How Memory Size Affects Disk Reads

Y.C. Tay*
National University of Singapore

Abstract

Although the issue of how memory size affects performance is an old one, an open question remains: How do disk reads grow when memory size shrinks? This paper presents a simple analytic model to answer this question, and its equations are validated with measurements made on Windows 2000.

The model centers on a 1-read footprint M^*. It can be used for capacity planning, and optimizing code to reduce memory requirements can be interpreted as reducing M^*. It points to new ways of managing memory (e.g. trimming working sets). By recasting the relationship between memory size and disk read as between cache size and miss probability, the model also offers a new way of looking at caching policies.

The model's simplicity makes it a useful tool for analyzing system behavior, and this is demonstrated with a study of how memory contention interacts with processor contention.

1 Introduction

The issue of how memory size affects computer performance is an old one that goes back to the study on working sets [D]. Since then, the price of memory has fallen tremendously, but the issue remains [SKW] and is likely to stay for two reasons: software and data (e.g. video) are growing ever bigger, and portable devices are getting smaller and ubiquitous.

One fundamental question remains open after these thirty-odd years of research: How does memory size affect disk traffic? When memory is tight for a virtual memory system, incoming pages must be accommodated by replacing some pages that are already in memory, and the latter will have to be reread if they are needed later. Hence, the number of reads from disk increases as memory size decreases, and Figure 1 illustrates this relationship with measurements done with the Microsoft Office Test in the Winstone99 benchmark. There is no memory model in the literature that captures this relationship; this paper presents such a model.

Existing work on memory models clusters around two topics:

(1) Tracing and analyzing memory references for a single program [BMM, D, GC, MB].
However, even on a personal computer, programs are rarely run alone. A user may use a database application, a report generator and a web-page authoring tool at the same time on

* Work done while on sabbatical at Microsoft Corporation

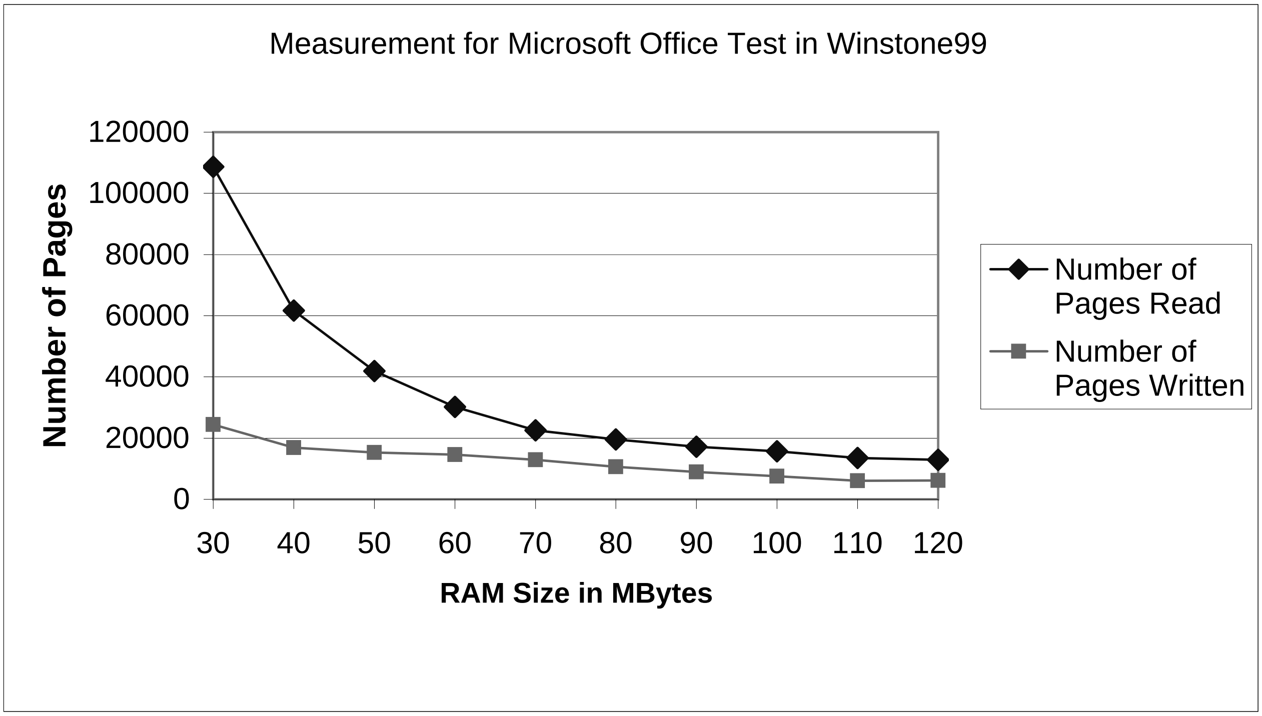


Figure 1 How memory size affects disk traffic.

a workstation, and handle email while doing so. Memory models for single programs do not capture the interaction among programs in such a workload. Even in a "uni"-programming environment, a model for, say, the virtual address access pattern in a program cannot capture how the application interacts with the operating system — which may occupy more memory pages, and who must provide clean pages for data, flush dirty pages, etc. — to produce the disk traffic. (Studies on program behavior are usually trace-based simulations of running the program alone [SKW].) Our model goes beyond a single program and captures the read traffic for a multiprogramming mix.

(2) Incorporating memory constraint in queueing models.
Queueing networks are frequently used to model computer systems, and memory size is typically incorporated by using a memory queue with a rate-dependent server [MAD]. This server can be decomposed into a product-form closed network for the processor and disks whose solution (by Mean Value Analysis, say) assumes branching probabilities do not change. But they do: more jobs in the closed network means less memory and more disk IO per job. This relationship between memory size and disk traffic remains missing in more sophisticated models [BM], but is the focus in our model.

There are many details in an operating system that may determine how memory size affects performance: an application's reference pattern on code and data, memory management reading ahead on the application's behalf, sharing of pages and copy-on-write, write clustering, when the association between a process and a page frame is broken (so soft page faults are not possible), data structures for address translation, page replacement, working set expansion and trimming, etc. In principle, one can construct a memory model bottom-up by building and integrating submodels for these details, but this will take a long time.

Instead, we use a top-down approach by focusing on the fundamental issue with the simplest possible model, then refine it as and when necessary. Since main memory is just the core of a much larger memory that stores files and supports demand paging, the fundamental issue is the open question mentioned previously: How does memory size affect the number of reads for a workload?

Memory size affects the number of writes too, but our experience is that the effect is smaller than for reads, as is the case in Figure 1. Moreover, most operating systems write dirty pages asynchronously and in clusters, so their impact on performance is muted (compared to reads). We therefore focus this paper on reads, and leave writes as a future refinement of the model.

We begin in Section 2 (The Memory Model) by listing the approximations used in the model to derive the equations. The crucial approximation (A5) models the relationship between the probability p that a page is reread from disk and the time t' a page spends waiting on disk to be reread. Section3 (Validation) then presents measurements done with Windows 2000 to support the equations.

