back into memory. Programs that access one or more large memory regions repeatedly are quite likely to suffer poor performance under traditional LRU page replacement. Ideally, whatever fraction of the address space range that can fit into memory will remain in memory, and the rest of the address space will be paged in and out as necessary.

This thesis proposes and evaluates an *adaptive* page replacement policy called *SEQ*. SEQ adapts its replacement decisions based on observed sequential reference patterns. When SEQ detects sequentially-accessed address space regions, it performs Most-Recently-Used (MRU) page replacement, on the assumption that the program will access the sequentially-detected region again in a similar sequential fashion. SEQ reverts to LRU replacement when it detects non-sequential access to previously sequential regions.

One may reasonably question whether virtual memory performance is worth studying given the constantly dropping price of main memory (DRAM chips). Consider some essential economic and technical trends of the computer industry: on average, processor speed doubles every eighteen months; main memory (DRAM) capacity quadruples every three years; disk capacity increases sixty to eighty percent every year; and finally, disk speed increases only about seven to ten percent per year[20, 37].

With these trends in mind, consider a program that performs poorly under LRU page replacement. Assume that it is necessary to run the program with input that causes the program to exceed main memory capacity. Compared to three years in the past (relative to this hypothetical scenario), we have a program that on average requires about four times as much memory, assuming that the program's memory demand increases as more memory becomes available. However, we have a disk that is only 33 percent faster, assuming ten percent annual speed improvement. As we shall see in great detail later, when sequential flooding occurs, programs often behave as though they were running with essentially negligible amounts of main memory: on the first access to a page per iteration over some sequentially-accessed region, the page must be faulted in from disk. If our hypothetical application is such a program, then we are attempting to run a program four times larger with a swap disk 33 percent faster; if we are unfortunate enough to suffer LRU flooding, then the program spends in absolute terms compared to three years ago *three times* as much time waiting on disk I/O. Clearly we would like to avoid such situations.

Often problems like this can be solved by purchasing more memory. But buying more memory is not always an option; buying and installing memory requires money, time, and space inside the computer—often not available—to plug in the additional memory. Moreover, buying more memory is only a temporary solution if one's programs continue to grow and once again exceed main memory size. Improving memory management is a better solution; the SEQ algorithm proposed in this thesis is one step in that direction.

In this thesis, I first introduce a set of benchmark applications that will be studied in depth. I then describe the SEQ algorithm in detail, and then describe performance results obtained via trace-driven simulation. These results suggest that SEQ performs quite well for programs that LRU handles poorly. I then describe an implementation of SEQ done in the context of the Linux operating system kernel. Detailed performance results show that SEQ does in fact perform well in practice, obtaining in many cases double the performance of LRU. (Results also show that in a few cases, SEQ performs slightly worse than LRU.) Finally, I summarize related past and contemporary work regarding page replacement.

Chapter 2

Empirical Program Behavior

Improving virtual memory performance requires an understanding of applications' memory behavior. Particularly important are long-term memory reference trends that can only be observed over complete program executions. In this chapter, I introduce and describe the applications studied in the rest of this thesis. I then describe the method used to obtain memory reference traces, and finally present several types of empirical descriptions of the programs' memory access patterns.

A word of warning before we begin: the bulk of this chapter consists of numerous detailed plots. While the plots are certainly interesting in themselves, they contain quite a bit of detail. Much of the material in this chapter, in particular the raw memory access graphs in Section 2.3, can be used for reference while reading later chapters.

2.1 Applications

This thesis studies eighteen memory-intensive programs. The programs are taken from a variety of application domains, but they all share two common characteristics. First, all programs studied require several megabytes of memory (usually much more—exact quantities appear below). The second common characteristic of these applications is that they are can be described as *fixed-input*: the applications only receive input from command-line parameters or from configuration files read at startup time. (Some programs have their inputs hard-coded in the source code.) None of these applications involve interactive user interfaces. Measuring only fixed-input programs has the weakness that it ignores the important class of interactive applications. However, it is difficult to reproduce experiments that involve substantial user interaction. Moreover, fixed-input programs are undoubtedly common, and, significantly, fixed-input programs often require substantial amounts of memory to store the data on which they compute. Thus, fixed-input programs are reasonable benchmark choices for a virtual memory study.

In the following pages, I list the applications studied and the particular details relevant to each. Applications from the SPEC95 benchmark suite[10] are indicated as such.

- **applu** (SPEC95): Solves five coupled parabolic/elliptic partial differential equations. The *test* input file is used, modified to perform ten iterations (200 was the default). (Changing the number of iterations does not affect the fundamental behavior of the program.)

- **blizzard**: A binary-rewriting tool based on the EEL library [25]; used to instrument shared-memory programs for the Blizzard distributed shared memory system [38]. In the run tested, Blizzard instruments a shared-memory relational database server.

- **coral**: a deductive database system evaluating a complex query [34]. The query is evaluated over a 600KB text input file containing a parts/assembly database.

- **es**: calculates microstructure electrostatics [43, 42]. Behavior is dominated by a number of sequential iterations over a large region of memory.
- **gcc**: the GNU optimizing C compiler, version 2.6.3. GCC compiles the preprocessed file *cccp.i*, one of the input files from the SPEC95 distribution.
- **gnuplot**: a graphics program plotting a large X-Y plot similar to the ones appearing in Section 2.3 below. A 12MB text input file (X-Y data, one point per line) is converted into a 15MB PostScript output file.
- **giftopnm**: image file format conversion software, converting a 3.3MB GIF format file into an uncompressed intermediate format used by the NetPBM application suite. (The uncompressed output, about 27MB, is redirected to the UNIX data sink, */dev/null*.)
- **ijpeg** (SPEC95): JPEG image compression software that repeatedly compresses a 2.4MB color image (*vigo.pnm* from the SPEC95 distribution). To speed up execution time, I reduced the "compression quality" and "smoothing factor" command-line arguments from the default values (90) to 30. This reduces the number of iterations performed but does not fundamentally change the program's behavior.
- **m88ksim** (SPEC95): a cycle-level simulator for a microprocessor. The SPEC95 *ref* input set is used; the program simulates the execution of a random number generator and of the Dhrystone synthetic benchmark.
- **murphi**: automatic protocol verification software[14], version 2.91S. I use the *mcslock1* input file provided with the source distribution. (We study the performance of the compiled program *mcslock1* resulting from running the murphi

compiler on the *mcslock1.m* source file.)

- **perl** (SPEC95): the perl[44] interpreted scripting language executing the *jumble.pl* program with the SPEC95 *train* input file. The program makes heavy use of perl's built-in array types: integer-indexed arrays and associative arrays (hash tables).

- **ppmquant**: image quantization software [19], shrinking the colormap of a 27MB uncompressed input file.

- **sor**: successive over-relaxation on a 3000 by 3000 square matrix; the program is from Park [32]. (I removed the Mach-specific system calls used in his study.)

- **swim** (SPEC95): a shallow-water simulation using a finite difference model; swim iterates over a number of arrays containing different quantities for points in the modelled space. The SPEC *test* input set is used (ten iterations).

- **trygtsl**: solves a general tridiagonal matrix problem; part of an early version of the Linpack floating-point benchmark [15] (the source code is dated 4/18/79).

- **turb3d** (SPEC95): a turbulence simulation using the SPEC95 *test* input set. The program is probably best described by a comment in the source code: "This program can be used for simulating isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions. It solves the Navier-Stokes equations using a pseudo-spectral method. Leapfrog-Crank-Nicolson scheme is used for time stepping."

- **vortex** (SPEC95): a relational database system performing a query on several in-memory tables. The SPEC *train* input set is used.

- **wave5** (SPEC95): a plasma simulation. The SPEC *train* input set is used.

The SPEC95 programs were chosen based on one criteria—the amount of memory they consumed. I examined the source code and accompanying documentation (if any) and chose programs that appeared to use substantial amounts of memory. GCC is a special case; it requires less memory than the other programs, but its popularity as a benchmark makes it worthwhile to study.

The non-SPEC95 programs were provided by a number of people in response to requests for programs requiring large memory sizes. Gnuplot was chosen after I observed that it consumes a large amount of memory—and caused heavy thrashing on my computer—while creating some of the graphs shown later in this chapter.

2.2 Trace methodology

I collected memory reference traces using Shade [9], an instruction-level trace generator for the SPARC architecture. Shade is a library; it allows a programmer to specify instruction types of interest. (For purposes of observing memory references, we need to know each instructions' Program Counter (PC) value and, for loads and stores, the data address.) At runtime, Shade instruments the target program and begins executing it. As target instructions execute, any instructions that match the programmer's specification are returned to a programmer-supplied function. This function, in turn, is free to record any state it desires. A common use, for example, is to simulate a processor cache.

All programs ran on Sun SPARCStation 20 computers running the Solaris 2.4 operating system. For purposes of generating memory references, all C programs were compiled with GNU CC version 2.6.3 or 2.7.2, and all Fortran programs with

SunPro SC3.0.1.

Because of the length of our traces—billions of instructions for most programs—recording all memory references individually would result in unmanageably large trace files. Instead, the trace generation software records “IN” and “OUT” records. Program execution is divided into fixed-length *time intervals*, for most programs one million instructions long. One million instructions was determined empirically; it gives in a good balance between trace file size and achievable simulation precision, discussed below.

At the end of every interval, for every page that was referenced in the interval but was *not referenced* in the previous interval, an IN record is generated and time-stamped 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 accessed* in the current interval, an OUT record is generated with the timestamp of the instruction that last referenced the page. IN and OUT records are written out to a trace file sorted by their timestamps.

A page size of 4KB is used throughout this study. 4KB is probably the most common page size in current use; some systems use 8KB pages.

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 not recorded. However, a page is guaranteed not to be accessed in the time between an OUT record and the next IN record for that page.

This trace format is not only compact but also allows *accurate* simulation for sufficiently large memory sizes. At any point in a trace, we 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". 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 *after* 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. In practice, there are usually ACTIVE pages with extremely early IN records, typically generated within a few thousand instructions after program startup, so usually LRU can only replace IDLE pages that have been IDLE for two or more intervals.

A limitation of this 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 trace method outlined above is similar to Carr's IRIM model (Independent Reference Interval Model) [8]. However, within an interval, IRIM apparently does not maintain precise temporal ordering of its equivalent of IN and OUT records. Resultant loss of precision is probably insignificant, though.

Table 1 lists some facts about the programs studied. The table gives the trace length (in millions of instructions executed), the program executable size (in kilobytes), and the *desired* memory size (in kilobytes). In this thesis, I use the term

Program	Execution time (10^6 instrs)	Trace length	Image size (KB)	Memory req'd (KB)
applu	1068	266792	136	14524
blizzard	2122	985620	1153	15632
coral	4327	2117710	940	20284
es	71003	1316173	56	104488
gcc	1371	397052	1599	3936
giftopnm	1676	56066	33	52628
gnuplot	4940	106546	602	62516
ijpeg	42951	904266	152	8260
m88ksim	10020	236161	165	19352
murphi	1019	230972	238	9380
perl	18980	3220759	569	39344
ppmquant	3814	82238	32	52760
sor	5838	176158	11	70592
swim	438	180342	56	15016
trygtsl	377	117882	26	69688
turb3d	17989	1946504	71	26052
vortex	2507	371868	600	9676
wave5	3774	155678	511	28700

Figure 1: Summary of benchmark traces: program execution time in millions of instructions; trace file length in number of records; binary executable image size (KB); and memory consumed (KB).

desired memory size to refer to the number of unique virtual memory pages used by a program in a particular execution. No pages replacements need to occur if a programs runs with physical main memory greater than or equal to its desired memory size.

Most traces are in excess of one billion instructions long (the exceptions are swim and trygtsl). Also, most programs consume much more memory than is required to fit their binary executable images into memory. This seems to be a trend among fixed-input programs—they operate on large quantities of data created at run-time or consumed as input. (Interactive programs with elaborate user interfaces, for example word processors, may in many cases have much more text than data.)

Several of the above traces were truncated to keep the trace files at a manageable size. In particular, coral, es, and perl were shortened to the length shown above. Coral was originally slightly over seven billion instructions, and perl was about thirty. Es was much longer than the 71 billion instructions shown, but its looping behavior (described below) is such that the truncated trace captures es's fundamental behavior. I verified for coral and perl that behavior as indicated by the truncated traces reflects an accurate picture of the respective complete run.

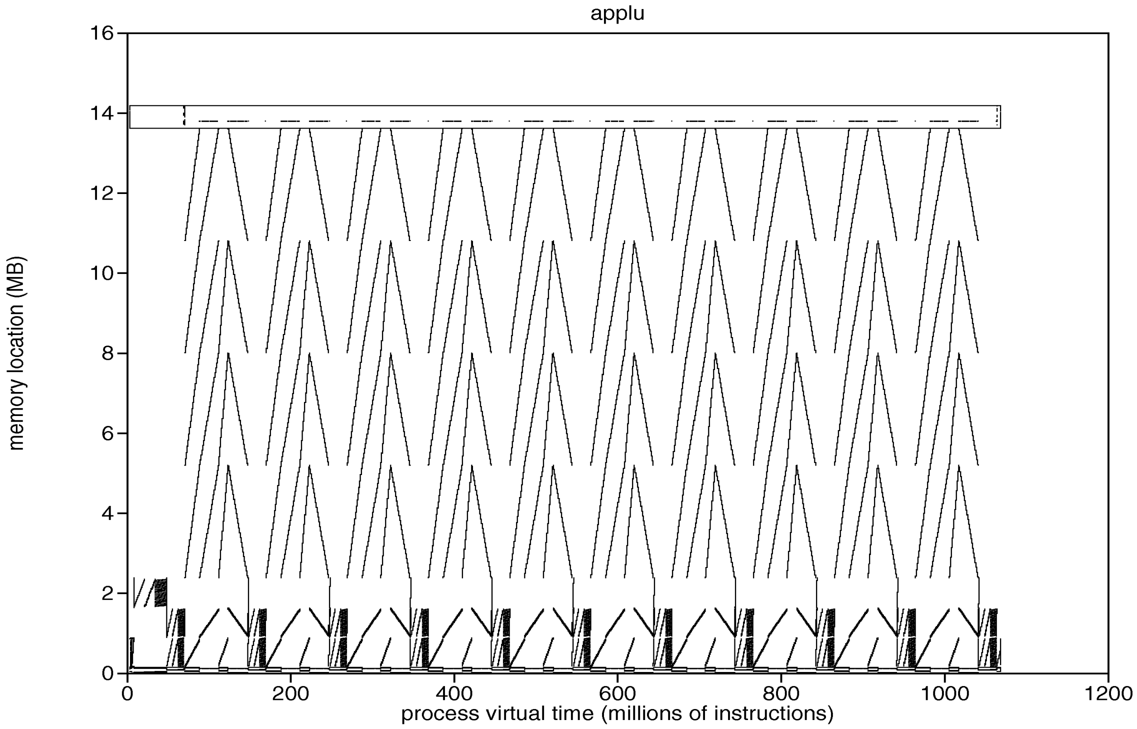
2.3 Memory reference graphs

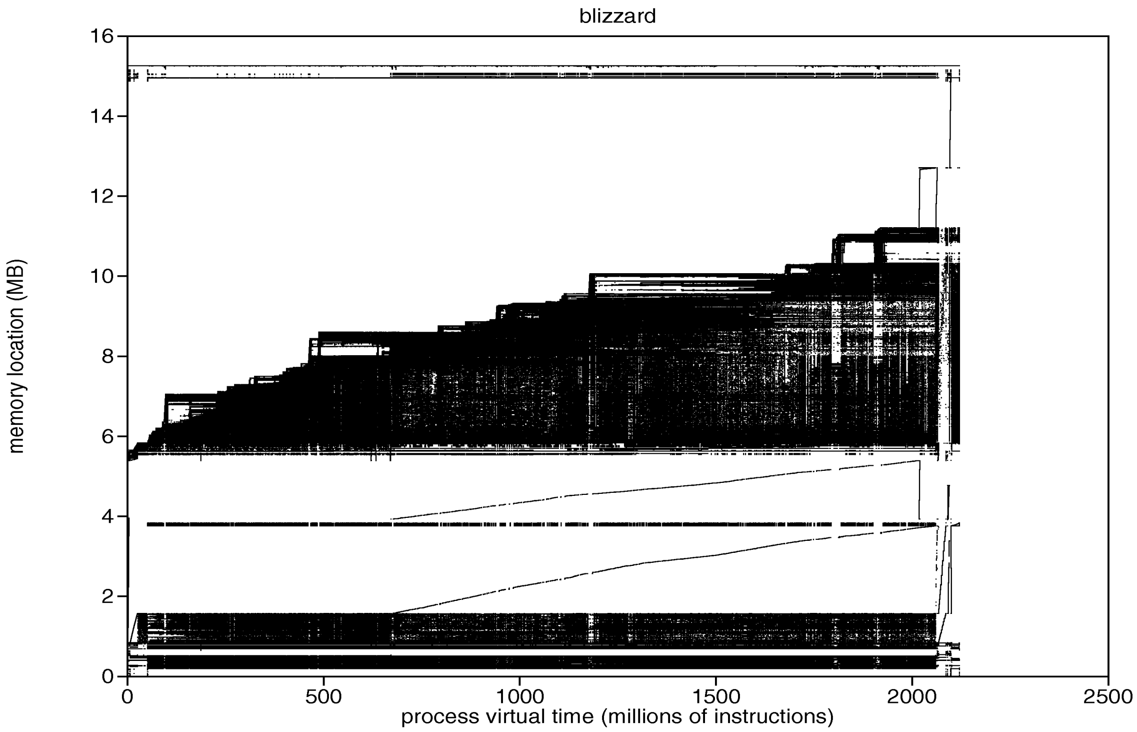
This section presents what I will refer to as *space-time* plots. A space-time plot shows, for each execution interval (the x-axis), the set of pages that were referenced in that interval (the y-axis).

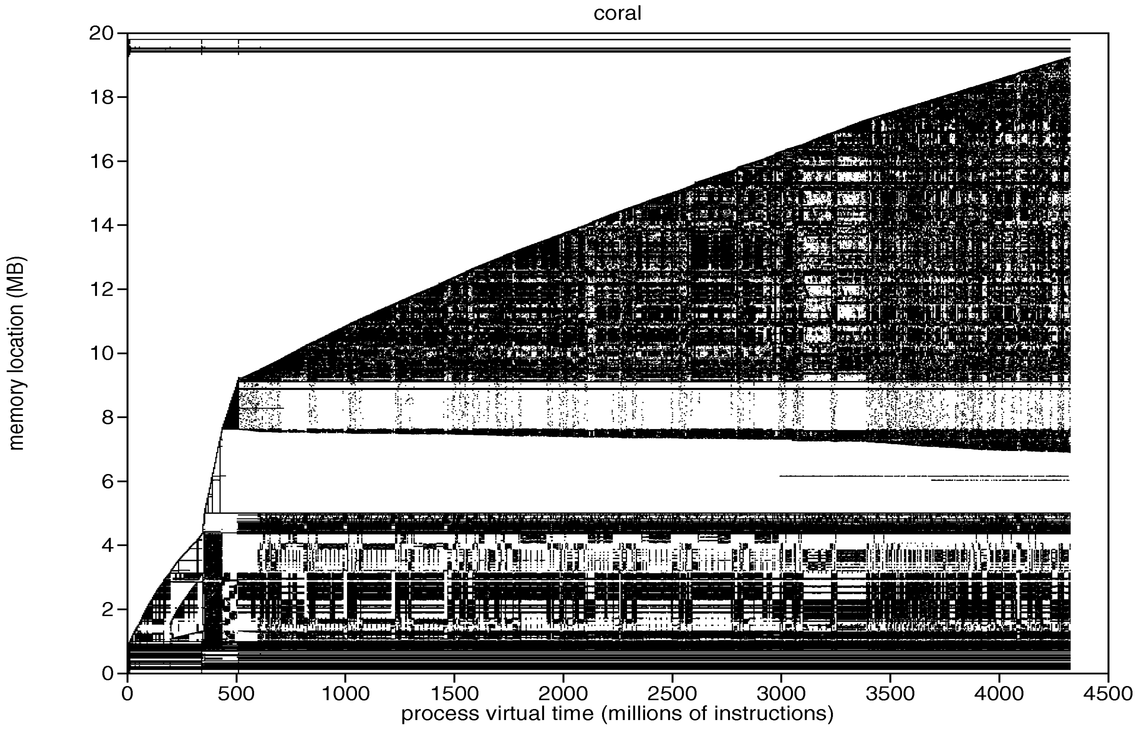
Solaris 2, like many operating systems, creates for each process a sparse address space. The address space spans 32 bits (in Solaris 2.4), but very few programs come close to using all of the available address space. Most programs contain large regions of unused virtual address space. In the plots that follow, these regions have been eliminated; if they were present in the graphs, they would squeeze out the interesting data—the portions of the address space that were actually used by the program. The y-axis is labelled in terms of *relative* address space location, starting from 0 at the low end of the address space.

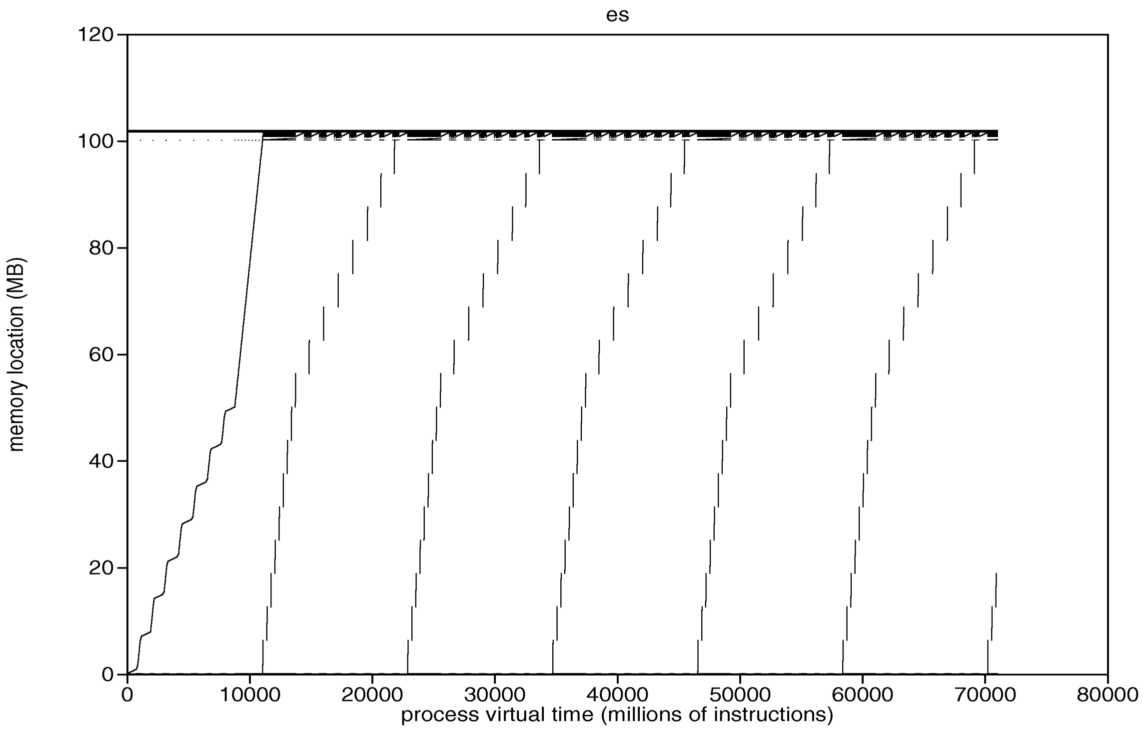
Operating system policy determines the location of various address space components in the plots. In Solaris 2, a process's executable code—its text segment—is at the bottom of the address space; the text is followed directly by the *initialized* data segment, which contains data from the binary executable image. Following the initialized data comes the *uninitialized data* segment, sometimes called the *heap*. (A system call may be used to expand or shrink the heap.) At the top end of the address space is the stack, which grows downward. Finally, any shared libraries (in UNIX, really just memory-mapped files) are mapped at kernel-chosen addresses midway between the top of the heap and the bottom of the stack. The applications studied in this thesis all use a modest amount of library code for such things as formatted input/output, mathematical functions, memory allocation, and string operations. The Image Size column in Table 1 above does not include pages mapped in from shared libraries.

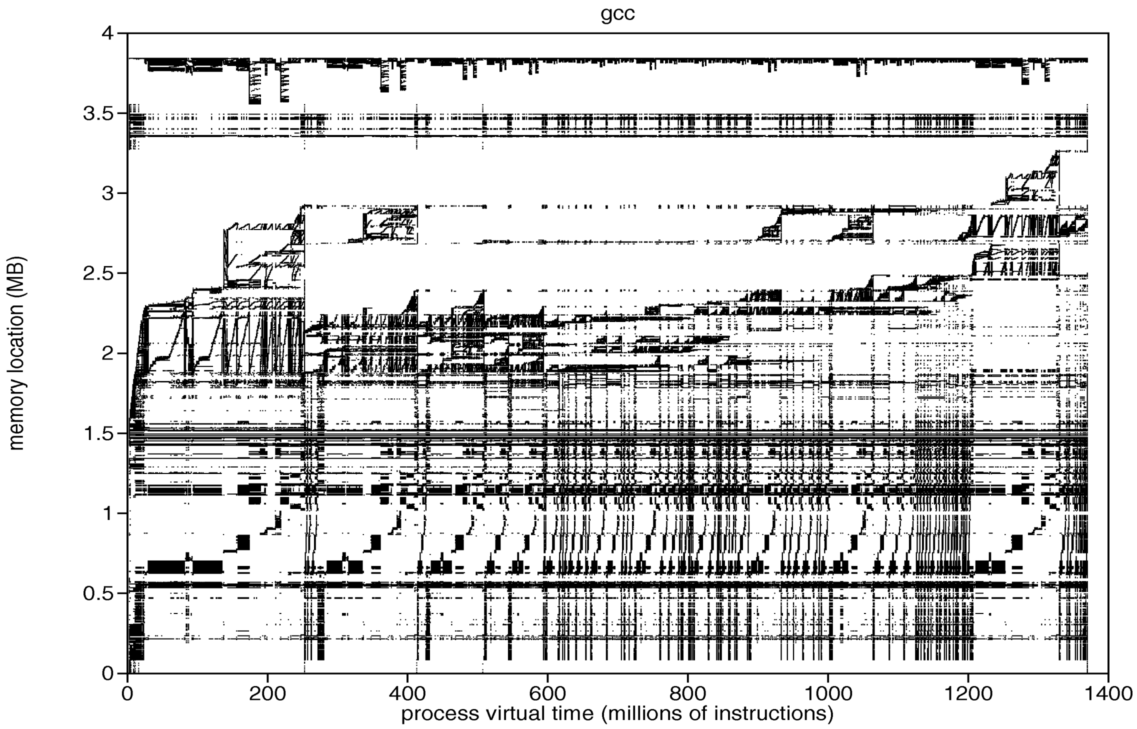
In the plots below, the program text is at the bottom, followed by initialized and uninitialized data, followed finally by shared libraries and stack. For most programs, the uninitialized data segment dwarfs the other address space components. (GCC is the only program whose text size approaches the data size.) For a few programs, there is evidence of recursive function calls—stack activity near the top of the plot. However, for most programs the stack is usually visible only as line.

