The Expected Lifetime of "Single-Address-Space" Operating Systems

David Kotz and Preston Crow
Dartmouth College
Hanover, NH

Technical Report PCS-TR93-198

Revised March 15, 1996*

Abstract

Trends toward shared-memory programming paradigms, large (64-bit) address spaces, and memory-mapped files have led some to propose the use of a single virtual-address space, shared by all processes and processors. To simplify address-space management, some have claimed that a 64-bit address space is sufficiently large that there is no need to ever re-use addresses. Unfortunately, there has been no data to either support or refute these claims, or to aid in the design of appropriate address-space management policies. In this paper, we present the results of extensive kernel-level tracing of the workstations on our campus, and discuss the implications for single-address-space operating systems. We found that single-address-space systems will probably not outgrow the available address space, but only if reasonable space-allocation policies are used, and only if the system can adapt as larger address spaces become available.

1 Introduction

Many researchers have proposed single-address-space operating systems. With such systems, the entire memory hierarchy is mapped into a single large address space, including files and processes, and often remote memories of other machines. A good discussion of the advantages, disadvantages, and other issues concerning such systems can be found in [Chase *et al.* 1994].

One of the major problems with single-address-space operating systems is managing the address space. Once space has been allocated, it is often preferable not to reallocate the same space for other purposes. Hence, over time, the address space will eventually be consumed. Previous work has not studied the rate at which this consumption will take place.

In this paper, we examine the issue of address-space consumption, based on traces of Ultrix-based workstations running computer-science, numerical-analysis, and server workloads. Though we recognize that

*Revisions from the original release of this report include additional data for server 1, extensive modifications to the text, and the addition of Figure 10.

This research was supported in part by NASA Graduate Student Research Assistantship NGT-51160, and by Digital Equipment Corporation through ERP contract number 2043.

applications under a single-address-space operating system would behave somewhat differently, we believe that the data gathered from these workloads lays a basic foundation for understanding consumption rates.

In the next section we examine some of the previous work in single-address-space operating systems, focusing on their assumptions of address-space usage. In Section 3, we discuss our trace collection and the analysis of current usage patterns. In Section 4, we show how we used this data to predict the lifetime of single-address-space operating systems. Finally, in Section 5, we summarize.

2 Background

The MONADS-PC project [Broessler *et al.* 1987, Rosenberg *et al.* 1992, Rosenberg 1992] was one of the first systems to place all storage (all processes and all files) in a single, distributed, virtual-address space. They use custom hardware that partitions the bits of an address into two fields: a 32-bit address-space number and a 28-bit offset. The address-space numbers are never re-used. A newer version of the system, MONADS-MM [Koch and Rosenberg 1990], uses 128-bit addresses, extending the address-space numbers to 96 bits and the offsets to 32 bits. The MONADS project does not report on any experience with a long-running system and its address-space consumption.

Hemlock [Garrett *et al.* 1992] proposes a single 64-bit address space. Persistent and shared data are allocated a non-reusable segment of the address space. Files are mapped into contiguous regions in the address space, requiring them to allocate a large address range (4 GB) for each file to leave room for potential expansion. This fragmentation may limit the effective size of their (64-bit) address space. Another characteristic of their model is that they "reserve a 32-bit portion of the 64-bit virtual address space for private code and data." This exception from the otherwise single address space simplifies some relocation issues and provides a limited form of re-use. Hemlock dynamically links code at run time to allow for different instances of global data.

Opal [Chase *et al.* 1994] uses other techniques to avoid Hemlock's "private" 32-bit subspace and dynamic linking. For example, all global variables are referenced as an offset from a base register, allowing separate storage for each instance of the program (this technique is also used in the Macintosh operating system [Wakerly 1989]). They concede that conserving and re-using address space is probably necessary.

Bartoli *et al.* point out that that "if ten machines create objects at a rate of ten gigabytes a minute, the [64-bit] address space will last 300 years" [Bartoli *et al.* 1993]. Hence, a collection of 200 machines would only last 15 years, and larger collections would likely be out of the question.

Patterson and Hennessy claim that memory requirements for a typical program have grown by a factor of 1.5 to 2 every year, consuming 1/2–1 address bits per year [Patterson and Hennessy 1990]. At this rate, an expansion from 32 bits to 64 bits would only last 32–64 years for traditional operating systems, and a single-address-space operating system would run out sooner.