But the analysis does not end with capturing the read curve in Figure 1 — it begins there. Section 4 (Capacity Planning) shows how the model can be used in sizing a new machine and predicting performance. The sizing is based on the 1-read footprint M^*, which measures a workload's memory requirement. Indeed, Section 5 (An Interpretation of Code Optimization) shows that some techniques in modifying source and binary code to reduce their memory requirement can be viewed as reducing M^*.

Section 6 (Refining the Model) then introduces some modifications to the model. One of these modifications is used in Section 7 (A New Perspective) to show how M^* can be used to trim working sets. This illustrates how memory management can be done differently if the operating system can estimate a process's memory requirement. The model also provides a different view of caching, and gives a formula for cache misses. We then use the model in Section 8 (Behavioral Analysis) to examine the interaction between memory contention and resource contention: When pages are removed from a process, can it be compensated through higher priority?

We end in Section 9 (Conclusion) with a summary and a discussion of several possible directions for future work. For example, our model is operating system-independent, and much can be done in using it for Linux. Another example: the model provides a new approach to cache analysis — a replacement policy (least-recently-used, etc.) can be analyzed for an equation between p and t', and this relationship plugged into our model by replacing approximation (A5).

2 The Memory Model

Consider a set of M pages in random access memory (RAM or, simply, *memory*). Each page may contain code or data, and belong to the kernel or user space. M may be the average of a random variable (e.g. working set size), and treated like a real number (rather than an integer). In our experiments, each page had 4 KBytes, and M was specified in MBytes. Our model is based on several approximations, the first of which is:

M	memory size
M^*	1-read footprint
r	number of times a page is read
p	probability that a page is reread
λ	rate of fresh reads
I	time between fresh reads, measured by number of instructions
J	time a page is needed in memory, measured by number of instructions
t	time a page is memory resident, per visit
t'	time a page waits on disk, between being replaced and being reread
u	processor utilization
S	processor speed
n^*	number (pages) of fresh reads for a workload
n	number (pages) of reads for a workload
D	read delay (seek time, etc.)

Figure 2 Glossary

(A1) All M pages are used.

We therefore exclude the case where some pages are idle because memory is larger than the workload. Some pages may not be replaceable; for example, a log manager may make its buffer pool nonpageable. We assume such pages form a small fraction of M. Hence the following:

(A2) Any of the M pages can be replaced.

The pages used by a workload may be read from disk, or generated in memory (e.g. by dynamically allocating a large data structure). We exclude the second possibility:

(A3) All pages used by the workload are read from disk.

All operating systems ensure that the instruction overhead for disk IO is minimal. Hence the next approximation:

(A4) The instruction overhead for disk IO is negligible.

The read delay (seek time, rotational delay and transfer time), however, may not be negligible.

A page in memory may be replaced to accommodate another page, and will have to be read from disk again if it is referenced later. A *fresh read* refers to the first time a page is read from disk, and we call subsequent retrievals of that page a *reread*. Let λ is the rate of fresh reads and r the number of times a page is read. Since all needed pages are read from disk (A3), we have $r \geq 1$, and the rate of rereads is $(r-1)\lambda$. Here, λ and r are both averages: λ over time and r over pages. Throughout this paper, all variables — see Figure 2 for a glossary — are averages for corresponding random variables.

Assume the workload executes I instructions between two fresh reads. There is considerable evidence that a program's memory references can be viewed as forming a series of phases, where

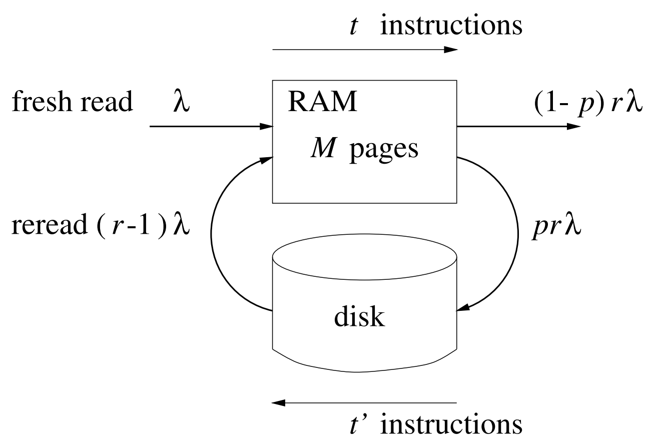


Figure 3 A simple model of page flow.

each phase is localized on a small set of pages [MB]. This phase behavior requires that a page remain in memory for a certain time span. Should the page be replaced during this period, it will have to be reread. Like I, we measure this time span with J instructions. Note that J is not the number of instructions on that page (which may be a data page anyway), and not the number of times the page is reference (a processor may execute register instructions without referencing any page); I and J are simply measures of time [GC].

Finally, let p be the probability that a (replaced) page is reread. By (A2), $p(r\lambda)$ is the rate at which pages are replaced and later reread. If a page stays in memory for t instructions on each visit and, when replaced, waits on disk for t' instructions before being recalled, then we have the simple model shown in Figure 3.

The crux of this model lies in estimating p when memory is tight, i.e. the operating system does not have enough free page frames to quickly respond to allocation requests. Consider a page that is rarely referenced. When there is a shortage of page frames, a memory manager will likely make the mistake of replacing such a page, and have to reread it, i.e. $p \approx 1$. If a page is frequently referenced, a memory manager will not replace it — but choose some less referenced page instead — unless it is removed at process termination, i.e. $p \approx 0$. (The small positive p allows for the page being recalled by another process in the workload,)

Now, the number of instructions between two consecutive reads from disk is $\frac{J}{r}$, so $t' < \frac{J}{r}$. If $t' << \frac{J}{r}$, then the page spends very little time waiting on the disk before being recalled. When memory is tight, this happens only for pages that are frequently referenced; by the argument above, $p \approx 0$. If $t' \approx \frac{J}{r}$, then the page spends most of its time waiting on the disk. Such a page is rarely referenced, and the previous argument gives $p \approx 1$. We therefore adopt a simple linear approximation: $p \approx t'/(\frac{J}{r})$, i.e.

(A5) $$p = \frac{rt'}{J} \ .$$

The rest of the derivation is straightforward. Flow balance in Figure 3 requires that $(r-1)\lambda =$

$pr\lambda$, so we have the basic relationship

$$p = 1 - \frac{1}{r} \qquad \text{or} \qquad r = \frac{1}{1-p} \tag{$E1$}$$

($p \geq 0$ since $r \geq 1$). Let processor speed be S and the workload's processor utilization be u. Then, a page spends (clock) time $\frac{t}{uS}$ in memory on each visit, so Little's Law and (A1) give

$$M = \frac{r\lambda t}{uS} \ . \tag{$E2$}$$

The inter-arrival time for fresh reads is $1/\lambda$, so

$$I = (\frac{1}{\lambda})uS. \tag{$E3$}$$

Since each page makes rt visits, we have

$$\begin{aligned} J &= rt + (r-1)t' \\ &= rt + (r-1)\frac{pJ}{r} \qquad \text{by (A5)} \\ &= rt + p^2 J \qquad \text{by (E1)} \\ &= \frac{rt}{1-p^2} \\ &= \frac{MuS}{(1-p^2)\lambda} \qquad \text{by (E2).} \end{aligned}$$