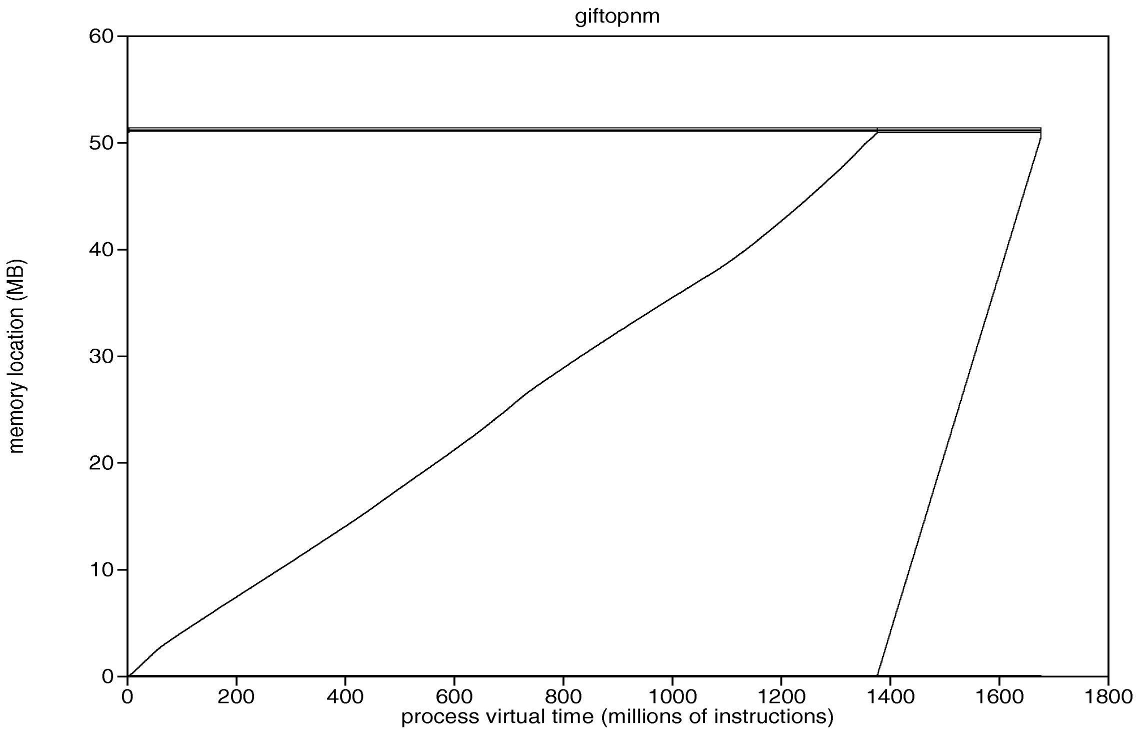

applu
memory location (MB)
16
14
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
process virtual time (millions of instructions)

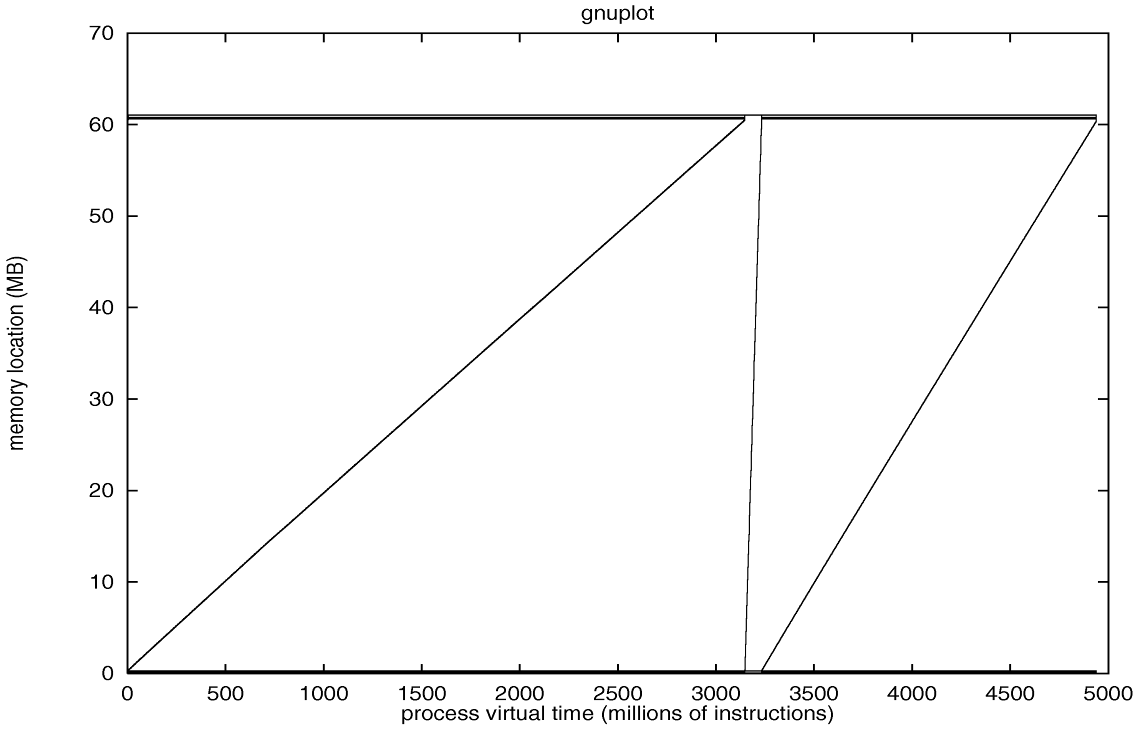

blizzard
memory location (MB)
16
14
12
10
8
6
4
2
0
0
500
1000
1500
2000
2500
process virtual time (millions of instructions)

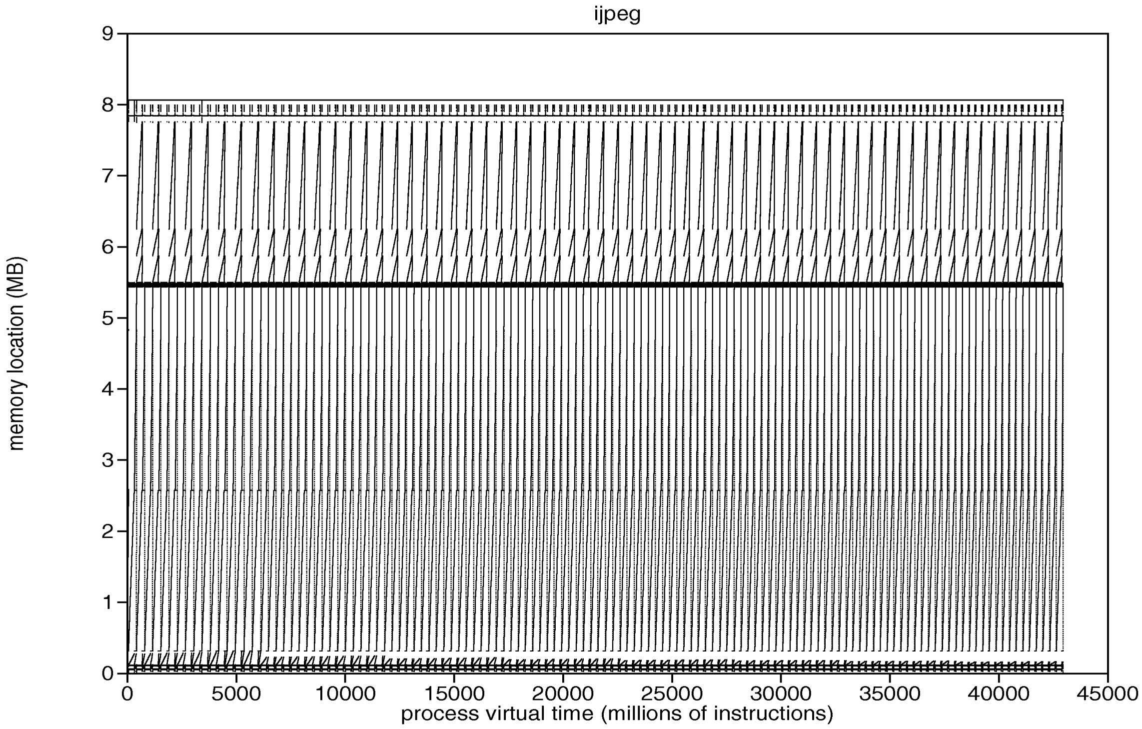

coral
memory location (MB)
process virtual time (millions of instructions)
0
2
4
6
8
10
12
14
16
18
20
0
500
1000
1500
2000
2500
3000
3500
4000
4500

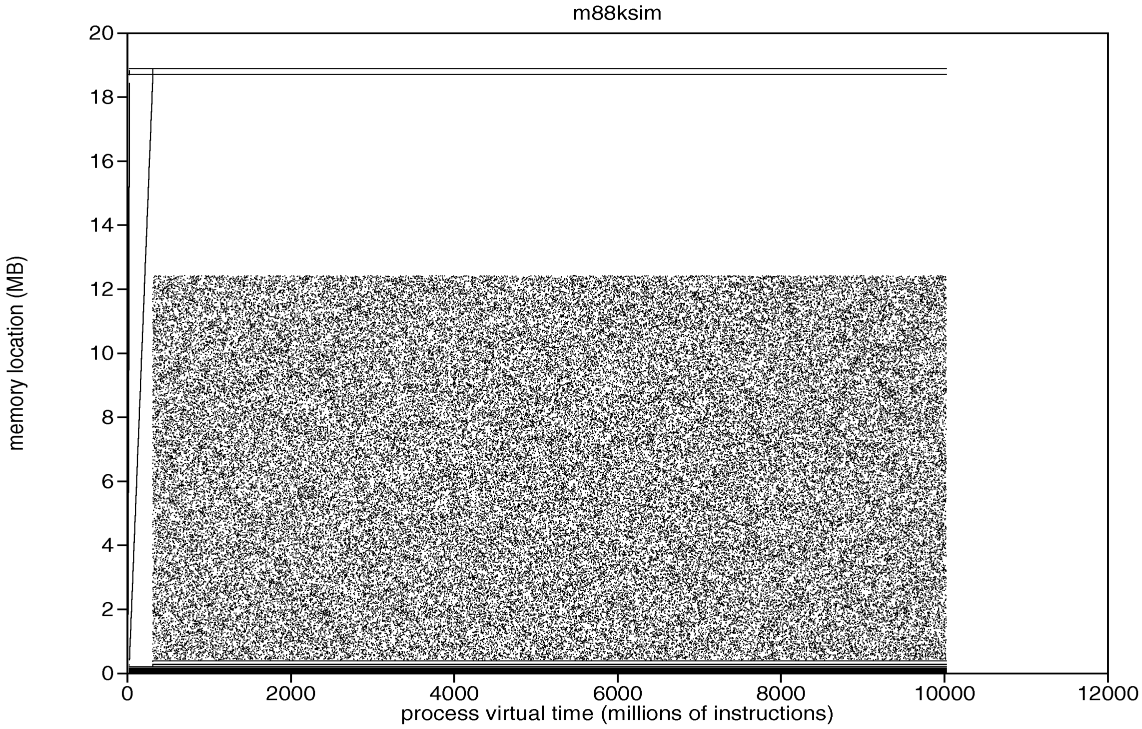

es
memory location (MB)
process virtual time (millions of instructions)
0
20
40
60
80
100
120
0
10000
20000
30000
40000
50000
60000
70000
80000

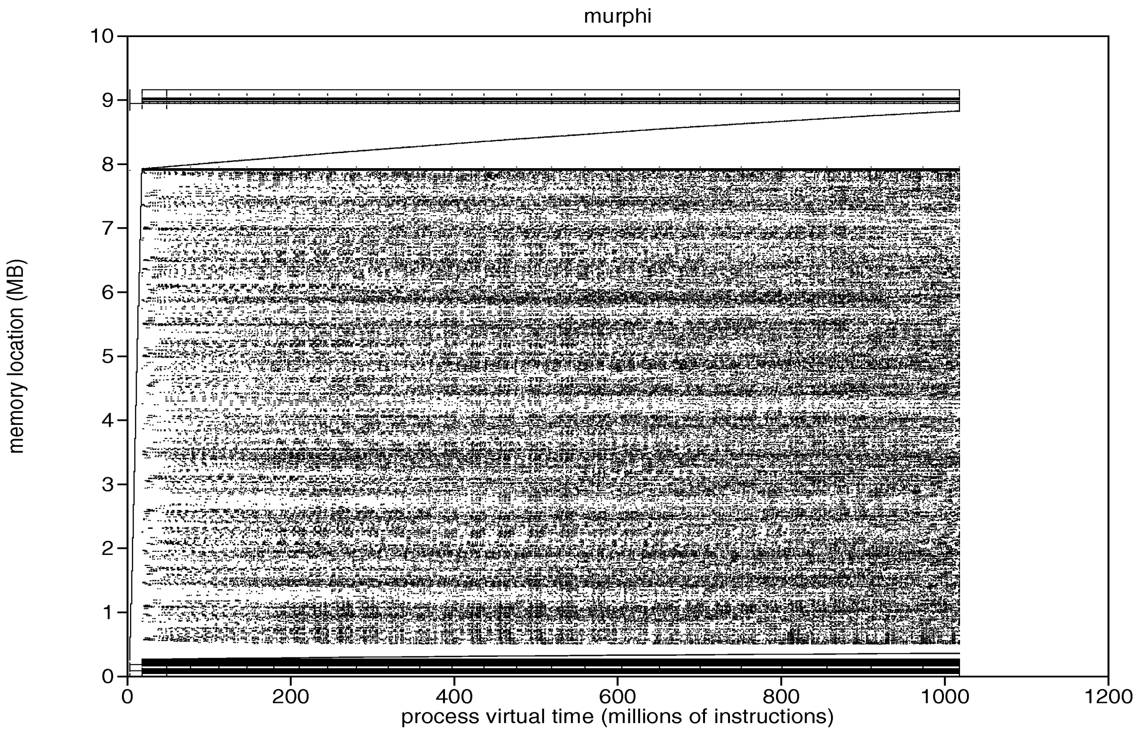

gcc
memory location (MB)
process virtual time (millions of instructions)
0
0.5
1
1.5
2
2.5
3
3.5
4
0
200
400
600
800
1000
1200
1400

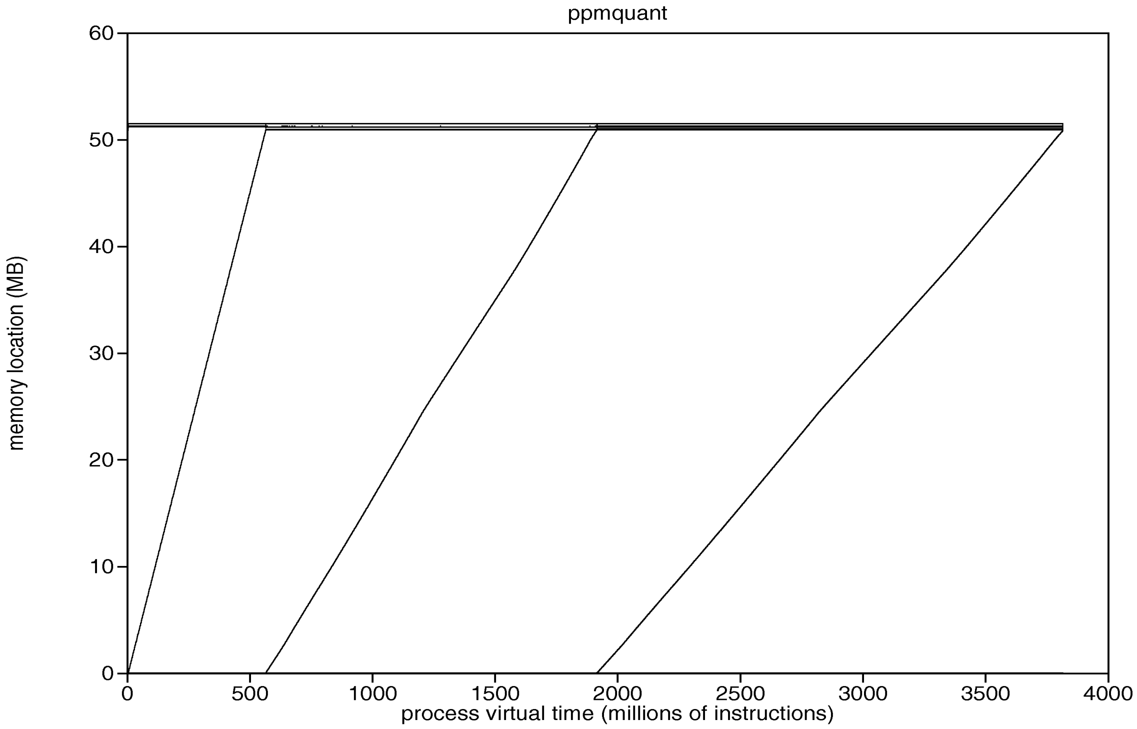

giftopnm
memory location (MB)
process virtual time (millions of instructions)
0
10
20
30
40
50
60
0
200
400
600
800
1000
1200
1400
1600
1800

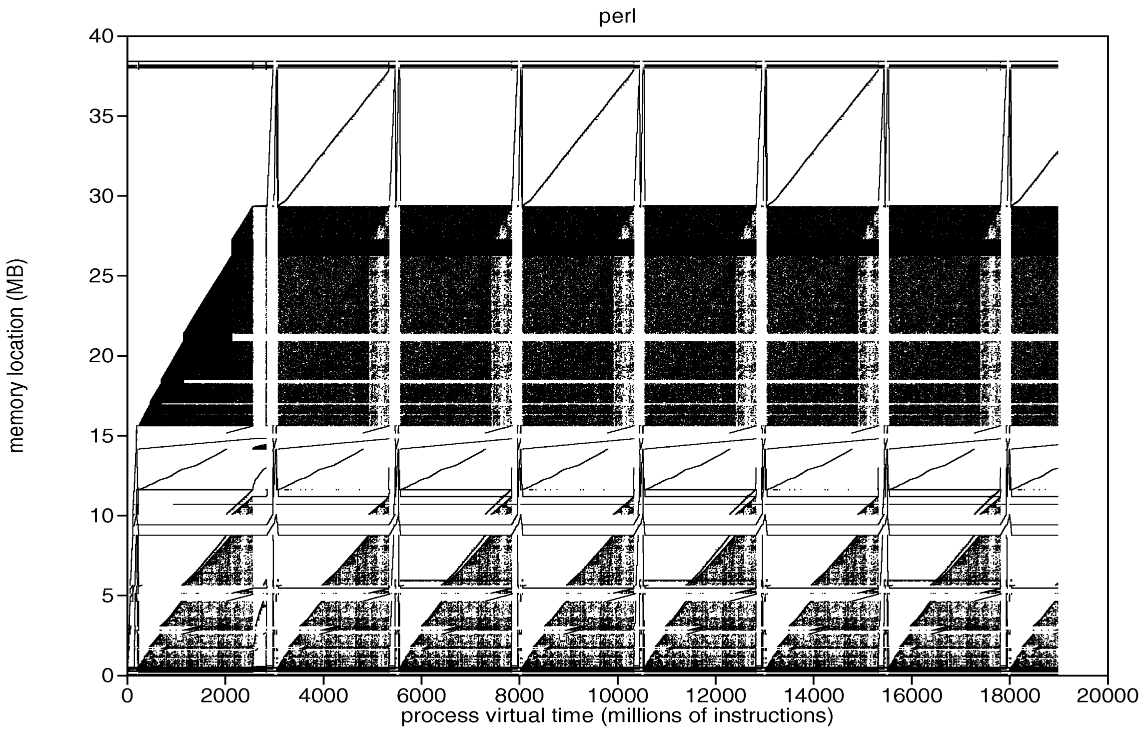

gnuplot
memory location (MB)
0
10
20
30
40
50
60
70
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
process virtual time (millions of instructions)

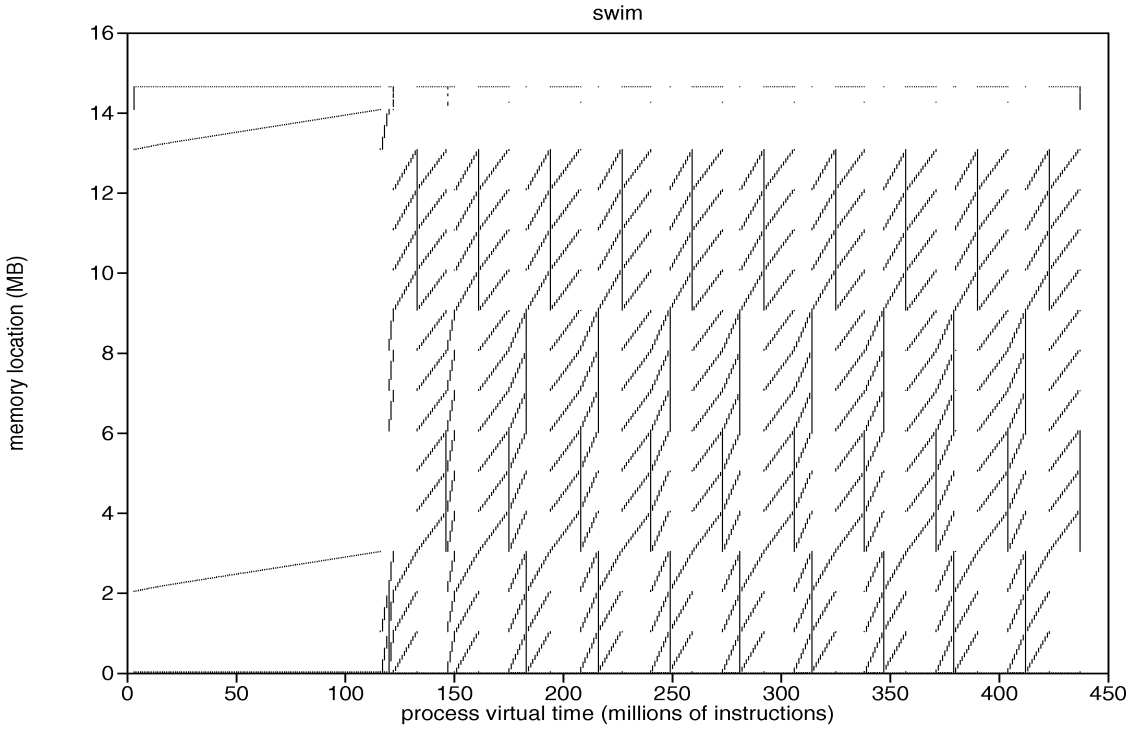

ijpeg
memory location (MB)
0
1
2
3
4
5
6
7
8
9
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
process virtual time (millions of instructions)

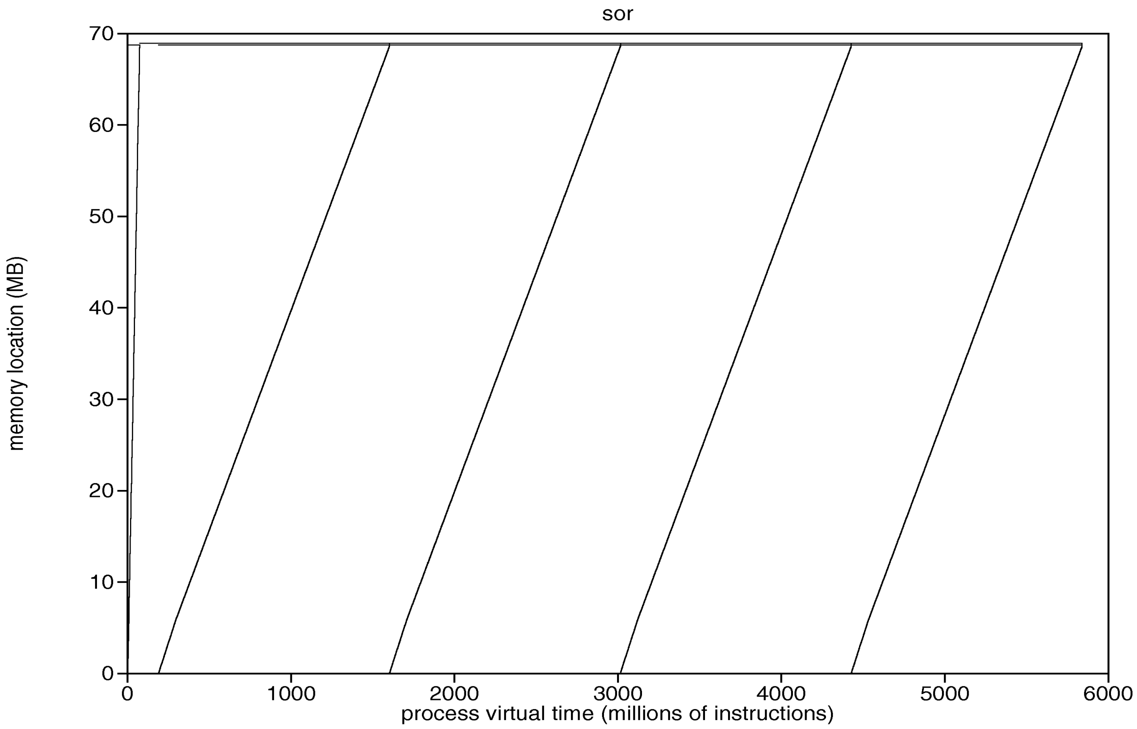

m88ksim
memory location (MB)
process virtual time (millions of instructions)
0
2
4
6
8
10
12
14
16
18
20
0
2000
4000
6000
8000
10000
12000

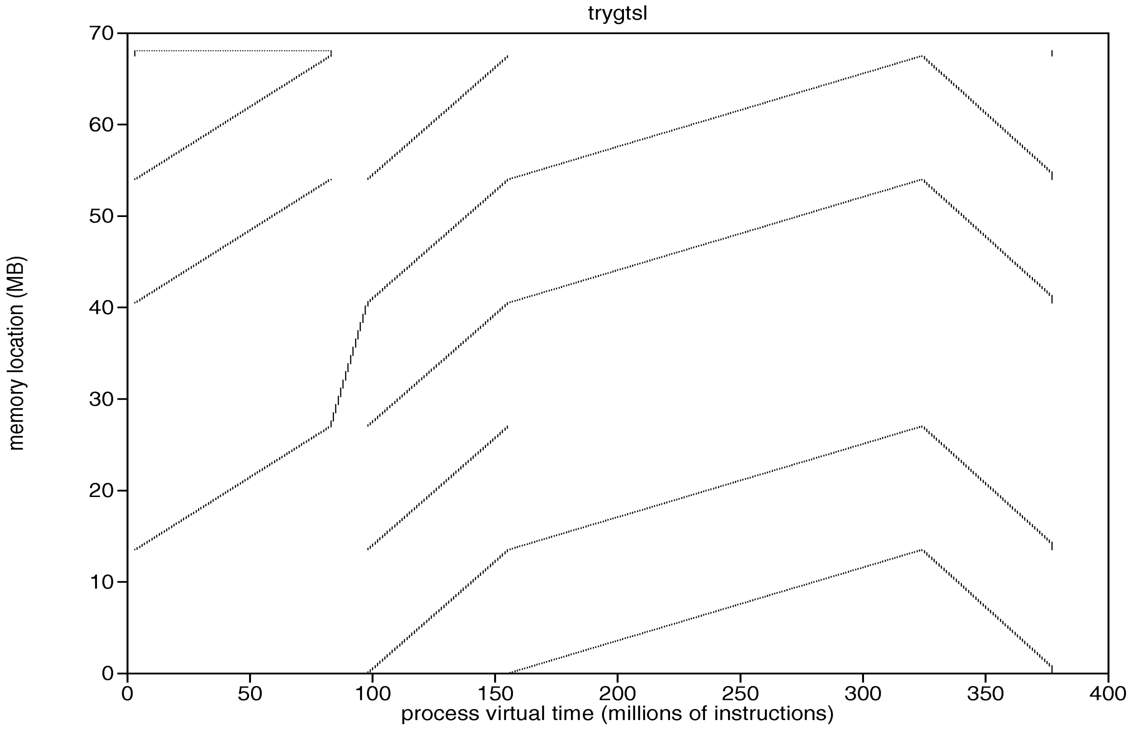

murphi
memory location (MB)
process virtual time (millions of instructions)
0
1
2
3
4
5
6
7
8
9
10
0
200
400
600
800
1000
1200

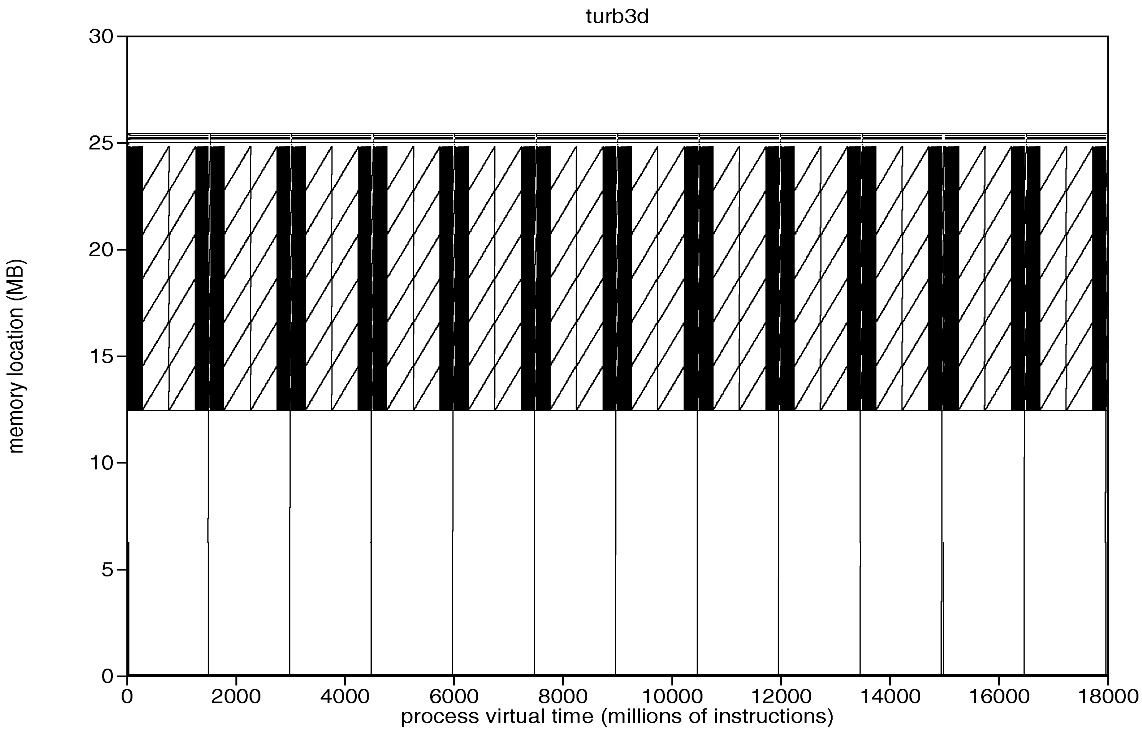

ppmquant
memory location (MB)
process virtual time (millions of instructions)
0
10
20
30
40
50
60
0
500
1000
1500
2000
2500
3000
3500
4000

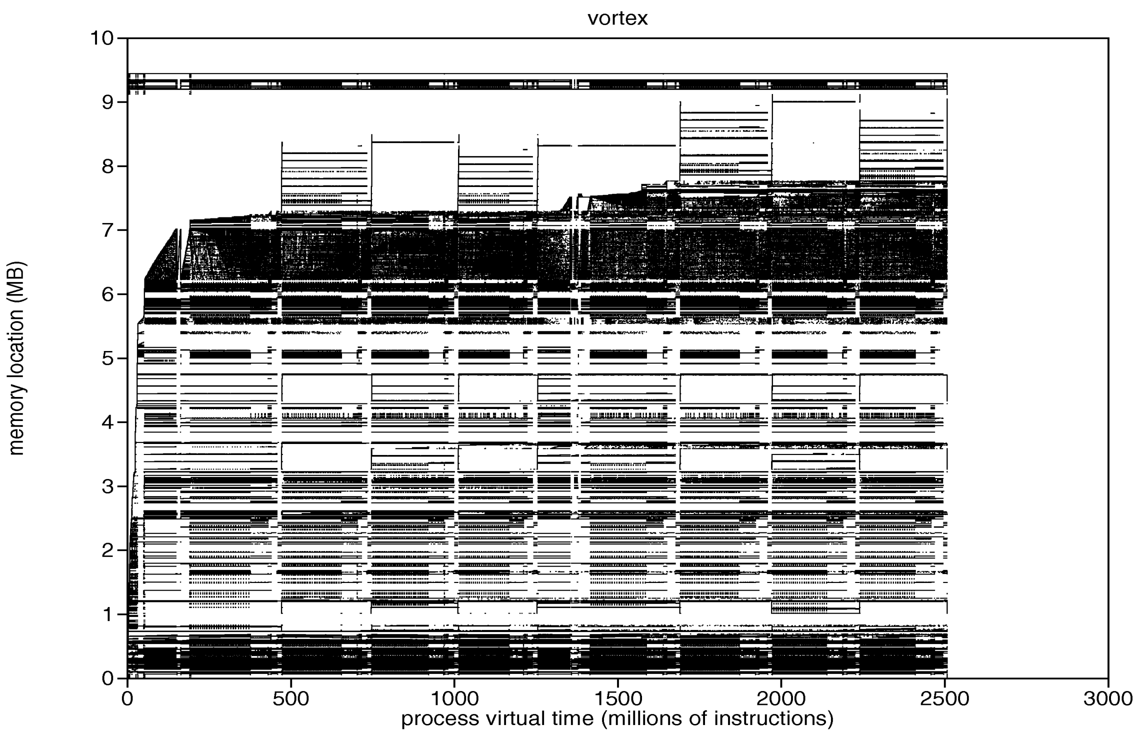

perl
memory location (MB)
process virtual time (millions of instructions)
0
5
10
15
20
25
30
35
40
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000

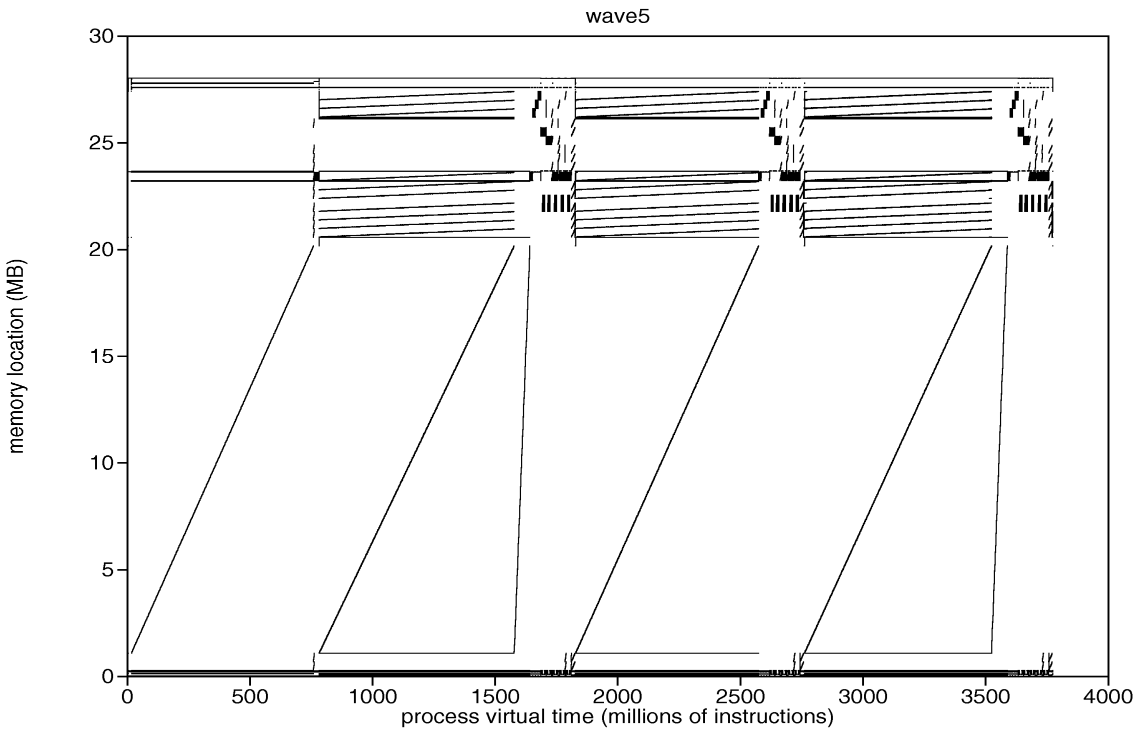

swim
memory location (MB)
process virtual time (millions of instructions)
0
2
4
6
8
10
12
14
16
0
50
100
150
200
250
300
350
400
450


sor
memory location (MB)
process virtual time (millions of instructions)
0
10
20
30
40
50
60
70
0
1000
2000
3000
4000
5000
6000


trygtsl
memory location (MB)
70
60
50
40
30
20
10
0
0
50
100
150
200
250
300
350
400
process virtual time (millions of instructions)


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


vortex
memory location (MB)
process virtual time (millions of instructions)
0
1
2
3
4
5
6
7
8
9
10
0
500
1000
1500
2000
2500
3000


wave5
memory location (MB)
process virtual time (millions of instructions)
0
5
10
15
20
25
30
0
500
1000
1500
2000
2500
3000
3500
4000

These graphs suggest several basic types of program behavior. On the one hand there is the simple looping of sor, ppmquant, and others; at the other extreme are the massive dark regions in programs like coral and perl. These four examples illustrate the difference due to basic data structures: sequentially traversed arrays in sor and ppmquant compared to hash tables used by coral and perl. M88ksim exhibits almost random memory reference patterns after a brief sequential initialization phase. Murphi and GCC show complex patterns of locality (horizontal lines appear for frequently-used pages). Gnuplot makes three passes over a single large memory area, but the time required by the second pass differs substantially from the input phase (first pass) and the output phase (third pass).