Though most researchers recognize that even a 64-bit address space presents limits for a single-address-space operating system, there is not any real understanding of the rate of address-space consumption, and that some data is needed. This problem was the motivation for our work.

3 Current usage

To provide a basis for our analysis of single-address-space systems, we first measured address-space usage in current operating systems. Our goals were to determine the rate that address space was used in our current operating systems, and to collect traces to use in trace-driven simulations of future address-management policies. For two servers and two workstation clusters on campus, we traced the events that may consume address space in a single-address-space system, recording every system call that could create or change the size of files, shared-memory segments, process data segments, and process stack segments.

The data we collected differs from most previous studies in that it measures virtual rather than physical resources. We did not take into account the text-segment size, assuming that it would be allocated at compile time.[1] Table 1 summarizes the traces we collected.

To collect this data, we modified the DEC Ultrix 4.3 kernel[2] to generate a trace record for all relevant activities. In particular, we recorded every exec, fork, exit, sbrk, stack increase, shared-memory creation, shared-memory deallocation, unlink, open (for write only), close, truncation, and write.

Our method was modeled after the Ultrix error-logging facility. The kernel stored trace records in an internal 20 KB buffer, which was accessible through a new device driver that provided a file-like interface to the buffer. A user-level trace daemon opened the device, and issued large (20 KB) read requests. When the internal buffer contained sufficient data (15 KB), the kernel triggered the device driver, which then copied the data to the trace daemon's buffer, and woke the trace daemon. The kernel buffer was then available for new data, while the trace daemon wrote its buffer to a trace file. The activity of the trace daemon, and thus of

[1] With dynamic linking, as in Hemlock, the addresses allocated for the text segment could likely be re-used.

[2] DEC and Ultrix are trademarks of Digital Equipment Corporation. Ultrix 4.3 is a variant of Unix 4.2BSD. Unix is a trademark of X/Open.

Group	Days	Records	Lost records	Processes	Lost processes
Server 1	23.8	36895501	18564 (0.05%)	1640775	930 (0.06%)
Server 2	25.3	6595110	61709 (0.94%)	114435	99 (0.09%)
Cluster 1	22.9	915718	614 (0.07%)	39041	0 (0.00%)
	22.9	3667000	6 (0.00%)	40110	0 (0.00%)
	22.9	378430	1409 (0.37%)	33707	2 (0.01%)
	22.9	3293680	19351 (0.59%)	122402	92 (0.08%)
	22.9	417550	26 (0.01%)	45423	3 (0.01%)
	23.0	884393	2 (0.00%)	49144	0 (0.00%)
	22.9	1402850	132692 (9.46%)	51669	0 (0.00%)
	22.9	1343890	3180 (0.24%)	61480	0 (0.00%)
	23.0	849289	5995 (0.71%)	54974	0 (0.00%)
	22.1	601798	2100 (0.35%)	49277	110 (0.22%)
	23.0	1850030	0 (0.00%)	190958	0 (0.00%)
	22.9	605955	88 (0.02%)	42666	0 (0.00%)
	Total	16210583	165463 (1.01%)	780851	331 (0.04%)
Cluster 2	29.4	9792880	175785 (1.80%)	405368	111 (0.03%)
	29.4	1082960	16144 (1.49%)	49859	78 (0.16%)
	29.4	610202	6051 (0.99%)	48196	58 (0.12%)
	29.4	486763	5458 (1.12%)	42920	57 (0.13%)
	Total	11972805	203438 (1.67%)	546343	304 (0.06%)

Table 1: Summary of the traces collected. Server 1 was used as a general-purpose Unix compute server by many people on campus. Server 2 was the primary file, mail, and ftp server in our computer-science department. Cluster 1 included general-use workstations in the computer-science department, most located in faculty offices. Cluster 2 contained workstations used primarily by a compute-intensive signal-processing research group. All workstations were DECstation 5000s running Ultrix 4.3. A small fraction of records were lost in the collection process, accounting for a generally even smaller fraction of processes not being accounted for (see Section 3 for details). These data were collected in fall 1993.

the trace files, was explicitly excluded from the trace by the kernel. This buffering strategy decoupled trace generation from disk writes so that no activity was ever significantly delayed to write trace records to disk, and so that the overhead was amortized across large groups of trace records. While it is not a new technique, we highly recommend this simple, unobtrusive, portable mechanism for other trace-collection efforts.