From (E3), we get

$$\frac{M}{1-p^2} = \frac{J}{I} \ . \tag{$E4$}$$

By (A4), the right-hand-side in (E4) is independent of M. We therefore define a constant

$$M^* = \frac{J}{I} \quad \text{(definition)}, \tag{$E5$}$$

so (E1) and (E4) give

$$\frac{M}{1-(1-\frac{1}{r})^2} = M^*. \tag{$E6$}$$

This equation expresses the fundamental relationship between memory size and rereads — if M is changed to M', then r will change to some r' in the following way:

$$\frac{M'}{1-(1-\frac{1}{r'})^2} = \frac{M}{1-(1-\frac{1}{r})^2} \ . \tag{$E7$}$$

In particular,

$$r = 1 \qquad \text{at} \qquad M = M^*. \tag{$E8$}$$

Clearly, r remains 1 for $M > M^*$, so M^* is a maximum for M in (E6). It would seem from (E6) that $\lim_{M\to 0} r = \infty$. This is false, since r cannot exceed the number of times a page is referenced by the workload, and (E6) and (E10) can be suitably modified to account for this. In practice, a user will tolerate only small values for $r(< 10$, say), so we can be careless about the formulas for arbitrarily small M. Measuring r directly may require a significant amount of overhead, and none of the operating systems we know of makes such a measurement. However, r can be approximated indirectly by measuring n, the number of pages a workload reads from disk, and estimating n^*, the number of fresh reads among n:

$$r = \frac{n}{n^*} \ . \qquad (E9)$$

Substituting (E9) into (E6) gives

$$n = \begin{cases} \frac{n^*}{1-\sqrt{1-\frac{M}{M^*}}} & M \leq M^* \\ n^* & M > M^* \end{cases} \qquad (E10)$$

which shows how n decreases with increasing M. Note that n^* and M^* are both determined by the workload; they are input parameters for our model.

3 Validation

Before proceeding further, we need to validate the basic equations derived so far. The following measurements were made on a machine with one Celeron 400 MHz processor, 128 MBytes RAM, a Seagate ST-39102LW disk and early versions of Windows 2000 (Builds 2008 and 2031). The memory size M was adjusted with `/maxmem` in `boot.ini`, and n measured with `tracelog` (which records events like disk IO, network messages, etc.).

Three runs were made for each M, with a reboot before each run so that memory contents started out similar across runs. The average n was plotted against M, and n^* estimated with a horizontal asymptote for this graph (since $n^* = \lim_{M\to\infty} n$). This way of estimating n^* sufficed because, at this stage, we were only *validating* the equations. When *applying* the equations, n^* will have to be estimated differently (by measurements in the operating system, say).

Note that many memory pages that should count towards n were already in memory before `tracelog`'s counters were initialized: When the machine booted up, some 60 MBytes of its memory were already occupied by processes (e.g. `services.exe`) and device drivers. Furthermore, Windows 2000 had a nonpaged pool of about 3 MBytes in our runs, but the approximation (A2) that any page can be replaced remained acceptable for the range of M (≥ 40 Mbytes) in the experiments.

(V1) *Compile and link* `mm\wslist.c`

Our first workload was a simple `build` for the working set utilities in Windows 2000's memory management. Table 1(a) presents the results from measurement and calculation. The last column supports our analysis in two ways:

(1) Despite the changes in M and r, the value for the expression was approximately constant, thus verifying (E7).

(2) The constant was $M^* \approx 91$ MBytes, so — according to (E8) — we should have $r \approx 1$ at $M = 90$ MBytes. Indeed, we got $r \approx 1.03$.

M (MBytes)	number of pages read	average n	$r \approx \frac{n}{n^*}$	$\frac{M}{1-(1-\frac{1}{r})^2}$
50	38214, 36040, 37924	37393	2.97	89
60	28635, 32809, 31020	30821	2.45	92
70	23728, 23073, 23680	23494	1.86	89
80	20114, 19987, 20183	20095	1.59	93
90	13096, 12997, 12716	12936	1.03	90

(a) Results for (V1) from compiling and linking `mm\wslist.c`; $n^* = 12600$.

M (MBytes)	number of pages read	average n	$r \approx \frac{n}{n^*}$	$\frac{M}{1-(1-\frac{1}{r})^2}$
50	34902, 37611, 34820	35777	3.58	104
60	29847, 29872, 29965	29894	2.99	108
70	18679, 19039, 19338	19018	1.90	90
80	16986, 17201, 16912	17033	1.70	96
90	11055, 10991, 11129	11058	1.10	91
100	10194, 9968, 9952	10038	1.00	100

(b) Results for (V2) from compiling and linking `mm\*`; $n^* = 10000$.

Table 1 Validating (E7) and (E8).

For $M > 90$ MBytes, although the expression $M/(1-(1-\frac{1}{r})^2)$ can still be evaluated from measurements, (E6) is no longer valid because not all pages are used (A1), so no comparison was made there. In essence, while measurements can be made with an arbitrarily large M, the theory extends only as far as its assumptions.

(V2) *Compile and link* `mm\*`

For comparison, we ran a bigger workload by building all of Windows 2000's memory management modules in `mm\*`. The results are presented in Table 1(b). The values in the last column have a larger spread than for (V1), but remain around 98 MBytes, thus supporting (E7). Furthermore, $r \approx 1.00$ at $M = 100$ MBytes, which agrees with $M^* = 98$ MBytes, and validates (E8).

As expected, r in (V2) was consistently higher than in (V1), since `mm\*` includes `mm\wslist.c`. However, the number of fresh reads n^* was *smaller*: 12600 in (V1) and 10000 in (V2). This is because the partial build in (V1) had to read object files from disk, whereas they were generated by (V2), and many of their pages were already in memory by the time compilation reached `wslist.c` in (V2). This workload thus seriously violated the approximation (A3) that all pages are read from disk, which may be why there is greater variation in the values of $M/(1-(1-\frac{1}{r})^2)$.

(V3) *AVS/Express Test in Winstone99*

We next used an interactive workload taken from the Winstone99 benchmark. The test was an exercise in volume rendering (of a lime green lobster) using a visualization suite from Advanced

M (MBytes)	number of pages read	average n	$r \approx \frac{n}{n^*}$	$\frac{M}{1-(1-\frac{1}{r})^2}$
60	10797, 10174, 10120	10364	2.07	82
65	9335, 9816, 9674	9608	1.92	84
70	7818, 8169, 7868	7952	1.59	81
75	6456, 7234, 7398	7029	1.41	82
80	6069, 6057, 6331	6152	1.23	83
85	6161, 6023, 6154	6113	1.22	88

Table 2 Results for (V3) from AVS/Express Test (Volume Rendering) in Winstone99; $n^* = 5000$.

Visual Systems. The measurements and calculations are presented in Table 2.

Again, the values in the last column are fairly constant, with an average of $M^* = 83$ MBytes. However, at $M = 85 (> M^*)$, we got $r \approx 1.22$, which is significantly different from the theoretical $r = 1$. We believe this is because each Winstone99 test begins by copying and uncompressing some files. If `tracelog` were activated before the copying, the measurement would include the file copy, thus distorting the measurement of n. For (V3), we turned on `tracelog` after the uncompression, so some of the workload's code and data pages were already in memory when measurement began. The estimated number of fresh reads n^* was therefore smaller than it should be, so the calculated values of r might be larger than their actual values.