Denning introduced the Working-Set model as a model program behavior [12]. Central to the model is the idea of *phase transitions*, in which programs stop doing one computation and move on to another; during such transitions, one would expect the current "working set" of pages to go out of use, and another set of pages to come into use. While there is some evidence of phase transitions (for example, a transition phase change comes after the initialization phases in programs like m88ksim and swim), a more accurate observation is that fixed-input programs perform their computations on the same data: the same data is accessed over and over again. Different phases access the same data at perhaps different rates or in perhaps different ways (sequential versus random, for example).

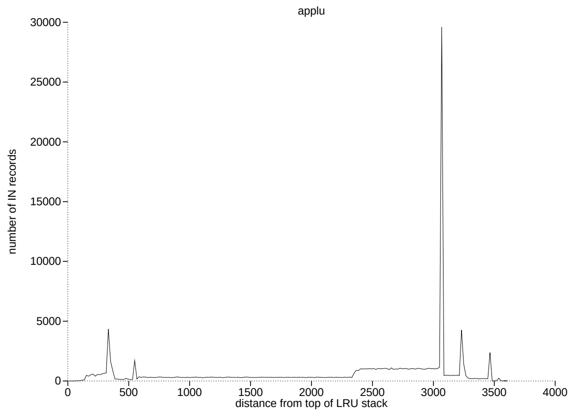
Graphs similar to the space-time graphs shown above were created by Perl and Sites [33]. Their plots differ from mine in several important ways due to different trace generation techniques. First, their plots contain both user and kernel references. This allows for more realistic cache and TLB studies, but is not relevant for paging behavior since the operating system kernel (or at least a large portion of it) cannot be paged out. More significantly, their plots show at most 25 million instructions, not nearly enough to observe long-term program behavior for purposes of exploring virtual memory performance.

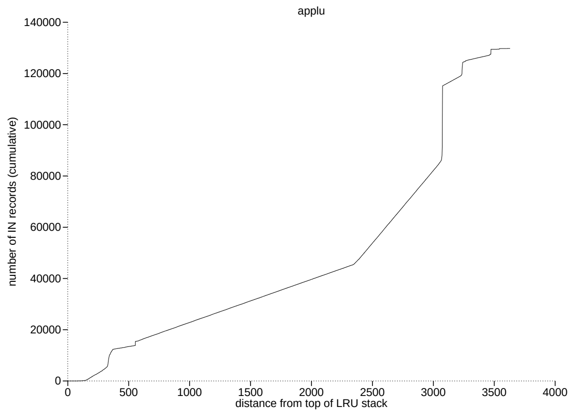
Further principles of program behavior will arise from several more sets of empirical data.

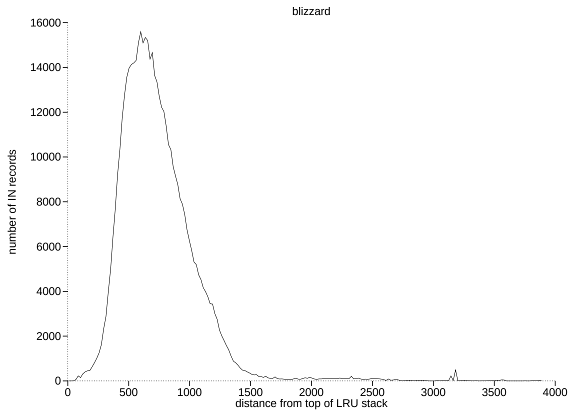
2.4 LRU Stack Graphs

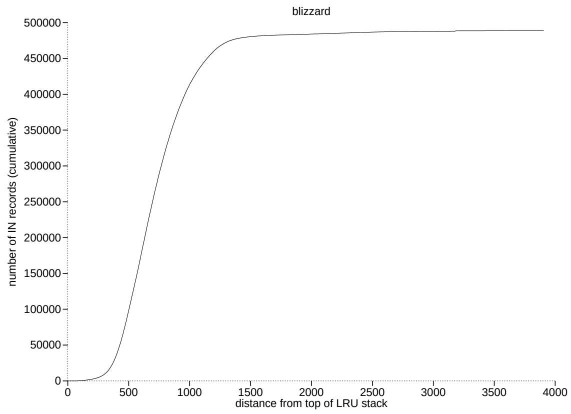
In this section we study program *locality*, which for virtual memory is the correlation of past memory references to a page with future references to that page. An interesting question to ask about a sequence of memory references made by a program is to examine, for a specific reference to a particular page, the *time of last reference* made to that page. The *LRU stack* for a program at any instant contains all the pages used by the program sorted by the time of last reference with most-recently accessed pages at the top of the stack. The following graphs show the frequency of reference as a function of the referenced page's position in the LRU stack. Note that because of the trace format outlined earlier, not all memory references appear on the graphs. Only references with corresponding IN records are shown. (If all references were shown, the far left x-axis values would dominate the curves.)

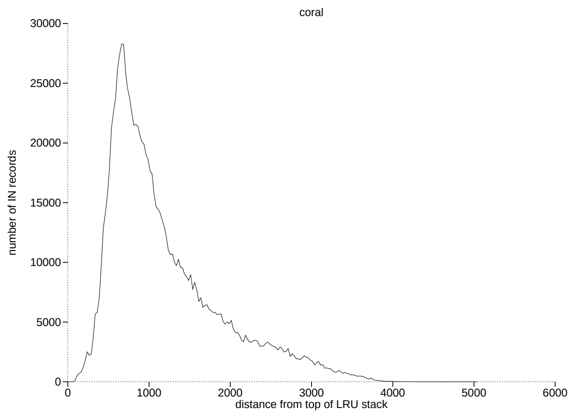
Two graphs are given for each program: the first plots the count of references made as a function of LRU stack position; the second plots the *cumulative* count of references as a function of LRU stack position.

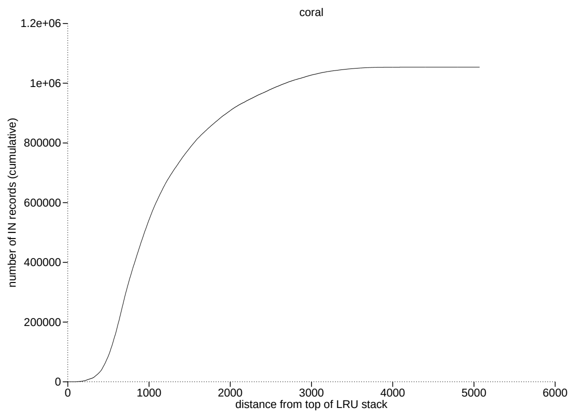

applu
number of IN records
distance from top of LRU stack

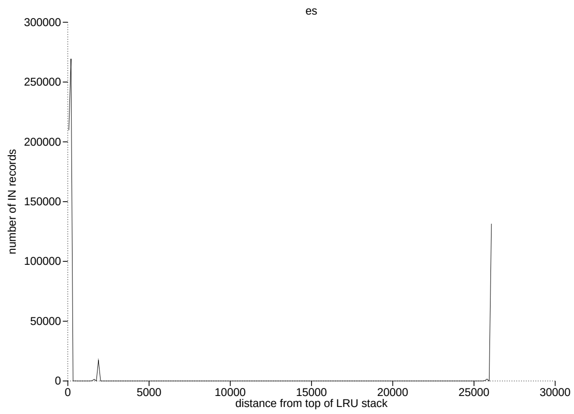

applu
number of IN records (cumulative)
distance from top of LRU stack

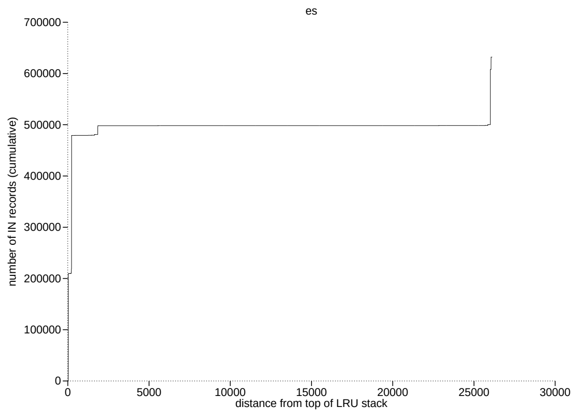

blizzard
number of IN records
distance from top of LRU stack

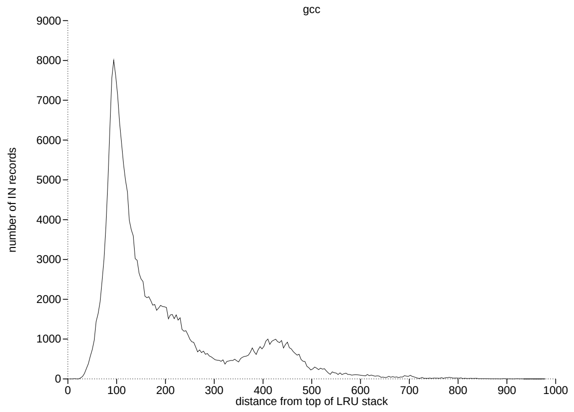

blizzard
number of IN records (cumulative)
distance from top of LRU stack

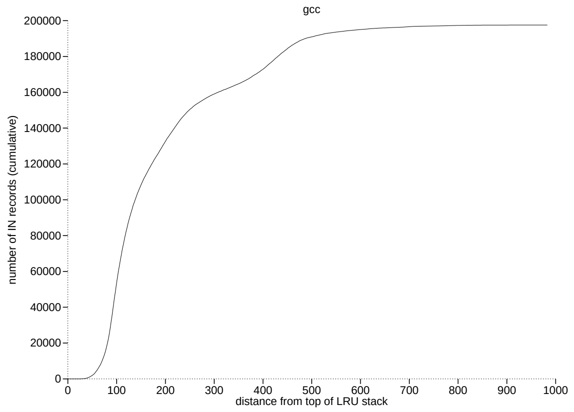

coral
number of IN records
distance from top of LRU stack

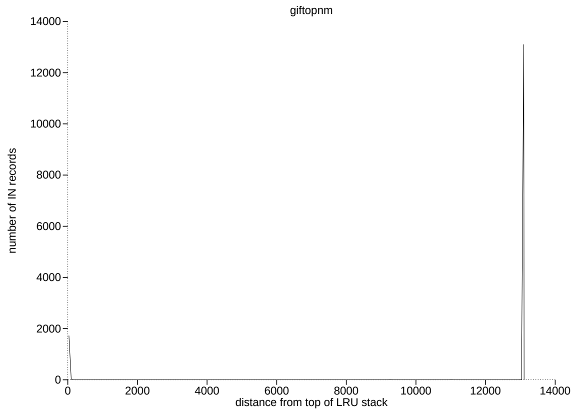

coral
number of IN records (cumulative)
distance from top of LRU stack

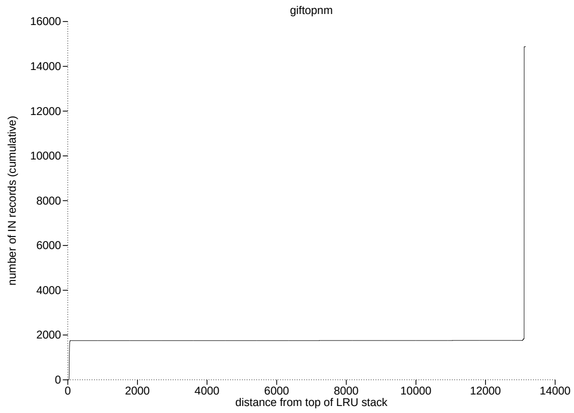

es
number of IN records
300000
250000
200000
150000
100000
50000
0
0
5000
10000
15000
20000
25000
30000
distance from top of LRU stack

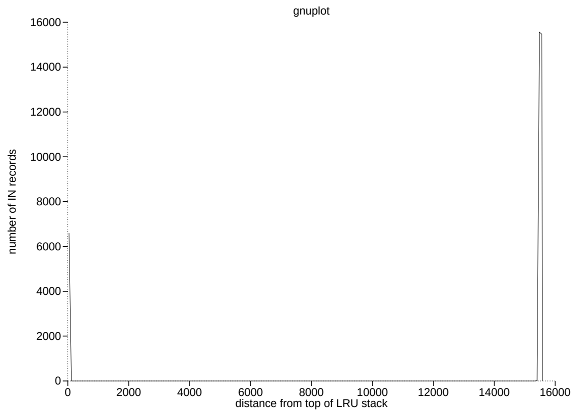

es
number of IN records (cumulative)
700000
600000
500000
400000
300000
200000
100000
0
0
5000
10000
15000
20000
25000
30000
distance from top of LRU stack

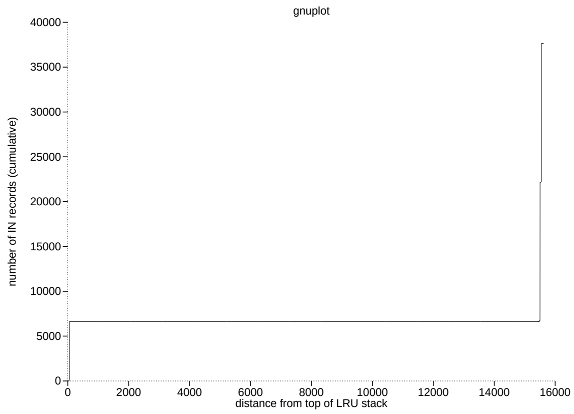

gcc
number of IN records
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
700
800
900
1000
distance from top of LRU stack

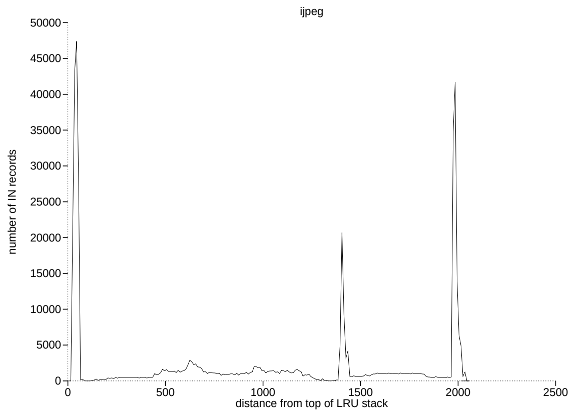

gcc
number of IN records (cumulative)
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
0
100
200
300
400
500
600
700
800
900
1000
distance from top of LRU stack

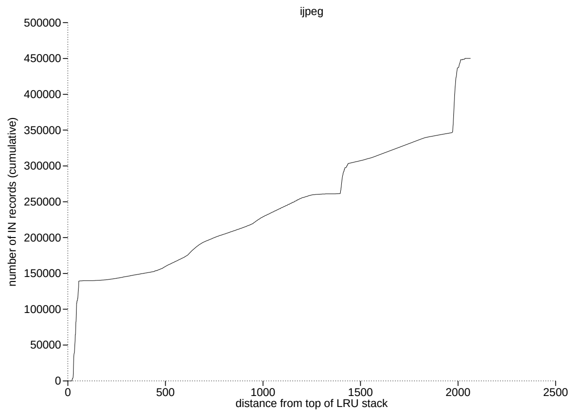

giftopnm
number of IN records
14000
12000
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000
12000
14000
distance from top of LRU stack

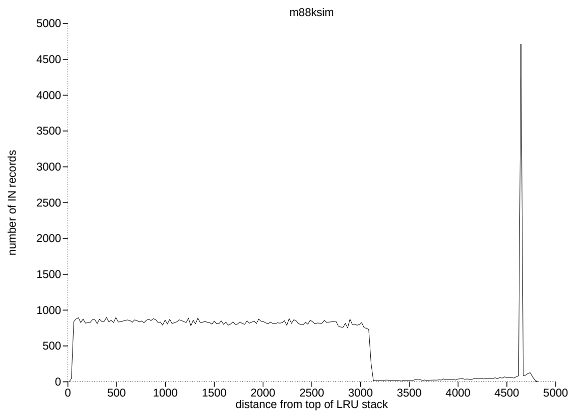

giftopnm
number of IN records (cumulative)
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000
12000
14000
distance from top of LRU stack

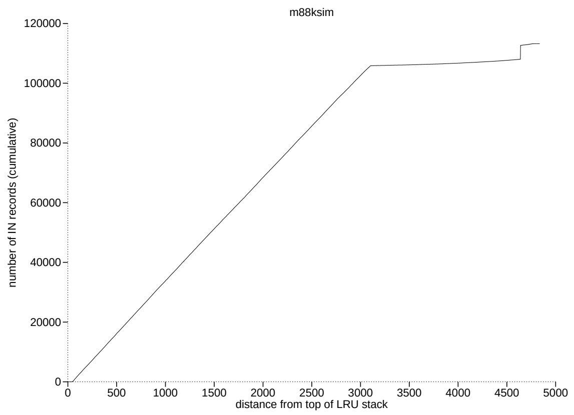

gnuplot
number of IN records
distance from top of LRU stack

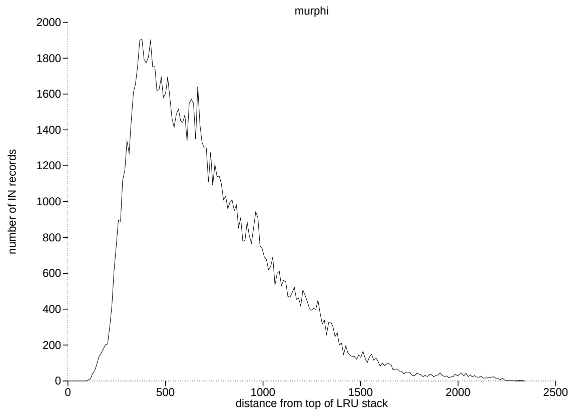

gnuplot
number of IN records (cumulative)
distance from top of LRU stack

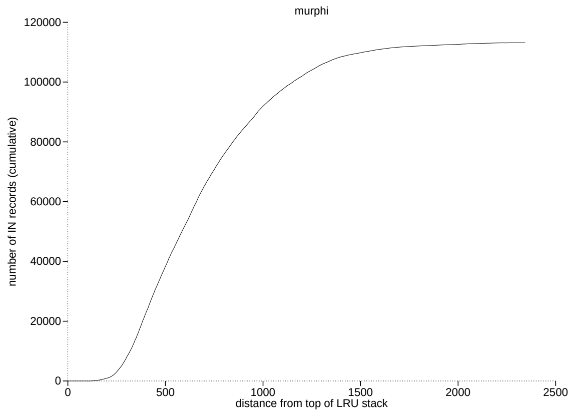

ijpeg
number of IN records
distance from top of LRU stack

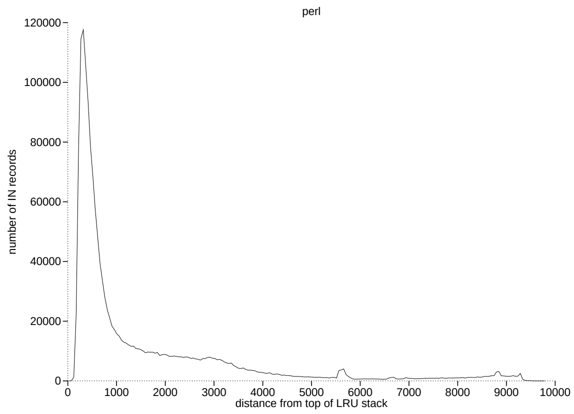

ijpeg
number of IN records (cumulative)
distance from top of LRU stack

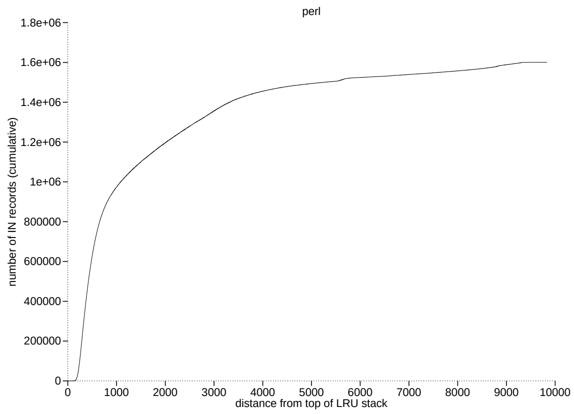

m88ksim
number of IN records
distance from top of LRU stack

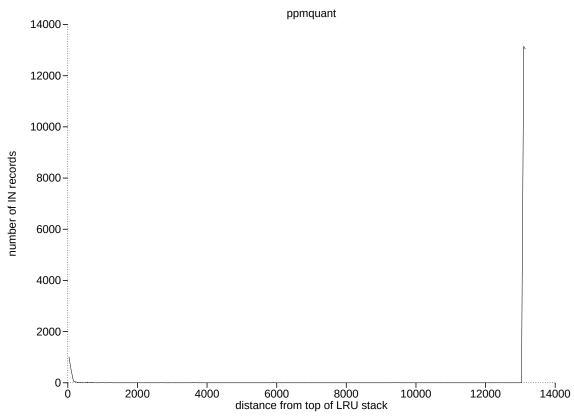

m88ksim
number of IN records (cumulative)
distance from top of LRU stack

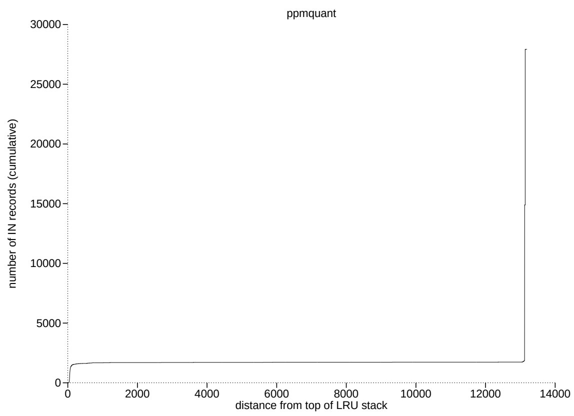

murphi
number of IN records
distance from top of LRU stack
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
500
1000
1500
2000
2500

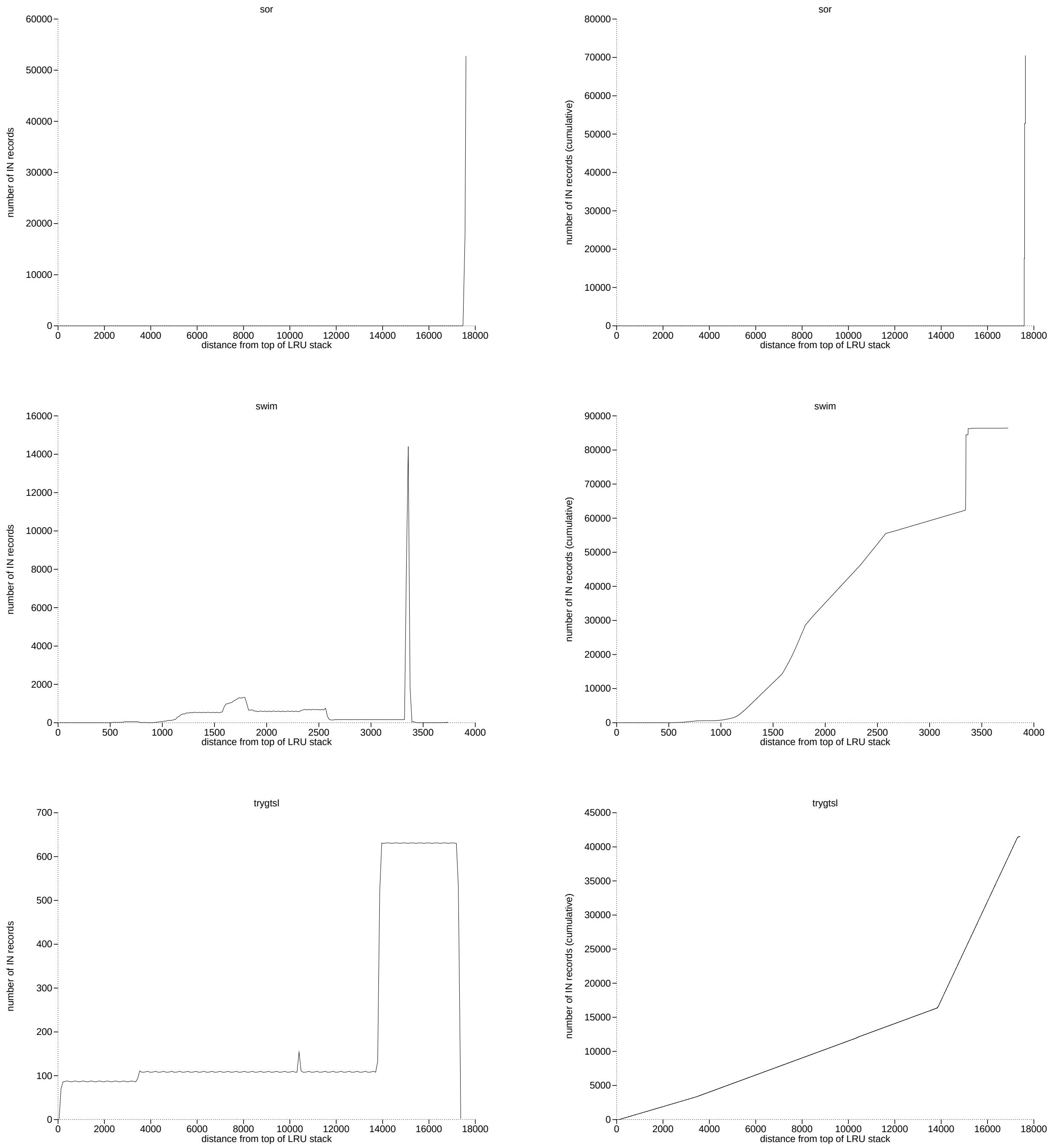

murphi
number of IN records (cumulative)
distance from top of LRU stack
0
20000
40000
60000
80000
100000
120000
0
500
1000
1500
2000
2500

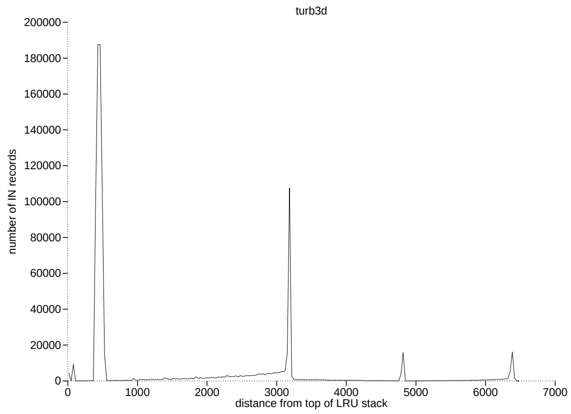

perl
number of IN records
distance from top of LRU stack
0
20000
40000
60000
80000
100000
120000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

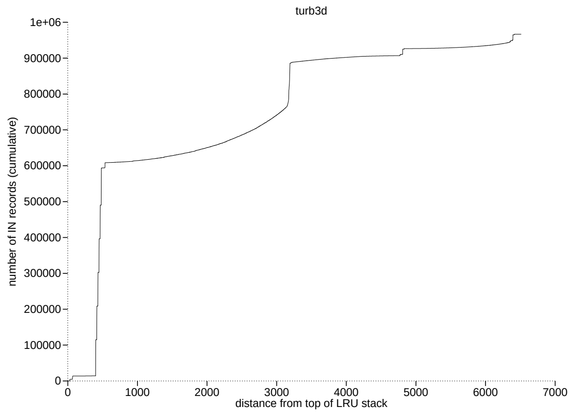

perl
number of IN records (cumulative)
distance from top of LRU stack
0
200000
400000
600000
800000
1e+06
1.2e+06
1.4e+06
1.6e+06
1.8e+06
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

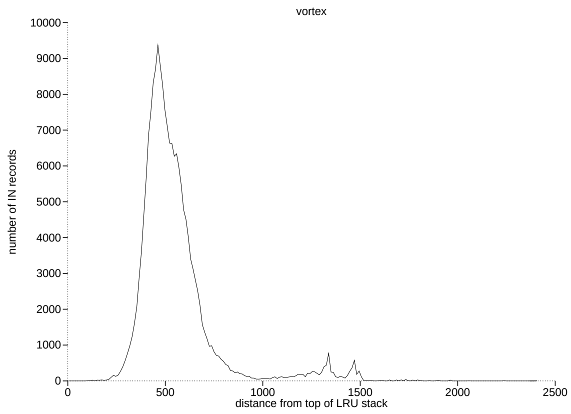

ppmquant
number of IN records
distance from top of LRU stack
0
2000
4000
6000
8000
10000
12000
14000
0
2000
4000
6000
8000
10000
12000
14000

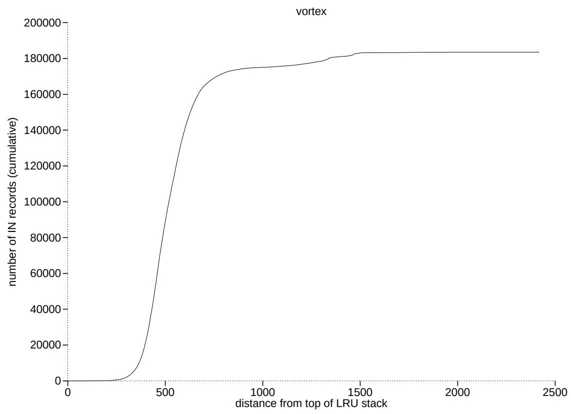

ppmquant
number of IN records (cumulative)
distance from top of LRU stack
0
5000
10000
15000
20000
25000
30000
0
2000
4000
6000
8000
10000
12000
14000

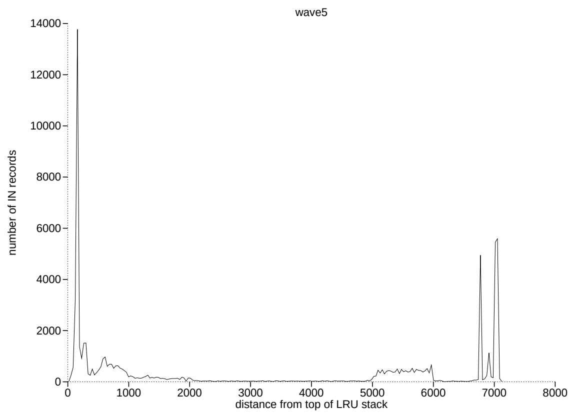

sor
number of IN records
distance from top of LRU stack
sor
number of IN records (cumulative)
distance from top of LRU stack
swim
number of IN records
distance from top of LRU stack
swim
number of IN records (cumulative)
distance from top of LRU stack
trygtsl
number of IN records
distance from top of LRU stack
trygtsl
number of IN records (cumulative)
distance from top of LRU stack

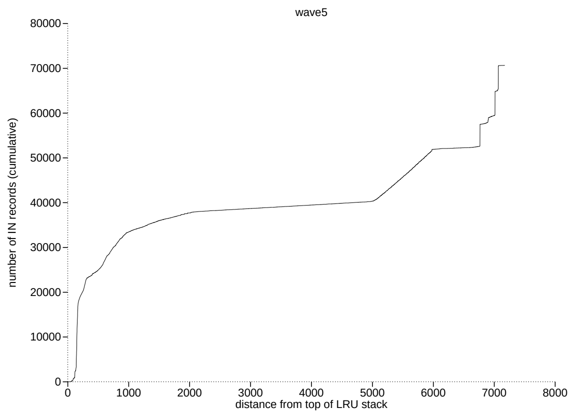












Three distinct behavioral patterns stand out in these plots. For programs whose behavior is dominated by linear traversal of a single, large region of memory, the left-hand stack graphs above show a large spike at a high x-axis value, corresponding to the loops' references to old pages. Five programs comprise this class: es, giftopnm, gnuplot, ppmquant, and sor.