To measure the performance overhead of our tracing activity, we ran 25 trials of the Andrew benchmark [Satyanarayanan 1989] on the standard Ultrix 4.3 kernel and on our instrumented kernel. The Andrew benchmark extensively uses most of the system calls we modified for tracing, by creating, searching, and deleting files, and compiling programs. We ran 25 trials with the standard kernel and with the tracing kernel. We discarded the first trial in each case, due to a cold file cache. An unpaired t-test [Jain 1991] showed the difference to be insignificant at the 99% confidence level, implying that our tracing apparently had no significant effect on performance. This result matches our qualitative experience (no users reported any perceived difference).

Group	bytes per second mean	95th	max	records per second mean	95th	max
Server 1	675	2265	87619	17.9	56	3122
Server 2	115	630	60968	3.0	13	2159
Cluster 1	19	0	41311	0.5	0	1470
	79	520	19628	1.9	11	701
	8	0	14364	0.2	0	513
	65	171	58408	1.7	4	2065
	10	0	32398	0.2	0	755
	18	0	22484	0.4	0	803
	24	28	92524	0.7	1	3295
	27	0	54580	0.7	0	1943
	17	0	85156	0.4	0	3035
	14	0	11005	0.3	0	385
	40	115	34592	0.9	3	1217
	12	0	45194	0.3	0	1065
All	28	40	92524	0.7	1	3295
Cluster 2	156	1470	66956	3.8	35	2385
	16	0	50096	0.4	0	1782
	10	0	11312	0.2	0	404
	8	0	21196	0.2	0	757
All	48	0	66956	1.2	0	2385

Table 2: Amount of trace data collected per second in bytes and records. The mean, 95th percentile, and maximum values are listed for each machine. The relatively low 95th percentiles indicate that trace data generation was very bursty. Figures are listed for each machine, as well as the overall figure for each cluster.

After collection, the raw trace files were post-processed to clean up the data. In particular, the raw trace files were missing a small percentage of the trace records, as indicated by record-sequence numbers. This loss was caused by the trace buffer occasionally filling up before the trace daemon could read it, or, in one case, the trace disk running out of space. In most cases, the effect of the missing records was simulated, the data being inferred from subsequent events. For example, a missing process-fork record was inferred from a subsequent process-exec or process-exit record. Only a fraction of a percent of processes were missed entirely due to the missing records. When a large number of records were lost, the usage that they would have reflected was not recorded. As shown in Table 2, trace data generation was very bursty, suggesting that a larger collection buffer may have been preferable. Fortunately, fewer than two percent of the records were missing from any trace group, with less than a tenth of a percent of processes unaccounted for, indicating that the effect on the usage rates should be quite small, most likely underestimating usage by less than 1%.