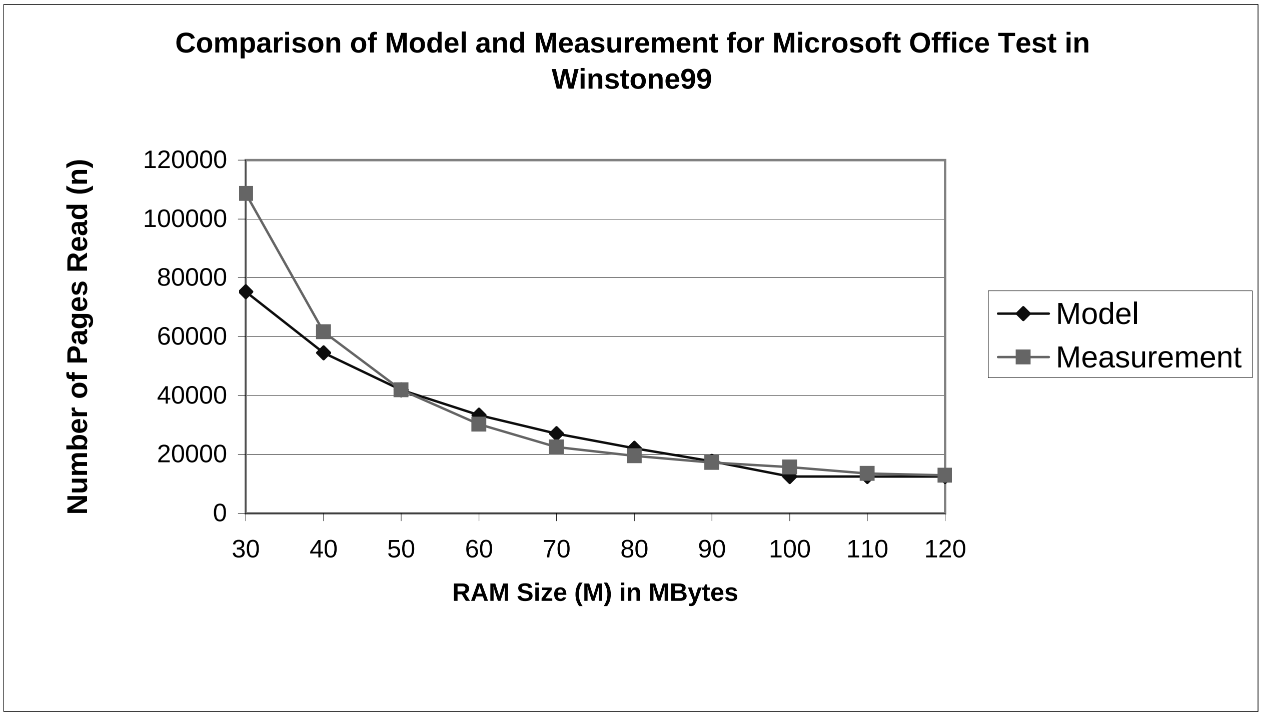
To get around this problem for the next workload (V4), we installed a debug version of Winstone99 that skipped the copy and uncompress. Unfortunately, this disabled the original version, so that we could no longer run the AVS/Express Test, and the measurements for $M < 60$ MBytes were incomplete.

(V4) *Microsoft Office Test in Winstone99*

Recall our claim in the introduction that, unlike other memory models, ours can capture the relationship in Figure 1. We now demonstrate this, as well as test the model's ability to extrapolate performance from one data point. To do so, we first estimated n^* to be 12500 pages for Figure 1, then used the data point at $M = 50$ MBytes, $n = 41901$ pages to calculate M^* from (E9) and (E6). (With appropriate instrumentation, n^* could be measured at $M = 50$ MBytes as well.) We obtained $M^* = 98.4$ MBytes. Using this value and (E10), we then plotted n against M, as shown in Figure 4.

The agreement between the model and measurements is strikingly good for such a simple model using very crude approximations. The agreement is better or worse if some other point is used for extrapolation, but that is inevitable since any model will give a graph that is nearer the measurements at some points and further at others. If, instead of using just one point, we used all points to estimate M^* (like for (V1)–(V3)), then $M^* \approx 97$ MBytes, and (E10) gives a graph similar to Figure 4.

The model and measurement are very different at $M = 30$ MBytes. There, $r \approx 8.7$, so each page is repeatedly reread — a sign that the system is thrashing. We believe it is not important that a model is inaccurate in a thrashing region, since a system should not be operating in such a



The model's curve is extrapolated from the measurement at $M = 50$ MBytes.

A similar curve is obtained if M^* is averaged over all measurements.

Figure 4 Validating (E10): The model captures the relationship between memory size and disk reads.

region anyway. (The recommended RAM size for Windows 2000 is 64 MBytes.)

The Winstone99 suite was chosen for the validation because it was one of the benchmarks that were used to measure and tune the performance of Windows 2000. Within the suite, the Microsoft Office Test — which was a workload that included Microsoft Access, Microsoft Excel, Microsoft PowerPoint, Microsoft Word and Netscape Navigator — carried most of the weightage for the final benchmark score. We also obtained similar results with MARS, which was a Microsoft benchmark widely used in-house as an alternative to Winstone99. (The workload mix was similar, but it paced the input at "typing speed", and gave response times instead of numerical scores.) We do not present the MARS measurements here since there is, unfortunately, no public documentation on MARS.

4 Capacity Planning

The equations from the model can be used for capacity planning, in sizing the memory for a new system, and in predicting the performance when moving a workload to a different machine.

(M1) *M^*: A Measure for Memory Footprint*

It is common for a workstation to be upgraded every two or three years. An operating system should have some software — a "wizard" — that advises the user on the configuration for the upgrade. For the new memory size, a natural recommendation would be M^*, the memory size for which a workload does not reread its pages (E8). The wizard can ask the user to run the workload that she wants to use for sizing the new machine, measure r, then calculate M^* with (E6).

In practice, software is strenuously tuned by developers to reduce its footprint, i.e. the number of memory pages it occupies when in use, but this number depends on the memory size that the software is tuned for. It is useful to have a measure for footprint that is independent of the memory size, that is an intrinsic measure of the memory a workload needs rather than what it uses. M^* is such a measure, and we call it the *1-read footprint* for a workload. Note, however, that the value of M^* for a portable application depends on the operating system that it runs on.

What do our measurements say about the value of M^*? As expected, M^* for (V2) was larger than for (V1), but the four values were surprisingly similar: 91, 98, 83 and 97 MBytes. We believe this is because Windows 2000 and its applications have been heavily tuned to work well in a narrow range of RAM sizes.

We checked the possibility that $M^* = n^* + B$, where B is the number of pages needed by the operating system (kernel, device drivers, etc.) to support the workload. In other words, is the 1-read footprint simply B plus the fresh reads? This is unlikely, since $M^* - n^*$ was 9600 pages for (V1), 13900 for (V2), 15300 for (V3) and 11200 for (V4). The values are quite different for (V1) and (V2), which required similar services from the operating system, yet quite similar for (V1) and (V4), which required very different services — (V1) was non-interactive and executed with one command-line shell, whereas (V4) had multiple active windows running interactive applications.