Programs that have complex space-time graphs—graphs with large dark areas—tend to have LRU stack graphs with a single peak corresponding to a relatively "young" age (a low x-axis value). Such a peak indicates the presence of locality in the program's behavior—the program tends to reference pages that it referenced in the near past. The complex, high-locality program class includes blizzard, coral, gcc, murphi, perl and vortex. Observe that even though the space-time graph for murphi may, at first glance, seem quite similar to the random behavior of m88ksim, murphi in fact displays a high degree of locality. M88ksim's behavior appears utterly random, with an even probability of referencing a page in a wide range of ages.

The remaining programs (applu, ijpeg, swim, trygtsl, turb3d, and wave5) fall in between the massively sequential class and the complex class. These other programs iterate over multiple ranges of memory; some have several spikes in the left graph, corresponding to different average page ages in different regions (for example, ijpeg, turb3d, and wave5).

For the remainder of this thesis, I will refer to these three program classes as *massively-sequential*, *complex*, and *mixed.* Complex applications show the most locality; massively sequential show the least. These classifications are primarily for notational convenience; the program classes should not be considered a rigid division of the space of possible program behavior. (Such a division is be bound to fail—consider the graph for perl, which shows high locality in some areas and fairly large-scale looping in others.)

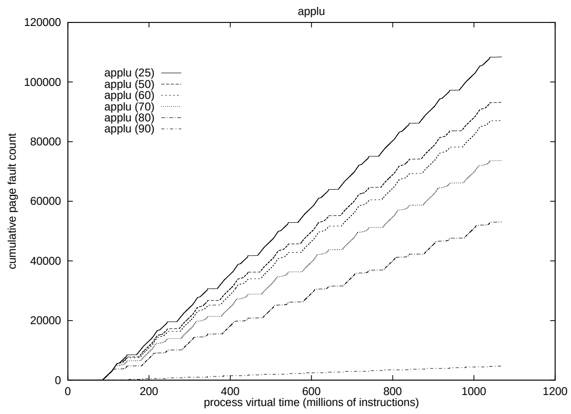
I should note a few details on some of the massively-sequential programs. Both gnuplot and es have spikes in their LRU stack graphs near the top of the stack (the left end of the graph). For es, the references responsible for the spike are visible in the space-time graph as a repeated looping pattern over a small region near top of the plot. In contrast, gnuplot's space-time graph does not reveal the source of the accesses to young pages. As it happens, in gnuplot's initial pass over the large memory region as it reads its input file, gnuplot accesses the region not quite sequentially—it periodically skips ahead of the main sequential line that the space-time graph shows. References corresponding to the left spike come from pages that are initially accessed *out of order* and are then accessed again in sequence—after enough time has passed that these pages leave ACTIVE state thereby necessitating IN records for subsequent accesses.

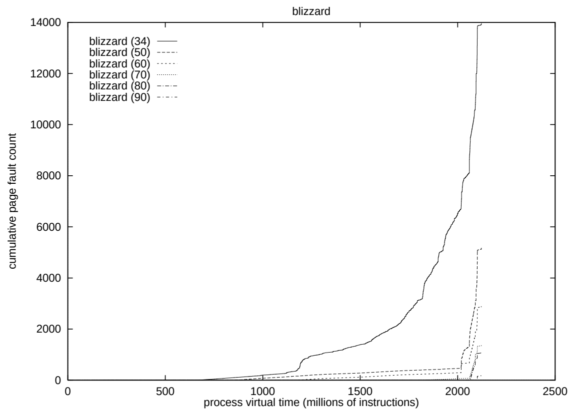
2.5 Cumulative fault rates

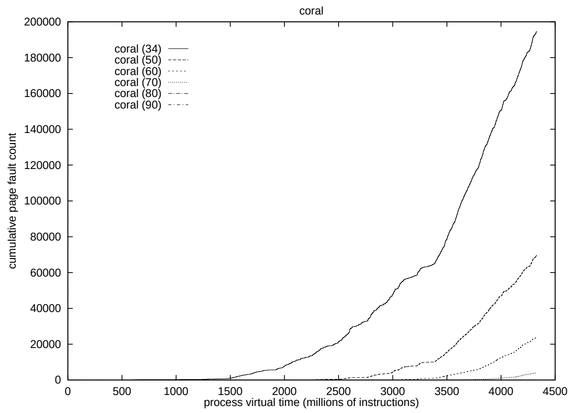
An interesting aspect of program behavior in a virtual memory environment is *page fault frequency*. Given a page replacement policy and a certain amount of physical memory, how frequent do page faults occur? In this section, the applications are examined under the ubiquitous LRU page replacement policy for several memory sizes, and the curves of cumulative page fault count are given. Similar curves for different applications appear in [21].

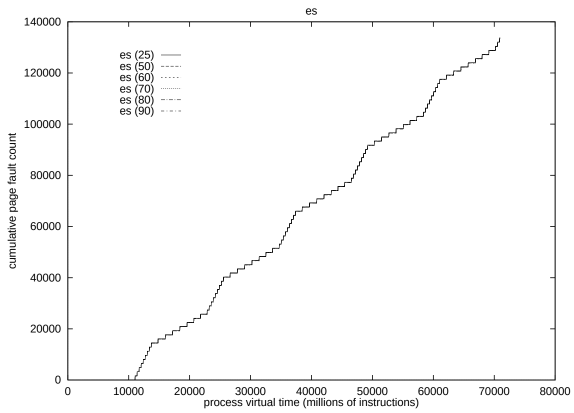
I simulated each application running with 25, 50, 60, 70, 80, and 90 percent of the desired amount of memory. (In a few cases 25 percent was impossible due to limitations imposed by the trace methodology, as described in Section 2.2.)

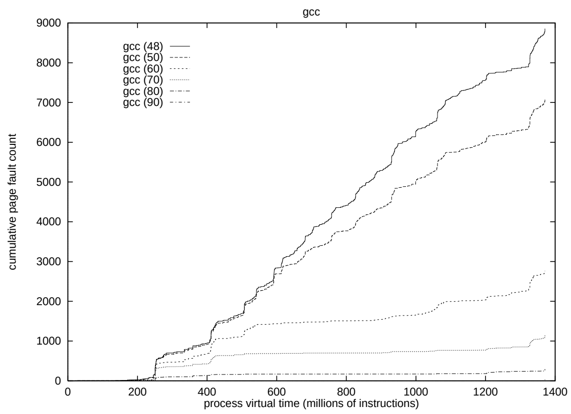
In some cases the curves for higher memory sizes may not be visible due to very low page fault count relative to lower memory sizes. In other cases curves may overlap; this is to be expected for massively-sequential programs for which LRU performs exceedingly poorly—adding memory eliminates no faults.

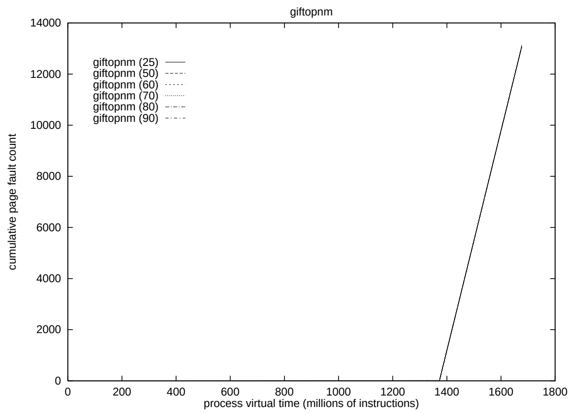


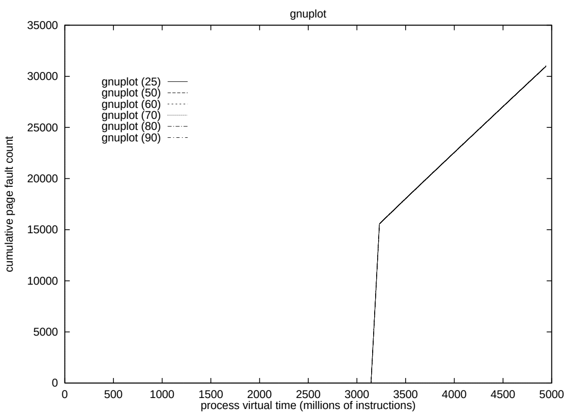


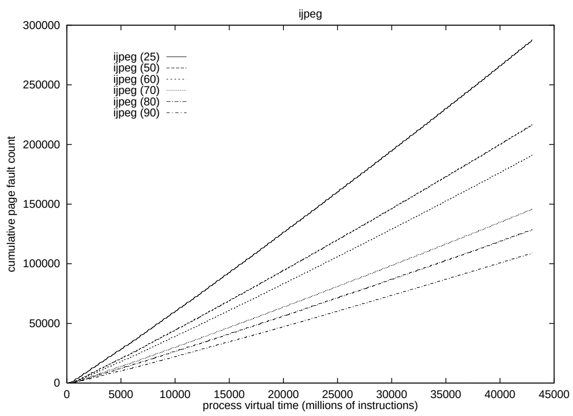


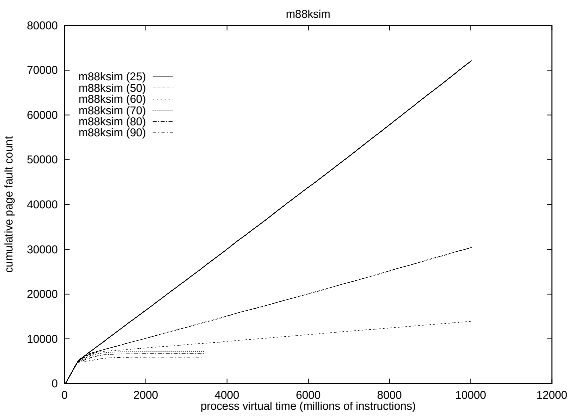


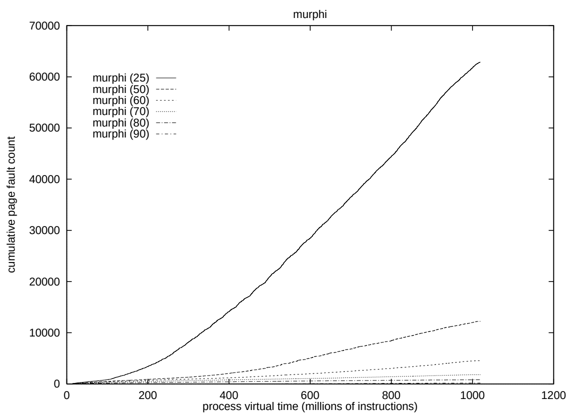


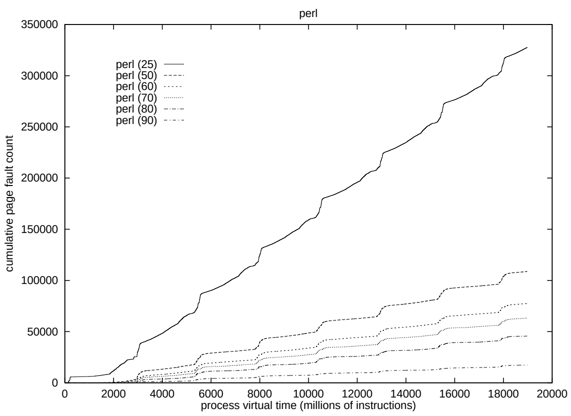


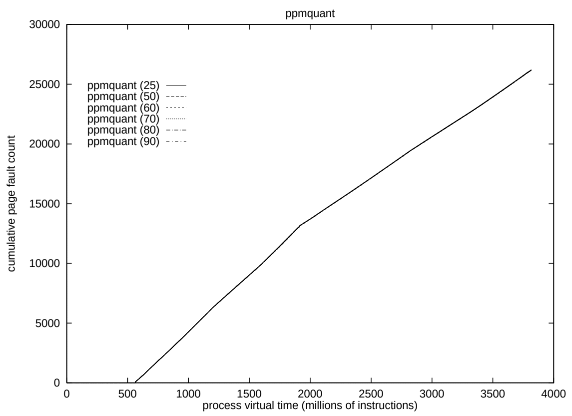

gnuplot
gnuplot (25)
gnuplot (50)
gnuplot (60)
gnuplot (70)
gnuplot (80)
gnuplot (90)
cumulative page fault count
process virtual time (millions of instructions)

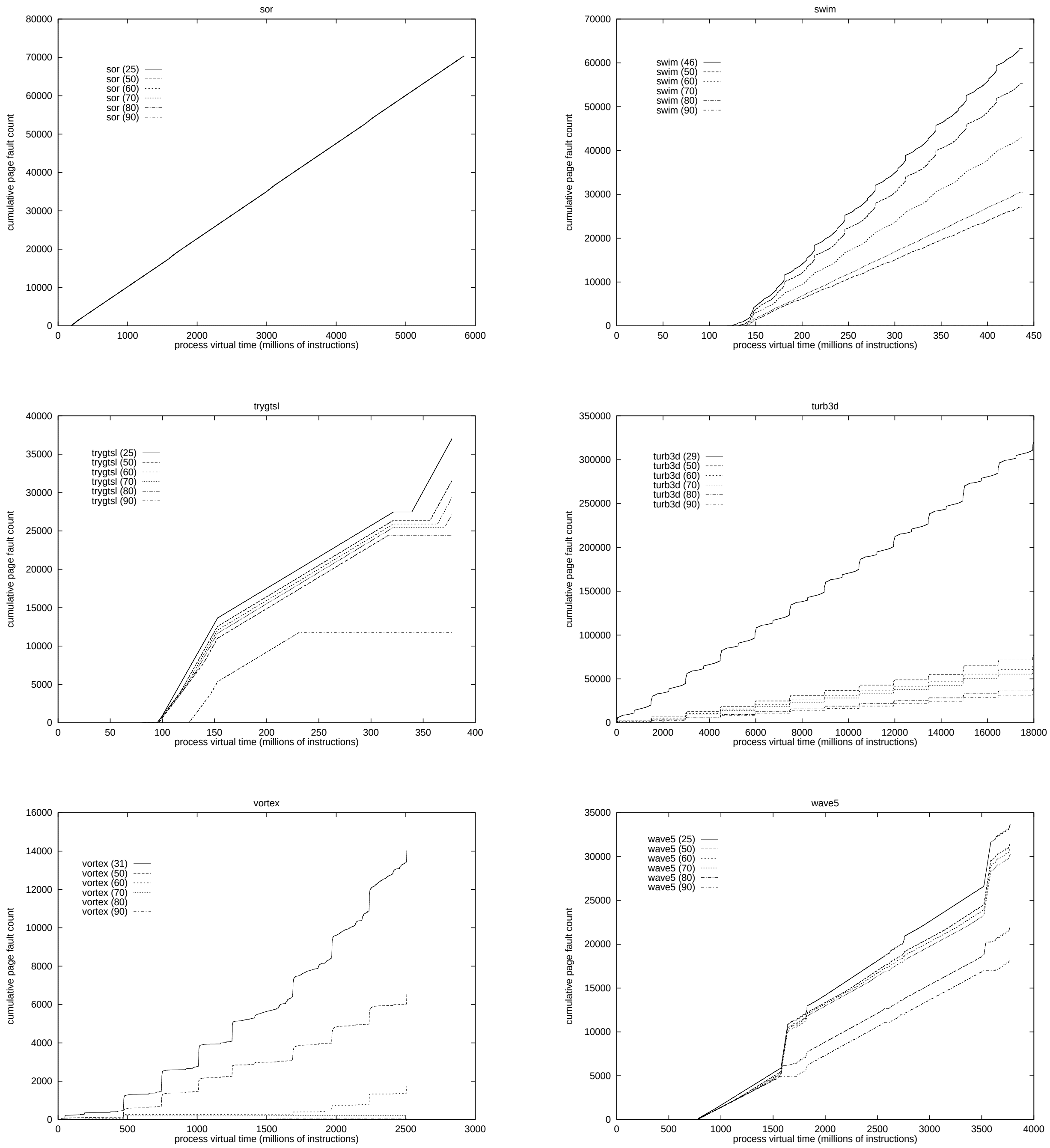

ijpeg
ijpeg (25)
ijpeg (50)
ijpeg (60)
ijpeg (70)
ijpeg (80)
ijpeg (90)
cumulative page fault count
process virtual time (millions of instructions)


m88ksim
m88ksim (25)
m88ksim (50)
m88ksim (60)
m88ksim (70)
m88ksim (80)
m88ksim (90)
cumulative page fault count
process virtual time (millions of instructions)


murphi
murphi (25)
murphi (50)
murphi (60)
murphi (70)
murphi (80)
murphi (90)
cumulative page fault count
process virtual time (millions of instructions)


perl
perl (25)
perl (50)
perl (60)
perl (70)
perl (80)
perl (90)
cumulative page fault count
process virtual time (millions of instructions)


ppmquant
ppmquant (25)
ppmquant (50)
ppmquant (60)
ppmquant (70)
ppmquant (80)
ppmquant (90)
cumulative page fault count
process virtual time (millions of instructions)



Jamrozik *et al*[21] observed that some programs encountered periods of high fault rate (at certain times in their executions) and other periods of low fault rate. We see similar results with our applications, particularly those of complex and mixed classes. Massively sequential programs, on the other hand, tend to display steady, high-magnitude fault rates; consider sor and ppmquant, for example. Plots for massively sequential gnuplot and mixed-class trygtsl contain basically straight lines, but the lines' slopes change, providing at least some evidence of phase changes.

For some complex-class programs, the curves for higher memory sizes are barely visible above the x-axis baseline (observe blizzard and coral). This suggests that LRU performs well for these programs: adding memory is an effective way to eliminate many page faults.

The high variance in fault rate evident from these graphs has implications for certain models of program behavior. A model may make an assumption about mean inter-fault times, that is, the average amount of time between faults. For the applications I studied, I observed that under LRU page replacement, standard deviations of mean inter-fault time were often an order of magnitude or more greater than the corresponding mean inter-fault times. Therefore, these applications are incompatible with models that assume exponentially distributed inter-fault times (for example, [6]).

2.6 Summary

This chapter has described eighteen benchmark programs from numerous application domains. Lengthy traces—complete end-to-end executions in all but three cases—have been obtained in a compact format that permits efficient virtual memory simulations.

We have looked at detailed plots of the memory references made by these programs, and have observed three general classes of application behavior: *massively* *sequential*, *mixed*, and *complex*, arranged in order of increasing reference stream locality. In addition to observing raw space-time data, we have observed LRU stack behavior in these programs; LRU stack behavior helps clarify the existence of the three behavior classes. Finally, this chapter has observed the changes in fault rate (under LRU replacement) that occur as execution progresses; these fault rate results are similar in nature to results from previous studies.

Chapter 3

SEQ: An Adaptive Replacement Algorithm

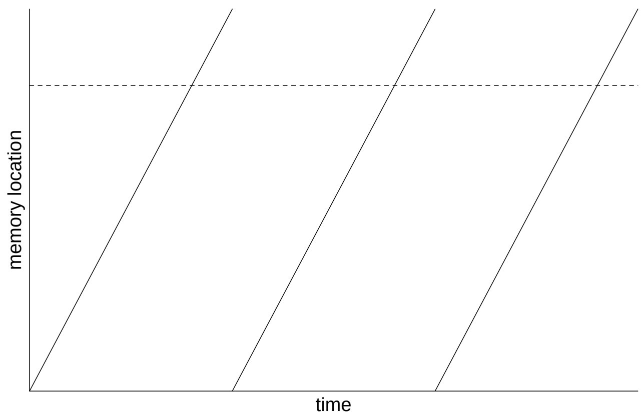
As we observed in Chapter 1, LRU page replacement typically performs poorly for massively sequential programs and often performs poorly for mixed sequential programs. The *SEQ* page replacement algorithm, the subject of this chapter, attempts to improve paging performance for massively sequential and mixed sequential programs. At the same time, SEQ tries to limit performance degradation for programs that LRU handles well.

SEQ takes its name from the fact that it observes *sequences* of page faults. On the assumption that page fault sequences indicate inherent sequential access in the direction of the faulting, SEQ replace pages in Most-Recently-Used (MRU) order from the fault sequences it detects. SEQ must be able to detect cases in which fault sequences *do not* indicate inherent sequential memory access behavior; in such cases SEQ must refrain from performing MRU replacement, or performance will suffer.

This chapter presents the SEQ algorithm first through an example of how it handles a massively sequential program. A second example shows how SEQ handles complex-class applications. After the examples, detailed pseudocode of SEQ's components provide insight into how the algorithm works, and reveal parameters that control SEQ's behavior. Following this, the SEQ parameter space is explored in some detail. Finally, I conclude this chapter with two additional methods for detecting cases in which SEQ should not apply MRU replacement.

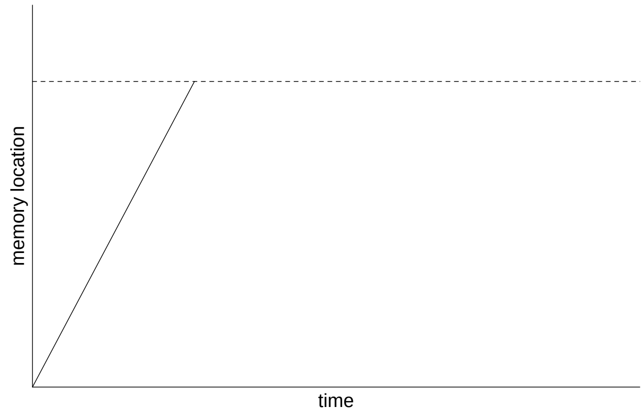
3.1 SEQ in Action

Consider the following massively sequential access pattern. As with the space-time graphs in the previous chapter, time increases along the x-axis and space, in terms of process address space, occupies the y-axis. Time and space units are omitted since they are immaterial for the purposes of this example. In the graph, the unbroken line indicates memory references. The dashed line indicates the amount of main memory relative to the process size—the desired memory size is about twenty-five percent larger than the available main memory:



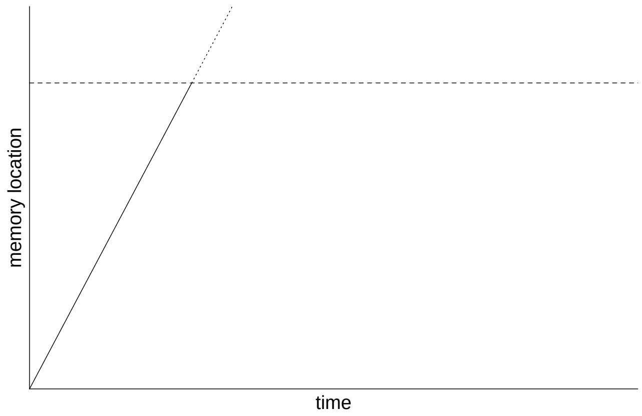
As the program accesses for the first time the memory region shown, pages are assumed to be faulted into the address space. SEQ notes these faults when they occur, and it records the address space range occupied by the faulted pages. As each new page is faulted in, the recorded range expands upwards by one page. As long as the process fits in main memory, SEQ does nothing more than record faults.

Here is the process after having created an address space large enough to occupy all available main memory:



At this point, since all of physical memory is occupied, a replacement decision will have to be made when the process faults in one more page. In this case, SEQ will replace a page from the "top" of the existing fault sequence. More precisely, it picks a page from the sequence's *head*—the most recently faulted end of the sequence. (To avoid replacing pages that are in active use—and hence would be faulted back in immediately after being replaced—SEQ actually replaces pages slightly behind the sequence head.)

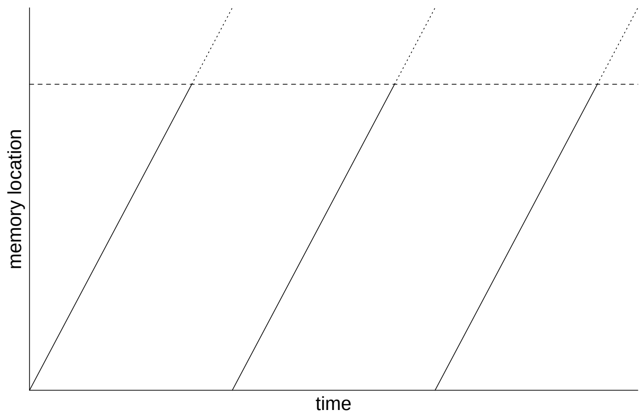
SEQ continually replaces pages from the sequence in MRU order as the sequence grows upwards. At the end of the first complete pass over the address space region, the bottom portion of the region remains in memory; in contrast to what LRU would do, SEQ has paged out the top of the region. The dotted line below shows the address space range that SEQ pages out; the unbroken line indicates the region that has remains in memory.



Successive iterations after the first proceed slightly differently. One difference is obvious: no initial faults occur, so accesses to the lower part of the memory region will proceed uninterrupted. Another difference is that when the program reaches the upper part of the region, it must fault in from the swap device the pages that had been replaced in the previous iteration.

The first of the faults to previously-replaced pages triggers an important aspect of the SEQ algorithm. On the program's second iteration, the first faulting page is at an address *inside* the range of the sequence that had been built on the first pass. SEQ handles this fault by *splitting* the original sequence into two new sequences. One sequence occupies the portion of the old sequence that extends from the old start up to, but not including, the new fault. The other sequence includes just the newly faulted page. In other words, on a fault to page P, if the sequence had originally been $[A..B]$, and assuming that faulting is going upwards in memory (as it is in this example), the two new sequences created are $[A..P-1]$ and $[P..P]$. (The next section describes sequence splitting in more detail.)

Pages that were replaced in the previous iteration fault into memory and extend the newly-created shorter sequence. SEQ chooses replacement pages from this new sequence: it chooses sequences based on the time of recent faults that extended the sequence. SEQ performs well for this hypothetical massively sequential program, replacing the upper address space region on each loop iteration:



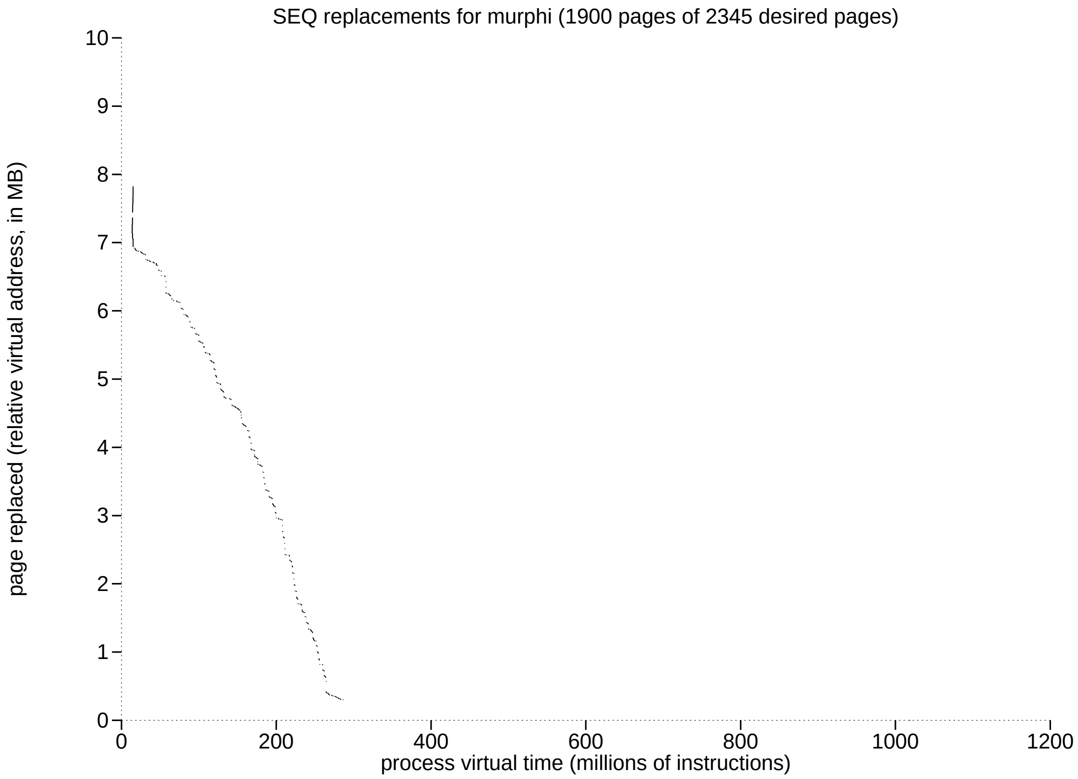
This example shows how SEQ works for a massively sequential program: sequence splitting helps determine which sequence to choose as a source of replacement pages—the younger sequence is chosen. Other techniques could be employed to choose replacement pages. We could avoid splitting the sequence and instead attempt to replace the most recently used page within each sequence. This approach has an advantage over SEQ. Since SEQ chooses the youngest sequence for replacement pages, a program that loops over two memory regions may suffer (for example, an outer loop and an inner loop accessing non-overlapping areas of memory). The problem, using the inner-loop outer-loop example, is that the inner loop will be replaced. However, with the possible exception of wave5, none of the applications used in this study exhibit this type of behavior[1].