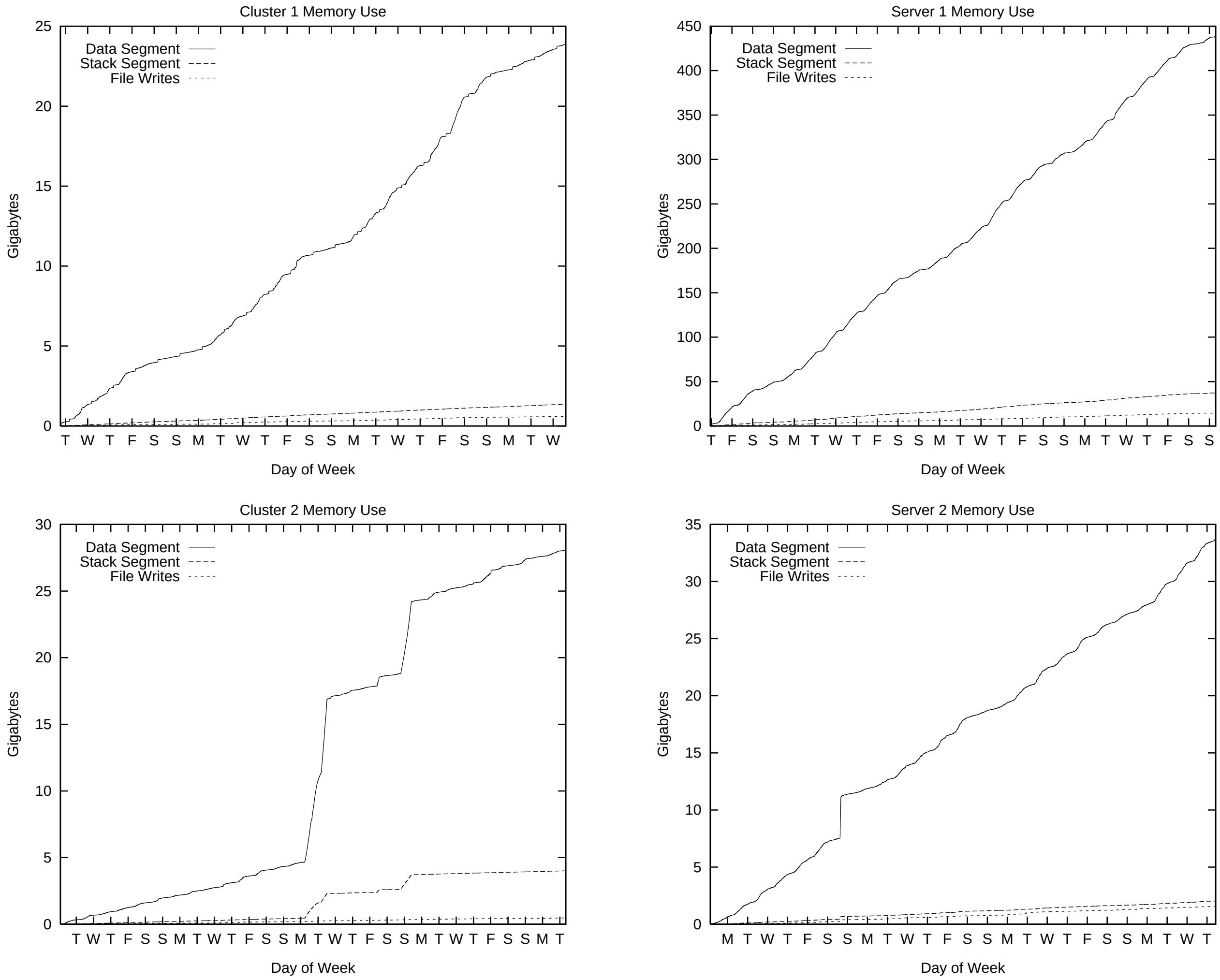

Figure 1: Cumulative address-space usage for all workstations in each trace group, separated by category of memory usage. Curves for Cluster 1 and Cluster 2 are scaled down by the number of machines in each cluster, for easier comparison. Shared Memory, if plotted, would be indistinguishable from zero. x-axis tic-marks represent midnight before the given day of the week.

3.1 Results

In Figure 1, we show the raw amount of address space allocated (in units of 4 KB pages) over time, for each of the four trace groups defined in Table 1. This figure is based on a running sum of the size of private-data segments, stack segments, shared-data segments, and file creations or extensions. Clearly, most of the usage was from data segments, with stack segments second. Shared data was rarely used on our systems (only by the X-windows server, apparently to share the frame buffer with the device driver), and is not shown in the figure. Daily and weekly rhythms are clearly visible. Server 1, heavily used for timesharing, used over ten times as much space. Cluster 2, used by a signal-processing research group, occasionally saw large bursts of activity caused by applications with large data segments.

To discover the nature of the significant address-space users, we compiled a list of the top 40 programs by address-space allocated, shown in Table 3. Most of the big users were not huge user applications, but instead common programs like the shells *sh* and *csh*, which were run often for scripts, the *gzip* compression program, which was run by nightly space-saving scripts, *is_able*, which was run by nightly system jobs on server 1, pieces of the C compiler (*ugen2.1, cc1, ccom2.1, as12.1,* and *ld2.1*), and periodic background processes (*init, sendmail, in.cfingerd, amd, named-xfer, in.fingerd*, and *atrun*). Only two programs on this list (*mm2*, a signal-processing application, and *ip*, an image-processing application), were user-written applications; all of the others were common applications used by many users. Only one (*ip*) could be called a large application. These data make it clear that policies that statically allocate a large region to every process would waste a lot of virtual-address space on many small but common applications.