To understand the value of M^*, we turn to its definition in (E5). If we use an instruction as the unit of time, then new pages arrive at the rate of 1 per I instructions, and each page stays for J instructions, so (E5) is in fact an application of Little's Law: M^* is the number of resident pages when there is no memory constraint, i.e. Denning's working-set — the subset of pages needed by the workload to be simultaneously memory-resident. However, this subset is surprisingly large, contradicting a fundamental assumption underlying demand paging: that a process needs only a small subset most of the time. The M^* values therefore appear to confirm previous observations that some applications had no significant phase behavior and no identifiable working-sets [BHMW, GC]. Note from Figure 3, (A5), (E4) and (E5) that the number of pages waiting on disk to be reread is

$$pr\lambda(\frac{t'}{uS}) = p^2\frac{\lambda J}{uS} = p^2\frac{J}{I} = p^2 M^* = M^* - M,$$

so we can think of the 1-read footprint as the set of active pages, M of which are in RAM, and the other $M^* - M$ on disk.

(M2) *Performance Prediction for a New System*

In the previous scenario (M1), the wizard is a capacity planner that sizes a new system. Very often, the size of a new system is decided by other constraints (e.g. discounts), and the issue becomes: Should the workload be ported to the new system (assuming there are no changes in the executable binaries)? For this, the wizard can ask for the new RAM size M', then compute the predicted r' with (E7). A queueing network model (say) can then use this r', together with the new processor speed, disk latency, etc., to predict the workload's performance on the new system.

The formula can be used another way: Consider a loosely-coupled network of processors sharing a set of disks; if jobs are assigned to processors according to idle processor time and available memory, then (E7) can be used to used to estimate the disk IO and thus the performance for moving a job to a processor whose available memory is less than desirable (but who has a lot of spare processor power, say).

5 An Interpretation of Code Optimization

Code performance is an important issue in the software business, and vendors are constantly pushed by published benchmarks into optimizing their code. We now see how the model can be used to interpret some techniques in code optimization.

(M3) *Source Code Optimization*

There are numerous aspects to optimizing source code — reducing the time complexity, minimizing lock contention, etc. Because RAM sizes vary in quantum jumps, making an application that works for 48 MBytes also run on 32 MBytes can boost sales significantly, so reduction of memory requirement is one important aspect of source code optimization. Here, we give an example to illustrate how such optimization can be interpreted as reducing M^*.

Consider an index page (e.g. B-tree node) containing pointers `B[0],...,B[n-1]` to n data pages that are connected by a linked list. The data pages can be scanned by an indexed loop:

```
for(i=0;i<n;i++)
    scan(B[i])
```

However, this can be replaced by a list walk:

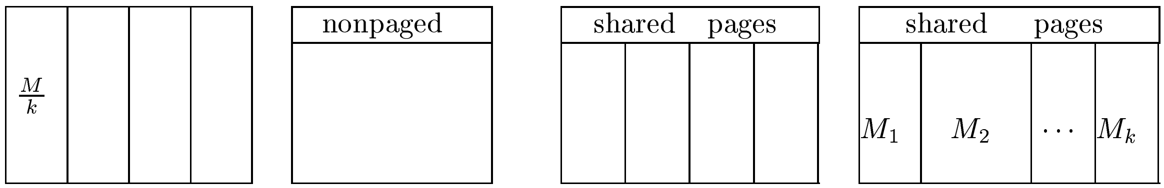
```
for (ptr=B[0]; ptr!=NULL; ptr=ptr->next)
    scan(ptr)
```

Intuitively, this change should reduce the memory requirement since the index page can be replaced after `B[0]` is used. Indeed, the optimization reduces J (how long a page is needed) and, thus, the 1-read footprint M^* (E5).

(M4) *Binary Code Optimization*

Unlike source code, binary code leaves very little room for optimization. Essentially, one can only analyze how the code is used by a workload, then reorganize it to improve its locality [PH]. We show how three common techniques in binary code optimization can be interpreted as reducing M^*.

By monitoring an application's code references for a typical workload, an optimizer can identify blocks of code that are never or seldom executed, separate them out and pack the frequently used code into a smaller number of pages, thus improving the *spatial locality* in the code. This reduces the number of disk reads for the workload, increases I and thus reduces the 1-read footprint



(a) multiprogramming (b) nonpaged pool (c) page sharing (d) nonhomgeneous processes

Figure 5. Refining the model.

M^* (E5). A second technique is branch straightening: Where there is a branch in the code, the branch more frequently taken is kept together with the code before the branch, and the other branch(es) moved elsewhere. This again improves the spatial locality, and reduces disk reads and M^*.

However, care must be taken when rearranging the binary, since it can *increase* the memory requirement. Suppose three blocks of code A, B and C are in three separate pages, and A is accessed from time 0 to 1, B from time 5 to 7, and C from time 9 to 10. Here, one page can replace another, so the memory requirement imposed by A, B and C is just 1 page frame for $1+2+1=4$ time units. If A, B and C were packed into the same page, then the memory requirement becomes 1 page for 10 time units. I increases but J also increases, thus possibly increasing M^*. The problem here is a lack of *temporal locality* among A, B and C, so code rearrangement usually orders the blocks by when they are called at run time, and packs only those blocks that are close in this ordering. In other words, the optimization tries to reduce J, and hence M^*.

We thus see that the definition of the 1-read footprint M^* in (E5) captures both spatial and temporal locality in the memory requirement of the workload. (Note: Phalke and Gopinath have a different meaning for spatial and temporal locality [PG].)

6 Refining the Model

Recall that our top-down approach is to adopt a simple model and refine it as and when necessary. We now introduce a few such refinements.

Our model can be used to incorporate the relationship between multiprogramming level and branching probabilities into a queueing network model, as follows: For multiprogramming level k, divide the memory M into k slices, like in Figure 5(a), and restrict (E6) to one slice:

$$\frac{M/k}{1-(1-\frac{1}{r})^2} = \text{ constant} \qquad (E11)$$

where the constant is the 1-read footprint for a job (e.g. transaction) in the multiprogramming mix. This footprint can be measured or estimated (M1) by, say, running a job alone. The relationship between k and r in (E11) can then be used to modify the branching probabilities as k changes in the multiprogramming model. (This modification means exact Mean Value Analysis cannot be used to solve the model in one recursion, but an approximation like Schweitzer's can still be used [LZGS].)