Sequence splitting is a simple technique that, as we will see, is often effective at helping SEQ avoid performing MRU replacement on sequences in cases where MRU should be avoided. In a complex-class program that happens to display some

[1]Wave5 displays something of this sort of behavior, but the "inner loop" sequences are eliminated shortly after being created due to non-sequential accesses made to them. Refer to the space-time graph in Section 2.3.

sequential behavior, SEQ will replace pages from any sufficiently long sequences. However, when the program reverts to non-sequential reference patterns, SEQ-chosen pages will be faulted into memory. These faults will split the sequence, with the portion between the faulted page and the sequence head being "chopped off," thereby reducing the sequence's length.

Consider the graph below of SEQ replacements for murphi. From murphi's space-time graph (Section 2.3) we see that murphi makes one initial pass over most of its address space, followed by mostly complex, high-locality references.



SEQ replaces pages from the sequence head to the tail. As pages fault in, the sequence is shortened. Vertical spaces between replacements occur because SEQ does not replace precisely the MRU page—it leaves some number of pages in memory between the sequence head and the page that it replaces. (This is discussed below.)

From this graph we see that sequence splitting helps SEQ avoid doing MRU replacement too aggressively. In this example, SEQ performs slightly worse than LRU—with SEQ, 932 page faults occur, whereas LRU incurs only 782, a difference of about 20 percent. Most cases where SEQ performs worse than LRU result in smaller performance differences, as the results in this chapter and the next show.

3.2 The SEQ Algorithm

Both in abstract terms and in its implementation, SEQ can be divided into two distinct parts. One part, *seq_registerfault*, updates the fault sequence list at fault time. The other part, *seq_findreplace*, decides at replacement time which page (if any) to replace from a fault sequence. If no sequences exist or if none contain suitable replacements, then the system's default replacement scheme is used. (Typically the default scheme will be some variant of LRU.)

Before I describe *seq_registerfault* and *seq_findreplace*, I introduce a bit of terminology to ease the discussion. A *fault sequence* is a set of page faults to consecutive virtual pages; a fault sequence grows at only one end. That is, sequences are consecutive in both space and time. For any given sequence, the *head* of that sequence is the most recently faulted page in the sequence; the sequence's *tail* is the least recently faulted page. For a given sequence, if the *direction* of faulting is *upwards*, then the sequence tail is the page with the lowest virtual address, and the sequence head is the page with the highest virtual address. If the direction of faulting is *downwards*, then the head is the lowest page and the tail is the highest page.

3.2.1 Observing page faults

In SEQ, whenever a page fault occurs, *seq_registerfault* is called. For purposes of this thesis, I consider as a page fault any fault that adds a previously unmapped page to an address space. This includes faults that require I/O as well as faults that do not require I/O. An example of the former class would be a fault to an executable text page that must be satisfied from the program's disk image. Examples of the latter class include zero-fill faults (mentioned earlier) and faults to memory mapped files that are in the disk cache but are not (yet) mapped in the faulting process's address space. *seq_registerfault* has the following pseudocode.

```
seq_registerfault(page P):
    if (P overlaps an existing sequence S)
        handle_overlap(S, P)
    else if (P extends an existing sequence S)
        extend_sequence(S, P)
    else
        create a new sequence [P..P]
```

The auxiliary function *handle_overlap* implements sequence splitting. The function *extend_sequence* extends a sequence—by lengthening it by page in the direction of the sequence head. (A fault to the page adjacent to a sequence's *tail* will not extend that sequence.) *extend_sequence* handles cases where extending one sequence would lead to overlap of another sequence, or cases where either of two sequences could be extended by the specified fault. These functions are described in turn. *handle_overlap* has the following pseudocode:

```
handle_overlap(sequence S, page P):
    let [A..B] denote the bounds of S
    let H and T denote the head and tail of S, respectively
    if (P ≠ T)
        if (fault direction of S is upwards)
            create new sequence [A..P − 1]
        else
            create new sequence [P + 1..B]
    create a new sequence [P..P]
```

handle_overlap does sequence splitting if there are any pages *behind* the faulting page P—pages between P and the sequence tail. The pseudocode for *extend_sequence* follows. (Recall that S refers to a sequence that can be extended by page P.) The code is complicated by the need to check for overlap between adjacent sequences. One decision embedded in *extend_sequence* is that newer sequences have precedence over older sequences; if sequences "collide," the older is deleted.

```
extend_sequence(sequence S, page P):
    let [A..B] denote the bounds of S
    if (there exists another sequence S2 that P can extend)
        if (S has been modified more recently than S2)
            extend S by P:
            if (S is faulting upwards)
                S = [A..P]
            else        (S is faulting downwards)
```

```
            S = [P..B]
        if (S now overlaps S2)
            delete S2
    else        (S2 more recent than S)
        extend S2 by P (same as above, except extend S2, not S)
        if (S2 now overlaps S)
            delete S
else        (no other sequence can be extended by page P)
    extend S by P (same as above)
```

This concludes the code for *seq_registerfault*. The algorithm is basically straightforward book-keeping, although there are a few important, unobvious concerns. First, the number of sequences that are tracked at any given time—a parameter I will call X—needs to be bounded; otherwise, SEQ may consume excessive amounts of memory. This is an important concern in SEQ's design: SEQ must be possible to implement efficiently. To limit space consumption, the default value of X is 50; this was determined by examining simulation results for varying X values (these are presented later in this chapter).

A second issue with *seq_registerfault* is the goal of sequence splitting. As the massively-sequential example above showed, sequence splitting results in optimal performance for massively-faulting programs. This is clearly one motivation for doing sequence splitting. Another motivation for splitting is that if SEQ is replacing pages which are quickly being faulted back into memory, then the source sequences will—we hope—be split repeatedly (and quickly) until they are *too short* to be used as a source of replacement pages (as happened with the murphi example—where the splitting took a little longer than desired and resulted in 20 percent more faults). In general sequence splitting is how SEQ avoids performing poorly on programs that fault sequentially but do not really display iterative sequential behavior that would benefit from MRU replacement.

Sequence splitting suggests the second SEQ parameter: L, the minimum sequence length for sequences that can supply pages for replacement. The parameter L is used by *seq_findreplace* as described in the next section.

3.2.2 Choosing pages for replacement

The second half of the SEQ algorithm, *seq_findreplace*, is responsible for picking a page to replace. As noted earlier, if no pages can be replaced from fault sequences, the system's default page replacement policy (typically LRU) is used.

Several parameters affect *seq_findreplace*'s behavior. As noted above, L is the minimum *length* of sequences that will be examined for replaceable pages. Another parameter, M, is the *minimum distance* from the sequence head of replacement pages. M is motivated by the desire to avoid paging out pages that may still be in use by the application. Some programs, such as sor, do not merely step through memory one page at a time. For example, sor's five-point stencil calculation makes simultaneous accesses to three adjacent rows of a large (square) array [32] [2]. In any case, M should not be too large, or SEQ may not be invoked soon enough. I use a default value of 20 for L and M.

One other parameter control's SEQ's behavior. When *seq_findreplace* examines sequences for replacement pages, it must examine them in a certain order. It searches

[2]Each row will consume *rowsize/pagesize * sizeof(array element)* pages.

them based on the time of each sequence's Nth most recent fault. The *seq_findreplace* pseudocode illustrates this.

```
seq_findreplace:
    ignore sequences of length < L
    sort remaining sequences on the time of the Nth most recent
        fault for each sequence
    for each sequence S in the sorted list
        let the bounds of S be [A..B]
        if (S is faulting upwards)
            for each page P from B − M + 1 to A
                if (P is resident in memory)
                    return P
        else            (S faulting downwards)
            for each page P from A + M − 1 to B
                if (P is resident in memory)
                    return P
    return 0        (no suitable replacement page found)
```

The idea behind the parameter N is that sequences that are faulting steadily are more likely to be good sources of replacement pages than sequences that are faulting sporadically; a sporadically-faulting sequence will likely not have recent faults for $N > 1$. In practice, N does not influence performance greatly. The default value for N is 2. (N is perhaps the result of undue paranoia about never making incorrect replacement decisions; sporadically-faulting sequences do not seem common in practice, and even if they do occur, regularly-faulting sequences will be chosen most of the time even for $N = 1$.)

Having studied SEQ in detail, we are now ready to examine its performance.

3.3 Simulation results

This section shows performance results for SEQ page replacement, comparing SEQ against LRU and OPT (the optimal offline algorithm [5]). The applications we study are described in Section 2.1.

The following table lists the SEQ parameters and their default values used in the simulations in this section.

Parameter	**Default**	**Description**
L	20	minimum sequence length
M	20	minimum distance from sequence head
X	50	maximum number of sequences
N	2	*seq_findreplace*: fault time to sort by

As mentioned in Section 2.2, the nature of the memory reference traces prohibits simulation of very low memory sizes. In particular, precise LRU simulation demands the following conditions. First, a policy can never replace an ACTIVE page; such pages are considered to be in use, and if an ACTIVE page were replaced, there is no way in processing the trace file to tell when it would be faulted back into memory. Second, LRU must replace only the oldest IDLE page, which is easy to ensure since ACTIVE to IDLE transitions contain exact timestamps. Third, the oldest IDLE page must be older than all ACTIVE pages. This condition may be satisfied by replacing only IDLE pages that became IDLE two or more execution intervals in the past. Such pages are older than all ACTIVE pages because any currently ACTIVE page must have been referenced either in the previous interval or in the current interval. [3]

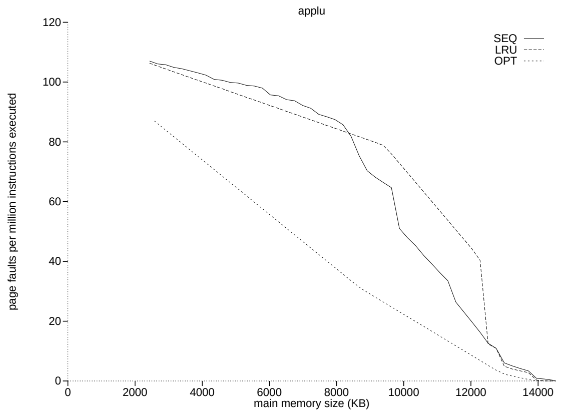
Restrictions similar to the ones above apply to OPT simulation. The only difference (relative to LRU) is that pages to replace may be referenced no earlier than two execution intervals into the future.

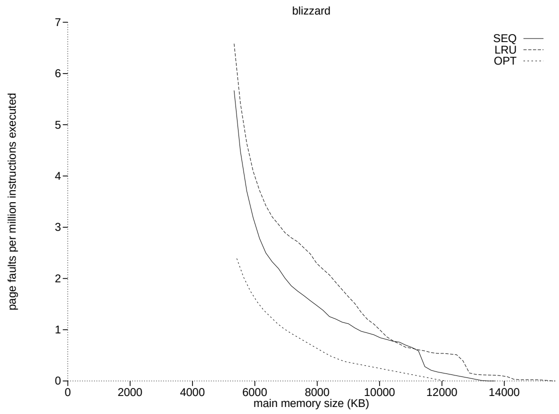
The graphs below show *page fault rate* as a function of main memory size. Page fault rate is measured in units of page faults per million instructions executed. A benefit of measuring page fault rate, as opposed to simply page fault count, is that rate can be translated, to a first approximation, into execution time. In addition, programs of vastly different execution time can be compared [4].

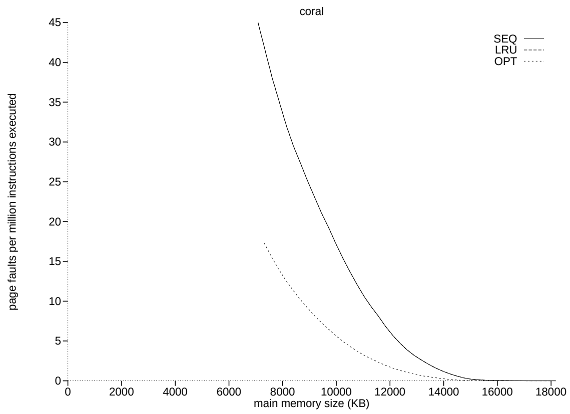
The requirements for accurate simulation restrict the memory sizes for which LRU replacement may be simulated. The graphs that follow indicate the minimum simulatable memory size by the leftmost x-axis position of the respective curves. For SEQ, the LRU-simulatable memory sizes were simulated. In the graphs below, the SEQ curves are indicated by unbroken lines; LRU and OPT curves are dotted lines, with OPT always below LRU. SEQ sometimes overlaps LRU, and other times almost overlaps OPT. If overlap occurs, SEQ's unbroken line indicates which curves overlap. (With these traces, LRU and OPT never overlap.)

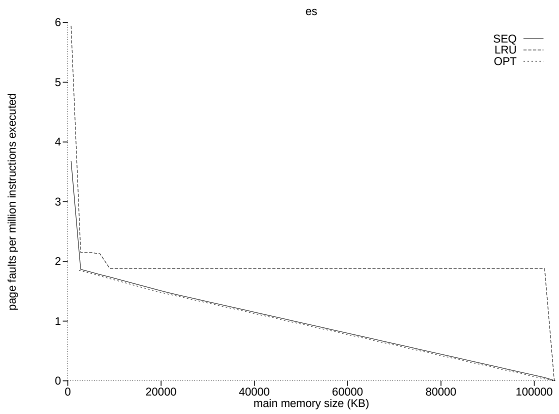
[3]We may be more aggressive and replace an IDLE page that became IDLE in the previous interval if we know that all currently ACTIVE pages became ACTIVE only *after* the IDLE page became IDLE. This never happens in practice because for all programs studied, at least a few pages remain in ACTIVE state for essentially the entire execution of the program.

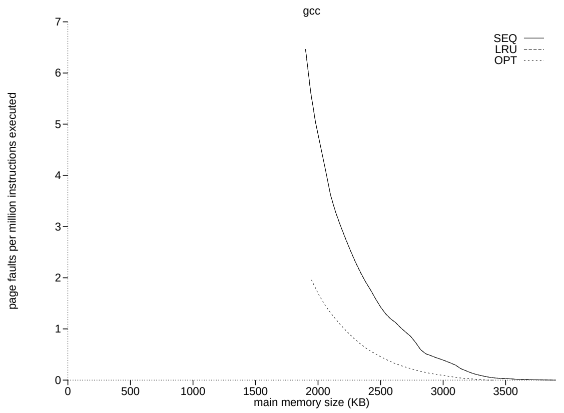
[4]The exact numeric values of the results depend on the compilers used, the machine architecture (SPARC v8), and so on. However, they can be translated into wall-clock time by assuming a particular clock speed and execution rate in terms of cycles per instruction. The instruction counts could also be scaled to account for differences between SPARC and another architecture of interest.

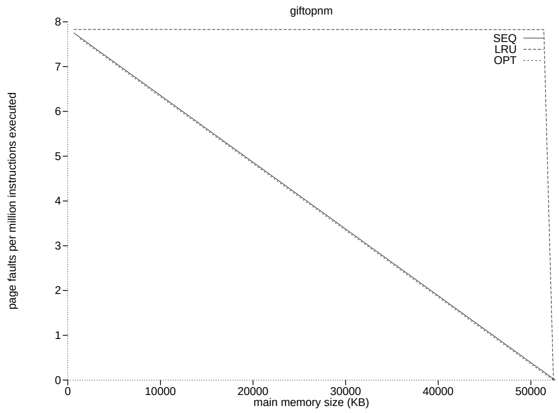

applu
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

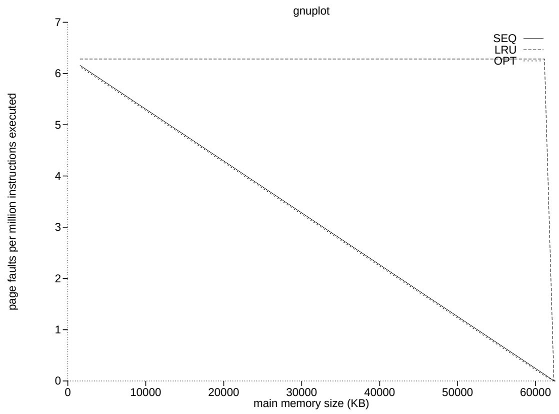

blizzard
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

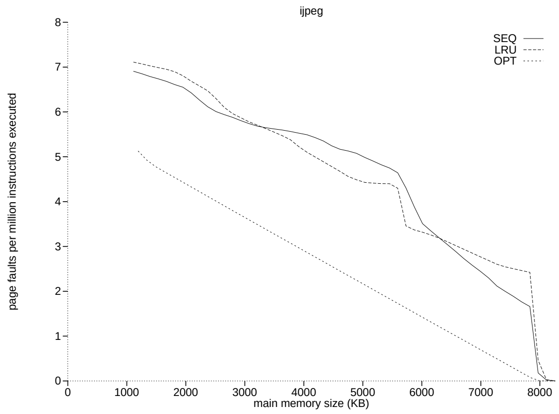

coral
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

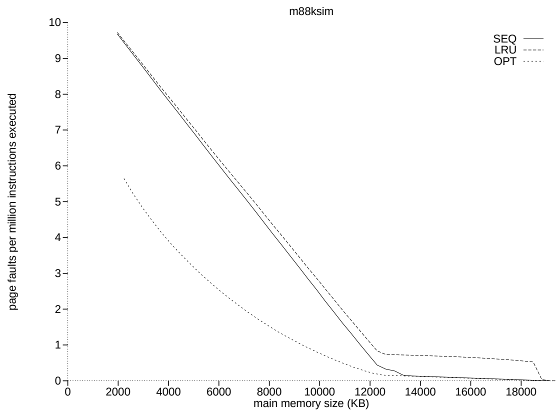

es
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

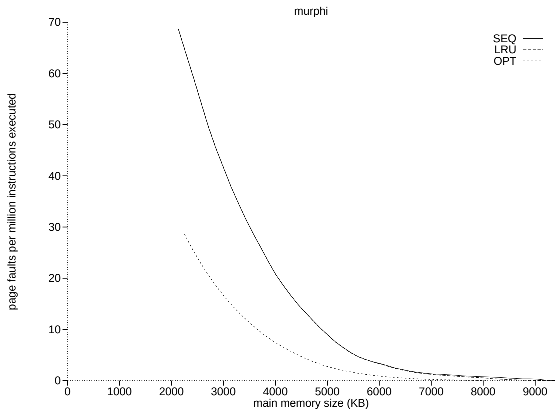

gcc
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

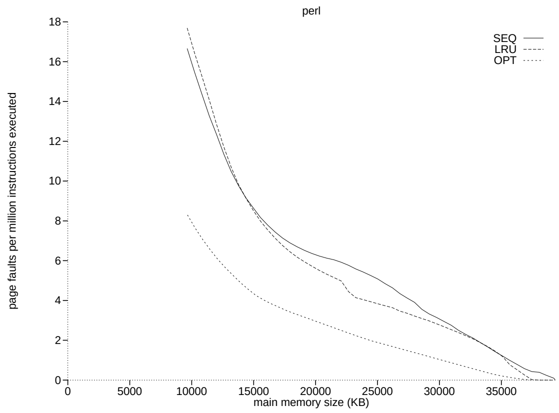

giftopnm
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

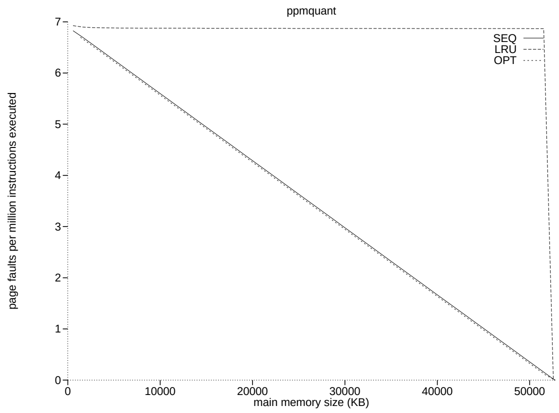

gnuplot
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

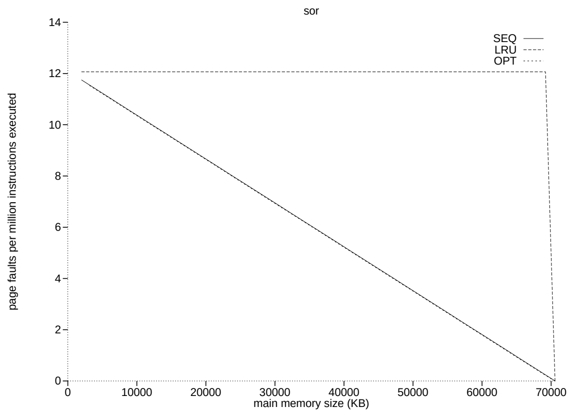

ijpeg
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

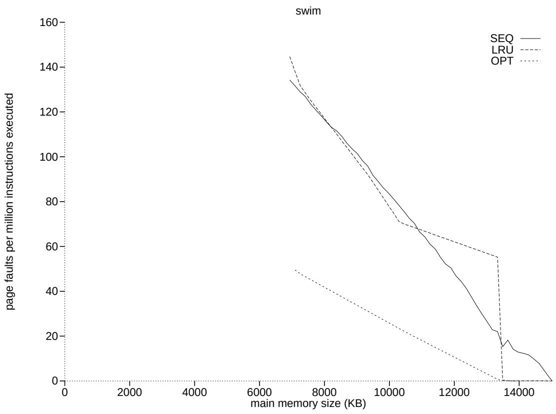

m88ksim
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

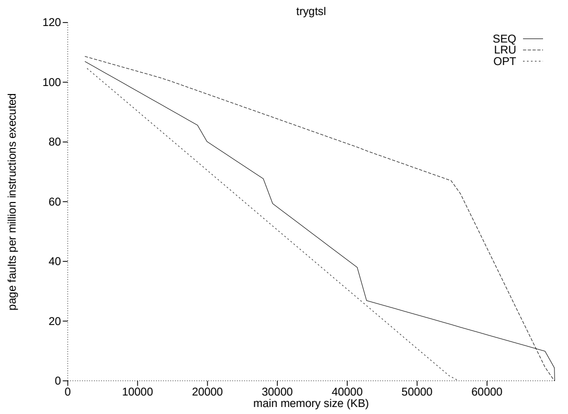

murphi
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

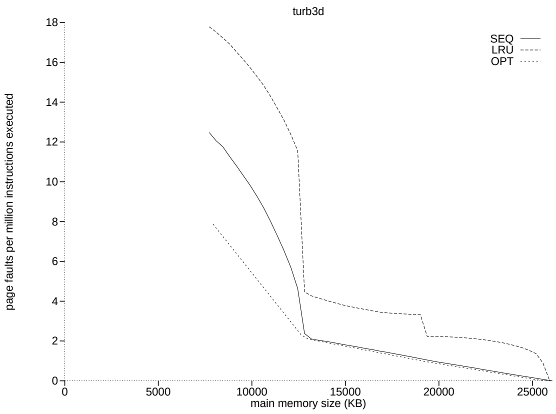

perl
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

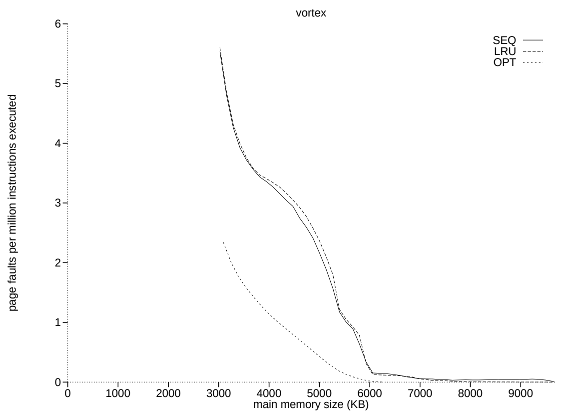

ppmquant
SEQ
LRU
OPT
page faults per million instructions executed
main memory size (KB)

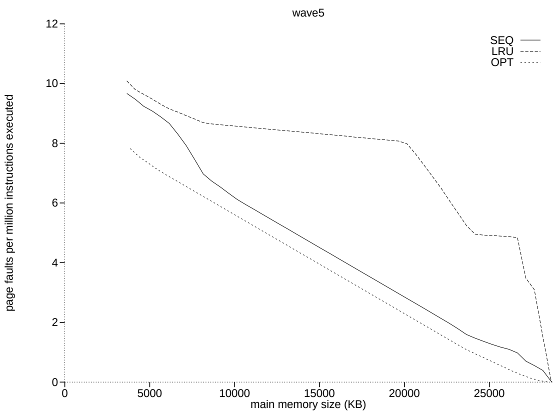












From these graphs, it appears that SEQ performs well for a variety of applications, particularly massively-sequential programs. Es, giftopnm, gnuplot, ppmquant, and sor achieve near-optimal performance. Some mixed complexity programs—trygtsl, turb3d, and wave5—perform very well under SEQ, while others perform moderately better or worse depending on memory size (applu, ijpeg, swim). Complex applications—coral, gcc, murphi, vortex—typically perform very similarly under LRU and SEQ. Blizzard performs a little better under SEQ, and perl somewhat worse at some memory sizes.

For these applications, then, SEQ appears to solve the LRU flooding problem without interfering significantly with complex-class, high-locality applications.

The next section explores SEQ's parameter space in more detail.

3.4 Varying the SEQ Parameters

We would like SEQ's performance, as observed in the previous section, to persist across a variety of parameter values, or at least not to degrade significantly. In this section, all four parameters (L, M, X, N) are varied and the results are compared to SEQ's results with the default parameters. The goals in this section are to observe whether changes in SEQ's parameters result in significant performance changes, and, if so, to observe and account for such changes—particularly changes that result in poorer performance. We would hope that SEQ performs robustly across variations in its parameters. Finally, if there are better choices than the defaults (as given in the previous section), we would like to incorporate them as new defaults.

In earlier work [17], we varied SEQ's parameters but used the same fault curve plot style used in the previous section. Differences in performance were observed to be minor, but the scale of those graphs makes the right end of the curve sometimes difficult to read. For programs with very high fault rates, the left and middle of the fault rate plots may not be interesting if the fault rate were high enough to make it impractical to run the program.

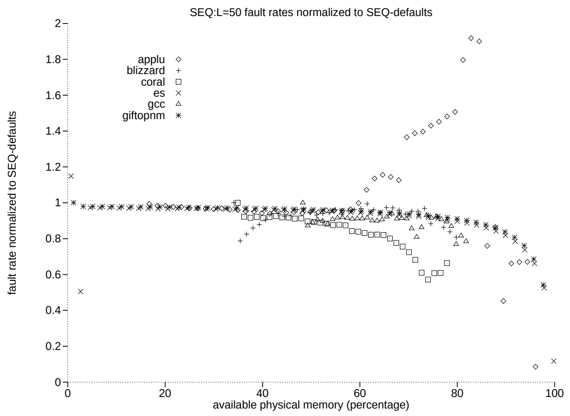
For this section we adopt a different graph style; instead of plotting fault curves against LRU and OPT, we plot fault count changes relative to SEQ with its default parameter values. This allows more subtle differences in the curves to stand out: often the differences arise at higher memory size values, and on the fault rate curves, the exceedingly high fault rates at lower memory values cause the relatively higher-memory small fault rate values to squeeze against the x-axis.

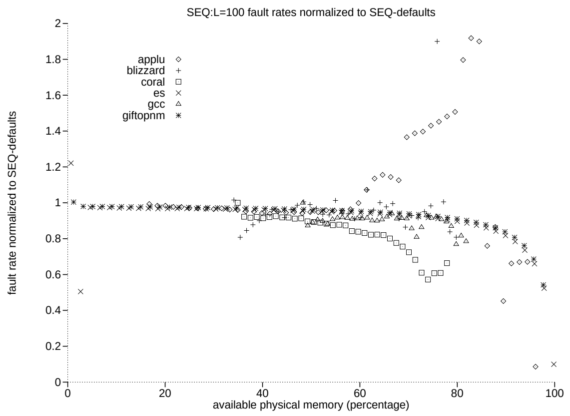
In all the graphs in this section, the following default parameters are used unless otherwise noted: $L = 20$, $M = 20$, $X = 50$, and $N = 2$. I illustrate performance by comparing SEQ performance with modified parameter values normalized to SEQ performance with all default parameter values.

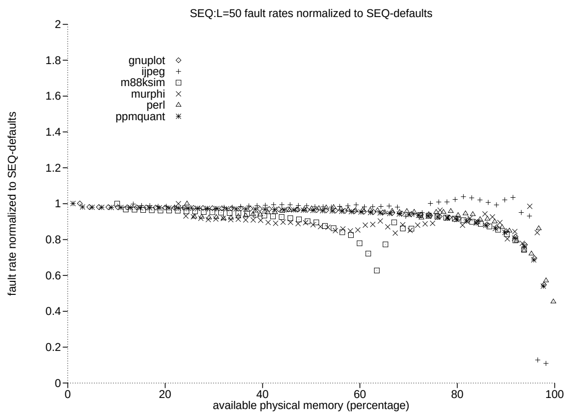
The graphs in this section omit results for runs where the policies differ by less than 200 page faults. 200 page faults would typically take less than one second of execution time on a circa 1997 personal computer.