In determining the amount of address space consumed by a process, we had to select a model for interpreting the *fork* and *exec* system calls. There were several alternatives:

- Both a *fork* and an *exec* would be interpreted as creating a new process, using new address space for stack and data segments. This would result in an overestimation of the space consumed due to the frequent use of a *fork*-*exec* pair of calls in creating a new process.
- The address space consumed by a *fork* call duplicating the data and stack segments of the parent process could be ignored, assuming that the use of a combined *fork*-*exec* call in a newer operating system would eliminate that usage. While this does effectively model the *fork*-*exec* paradigm, we found that some programs, accounting for 28% of all processes, did not follow this paradigm, and would have their space usage ignored by this interpretive model.
- The address space consumed by an *exec* call could be ignored, assuming that the process could overwrite the previous data and stack segments, only recording address-space consumption if the resulting process had a larger data or stack segment than the old process. This is the model we selected.

In attributing consumption to a program, all consumption by a process was attributed to the last program that was *exec*'d. In the case where a process did not call *exec*, the consumption was attributed to the program that the parent process was running when the *fork* that created the process was issued. This means that while a shell may have *fork*'d itself for each non-intrinsic command that the user issued, the space consumption for the new process was credited to the program the user ran, not to the shell.

Program Name	Instances	Pages Used Total	Pages Used Per Instance	CPU Seconds Used Total	CPU Seconds Used Per Instance
sh	180904	44994882	248.7	14147	0.078
init	326810	43011749	131.6	538302	1.647
sendmail	228187	35268689	154.6	283061	1.240
mm2	127976	11190223	87.4	113422	0.886
gzip	63545	11006499	173.2	2955	0.047
csh	117528	10481445	89.2	34793	0.296
in.cfingerd	204522	8252987	40.4	21685	0.106
awk	67463	7901092	117.1	16854	0.250
amd	102964	5892972	57.2	3201	0.031
mail	31261	3842822	122.9	14429	0.462
tset	37929	3480483	91.8	19099	0.504
ip	237	3308288	13959.0	425124	1793.772
ls	53235	2677470	50.3	5911	0.111
elm	29508	2336297	79.17	15492	0.525
virtex	2428	2110749	869.3	17598	7.248
nn	4848	2102887	433.8	18277	3.770
test	103296	1702253	16.5	7156	0.069
named-xfer	4029	1498105	371.8	184	0.046
cat	45116	1491297	33.1	987	0.022
ugen2.1	3792	1357733	358.1	824	0.217
compress	8110	1351396	166.6	4082	0.503
finger	38323	1314659	34.3	30063	0.784
filter	31846	1269500	39.9	6599	0.207
is_able	40422	1199002	29.7	562	0.014
cc1	3959	1157576	292.4	11479	2.899
stty	31212	1145785	36.7	491	0.016
df	3959	1072604	270.9	49	0.012
rn	2391	1022723	427.7	16358	6.841
as12.1	7214	1002944	139.0	2907	0.403
tcsh	14085	996873	70.8	7206	0.512
msgs	15688	951683	60.7	2797	0.178
ccom2.1	4235	942887	222.6	2755	0.651
ld2.1	4204	877609	208.8	2665	0.634
grep	34688	834894	24.1	1994	0.057
echo	84811	830943	9.8	1011	0.012
emacs	2954	817308	276.7	102160	34.584
uptime	3734	810873	217.2	381	0.102
hostname	20885	808601	38.7	243	0.012
in.fingerd	10046	796413	79.3	2153	0.214
movemail	1306	785457	601.4	42	0.0322

Table 3: Top 40 programs by total pages used. *init* was periodically forking a new process for the (usually unused) serial port on some workstations. *mm2* was a user's signal-processing application. *gzip* is a file-compression program. *in.cfingerd* was run every minute to update status information used by the GNU "finger" server; *in.fingerd* is the result of a query. *amd* is the auto-mount daemon, managing remote file-system mounts. *ip* was a user's image-processing program. *is_able* was part of a nightly system accounting program on Server 1. *cc1* and programs with *2.1* in their names are part of the C compiler. *named-xfer* is a daemon associated with network name services.

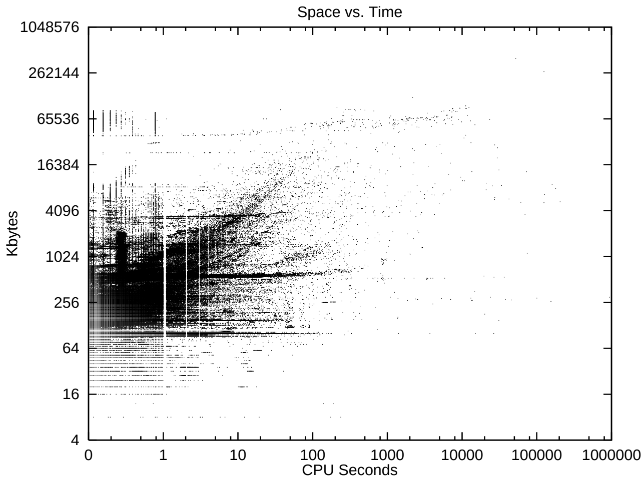

Figure 2: Scatter plot of the address space consumed and the CPU time consumed, for all processes traced in all groups. The correlation coefficient is 0.0236. The illusion of horizontal lines arises from the allocation of space in 4 KB pages. The illusion of vertical black lines and white gaps, we believe, arises from scheduler anomalies.