An operating system usually has a set of pages that are not subject to page replacement by the memory manager (e.g. the nonpaged pool in Windows 2000). Applications like database systems may also prevent some of their pages from being replaced. To take such pages into account, the model can restrict M to just those pages for which the assumption (A2) applies: Let M_{RAM} be the RAM size and M_{nonpaged} be the total size of nonpaged memory. Then (see Figure 5(b)),

$$\frac{M_{\mathrm{RAM}} - M_{\mathrm{nonpaged}}}{1-(1-\frac{1}{r})^2} = \text{ constant.} \tag{$E12$}$$

The multiprogramming model in Figure 5(a) does not reflect the fact that processes share libraries, etc. To a first approximation, we can assume a shared page is less likely to be replaced than a nonshared page, and therefore treat shared pages like they form a nonpaged pool of size M_{shared}. We can then combine the two previous refinements to give Figure 5(c) and

$$\frac{(M_{\mathrm{RAM}} - M_{\mathrm{shared}})/k}{1-(1-\frac{1}{r})^2} = \text{ constant.} \tag{$E13$}$$

Of course, the processes in the multiprogramming mix may not be homogeneous and have different memory requirements, as illustrated in Figure 5(d). We call the nonshared pages for a process its *private working set*. If the memory size for this set is M_i, we have

$$\frac{M_i}{1-(1-\frac{1}{r_i})^2} = M_i^* \ , \tag{$E14$}$$

where r_i is the average number of times a page is read into this set, and M_i^* is the *1-read private footprint*. Note that, unlike the 1-read footprint M^* discussed in (M1), the 1-read private footprint does not include shared pages in the operating system (e.g. device drivers).

7 A New Perspective

Our top-down approach offers a new perspective on issues of memory size. We illustrate this here for directory service caching and working set trimming.

(M5) *Directory Service Caching*

A *directory* is a database of *entries*, each of which is a set of attributes for an entity. For example, a directory for the domain `www.ietf.org` may contain an entry for each user and each device (e.g. printer), and an attribute may be an email address or an access list. A directory service allows clients to retrieve information from the directory [WHK]. Typically, most of a directory server's memory is used as a cache for entries. A client accesses a directory by setting up a connection, over which the client may issue multiple operations on the directory. To process these operations, the server retrieves relevant entries and extracts appropriate information for the client. Since the client may query these entries again, the server should try to cache those entries as long as the connection is open.

As with any cache management problem, the focus is *cache miss*, i.e. the probability that the server accesses an entry that must be retrieved from disk. For an entry that was retrieved previously, this becomes a reread, so the miss probability is related to p in our memory model. This, plus the fact that an entry is typically contained in one page, leads us to applying the model to the Directory Service cache.

For simplicity, assume each entry is smaller in size than a page, and no entry is accessed by two concurrent connections. (The model can be refined to remove these assumptions.) By studying the profile of client requests, one can estimate how many accesses (including repetitions) are made by a connection to the entries in a directory; let this number be N_{access}. Suppose the server reads n_{read} pages from disk (including rereads) for a connection. Then $n_{\text{read}} \leq N_{\text{access}}$ and the probability that an access requires a disk read is $P_{\text{miss}} = n_{\text{read}}/N_{\text{access}}$. If the number of entries accessed (no double-counting) is n_{entry}, then each entry is retrieved r times, where $r = n_{\text{read}}/n_{\text{entry}}$. If the number of concurrent connections is $N_{\text{connection}}$, set $M^* = N_{\text{connection}} n_{\text{entry}}$ (in pages), so that $r = 1$ if the cache size is M^*. If the cache can accommodate only $M_{\text{cache}} (< M^*)$ pages, set $M = M_{\text{cache}}$. Then, r is given by (E6), so

$$P_{\text{miss}} = \frac{n_{\text{entry}}}{N_{\text{access}}\left(1 - \sqrt{1 - \frac{M_{\text{cache}}}{N_{\text{connection}} n_{\text{entry}}}}\right)} \,. \tag{E15}$$

This estimate of P_{miss} can then be used to calculate the response time, IO rate, etc. For small M_{cache}, a first-order approximation of the square root gives $P_{\text{miss}} = 2N_{\text{connection}} {n_{\text{entry}}}^2 / N_{\text{access}} M_{\text{cache}}$, which indicates the quadratic effect from the number of entries accessed by a connection. Just as (E6) and (E10) can be modified to set upper bounds for r and n for small M, (E15) can also be modified so $P_{\text{miss}} \leq 1$ for small M_{cache}. In practice, a server must have a cache that is big enough to contain at least a few entries per connection, so M_{cache} is a sizable fraction of $N_{\text{connection}} n_{\text{entry}}$.

To complete our re-interpretation of the model for caching, I is the number of instructions executed at the server between requests for a new entry by a client, and J similarly measures the time span for a client connection.

(M6) *Working Set Trimming*

When memory is tight, the memory manager must begin to *trim* the working sets. This means some page frames are removed from a working set and prepared (e.g. schedule a write if the page is dirty) for release to the free page list. The trimming mechanism — selection of candidate working sets, how much to trim and which pages to trim — depends on the operating system, and may use such factors as working set size, page fault rate and how long a page is memory-resident. We now apply our model to address one of these issues: How much to trim?

Suppose there are k processes in the multiprogramming mix, of which $h (< k)$ have been selected by some criteria as candidate working sets for trimming. Let the private working set sizes (i.e. the number of nonshared pages in the working sets) be $M_1, M_2, \ldots, M_h$. We want to determine the number of pages Δ_i to be trimmed from M_i, so that $\Delta_1 + \Delta_2 + \cdots + \Delta_k = \Delta$, where Δ is the target number of page frames the memory manager wants to recover and release to the free list. An obvious way to determine Δ_i is $\Delta_i = (M_i / \sum_{j=1}^{h} M_j)\Delta$, but this trims a working set according to what it has (M_i) and not what the process needs (namely, M_i^*). On the other hand, $\Delta_i = (M_i^* / \sum_{j=1}^{h} M_j^*)\Delta$ trims according to what a process needs, but not what it has.

For trimming to be fair, a natural criterion would be that each candidate has the same rereads per page after trimming, i.e. $r_1' = \cdots = r_k' = r$, say. Then, by (E14),

$$\frac{M_i - \Delta_i}{1-(1-\frac{1}{r})^2} = M_i^* \qquad \text{for } i = 1, \ldots, h.$$

Summation over i gives

$$\frac{(\sum_{i=1}^{h} M_i) - \Delta}{1-(1-\frac{1}{r})^2} = \sum_{i=1}^{h} M_i^* \ ,$$

so

$$\frac{M_i - \Delta_i}{(\sum_{j=1}^{h} M_j) - \Delta} = \frac{M_i^*}{\sum_{j=1}^{h} M_j^*} \ .$$

This gives the following

Rule of Thumb for Trimming : $\qquad \Delta_i = M_i - \frac{M_i^*}{\sum_{j=1}^{h} M_j^*}(\sum_{j=1}^{h} M_j - \Delta) \qquad$ (E16).

This rule uses both what a process has and what it needs to decide the trim amount.

To use the rule, there must be some way of estimating M_i^*. This can be done if the operating system keeps track of r_i, since it can then use (E14) to calculate M_i^* on the fly. However, one should bear in mind that the derivation for (E6) is based on averaging over an entire workload, so some engineering is necessary when using (E14) to estimate M_i^* midstream. For example, do not use the rule if the process is young, and smooth out the moving average r_i.

8 Behavioral Analysis

Computer systems are becoming more complex and their behavior harder to understand. We need tools to help us cut through the complexity, so that we can see how competing forces shape system behavior, and thus develop better intuition. Our simple model is such a tool, and we now demonstrate its use in analyzing how two fundamental forces interact in a system.