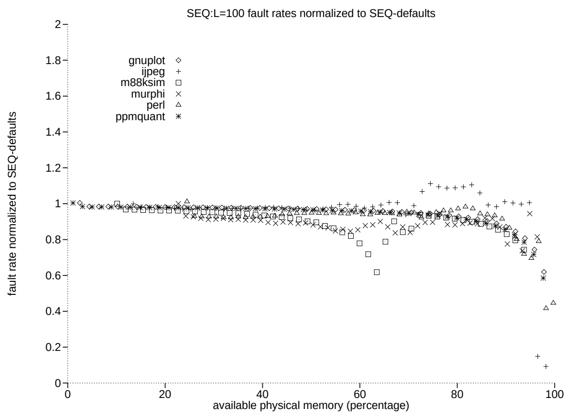
The first parameter I varied is L. L is likely the most important SEQ parameter for several reasons. First, for practical reasons, X and N require relatively low bounds. (X because of storage overhead associated with individual sequences; N because of storage overhead associated with individual fault times: the times of the last N faults must be maintained.) Second, M must be less than or equal to L: if M is greater than L, then no pages can be replaced from any sequence shorter than M, because any page in such a sequence is not M pages away from the sequence head. Third, if M is too high, SEQ will not be able to perform effective MRU page replacement on sequences.

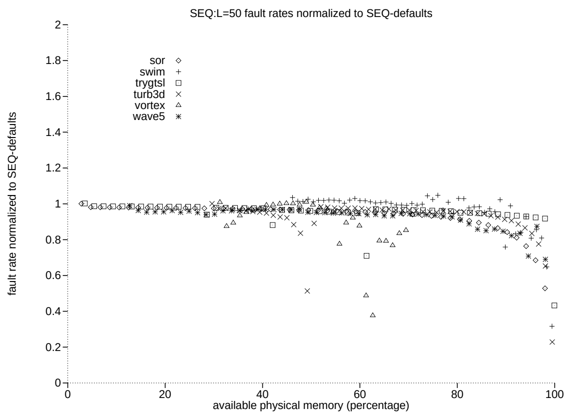
The following graphs show SEQ performance as L increases from the default value of 20 to 50 (left graph) and 100 (right graph). The two runs are presented side by side for ease of comparison.

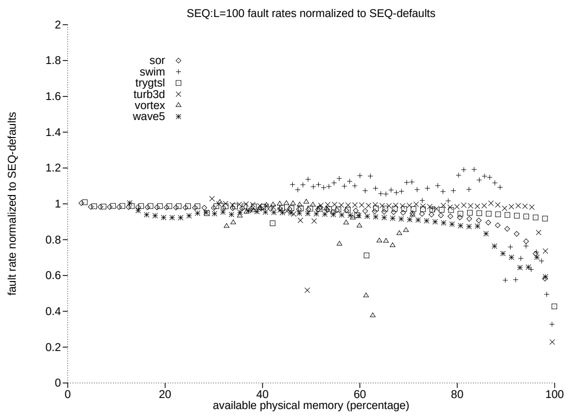








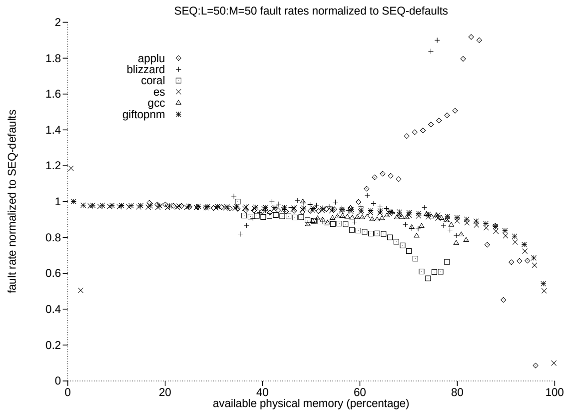


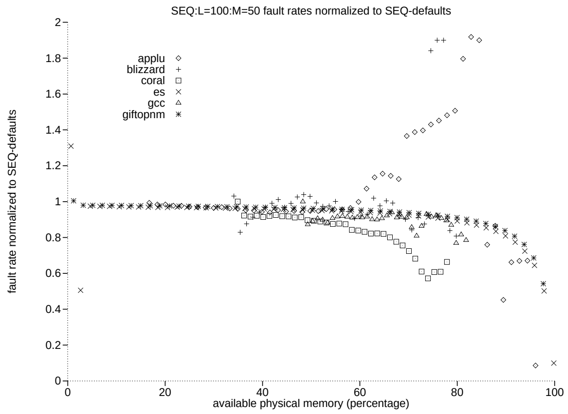


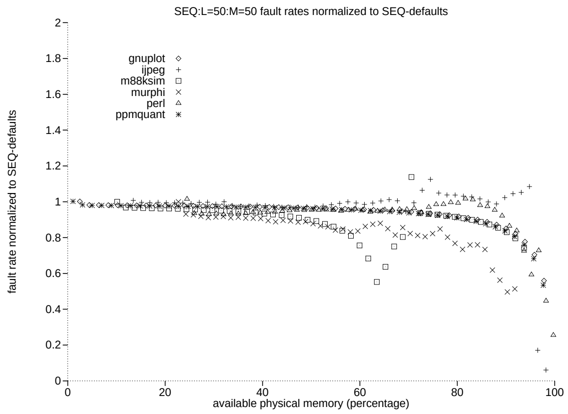
The primary difference in performance is that for both $L = 50$ and $L = 100$, SEQ suffers for applu. (In fact, the default for L had originally been 50, but was changed to 20 after observing these results.) Since SEQ sometimes performs worse than LRU for applu, changes that hurt applu's performance are undesirable. Other changes in the above graphs include, for $L = 50$, improvements for coral and m88ksim over a visible range of memory sizes, and sporadic improvements for other programs as memory size increases.

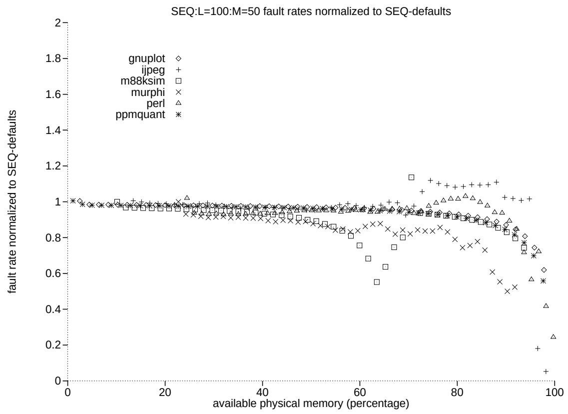
For $L = 100$, coral performance improves, but two mixed class programs (swim and ijpeg) suffer slightly. Improving coral should not be a goal since coral is a complex class program for which LRU performs well. Of greater concern is avoiding hindering mixed class programs where SEQ and LRU may differ. The value of L seems to have an impact in this regard: fault sequences in applu and swim are often between 20 and 100 faults long. (This behavior is not obviously visible on the space-time graphs in Chapter 2, but I have verified it by observing SEQ's runtime behavior and its sequence state.) Performance for applu, ijpeg, and swim varies noticeably with varying L, so it is probably possible to find a program for which $L = 20$ performs poorly. Thus we see that sequence splitting is not always successful at preventing poor replacement decisions. This concern will be addressed in the next section.

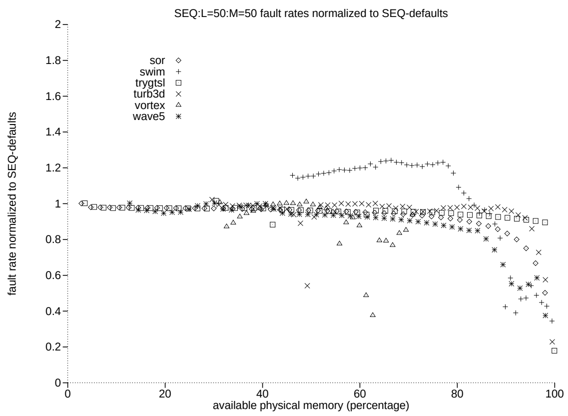
The next parameter I varied is M. As noted above, M is an effective lower bound on L, so L should be greater than or equal to M. Following are results for two runs, with $L = 50$ and $M = 50$ for the plots on the left, and $L = 100$ and $M = 50$ for the plots on the right.

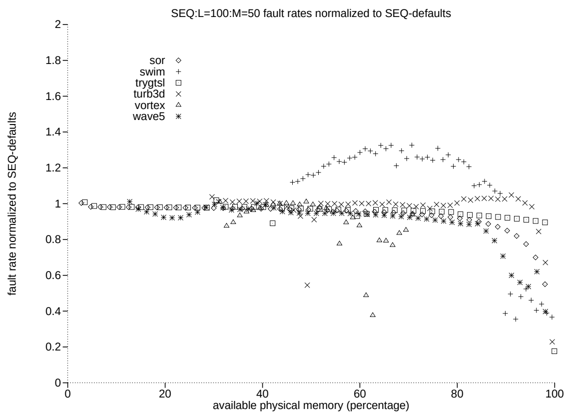










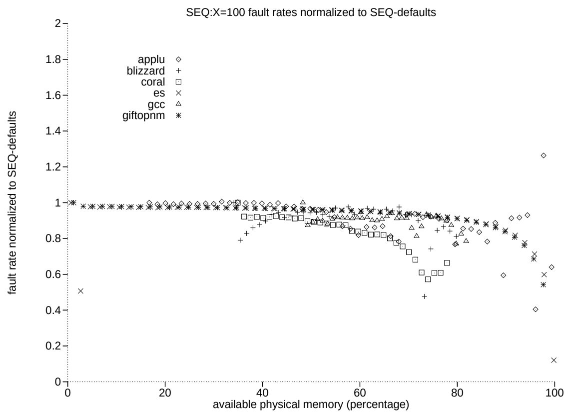


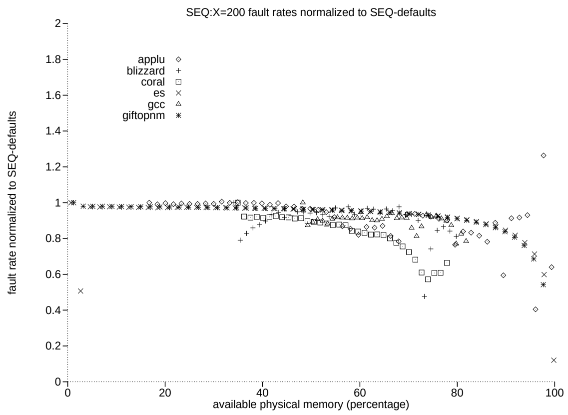
Once again, applu performance suffers—fairly substantially, varying about linearly up to about twice as bad as SEQ with default parameters. Also, for these runs, swim suffers 20–30 percent more faults, across a wider range of memory sizes than for the previous graphs ($L = 100, M = 20$). In addition, for these runs, ijpeg's fault rate increases slightly. These losses are partially offset by occasional improvements at high memory sizes for a number of programs.

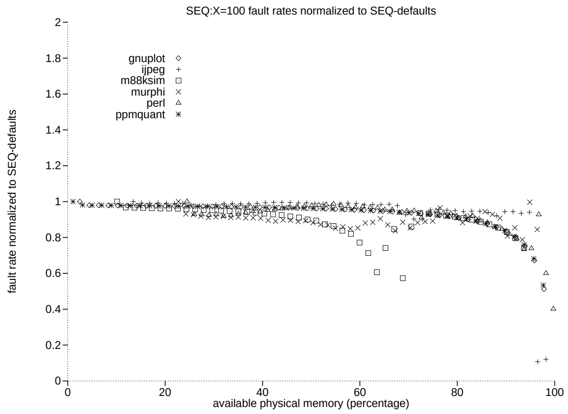
From these sets of runs, it appears that performance for mixed-class programs can, at times, depend on changes to L and M. For most programs, though, performance is fairly consistent across parameter changes.

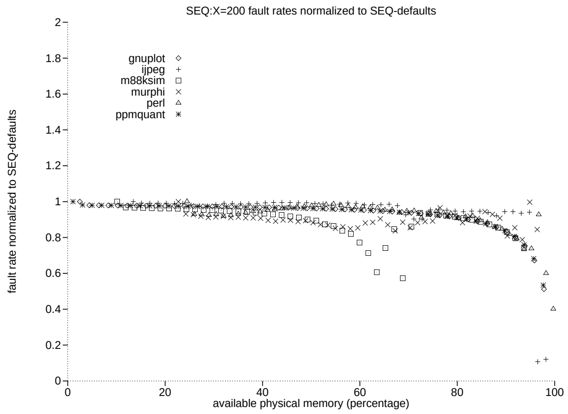
The next sets of runs vary X, the parameter controlling the maximum number of simultaneous recorded sequences (by default X is 50). An important motivation for keeping X small is that SEQ's runtime space overhead is directly proportional to X: information about X number of sequences must be stored. (In my SEQ prototype, described in the next chapter, each sequence requires 92 bytes of state; this could be reduced modestly in a more optimized implementation.)

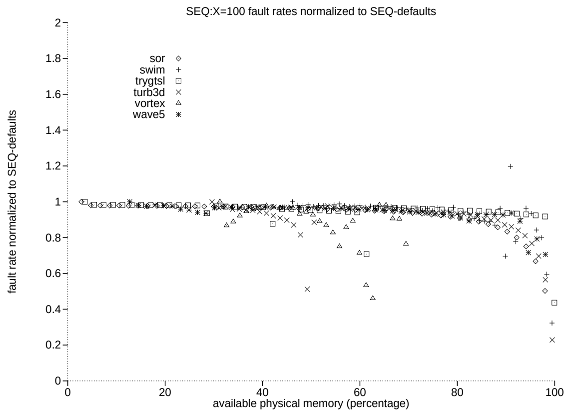
On the left are results for $X = 100$, and on the right are results for $X = 200$. (Other parameters are not varied here.)

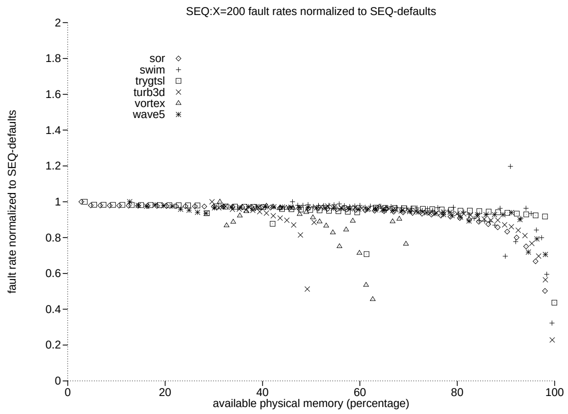






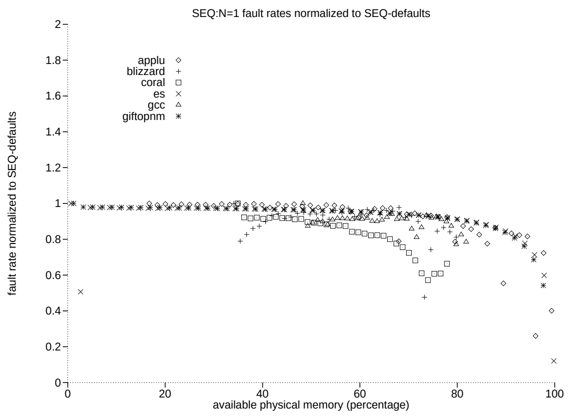


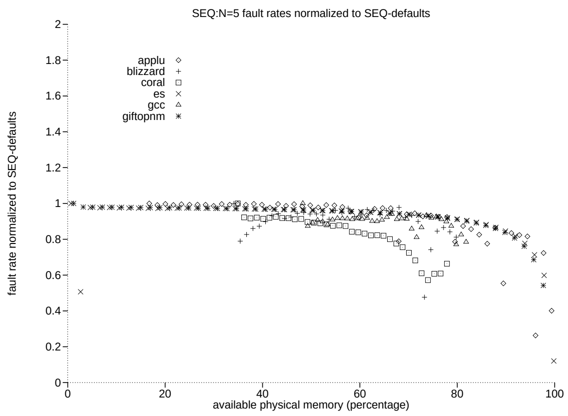


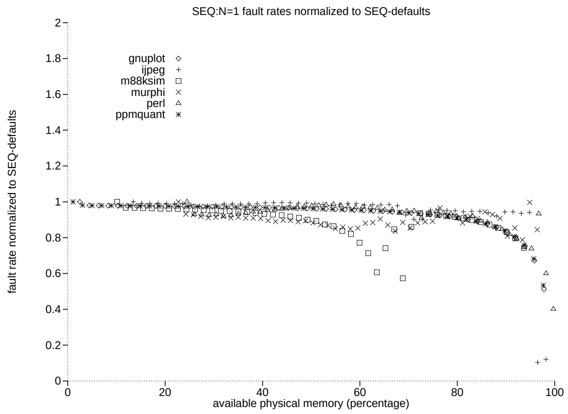


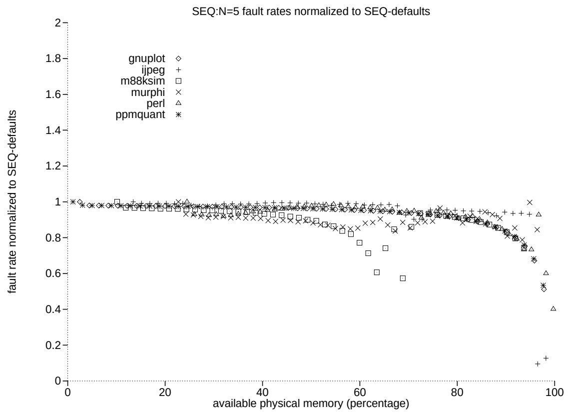
Nothing changes for the worse, ignoring one isolated bad case each for applu and swim (occurring on both graphs). No changes for the worse is good news, indicating that $X = 50$ is not merely a lucky choice. On the other hand, the changes for the better indicate that some performance gain may be obtained, at the cost of higher runtime space overhead.

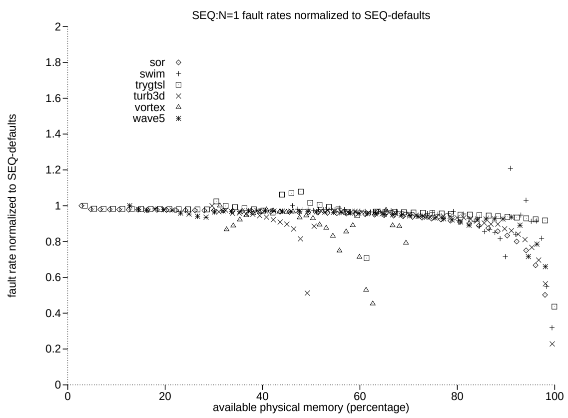
Finally, here are plots for varying N. Recall that *seq_findreplace* examines candidate sequences by the time (newest first) of each sequence's Nth most recent fault ($N = 2$ is the default). In the very earliest SEQ implementation, I actually set $N = 20$. Results for that case are similar to those for $N = 5$. The left graph has results for $N = 1$; the right graph for $N = 5$. Other parameters have the default values.

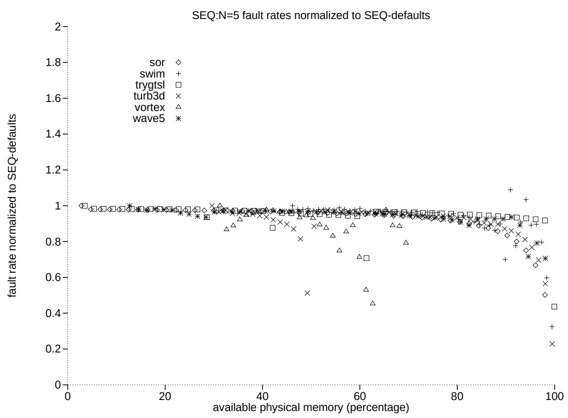












Performance improves slightly for m88ksim and coral in both cases. Apart from that, and a few scattered improvements at the right ends of the curves, performance does not change significantly. (Perhaps the default for N should be changed to $N = 1$.)

In summary, SEQ has been fairly consistent across changes to its parameters' values—almost all parameter changes have resulted in performance changes of less than 50% above or below the default cases, with most performance differences being within 25%. More parameter variations could be studied; for example, L and M could be lowered (to, say, 10), as could X (to 30 or 40).

Most observed cases of performance degradation occur with changes to L. The programs affected were mixed-class applications applu, ijpeg, and swim. These programs sometimes lack the complex-class access patterns that are necessary for sequence splitting alone to restrain SEQ from making bad replacements. The next sections consider strategies for restricting SEQ most effectively.

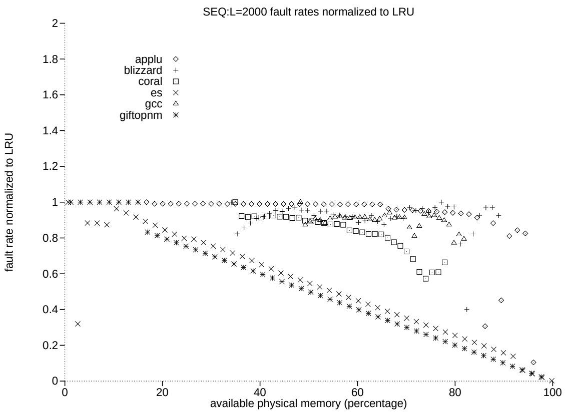
3.5 Throttling SEQ

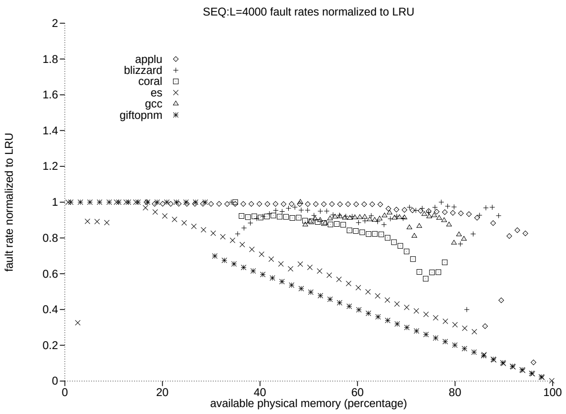
The previous sections revealed a few programs—for example, perl, applu, and swim—for which SEQ sometimes performs worse than LRU. Furthermore, some of these programs are sensitive to changes in SEQ parameters L and M, suggesting that there may exist programs that exhibit pathological behavior *for the default L and M values*—a worrisome case. This section explores additional techniques that SEQ can use to avoid performing significantly worse than LRU. These techniques complement SEQ's existing sequence-splitting technique.

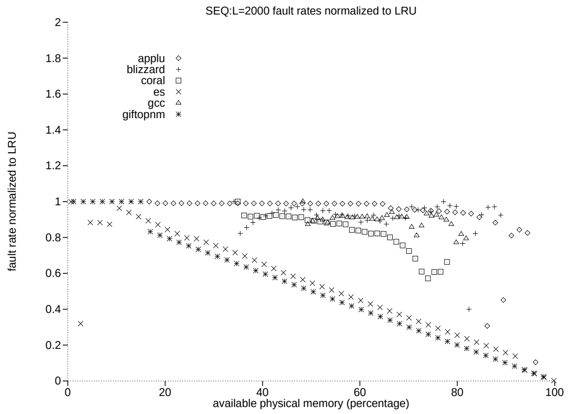
One simple approach to limiting SEQ's potential damage might arise from the following line of thought:

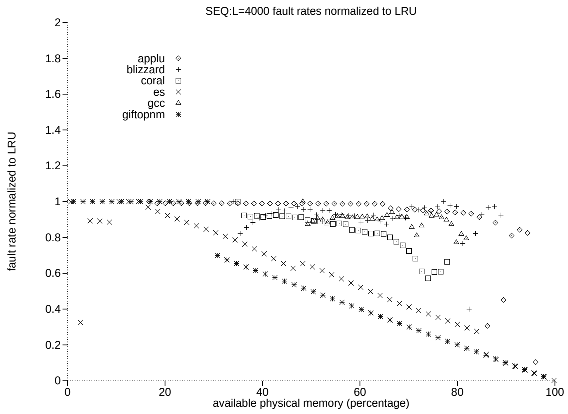
- massively sequential programs seem common
- SEQ handles massively sequential programs optimally
- let LRU/default policy handle all mixed/complex application classes—those whose fault sequences are not exceedingly large

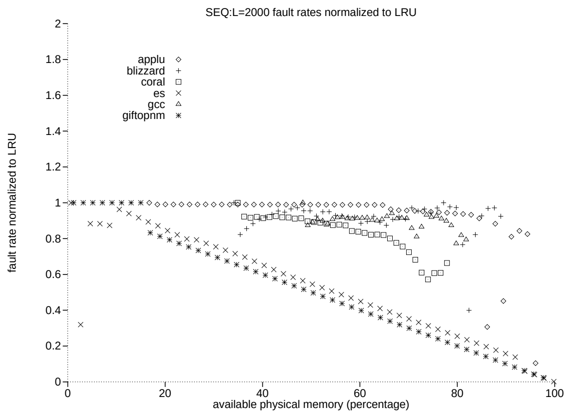
A simple way to decrease the likelihood that SEQ will replace sequences from mixed-class applications—and incur poor performance—is to inflate L to be well above the length of sequences likely to be present in such applications. The following two experiments set L to 2000 (left-hand graphs) and 4000 (right-hand graphs). Other parameters retain their default values ($M = 20, X = 50, N = 2$). SEQ fault counts here are normalized to LRU fault counts.

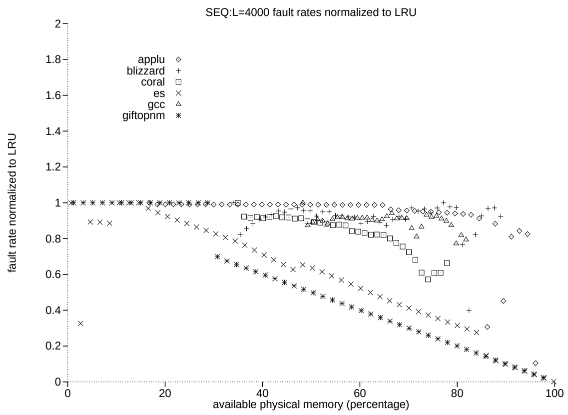












Clearly, inflating L effectively throttles SEQ for applications it had previously handled poorly. However, inflating L to such a high value may be overkill; applications that may benefit under SEQ with small L may lose out. For example, wave5 performs worse in these cases than it does under default SEQ parameters.

Another way to apply SEQ page replacement more discriminantly is to allow SEQ to be enabled or disabled on an application-specific basis. Going even further, applications could alter their own L or M (or even X and N, if desired) parameters. (The implementation described in the next chapter includes these features.) However, application-specific parameter settings assume that programmers, users, and/or system administrators are sufficiently interested and sophisticated to take the time to find good SEQ parameter settings—or even merely to bother to enable or disable SEQ. What we would like is an automated way for SEQ to throttle itself when it discovers that its replacement choices result in performance worse than LRU. The next section presents an alternative approach that requires no user, programmer, or administrator input.

3.5.1 The Reference Race

The optimal offline replacement algorithm introduced by Belady [5] works by replacing the page that will be referenced the farthest in the future. When SEQ replaces a page, the hope is that the page replaced will not soon be referenced; in particular, we hope that the SEQ-chosen page will be referenced *after* the LRU-chosen page will be referenced.

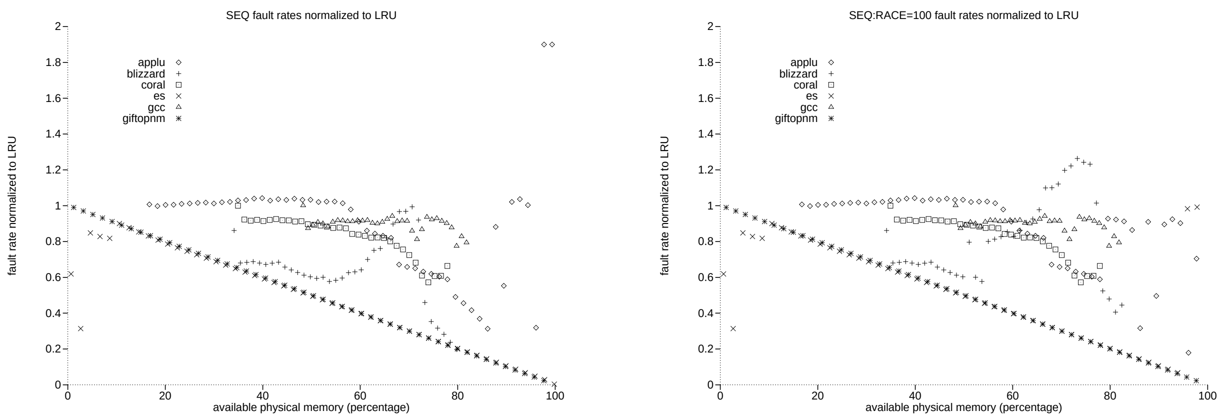
A SEQ implementation can dynamically measure SEQ's performance by enacting what I will refer to as a *reference race*. When SEQ chooses a page to replace (the "SEQ page"), the page that LRU would have chosen (the "LRU page") and the SEQ page are entered into a race. If the program references the SEQ page before it references the LRU page, then LRU wins the race: SEQ has made a poor replacement decision, incurring a page fault earlier than LRU would have (if it had incurred a fault at all). Conversely, if the LRU page is referenced before the SEQ page, SEQ wins, since it avoided a page fault that LRU would have taken. (The reference race as described essentially borrows the concept of a *placeholder* from Cao's LRU-SP policy [7].)

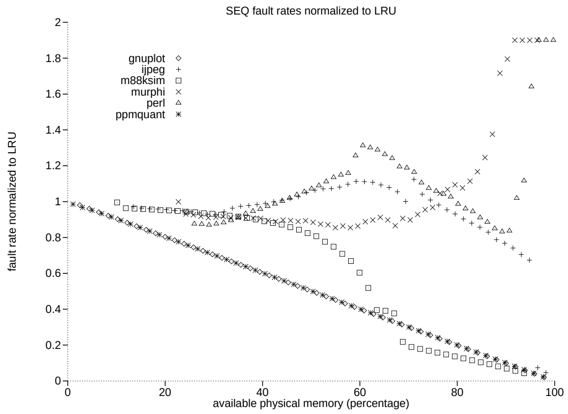
A few other details complete the description of the reference race. Suppose that at some point a reference race is set up and that the LRU page is not accessed. If at a later time SEQ chooses as a replacement this previously designated LRU page, the original race is considered a tie, since in the original race neither page was referenced earlier before both pages ended up on disk. (A new race involving the newly-chosen SEQ page and the current LRU page is set up as usual.)

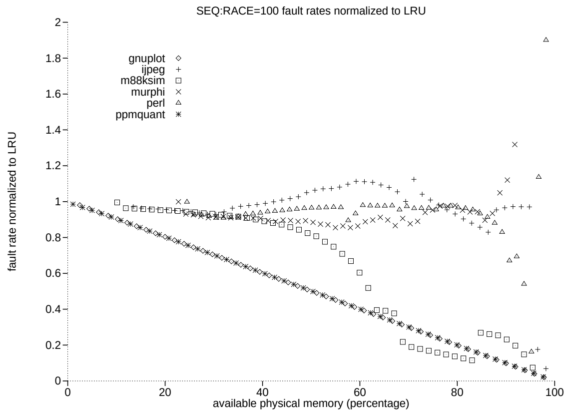
The difference between the number of races won by SEQ and the number won by LRU is SEQ's *score*. When SEQ's score drops below a certain threshold (e.g, −100), SEQ replacement is disabled for the application. The idea is that applications which perform poorly under SEQ will lose enough reference races early into their execution and thereby not suffer significant performance degradation.