(M8) *The Interaction Between Memory Contention and Processor Contention*

There was some interest recently on the interaction between processor scheduling and memory usage in the context of parallel and distributed applications [MZ, PS, PSN]. We take a step back to consider: How do they interact on a single processor? Intuitively, when a process runs alone, if it gets more memory, then it can better utilize the processor. Is this true when two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ share one processor and one memory? Can the following chain of events occur: (1) $\mathcal{P}_1$ gets some page frames from $\mathcal{P}_2$; (2) $\mathcal{P}_2$ makes slower progress; (3) $\mathcal{P}_2$ issues less disk IO; (4) $\mathcal{P}_1$ has less opportunity to use the processor by itself, so slows down? When page frames are taken from $\mathcal{P}_2$, can it be compensated by a higher processor priority, so that its progress is unaffected? To study this interaction, we begin with an assumption:

(A6) Reads from disk are synchronous and writes are asynchronous.

This and the assumption that IO instruction overheads are negligible (A4) make writes invisible except to the disks. For reads, (A6) is an approximation, since the operating system may anticipate file access by reading ahead. Fix the memory size M. Fix a time span too, and suppose the processor can execute T instructions in that time. Let D be the read delay (seek time, etc.) measured by the number of instructions the processor can execute in that delay. We also assume:

(A7) There is only one processor.

For process $\mathcal{P}_i$ $(i = 1, 2)$, let u_i be its processor utilization, M_i its private working set, M_i^* its 1-read private working set, p_i its probability of rereading a page, and I_i the number of instructions it executes between fresh reads; furthermore, suppose $\mathcal{P}_i$ reads n_i pages from disk during T, of which n_i^* are fresh reads.

Running $\mathcal{P}_1$ Alone

We first confirm our intuition that, when running alone, a process will have higher processor utilization if it gets more memory. Since $\mathcal{P}_1$ executes $n_1^* I_1$ instructions in time T and it is running alone, we have $T = n_1^* + n_1 D$ and utilization is $u_1 = n_1^* I_1 / T$. Together with the basic equation $M_1^*(1 - (1 - \frac{n_1^*}{n_1})^2) = M_1$, we get

$$\frac{du_1}{dM_1} = \alpha_1 \frac{D}{I_1} u_1^2 \qquad \text{where } \alpha_1 = \frac{1}{2M_1^* p_1 (1 - p_1)^2} \text{ (definition),} \qquad (E17)$$

so $\frac{du_1}{dM_1} > 0$, i.e. processor utilization u_1 does increase with memory allocation M_1, as expected.

Running $\mathcal{P}_1$ and $\mathcal{P}_2$ Together

Suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ have (possibly) different priorities at the processor and, whenever they are both ready to run, $\mathcal{P}_1$'s share of processor time is Q times $\mathcal{P}_2$'s share. Here, Q may be larger or smaller than 1. To see the interplay between the two forms of contention, we assume the processor is fully utilized and page sharing (including any nonpaged pool) is negligible. Then,

$$u_1 = \frac{n_1^* I_1}{T}\,, \quad u_2 = \frac{n_2^* I_2}{T} \quad \text{and} \quad T = n_1^* I_1 + n_2^* I_2 \qquad (E18)$$

so $u_1 + u_2 = 1$ (this captures the processor contention). During T, the two processes compete at the processor for time $T - (n_1 D + n_2 D)$; suppose $\mathcal{P}_2$ got the processor for time τ, so $\mathcal{P}_1$ got $Q\tau$ and $\tau + Q\tau = T - (n_1 D + n_2 D)$. Then

$$n_1^* I_1 = n_2 D + Q\tau \quad \text{and} \quad n_2^* I_2 = n_1 D + \tau. \qquad (E19)$$

Again, we have the basic equations

$$M_1^*(1 - (1 - \frac{n_1^*}{n_1})^2) = M_1 \quad \text{and} \quad M_2^*(1 - (1 - \frac{n_2^*}{n_2})^2) = M - M_1 \qquad (E20)$$

since $M_1 + M_2 = M$ (this captures the memory contention). From (E18), (E19) and (E20), we get

$$\frac{dn_1^*}{dM_1}((Q+1)I_1 I_2 + \frac{n_2}{n_2^*} D I_1 + \frac{n_1}{n_1^*} Q D I_2) = ((n_2^* \alpha_2 + Q n_1^* \alpha_1) D + \frac{dQ}{dM_1}\tau) I_2 \qquad (E21)$$

where $\alpha_2 = 1/(2M_2^* p_2(1-p_2)^2)$ like in (E17).

Suppose there is no change in priority In other words, $\frac{dQ}{dM_1} = 0$. Then, $\frac{dn_1^*}{dM_1} > 0$ (E21) and $\frac{du_1}{dM_1} > 0$ (E18). Therefore, the chain of events described before cannot occur, and $\mathcal{P}_1$ will always increase its processor utilization if its memory allocation increases. This is regardless of $\mathcal{P}_2$'s priority (as long as that priority does not change).

Suppose there is a change in priority If the priority changes with M_1, but $\frac{dQ}{dM_1} > -(Qn_1^*\alpha_1 + n_2^*\alpha_2)\frac{D}{\tau}$, then again, by (E21), $\mathcal{P}_1$'s processor utilization increases with M_1. However, if $\mathcal{P}_1$'s priority decreases and $\frac{dQ}{dM_1} < -(Qn_1^*\alpha_1 + n_2^*\alpha_2)\frac{D}{\tau}$, then $\frac{dn_1^*}{dM_1} < 0$, so $\mathcal{P}_1$'s utilization drops despite having more memory. Even so, the chain of events is not the one outlined before, since $\mathcal{P}_2$'s utilization in fact increases, so $\mathcal{P}_2$ is making more progress than $\mathcal{P}_1$. To answer the last question posed above — Yes, $\mathcal{P}_2$ can be compensated for its memory loss by higher priority, to the tune of $\frac{dQ}{dM_1} = -(Qn_1^*\alpha_1 + n_2^*\alpha_2)\frac{D}{\tau}$. Such insights are hard to get from models (e.g. stochastic matrices) that analyze behavior through numerical solutions.

9 Conclusion

We conclude with a summary and a discussion of future work.

Summary

This paper addresses the open question of how memory size affects disk reads. We have presented a very simple, operating system-independent model that is nevertheless supported by preliminary measurements (V1)–(V4) made with Windows 2000. Moreover, it offers the following features:

(1) The top-down view gives a new perspective on issues of memory size: The model shows how optimization of source code (M3) and executable binaries (M4) can both be interpreted as reducing the 1-read footprint M^*. It also gives a new interpretation of cache miss for directory service, and thus yields a closed form for the miss probability (E15). The model shows how an operating system can estimate the memory needed by a process with M^*, and use it to trim working sets (E16).

(2) The model's analytic tractability facilitates the analysis of system behavior: It reveals the quadratic performance effect of the number of entries accessed by a directory service connection (M6). The analysis of how memory contention interacts with processor contention shows that a smaller memory allocation can be compensated by a higher processor priority; without a change in priority, a larger memory allocation always allows a process to use more of the processor (M8). Such insights are hard to get from models (e.g. stochastic matrices) that analyze behavior through numerical solutions.

(3) The model is useful for capacity planning: It can be used to size the RAM for a new machine (M1), size the cache for a directory server (E15), or predict performance when changing a system (M2).

Future Work

Measurements The amount of experimental data we present here to support the model is a lot less than what we hoped for. Some data are omitted (e.g. MARS), and much time was lost to incomplete experiments. For example, we tried making NetBench measurements with up to 144 (physical) clients. These measurements were plagued by overflows in the `tracelog` buffers, interruptions from more urgent measurement tasks for the laboratory, and upgrades to NetBench, `tracelog` and Windows 2000. We never had a consistent set of software long enough to finish the measurements. We also could not get data to validate the formula for directory service caching: For the directory server that we experimented with and the directory available in the laboratory, the smallest RAM size that allowed the server to boot was large enough to hold the entire directory in its cache.

More measurements will have to be taken to test the applicability of the model for different workloads (e.g. scientific applications [BHMW]) and operating systems (e.g. Linux), and validate (E15).

Writes Memory size affects disk writes because, if memory is tight, a page may have to be written out and replaced before more changes can be made to the page. In flushing a dirty page, the operating system may first put the page on a modified page list to wait for a disk write; if the process accesses the page while it is waiting, it will be removed from the list and returned to the process's working set. This is a *soft fault*, whereas the rereads in this paper are *hard faults* (i.e. retrievals from disk).

To capture the effect on disk writes, one may therefore have to model soft faults. This can be done by excluding from M the various page lists (for Windows 2000: modified, standby, free and zeroed page lists) and modifying (A5) to reflect the relationship between the probability that a page is reclaimed from these lists, and how long it stays there. This relationship will likely depend on how the operating system manage these lists, and may therefore make the model operating system-specific.

Replacement Policies There are many page replacement policies in the literature, and they have been demonstrated to have different performance impact. Windows 2000 also has a sophisticated policy of its own. Yet, without modeling this policy, the formulas were in agreement with the measurements. We believe this is because most (reasonable) replacement algorithms have similar average behavior, unless they are stressed. One possible way of closing the gap between model and measurement at $M = 30$ MBytes in Figure 4, for example, may be to refine the model with the replacement policy. This can be done by studying how the policy affects p and t', and adopting a better approximation than the linear equation in (A5).

The significance in this observation actually lies in the caching problem (M5). Our model provides a common analytic framework for different cache replacement policies; once a policy is analyzed to give an equation relating p and t', that relationship can replace (A5) and the model solved to give a formula for the policy's miss probability, like in (E15). This is a new perspective on caching analysis that should be explored.

Memory Management A memory manager that allows a working set to grow up till a maximum threshold, then begins to trim it, should perhaps have stopped that growth earlier. This is hard to do since the operating system has no way of knowing how many pages a process needs. Our model offers the 1-read footprint M^* as a measure of that need, and we have demonstrated its use in trimming. We believe M^* can also be used to limit working set growth (and make the maximum threshold unnecessary).

However, as previusly mentioned, the engineering aspects of estimating M^* with minimal overhead requires more research, just as the least-recently-used policy spawned several different implementations for operating systems.

Space constraint prevents us from discussing other possible lines of development. One final example: The number of pages waiting on disk to be reread is $M^* - M$. If the disks are replaced by other RAMs on a network, $M^* - M$ becomes another handle for analyzing the use of memory servers [CG], global memory [V∗] and idle remote memory [AS].

References

[AS] A. Acharya and S. Setia, *Availability and utility of idle memory in workstation clusters*, Proc. SIGMETRICS, Atlanta, GA (May 1999), 35–46.

[BHMW] D.C. Burger, R.S. Hyder, B.P. Miller and D.A. Wood, *Paging tradeoffs in distributed shared-memory multiprocessors*, Proc. Supercomputing, Washington, DC (Nov. 1994), 590–599.

[BM] A. Brandwajn and W. McCormack, *Efficient approximation for models of multiprogramming with shared domains*, Proc. SIGMETRICS, Cambridge, MA (Aug. 1984), 186–194.

[BMM] R.B. Bunt, J.M. Murphy and S. Majumdar, *A measure of program locality and its application*, Proc. SIGMETRICS, Cambridge, MA (Aug. 1984), 28–40.

[CG] D. Comer and J. Griffioen, *A new design for distributed systems: the remote memory model*, Proc. USENIX Summer Conference, Anaheim, CA (June 1990), 127–135.

[GC] G. Glass and P. Cao, *Adaptive page replacement based on memory reference behavior*, Proc. SIGMETRICS, Seattle, WA (June 1997), 115–126.

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

[LZGS] E. Lazowska, J. Zahorjan, S. Graham and K. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, Englewood Cliffs, NJ (1984).

[MAD] D.A. Menascé, V.A.F. Almeida and L.W. Dowdy, *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*, Prentice Hall, Upper Saddle River, NJ (1994).

[MB] J.M. Murphy and R.B. Bunt, *Characterizing program behavior with phases and transitions*, Proc. SIGMETRICS, Santa Fe, NM (May 1988), 226–234.

[MZ] C. McCann and J. Zahorhan, *Scheduling memory contrained jobs on distributed memory parallel computers*, Proc. SIGMETRICS, Ottawa, Ontario, Canada (May 1995), 208–219.

[PG] V. Phalke and B. Gopinath, *An inter-reference gap model for temporal locality in program behavior*, Proc. SIGMETRICS, Ottawa, Ontario, Canada (May 1995), 291–300.

[PH] K. Pettis and R.C. Hansen, *Profile guided code positioning*, Proc. PLDI, White Plains, NY (June 1990), 16–27.

[PS] E.W. Parsons and K.C. Sevcik, *Coordinated allocation of memory and processors in multiprocessors*, Proc. SIGMETRICS, Philadelphia, PA (May 1996), 57–67.

[PSN] V.G.J. Peris, M.S. Squillante and V.K. Naik, *Analysis of the impact of memory in distributed parallel processing systems*, Proc. SIGMETRICS, Nashville, TN (May 1994), 5–18.

[SKW] Y. Smaragdakis, S. Kaplan and P. Wilson, *EELRU: Simple and effective adaptive page replacement*, Proc. SIGMETRICS, Atlanta, GA (May 1999), 122–133.

[V∗] G.M. Voelker, H.A. Jamrozik, M.K. Vernon, H.M. Levy and E.D. Lazowska, *Managing server load in global memory systems*, Proc. SIGMETRICS, Seattle, WA (June 1997), 127–138.

[WHK] M. Wahl, T. Howes and S. Kille, *Lightweight directory access protocol (v3)*, Request for Comments, RFC 2251 (Dec. 1997), `http://www.ietf.org`.

Acknowledgement

Microsoft Office, Microsoft Access, Microsoft Word, Microsoft PowerPoint are registered trademarks and Windows 2000(TM) is a trademark of Microsoft Corporation. NetBench and Winstone are registered trademarks of Ziff-Davis Publishing Company. Netscape Navigator(TM) is a trademark of Netscape Communications Corporation. AVS/Express is a registered trademark of Advanced Visual Systems Inc. Celeron is a trademark of Intel Corporation. Seagate is a registered trademark of Seagate Technology, Inc.