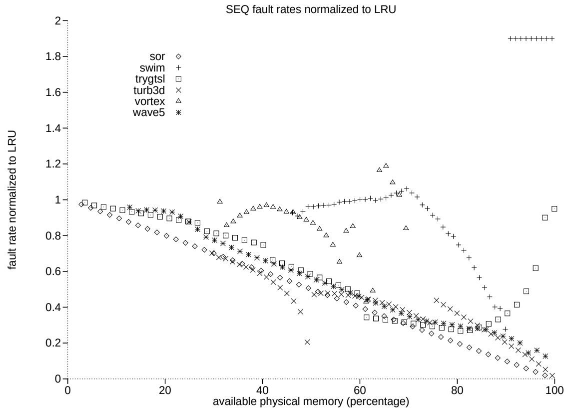
The following experiment shows SEQ's performance relative to LRU both with and without reference races (left graphs without and right graphs with reference races). In the runs with reference races, SEQ was disabled as soon as its score dropped below –100. SEQ parameters were their default values in both runs.

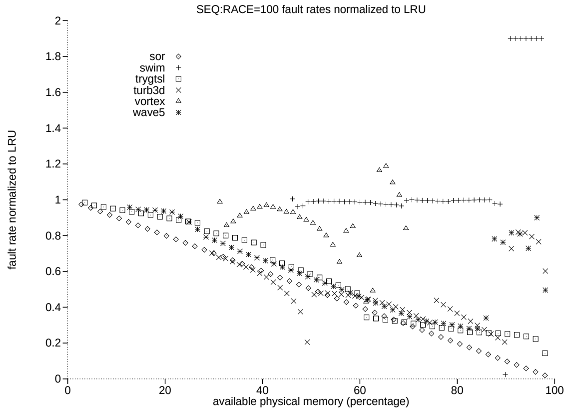
The graphs of the left hand side may be useful for comparing with the previous section's graphs. (If desired, one may correlate SEQ parameter changes with performance relative to LRU.)











With the reference race enabled, SEQ avoids poor performance for murphi, perl, and swim. Blizzard and vortex suffer, but only for a few memory sizes. Performance for the massively sequential programs is still excellent: observe that the diagonal lines indicate that SEQ remains close to OPT—the number of faults is the difference between desired memory size and actual memory size; the difference decreases linearly with increasing actual memory size. In summary, the reference race appears to be a good way to limit potential damage caused by over-aggressive SEQ replacement.

3.6 Summary

This chapter has introduced SEQ, a new algorithm for adaptive page replacement. SEQ's design has been described in detail. Briefly, SEQ works by observing sequences of page faults and performing MRU replacement on such sequences. SEQ relaxes MRU replacement by splitting sequences when page faults occur inside fault sequences.

Simulation results show that SEQ performs very well—often indistinguishable from OPT—for the massively sequential programs studied. For most complex applications studied, SEQ manages to restrain MRU replacement and as a result performs similarly to LRU (in a few cases performing worse). For some mixed-class applications SEQ performs quite well (e.g. wave5) and for others (e.g. applu) performance hovers near that of LRU.

SEQ retains its performance for a modest set of variations to its input parameters. Importantly, performance does not suffer substantially in any of the parameter changes studied.

Finally, I have introduced two techniques by which SEQ's MRU replacement can be more aggressively throttled. One simplistic technique relies on changing an important SEQ parameter. A more sophisticated technique incorporates a key concept from previous work and achieves good results, performing worse than LRU only in a few isolated cases, while not sacrificing SEQ's excellent performance on massively sequential programs.

Chapter 4

An Implementation of the SEQ Algorithm

The previous chapter demonstrated that the SEQ algorithm performs well in a simulated virtual memory environment. This chapter demonstrates that SEQ performs well in a real operating system kernel implementation, resulting in speedups of 50 to 100 percent for massively sequential programs and some mixed applications, without significantly reducing performance for other classes. Moreover, this chapter demonstrates that SEQ can be implemented efficiently in terms of minimizing time and space overhead.

This chapter describes a SEQ implementation done in the context of Linux, a freely-distributable UNIX operating system. I used Linux version 2.1.29, the most recent release available when this work was performed.

This chapter begins with an overview of how Linux manages memory. I then describe the kernel changes necessary to incorporate SEQ. In addition, I describe a simple implementation of swap prefetching that takes advantage of SEQ's knowledge of sequential access patterns. I have not (as of this writing) implemented the reference race technique introduced in Section 3.5.1.

Following the Linux prototype description, I report performance results—end-to-end execution times, page fault counts, and fault service times—comparing SEQ's performance (with and without prefetching) to unmodified Linux LRU replacement. Several sets of results are given, first for single applications running alone and then for a few combinations of applications running together. Finally, several additional benchmarks demonstrate that SEQ's time and space overheads are minimal.

4.1 Linux Memory Management

Any given of main memory can serve at different times one of many purposes in Linux [1]. However, apart from being used internally by the kernel, pages primarily serve one of three purposes:

- **buffer cache**: Linux maintains a cache of buffers corresponding to device blocks (the *buffer cache*[3]); the buffer size corresponds to the underlying device sector size, 512 bytes. The buffer cache primarily holds filesystem metadata (inodes and directory information).

- **page cache**: caches the contents of files that reside on filesystems. Note that data is never in the both the page cache and in the buffer cache.

- **process address spaces**: Finally, pages of memory may be devoted to storing the memory image of current processes. Note that address space pages for memory-mapped files (mapped via the *mmap* system call [40]) reside in both the respective address space (or spaces) and in the page cache.

Page replacement in Linux is implemented by the function *try_to_free_page*, which cycles through each source of memory consumption, requesting each source to reduce its usage. *try_to_free_page* is invoked when free memory runs low—below 128

[1] In the Intel x86 version of Linux I used, the page size is fixed at 4KB.

pages on a machine with 64MB. Both a dedicated kernel thread, *kswapd*, and the general purpose kernel memory allocator, *get_free_page*, invoke *try_to_free_page*. (Because *try_to_free_page* can sleep while paging out data to disk, *get_free_page* may be called with a flag to indicate that the request should fail if no memory is available; this allows various race conditions to be avoided.)

Process address space pages are paged out by the function *swap_out*, described below. I will not discuss the replacement mechanism and policy for page cache pages and buffer cache pages, since SEQ only applies to address space pages (and address space pages that are present in the page cache). In the SEQ implementation, I did not change the emphasis Linux places on freeing page cache and buffer cache pages as compared to freeing address space pages. (In general, though, Linux prefers to retain address space pages in memory.)

Unlike some other UNIX implementations, Linux does not maintain a mapping from a given physical page to all the users of that page, in particular to all the processes that may be sharing that page. As a result, *swap_out* cannot be implemented by traversing the putative reverse mapping structure: in order to free a page, *swap_out* would need to clear the "valid" bit on each page table entry mapping that page—but *swap_out* does not have that list of page table entries available. Instead, *swap_out* works by iterating over each process's page table, freeing pages that are deemed to be sufficiently old. Shared pages are freed by being removed one at a time from each sharer's address space.

In more detail, *swap_out* iterates over the list of processes; for each process, *swap_out* calculates a maximum number of pages to free in the current iteration: 32 pages per megabyte of resident memory per process. *swap_out* then traverses the process' page table, applying a linear aging algorithm to each page. If a page's age reaches 0, the page is written to backing store (if necessary) and returned to the free pool. The page aging process has the following algorithm. (Note that the numeric constants in the pseudocode below are the default values, which I left unchanged; these values may be tuned at run-time, but I ignored this capability.)

```
age_page(page P):
    if (P's reference bit is set)
        if (P.age < 20 − 3)
            P.age = P.age + 3
        else
            P.age = 20
        return          don't free this page
    else            page not referenced
        if (P.age > 1)
            P.age = P.age − 1
        else
            P.age = 0
    if (P.age = 0)
        replace P
```

When a page is first added to an address space, its age is set to 3.

4.2 Adding SEQ to Linux

The pseudocode above offers a hint at how to integrate SEQ. Effectively all replacement decisions are made by the last two lines of the pseudocode above—if a page's age has reached 0, the page is replaced. SEQ fits quite naturally into this structure (this replaces the last two lines of *age_page* above):

```
replacement decision (page P):
    if (P.age = 0)
        page S = seq_findreplace()
        if (S ≠NULL)
            replace S
        else
            replace P
    else            age not 0: do nothing
        return
```

Recall from Chapter 3 that *seq_findreplace* is the SEQ function to retrieve a replaceable page from a sequence. (It returns NULL to indicate that no replaceable pages are available.) Adding the code corresponding to this pseudocode required about 40 lines of code.

The SEQ integration shown above avoids a difficult policy decision. The code above does not change the decision of how many pages to replace for each process (i.e., at most 32 pages per megabyte per pass). As a result, even with SEQ enabled, Linux still does a good job of quickly paging out inactive processes. One can imagine a naive

SEQ implementation whereby at replacement time, if *any* SEQ page were available, then that page would be immediately replaced. A user running a massively sequential process would suffer because idle processes—including very old, inactive ones that definitely should be paged out—would remain in memory since every replacement could be satisfied from massively sequential process's sequences.

4.3 Additional modifications

Other modifications required to add SEQ to Linux are as follows. I added three one-line changes to insert calls to *seq_registerfault*. The three one-line additions correspond to three types of page faults: zero-fill faults (minor faults), faults to memory-mapped files, and faults to pages that have been previously replaced and currently reside on a swap device. Two other modifications were necessary to add per-process SEQ initialization calls (for the *fork* and *exec* system calls [3, 40]). Finally, one additional modification was necessary: in the *exit* system call, the per-process SEQ state is deallocated.

I made several other minor modifications to SEQ. After observing small (e.g. one to two percent) slowdowns for things involving the creation of a large number of processes—in particular, kernel compilation—I changed the SEQ implementation to record faults only for processes occupying more than 4MB. (This parameter can be changed like any other SEQ parameter.) Since 4MB is far less than the amount of memory a process would need to consume before paging incurs, SEQ does not suffer, and commonly-run applications do not suffer (in my case, these applications are the C compiler front end, the compiler back end, the pre-processor, and assembler).

To speed up *seq_findreplace*, I changed SEQ to record on a per-sequence basis the start and end of on-disk pages in that sequence. These are the pages that previous *seq_findreplace* calls had returned. Adding this cache of known previously-replaced pages speeds up *seq_findreplace* because it helps avoid the cost of traversing a large range of on-disk pages looking for a replacement. Sometimes replacement pages must come far away from the sequence head—whenever the rate of sequence grown is slower than the memory demand. Since Linux tries to replace a number of pages at a time, there are periods of high instantaneous memory demand, which may exhaust pages near the heads of active sequences. In cases like this, the MRU replacement must continue—we just want to avoid having to examine hundreds or thousands of previously-replaced pages.

I made one final change to SEQ. Similar to the phenomenon of high instantaneous memory demand is the issue of temporary memory availability. If a sequence is being faulted in, and sufficient free pages exist to store the faulted pages, then SEQ need not replace pages behind the sequence head, even as the sequence grows. Thus a *gap* develops between the sequence head and the start of on-disk pages contained within that sequence.

This gap causes minor difficulties on any subsequent iterations over the pages in the sequence currently under consideration: if the entire sequence is not paged out due to gap pages here and there, then on a subsequent iteration, the pages will not fault in together as one long, contiguous sequence, but rather as a number of smaller sequences. As a result, because of SEQ's M parameter, for each small sequence, M (default 20) pages must remain in memory. Thus a modest number of free pages must be obtained from some other source, typically (for the case of a massively sequential program) from the single, large in-memory sequence below the point of the initial sequence split.

To avoid the gap problem, I changed SEQ to attempt to replace pages (near the sequence head) that were contiguous with the existing on-disk region within that sequence. On subsequent iterations the single on-disk region will be faulted in as a single large sequence. The whole problem relating to the gap within on-disk pages is only an issue for programs that make a large number of iterations. For example, before adding the change mentioned above, I observed sor on a run requiring 20 iterations. Sor initially faults—and SEQ replaces as a sequence—a large contiguous range, the upper portion of sor's address space above the point that can fit in memory. On subsequent iterations a few gaps develop, but they widen since the first M pages of each sub-sequence remain in memory. The number of gaps grows, and at 20 iterations, the range of gap/on-disk pages has become about twice the size of the original on-disk region. Forcing SEQ to replace forwards from the on-disk region fixes this problem.[2]

Linux offers a useful facility for reading and writing kernel configuration parameters at run-time. As part of the */proc* filesystem, a directory hierarchy mirrors major kernel subsystems. Leaf nodes (that is, files) of the tree represent one discrete group of related configuration parameters. A system call (*sysctl*) may be used to read or write a named set of parameters. Alternatively, the traditional system calls—*open, read, write, close*—may be used on the leaf node file itself. I added an entry */proc/sys/vm/seqctl* to allow easy access to SEQ control parameters, including a new boolean parameter to turn SEQ off or on. To create the *seqctl* entry I merely had to add two lines of code to a static array initializer, setting up the equivalence between the *seqctl* entry and a global *seq_control* record.

[2]This change does not really violate MRU replacement, since the gap between on-disk pages and the sequence head only grows temporarily—when there happen to be a few free pages to hold the newly faulted pages at the sequence head. Since the gap only grows temporarily, replacing forwards from the on-disk region quickly catches up to the page M pages away from the sequence head.

With this implementation, SEQ parameters can be changed using either the *sysctl* system call, or by simply writing to the appropriate */proc* entry. I found the latter method to be quite convenient as I could turn SEQ on or off from the command line, or from various scripts used to obtain timing measurements.

4.4 Swap Prefetching

SEQ makes it easy to detect when a process is referencing memory in sequential fashion. If we detect sequential access to paged-out data, an obvious thing to do is to attempt to prefetch the data into memory before it is actually needed, thereby mitigating the latency suffered by a page fault. Prefetching depends on at least two things: temporary memory available to hold the prefetched data, and enough inherent computation to overlap the latency necessary for the prefetched read disk requests to complete.

I implemented a simple swap prefetching scheme in Linux. At the end of the page fault service routine, after the page has been read from the swap device, the fault address is, as usual, registered with *seq_registerfault*, which now returns the length of the fault's sequence. If the address lies in a sequence of at least ten pages, the fault handler issues prefetch requests for subsequent pages in the sequence. The number of requests issued is the minimum of the sequence length and sixteen. (I did not vary these settings.) These numbers are fairly aggressive; sixteen pages (64KB) seems to be a good tradeoff between access latency and transfer size [16].

With prefetching, the page fault handler now has to check if the faulting address is the subject of an outstanding prefetch request. If so, then the fault handler merely waits for the disk read to complete (if it is still in progress) and then retrieves the page from the prefetch module. The total number of outstanding prefetches is limited to 64 (across all sequences—each sequence is still limited to 16), but this limit was rarely encountered in practice.

Each prefetch request is handled in the following manner. The prefetch module must allocate a page into which it can deposit the prefetched data. For this purpose the prefetch module calls the *get_free_page* memory allocator, but with the "atomic" flag to prevent *get_free_page* from descending into *try_to_free_page* if no pages are available. If no pages are available, i.e., the free page count is below the threshold set by the variable *min_free_pages*—128 pages on a 64MB machine—then *get_free_page* returns a failure status code and the prefetch code drops the request. If a page is available the request is issued to the swap device asynchronously. After all prefetch requests have been issued, control returns to the application.

4.5 Results

This section presents the results of running the benchmark applications introduced in Section 2.1. The programs were run under the default Linux replacement policy ("LRU"), under SEQ without prefetching ("SEQ"), and under SEQ with prefetching enabled ("SEQPREF"). Most programs were run under varying memory sizes, although not enough to obtain complete curves like the ones in Section 3.3.

I first present results for running each program by itself; following this, I present results for several program combinations. I believe that the single-process results are more important than the multi-process results for the following reason. If a user needs to run a program that simply cannot fit in memory, he is essentially at the mercy of the page replacement policy in his operating system. On the other hand, when attempting to run multiple processes simultaneously, the user can intervene and terminate a process if virtual memory performance is poor. In any case, SEQ performs well in the multi-process case.

The system on which these results were obtained has the following characteristics. The processor is a 133MHz Intel Pentium; 64MB of main memory is present along with a 256KB L2 cache. A single SCSI disk contains a Linux EXT2 filesystem (containing all programs and data) and a 128MB swap partition (no auxiliary swap files were used). The disk is a 1GB Quantum VP31080 rotating at 5400rpm with average seek time of 8.5ms and rotational latency of 5.55ms. The disk has an internal 512KB track buffer cache. The SCSI chain is driven by an ncr53c810-based controller operating at 5.0MB/s.

A sequentially faulting micro-benchmark achieves approximately 270 page faults per second in steady state under default LRU replacement; combined with equivalent steady-state write traffic, paging achieves best-case bandwidth of about 2.2MB/s and latency just under 4ms. A randomly-faulting micro-benchmark runs at only 50–60 faults/second, or about 18ms/fault.

Most runs below are the average of five separate runs, all of which were done with the system in single-user mode. Standard deviations of execution times and fault counts for all runs were low (under five percent of mean), with several exceptions. [3] First, for runs with low system CPU time, standard deviations were higher, but this is to be expected given the way CPU usage is calculated (by examining the processor status register at every clock interrupt). Second, for applu at 17MB, execution time differed significantly across runs. Apparently, applu is so sensitive to the amount

[3]Standard deviations for disk access times were approximately equal to mean disk access times—suggesting an exponential distribution.

of memory it gets that minor difference in the amount of kernel-internal allocated memory significantly affect performance. The results for applu (in the next subsection) were taken from a set of runs where the kernel-allocated memory quantity was observed not to change.

In the results below a common phenomenon is that both page fault count and page fault service time determine execution time. Linux does an excellent job of maintaining swap locality for sequential programs: swap blocks are allocated sequentially, and often—because of the sequential page table iteration page replacement—consecutive "old" virtual pages are replaced together. Consecutive pages often fault back in together; this is clearly true of massively sequential programs, but happens for other programs too varying degrees. The disk used in this study generally reads consecutive *increasing* blocks very quickly, probably due to its 512KB track buffer.

LRU benefits directly from this swap block allocation scheme since it often replaces consecutive increasing virtual pages in consecutive increasing swap locations. SEQ without prefetching, on the other hand, sometimes suffers: as time increases (and hence swap blocks of increasing address are allocated), SEQ replaces pages in decreasing virtual addresses. When sequentially faulting such a range back into memory, swap blocks with higher addresses are read before blocks with lower addresses. The result is that the track buffer, apparently performing read-ahead, does not help, and a complete disk rotation is necessary before the block can be read. I have observed this phenomenon for a number of sequential programs (noted below as appropriate).

With prefetching, SEQ regains an advantage with fault service times. Linux often combines many prefetches into a single disk I/O, with the disk blocks requested having been reordered as necessary for optimal sequential access. Often we will see that SEQ with prefetching achieves I/O performance as good as default LRU.

I first present results for each application executing alone; following that are results for several sets of applications running simultaneously.

4.5.1 Single-process Results

Single process results are divided into major divisions based on amount of available physical memory. Each memory size has three sets of bars: execution times, page fault counts, and mean fault service times. Each group of bars is arranged in the order: LRU, SEQ, SEQPREF. The execution time bars contain user CPU time, system CPU time, and total elapsed time.

Each set of bars is normalized to the maximum value within that set. Above the maximum for each set is the actual numeric value. For execution times, the value is in units of seconds. For page faults the value is unitless fault count. Fault service times are reported in units of milliseconds.

Runs with memory sizes below 64MB are obtained by instructing the kernel to allocate for itself the difference between 64MB and the desired amount of memory. (I added this functionality to Linux.) This memory region is available as a log for recording timed events, such a replacement decisions, faults, and disk I/Os. Mean fault service times are obtained by post-processing this log. As such, at 64MB no log space is available, and so mean fault service times are not available.

Several applications from Chapter 2 are not included in this section:

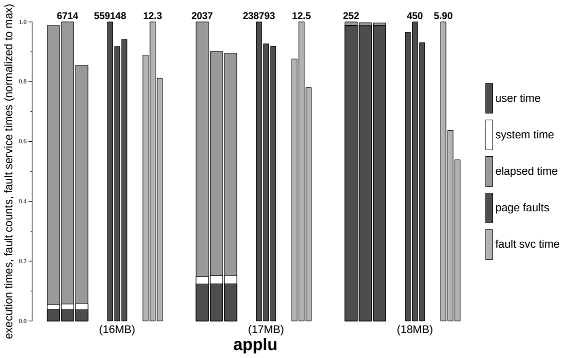
- As noted earlier, GCC requires so little memory that it is not particularly interesting to study for paging purposes, and would be hard to measure accurately given the granularity of its memory consumption.

- M88ksim failed to execute correctly under Linux (it receives a memory access error); the problem may be a bug in m88ksim or may be due to incorrectly-generated code from the compiler (GCC 2.7.2.1). In any case, I have not tracked it down.

- Results for ppmquant are presented in the next section, where it is run as it normally would be used—by receiving input from giftopnm.

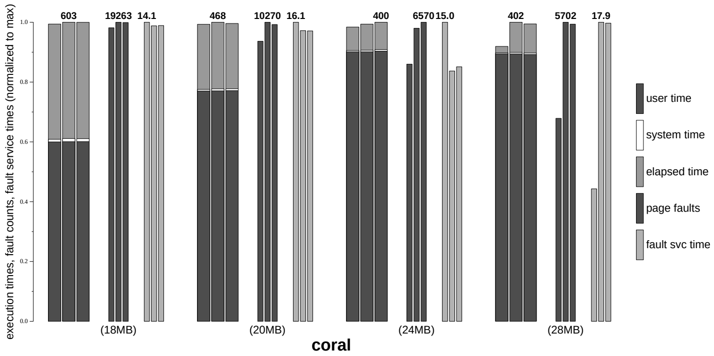
Note once again that the polices' bars are arranged in the following order: LRU, SEQ, and SEQPREF.

Consider first the graph for applu:



Three runs are present, for memory sizes 16, 17 and 18MB At 16MB the small user CPU bars indicate that almost all time is spent waiting for page faults. SEQ with prefetching speeds up fault service slightly. At 17MB SEQ and SEQPREF do slightly better than LRU, but execution time is still dominated by disk wait. Finally by 18MB applu fits comfortably in memory, in agreement applu's steep fault curve from Section 3.3. One further thing to note from applu's plot is the significant system CPU time (white portions of the thick bars) for the 16 and 17MB cases. The software overhead for servicing large numbers of page faults adds up quickly. At 18MB the faults are gone, taking with it system CPU time.

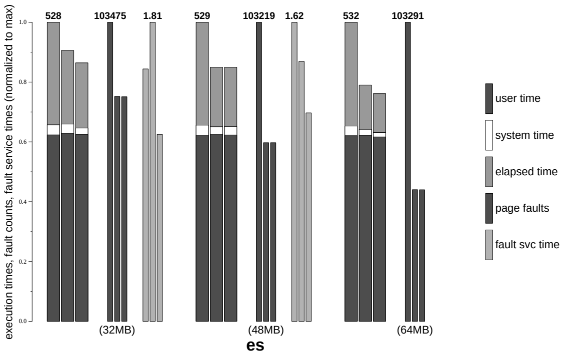
Coral is less sensitive to main memory size.



LRU slightly outperforms SEQ and SEQPREF. LRU suffers fewer faults for all memory sizes, with the ratio of SEQ faults to LRU faults increasing as the absolute fault count decreases.

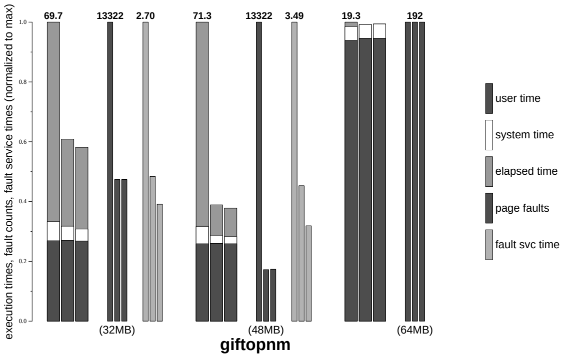
SEQ page splitting is to blame for the increase in page fault count: a single large sequence is built during most of coral's execution; the sequence extends from the beginning of coral's heap to the current heap end. SEQ only replaces from this sequence once memory fills up, which is near the end of coral's execution (cf. coral's graph in Section 2.3). When memory has become full, SEQ replaces pages near the head of the (single) sequence; the pages quickly fault back in and split the sequence. These splits essentially cut the top off the sequence, since the replacements have come from roughly M pages behind the sequence head. Subsequent SEQ replacements eventually split the sequence down to its tail. Then, since there are no more sequences—recall that coral accesses memory very erratically, so no other sequences are built—LRU takes over.

One other interesting result with coral is that at 28MB, LRU results in page fault service time approximately twice as fast as SEQ and SEQPREF. This is due to Linux's good swap locality, as described above. Even though coral does not fault extremely sequentially, it often happens that consecutively replaced pages fault in together, due to some amount of locality at a microscopic level.

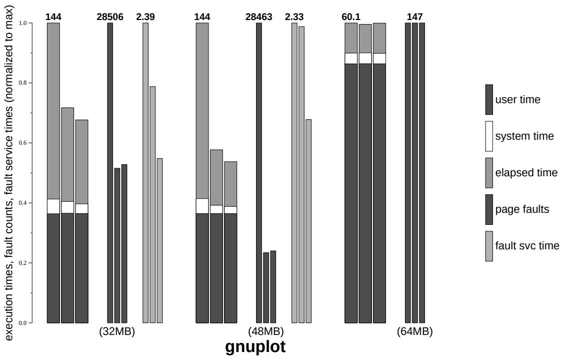


ES behaves as expected. SEQ and SEQPREF, as desired, fully utilize additional memory. Note the generally quite low fault service times, due to es's massively sequential nature combined with Linux's good swap block management. SEQPREF incurs the same fault count as SEQ, but at 32MB and 64MB prefetching cuts mean fault time enough to lower execution time. (I am guessing about mean fault service time at 64MB; recall that calculating service times in my implementation require the main-memory event log, which is not available at 64MB.) Generally, es's execution time is dominated by user CPU time. This illustrates an important phenomenon: sufficient per-page computation reduces virtual memory system performance impact. That is, even if a page fault must be suffered on every access to a data page, the fault overhead decreases relative to the computation done on that page. For es, the computation is high. This is not particularly visible in the previous chapter's plots.

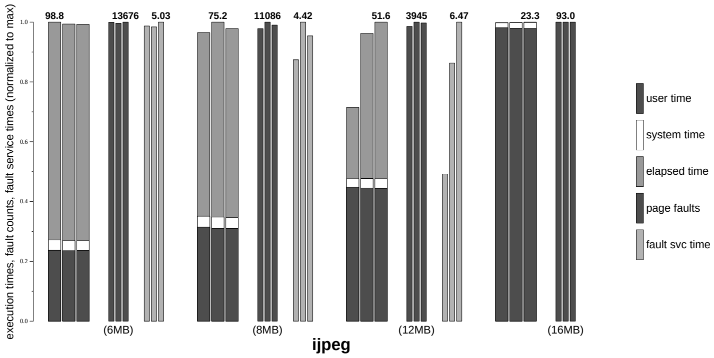
On the other hand, consider giftopnm.



At low memory, LRU replacement leads to a tripling of execution time, which SEQ reduces dramatically—giftopnm's computation time per page is rather less that es's. Note that SEQ improves both fault count (basically in proportion to its reduction in idle time) as well as fault service time. Most of the idle time for SEQ and SEQPREF comes from the filesystem overhead for reading the 3.2MB input file (those disk reads take much longer than 1ms each). Once giftopnm gets 64MB of memory, it executes without paging and without having to read the input file (which is cached now).



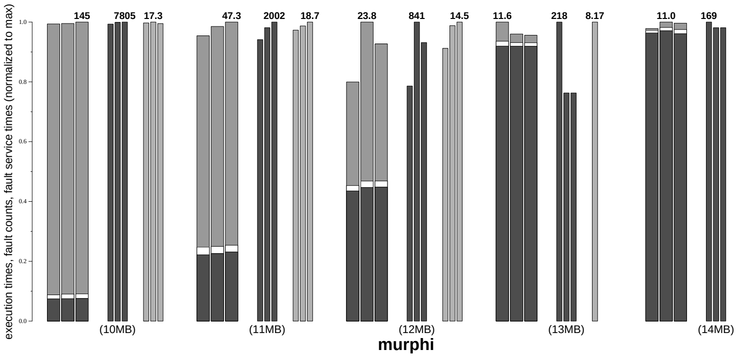
As expected, SEQ improves gnuplot's performance considerably. SEQPREF's fault service time reductions reduce execution time even more, even though it increases very slightly the fault counts (at 32MB and 48MB). Note once again the excellent fault service times all around. Gnuplot (as with giftopnm) requires some overhead for reading its input file, which is 12.1MB for gnuplot.



At middle memory sizes, LRU outperforms SEQ on ijpeg, but mostly due to fault service times that result from ijpeg's sequential access patterns. In a separate experiment, I counted the number of faults whose block address was between 1 and 10 greater than the previous fault's block address. For a run 2720 faults (similar to the 12MB result above), 2499 faults were to blocks 1–10 blocks past the previous fault's block address. Clearly Linux handles sequential swap access quite efficiently.[4]

One thing to note regarding ijpeg is that SEQPREF actually does worse that SEQ at 12MB. This is a rare case where prefetching hurts.

Murphi, like applu, appears to be sensitive to how much memory it gets.

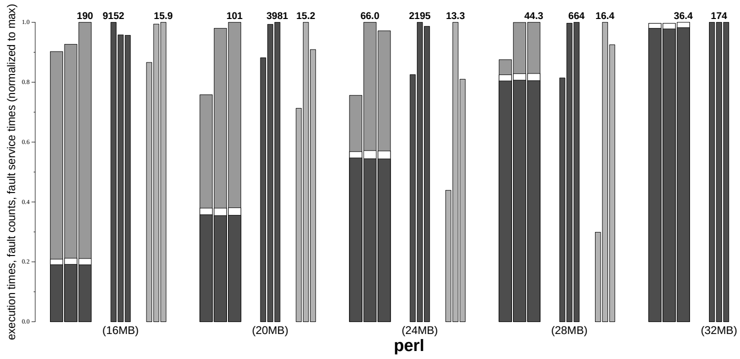


Between 10MB and 13MB execution time improves by a factor of ten or more.

[4]At 16MB for ijpeg, no page faults from the swap device occur, so fault service times are not reported.

At 11 and 12MB LRU does slightly better (in agreement with the simulation results from Section 3.5.1). The extra faults that SEQ incurs before it destroys the single large sequence hurt mostly at 12MB, where the total fault count is low, below 1000.

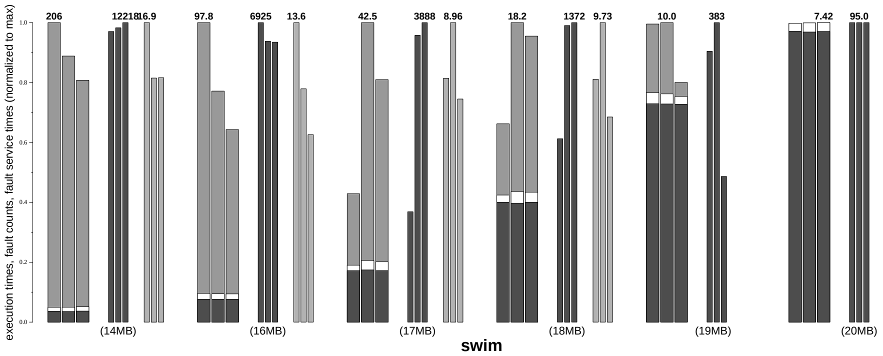
At 13MB SEQ slightly exceeds LRU, as almost no paging occurs. At 13MB for SEQPREF, and at 14MB for all policies, no faults occurred that required reading from the swap device, so no fault service times are recorded. The few faults that did occur were to executable image pages serviced by the filesystem.



SEQ visibly degrades perl's performance. Simulation results from the previous chapter indicated that such would be the case. LRU outperforms SEQ mostly through better fault service time—a result not anticipated in the simulation studies. Observations of swap block accesses verify that LRU often gets good swap device locality; under SEQ faults often occur to disk blocks behind previously-read blocks, incurring the latency of a complete disk rotation.

The constant 174 faults at 32MB are for perl's executable disk image.

As swim's simulation results indicated, SEQ sometimes surpasses LRU and other times loses to LRU:

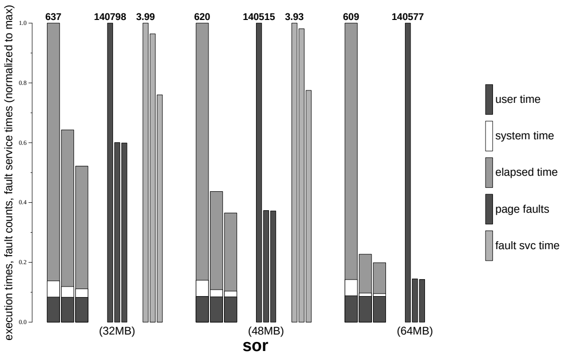


At low memory SEQ and especially SEQPREF do well; at 18MB LRU takes over, and by 20MB everything basically fits into memory. Ideally, SEQ would avoid the degradation at middle memory sizes.

To speed up the exceedingly slow low-memory runs of swim, I changed swim's input file to have it iterate only four times (the default was 10 iterations). The iterations are identical so lowering the iteration count does not fundamentally change swim's behavior.

At low memory swim's sequential performance does not help its fault service time[5] Many small sequences faulting together make it hard for fault block locality to work.

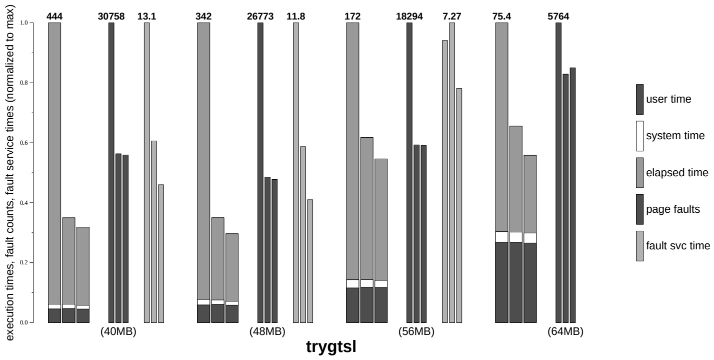
Sor's SEQ performance completely dominates LRU:



[5]At 14MB the graph obscures the fault count (12218) and the mean fault service time (16.9ms).

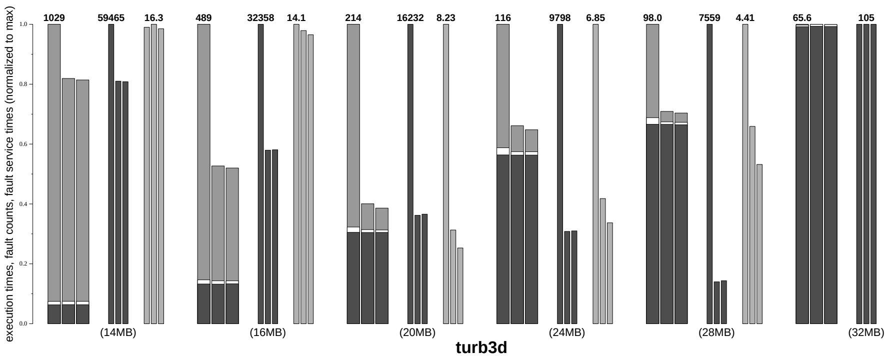
SEQPREF again achieves a reduction in fault service time and a corresponding reduction in idle time. Compared to es and some of the other massively sequential programs, sor performs relatively little computation per page, so SEQ and SEQPREF achieve extreme speedups. Fault service times are, as expected, uniformly low.

Also as expected, SEQ and SEQPREF do well for trygtsl:



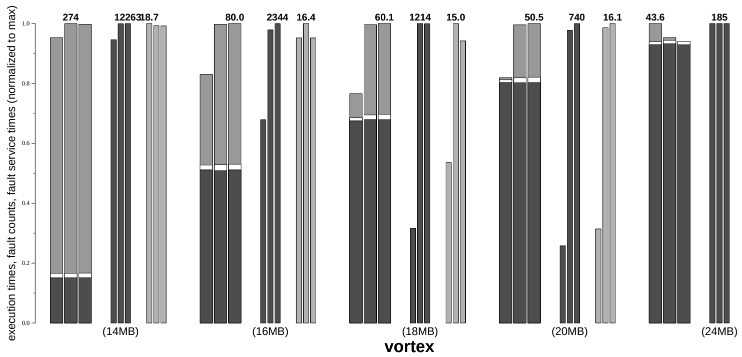
Unlike for sor and other massively sequential programs, LRU improves trygtsl's performance as more memory is added (as predicted by simulation results). Nevertheless, SEQ and SEQPREF still outperform LRU significantly. Fault service times are modest—higher than observed under massively sequential programs.

Similarly, turb3d improves modestly under LRU with increasing memory size, but SEQ does much better:



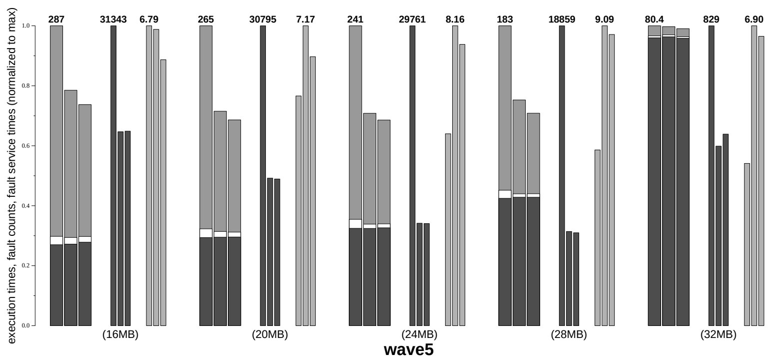
SEQ and SEQPREF achieve speedups by both reducing fault count and by improving fault service time. Note the disparities in system CPU time; LRU's excess page faults require non-trivial processing.

Along with perl, vortex is the only program for which LRU significantly outperforms SEQ for more than one memory size:



The range 14MB to 20MB represents a relatively large range of memory, making this result all the more significant. Like perl, SEQ slows down because it makes more bad replacements and because it increases fault service time. Slowdowns for SEQ and SEQPREF are limited to perhaps twenty to thirty percent, but we would certainly like a eliminate them. [6]

Finally, wave5 performs well, as expected:



[6]To reduce vortex's execution time, for all runs reported I changed the "outer loop" and "inner loop" parameters in vortex's input file. The new values are 1 and 2, respectively. Originally, in the SPEC *test* input file that I used, they were 2 and 4.

As in other cases, prefetching improves execution time modestly, although LRU gets very good swap locality. In a separate experiment, I observed that under LRU, fault blocks were 1–10 blocks ahead of the previous block for 11259 of 13469 faults. For SEQ and SEQPREF, fault blocks were very often behind the previous block, leading to higher service times due to rotational delays.

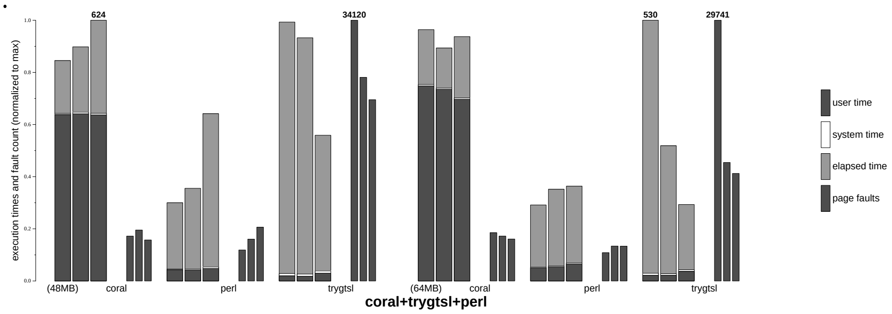
In summary, SEQ performs well for massively sequential programs, reducing fault count and execution time significantly, sometimes by several hundred percent. For mixed-class applications, SEQ often performs well. Prefetching almost always helps SEQ in cases where SEQ performs well. SEQ degrades performance for perl and vortex across a range of memory sizes. SEQ and LRU alternate doing better for other programs (applu, swim). For complex class programs (coral and murphi), SEQ generally limits itself to LRU performance, with a few cases handled worse than LRU. Finally, we have observed the impact that fault service time has on performance: SEQ cannot afford to perform poorly here, and typically it does well. Often SEQ's fault service performance is poor for programs like perl and vortex where it increases fault count as well. These results suggest that it is necessary to throttle SEQ more aggressively than simply relying on sequence splitting.

4.5.2 Multi-process Results

This section presents execution time and page fault count results for simultaneous executing of several program combinations. The goal is to discover whether SEQ works well outside the single process cases that we observed earlier.

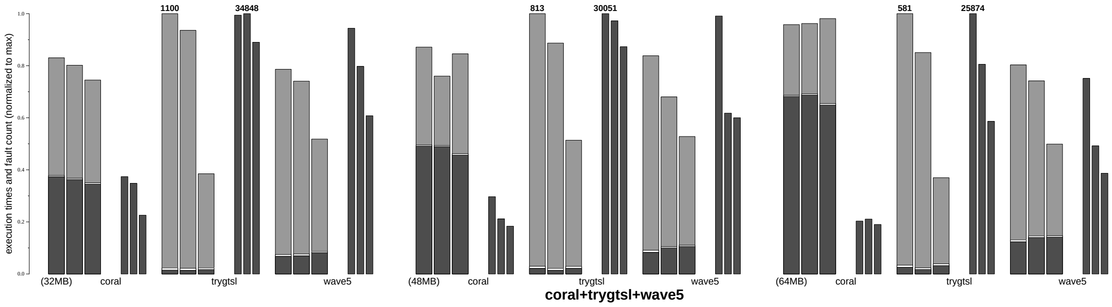
Graphs are divided first by memory size, then by application, and finally by replacement policy. As before the policies are ordered LRU, SEQ, SEQPREF. Execution times are normalized, for a particular memory size, to the time of the longest-running process across all applications and policies at that memory size. Similarly, at any given memory size, per-process page fault counts are normalized to the process that faulted the most. The programs are executed at the same time, so the time for all programs to complete in a given policy's run is the time taken by the longest running program (across all programs) under that policy. I have not recorded fault service times for these runs, since the swap device would be subject to multi-process contention.

The first combination is coral, trygtsl, and perl—one program from each application class:



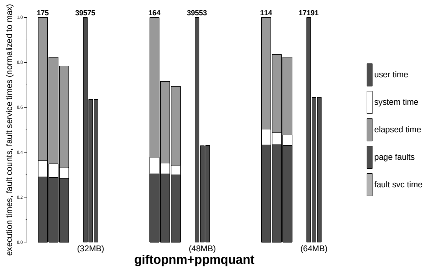
Interestingly, coral user CPU execution time dominates the time curves; this makes sense since coral's memory requirements are less than the other programs, and given its complex and heavy access patterns, it keeps most of its pages in memory. Occasionally perl and trygtsl get to run, but they soon fault and coral gets the CPU back. As expected, SEQ and SEQPREF cause more faults for perl. SEQPREF speeds up trygtsl at the expense of perl (at 48MB).

Somewhat similar is the case for coral, trygtsl, and wave5:



SEQPREF significantly helps trygtsl and wave5—the swap prefetching apparently brings in sufficient pages to allow these programs to run without interruption, preventing coral from monopolizing the CPU. By doing so, SEQPREF even reduces coral's fault count, although coral finishes slightly later as a result of waiting for the CPU.

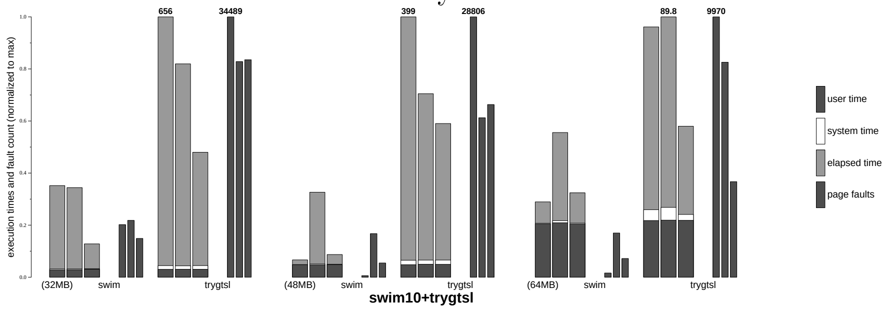
Next we have an interesting case: giftopnm creating ppmquant's input via a UNIX pipe:



Unlike other multiprocess graphs, this graph shows the execution time for both programs combined. Since both programs are massively sequential, we expect SEQ to do well, and it does, reducing execution time by time proportional to the fault count reduction. (At 48MB execution time is not quite reduced by this amount—LRU probably services page faults faster here.)

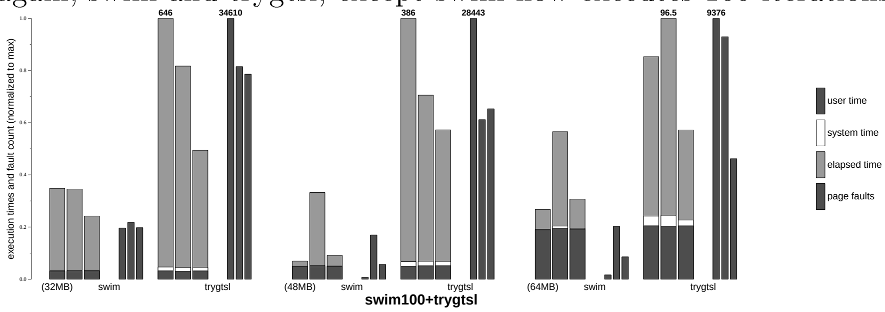
The next set of programs are swim and trygtsl. These programs were originally chosen because their trace lengths are roughly the same (300-400 million instructions, much lower than the billions of instructions for other programs). However, as it turns out, swim—with the default input file—finishes far earlier than trygtsl. This is because at 16MB, it fits into memory comfortably, whereas trygtsl faults heavily even at 64MB, so swim can run without interruption. Furthermore, since swim can run, it can keeps its pages referenced. Linux's page replacement will tend to replace trygtsl's pages since trygtsl occupies more memory and its pages are less frequently referenced.

I ran two runs of trygtsl and swim. First, the default swim case of ten iterations, in which swim finishes almost immediately:



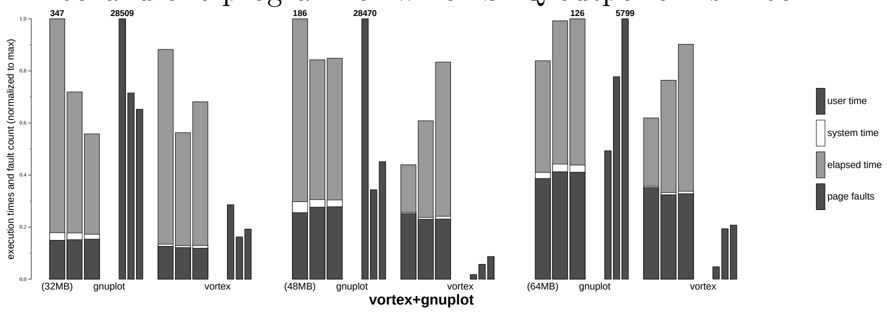
Even though the two programs do consume almost identical CPU time, trygtsl faults much more heavily, as evidenced by the visible system CPU levels and the vastly different fault count bars. Interestingly, at 48MB and 64MB, swim gets almost no faults under LRU—it finishes with almost no interruption from trygtsl. SEQPREF finishes first overall on all runs.

Once again, swim and trygtsl, except swim now executes 100 iterations:



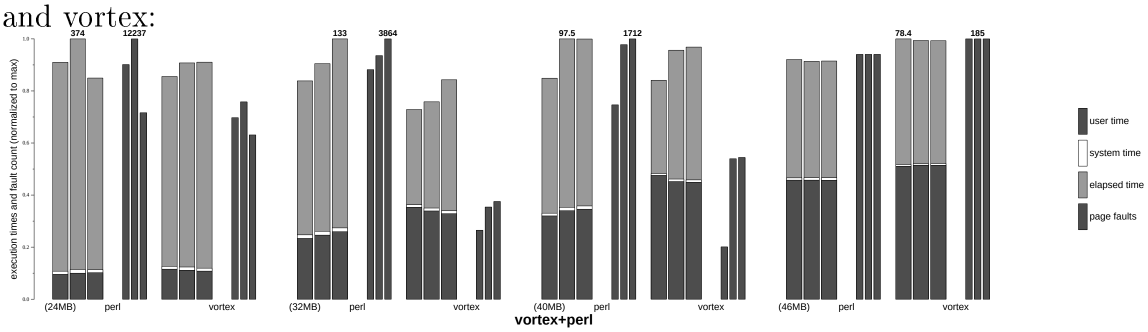
Once again SEQPREF is best for overall execution time; SEQ without prefetching does somewhat poorly.

The pair of programs is vortex and gnuplot—one program for which SEQ performs worse than LRU and one program for which SEQ outperforms LRU:



At 32MB, SEQPREF's prefetching helps gnuplot finish first, while vortex under plain SEQ finishes first. Poor performance on gnuplot hinders LRU at 32MB and 48MB. At 48MB, prefetching no longer helps gnuplot much, with SEQ and SEQPREF finishing at the same time. LRU finishes quite early on vortex—not surprising, since gnuplot faults heavily and thus does not occupy the CPU. Finally, at 64MB, LRU catches up and finishes first. (From gnuplot's curve in the previous section we see that at 64MB it runs without much faulting under all three policies.) LRU's superior vortex performance makes the difference here.

Finally, a kind of worst-case: the two programs that SEQ handles the worst, perl and vortex:



Except for 46MB where both programs fit in memory, LRU outperforms SEQ and SEQPREF on five of six runs (two programs times three memory sizes). Under LRU programs finish ten to fifteen percent faster than under SEQ and SEQPREF. While this is not disastrous for SEQ, there is clearly room for improvement.

4.6 Quantifying SEQ Overhead

We have seen that SEQ (especially with prefetching) performs very well for a number of programs. The previous sections show that SEQ incurs no CPU time overhead for heavily faulting applications when it performs well: avoiding page faults more than makes up for tracking sequences and examining them for replacements. But what about the fault-tracking overhead for programs that fit entirely in memory? (The faults that would be tracked in this case are minor faults.) This section measures this overhead for several application classes.

SEQ tracks sequences of minor faults that result from address space creation. To measure the absolute worst-case overhead, I wrote a simple program that creates and touches every page in a 40MB address space and then exits. Running the program 512 times takes 300 seconds without SEQ, and 306 seconds with SEQ running. So, in the worst case, SEQ adds two percent runtime overhead. The SEQ prototype could certainly be sped up to handle these cases with less overhead; however, this really is a worst-case scenario. For processes under 4MB (the normal case), SEQ does not track faults (as mentioned in Section 4.2).

A more realistic measure of address-space creation overhead due to SEQ is as follows. The simulation results in Chapter 3 are obtained by repeatedly running a policy simulator with a trace file as input. The simulator reads the trace file by mapping the file into its address space using the *mmap* call (using *mmap* here is far more efficient than reading the file with the *read* system call [40]). While running the LRU simulator—the one that does the least work per record processed—over turb3d's 33MB trace file 16 times, SEQ consumes about .15 seconds of CPU time. The 16 simulator runs sum to about 90 seconds execution time, so SEQ adds less than .2%

overhead. This could be reduced even further if necessary.

Regarding SEQ's space overhead, I have observed SEQ to consume between three and four KB per process. Since SEQ only tracks processes of size 4MB and above, 4KB per process is essentially insignificant (under .1%).

4.7 Summary

I have described a prototype implementation of the SEQ algorithm done in the context of the Linux kernel. SEQ fits cleanly into Linux and the implementation was not difficult. I have also described an implementation of swap prefetching that takes advantage of SEQ's knowledge of sequential access patterns.

The central contribution of this chapter has been extensive performance results showing that SEQ often obtains significant speedups—50 to 100 percent and more—over Linux's default LRU policy. Speedups occur because SEQ decreases page fault counts as predicted by the previous chapter's simulation results. In most cases SEQ with swap prefetching provides more efficient page fault service than does Linux's already efficient LRU implementation. Finally, SEQ's speedups are not limited to single-process execution.

For a few programs, we have observed SEQ to occasionally degrade performance by up to approximately 20 percent. Techniques introduced in the previous chapter to eliminate this performance reduction are the subject of ongoing research. Finally, I have shown that SEQ can be implemented extremely cheaply in terms of time and space consumption at runtime.

Chapter 5

Related Work

Virtual memory is a curious thing. A long time ago, there was no virtual memory. It was adopted widely once people accepted that it could be implemented efficiently. Recent proposals, though, generally involve giving policy control back to the programmer— at the expense of improving automatic (in-kernel) page replacement policies.

Before the advent of virtual memory, programmers managed all physical memory explicitly. This was exceedingly cumbersome, and there was desire to automate the process of choosing a page for replacement. (Virtual memory provides other benefits— linear address spaces, protection, relocation, and so on, but I ignore these features here.) Early studies of virtual memory confront the issue of who can do a better job of managing memory: the programmer or the system [23]. Results with the Manchester Atlas, the first computer with virtual memory, were encouraging—the system did better than the programmers.

As time-sharing grew in popularity in the late 1960s and 1970s, virtual memory became more or less a universal abstraction provided by all general-purpose computer systems. Studies from this era concentrate on time-sharing and its attendant concerns (for example, finding an optimal level of multiprogramming). Denning[13] summarizes work from this period.

The early 1980s witnessed the ascent of two operating systems that would become exceedingly popular, DEC's VAX/VMS with its FIFO replacement policy [28], and BSD UNIX, which implemented the global CLOCK LRU approximation[2, 27]. In both systems simplicity was an important goal—so much so that for the VAX, the per-page hardware reference bit was eliminated on the assumption that it was inconceivable for a replacement policy to examine every reference bit. At about this time people were starting to realize that LRU-based replacement policies were far from ideal for various important classes of applications, such as databases [41].

In the wake of these two systems, not much new work on automated, in-kernel methods has been done. For example, Windows/NT[11] adopts FIFO replacement.

Mach[1] introduced the idea of an *external pager*[45], software outside the operating system kernel proper, yet capable of implementing paging mechanisms. Others have extended the external pager interface to expose the replacement policy as well[29, 39]. V++ [18] implements a segment-based scheme in which user-level processes control segments.

Recently even more exotic interfaces have been proposed, including one built around a so-called *metaobject-protocol*[24] and another that involves applications downloading replacement policies coded in a simple assembly language [26]. This thesis at an implicit level fundamentally opposes to the idea of application-specific policies. Notwithstanding the fact that a SEQ implementation can—and probably should—include a switch to enable or disable SEQ on a per-process basis, in SEQ there is no concept of application-specific policies, and no energy has been spent dealing with the issues relating to allowing such policies. The goal of this thesis is to make the kernel sophisticated enough to recognize common access patterns and do the right thing with them.

In any case, it is not clear how much benefit is obtainable from application-specific policies. In three recent papers [24, 26, 32], three of a total of four benchmarks were massively sequential programs. (In fact, two of the three were the same basic program, SOR.) As this thesis has demonstrated, massively sequential programs can be handled optimally by automatic, in-kernel mechanisms. A cynic might contend that the reason these studies include so few benchmarks—especially programs that are harder to optimize than the massively sequential case—is that the application-specific policies are too painful to use, *even by their authors*. I prefer to observe that massively sequential programs are simply a very common case, and I agree that LRU's performance for such programs should be criticized as much as possible.

Recently an interesting compiler-based approach to prefetching and page replacement decisions been demonstrated [30]. Compiler-based schemes require at least some amount of kernel-exposed policy, but compiler-based mechanisms have the strong advantage that they do not require the programmer to implement the policy in order to get good performance. Unfortunately, compiler-based schemes may work well for array-based applications, but they remain undemonstrated for more complex programs—even ones with simple, easily-recognizable *at runtime* access patterns, like gnuplot.

SEQ's method of observing performance characteristics (in this case, page faults) at runtime is conceptually similar to methods explored by Romer *et al* for improving other aspects of memory-system performance, in particular page placement[35] and page size[36].

Finally, I observe that SEQ can clearly be applied to filesystem buffer cache management. Much evidence suggests that file access patterns are often sequential[4, 31]—perhaps more sequential than virtual memory access patterns. Although this thesis has concentrated solely on virtual memory, the techniques introduced here apply to buffer cache management. In fact, buffer cache management may be easier to handle, since a narrow filesystem interface (*read*, *write*, and *lseek* system calls) can be monitored to observe sequential behavior.

Chapter 6

Summary

This thesis began with a study of eighteen benchmark programs from numerous application domains. Lengthy traces, mostly complete end-to-end runs, were obtained in a format conducive to efficient virtual memory simulation studies.

By examining the applications' large-scale memory reference patterns, I have categorized applications based on the amount of locality present: massively sequential programs have little locality and are dominated by large sequential, iterative patterns. Mixed class applications show moderate locality, perhaps iterating simultaneously over some number of sequences. Finally, complex-class applications display high locality. This categorization is strengthened by examining LRU stack graphs for these programs. Finally, we have observed how fault rate under LRU varies over time with these applications.

Chapter 3 introduced SEQ, a new algorithm for adaptive page replacement. SEQ's design has been described in detail; SEQ works by detecting sequences of page faults and performing MRU replacement on such sequences. SEQ relaxes MRU replacement by splitting sequences when non-sequential faults occur inside sequences.

Simulation results showed that SEQ performs very well for the massively sequential programs under consideration in this thesis. For most complex applications studied, SEQ manages to restrain MRU replacement; as a result, SEQ performs similarly to LRU, occasionally performing worse. For mixed-class applications SEQ performs similarly to LRU, sometimes substantially surpassing it.

I have introduced several techniques by which SEQ's MRU replacement can be controlled more aggressively, to avoid performing worse that LRU. A promising technique incorporates results from previous research and is the subject of ongoing implementation.

In Chapter 4 I described a prototype SEQ implementation done in the Linux operating system kernel. Using SEQ's knowledge of sequential reference patterns, I implemented a swap prefetching technique that achieves excellent fault service times, even compared to Linux's already good performance in this area.

Extensive performance results for the Linux prototype show that SEQ often obtains significant speedups—50 to 100 percent and more—over Linux's default LRU policy. Performance degrades for a few programs for which sequence splitting is insufficient to throttle SEQ. As anticipated by Chapter 3's simulation results, SEQ improves performance by eliminating many page faults. Furthermore, swap prefetching provides additional benefit. SEQ performs well on multiprogramming workloads in addition to single-process cases. Finally, the unoptimized SEQ prototype incurs minimal overhead in terms of memory consumption and execution time.

Future work may proceed along a number of avenues. First, an implementation of the reference race technique (introduced in Section 3.5.1) may reduce or eliminate the occasional cases where SEQ performance drops below LRU performance. Another opportunity for future work is to apply SEQ replacement to filesystem buffer caching; filesystem workloads have significant sequential components. Yet another application of SEQ might be to *page placement*, to improve performance for large off-chip caches [22] by avoiding polluting a cache with data that will be accessed sequentially and then ignored.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Conference*, pages 93–112, Atlanta, Georgia, 1986.

[2] O. Babaouglu and W. Joy. Converting a swap-based system to do paging in an architecture lacking page-reference bits. In *Proc. 8th SOSP, Operating Sys. Review*, page 78, December 1981.

[3] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[4] Mary G. Baker, John H. Hartmann, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, December 1992.

[5] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(1):78–101, January 1966.

[6] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed shared-memory multiprocessors. *The Journal of Supercomputing*, 10(1):87–104, 1996.

[7] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proc. USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[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.

[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] Standard Performace Evaluation Corporation. *SPEC Newsletter*. Fairfax, Virginia, September 1995.

[11] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

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

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

[14] 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.

[15] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennesee, Computer Science Department, 1997.

[16] Gregory F. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the USENIX 1997 Annual Technical Conference*. Usenix Association, January 1997.

[17] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. Technical Report 1338, UW–Madison Computer Sciences Department, January 1997.

[18] 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 1992.

[19] Paul Heckbert. Color image quantization for frame buffer display. In *Proceedings of SIGGRAPH '82*, 1982.

[20] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Palo Alto, 2nd edition, 1995.

[21] Hervè A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans II, Anna R. Karlin, Henry M. Levy, and Mary K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the Seventh Internationl Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 258–267, Cambridge, MA, October 1996.

[22] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, November 1992.

[23] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One level storage systems. *IRE Transactions on Electronic Computers*, 11(2):223–235, April 1962.

[24] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of the 1993 OOPSLA*, 1993.

[25] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[26] 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.

[27] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, Reading, Massachusetts, 1989.

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

[29] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, Burlington, Vermont, October 1990.

[30] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 3–17, Seattle, October 1996.

[31] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2bsd file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985.

[32] Yoonho Park, Ridgway Scott, and Stuart Sechrest. Virtual memory versus file interfaces for large, memory-intensive scientific applications. In *Proceedings of Supercomputing '96*, 1996.

[33] 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.

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

[35] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[36] Theodore H. Romer, Wayne H. Ohrlich, Anna R. Karlin, and Brian N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In

Proc. 22nd Annual International Symposium on Computer Architecture, pages 176–189, June 1995.

[37] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[38] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994.

[39] Stuart Sechrest and Yoonho Park. User-level physical memory management for mach. In *Proceedings of the USENIX Mach Symposium*, Monterey, California, October 1991.

[40] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[41] Michael Stonebraker. Operating systems support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[42] F. Traenkle. Programming models and boundary integral equation methods for solving microstructure electrostatics. Master's thesis, University of Wisconsin–Madison, 1993.

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

[44] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, 1991.

[45] 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.