From Table 3, one wonders whether address space usage was correlated to CPU time used. Figure 2 demonstrates a lack of such a correlation in a scatter plot of these two measures for each process traced. The lack of correlation (coefficient 0.0236) between usage and CPU time meant that we could not expect to simply extrapolate per-process usage as a function of CPU speed.

4 Single-address-space systems

To be able to predict the lifetime of single-address-space systems, we had to consider more than just the current usage rate. First, we considered some space-allocation policies that might be used in a single-address-space system, to add the cost of fragmentation to the usage rate. Then we considered appropriate methods to extrapolate the current usage rate into the future. We begin by describing our methods.

4.1 Methods

4.1.1 Allocation policies

Clearly, systems that manage a single virtual-address space by allocating virtual addresses to processes and files without ever reclaiming the addresses for re-use will eventually run out of the finite address space. Allocation policies with significant fragmentation would shorten the expected lifetime, and allocation policies that allow some re-use would extend the expected lifetime. We used trace-driven simulations to measure the net rate of address-space usage under a variety of likely allocation policies. Each trace event allocates or extends a region of virtual-address space, in multiples of 4 KB pages, called a segment.[3] We were concerned with the internal fragmentation caused by allocating too many pages to a segment, but ignored the small internal fragmentation in the last 4 KB page of a segment.

Base allocation. For each processor in the distributed system, we allocated a generous 32-bit (4 GB) subspace to the kernel and its data structures. We also allocate 4 GB for every machine's initial collection of files, as a conservative estimate of what each new machine would bring to the address space. Note that this 8 GB was counted only once per machine.

Process allocation. Processes allocated four types of virtual-memory segments: text (code), shared data, private data (heap), and the stack. We assumed that the text segment did not require the allocation of new virtual memory, since it was either allocated at compile time or was able to be re-used (as in Hemlock). Shared libraries, though not available on the systems we traced, would be treated the same as text segments.

We assumed that shared-data segments would never be re-used, but could be allocated with the exact number of pages necessary. The actual policy choice made essentially no difference in our simulations, because our trace data contained only a tiny amount of shared data. In a single-address-space operating system, shared-data segments could be managed in much the same manner as private-data segments.

Private-data and stack segments have traditionally been extendible (to a limit), and thus an allocation policy in a single-address-space system may need to allocate more than the initial request to account for growth. Overestimates lead to fragmentation losses (virtual addresses allocated but never used). We examined several alternative policies, composed from two orthogonal characteristics. The first characteristic contrasted **exact-size** allocation, where each segment was allocated exactly the maximum number of pages used by that segment in the trace, and **fixed-size** allocation, where each process was allocated a 64 MB data segment and a 2 MB stack segment.[4] The *exact* policy could be approximated with reasonable user-supplied stack sizes and non-contiguous heaps. The second characteristic contrasted **no re-use**, where no segment was ever re-used, with **re-use**, where all freed private-data and stack segments were re-used for subsequent private-data or stack segments. Note that, of the four possible combinations, the two re-use policies are similar, in that neither causes any space to be lost from external or internal fragmentation *over the long*

[3] We assume a flat (not segmented) address space. We use the word "segment", in the tradition of names like "text segment" and "stack segment", to mean a logical chunk of virtual address space.

[4] These sizes are the default limits for these segments under Ultrix. Different sizes would not alter the qualitative results observed.

term. (Note that the 32-bit subspace of Hemlock [Garrett *et al.* 1992] is also similar to the fixed re-use policy.) Thus, we measured only **re-use**, **exact no-reuse**, and **fixed no-reuse**.

File allocation. Though Figure 1 implies that file data were insignificant, it does not account for fragmentation caused by address-space allocation policies in a single-address-space system. We considered several policies to determine their effect on fragmentation.

A file is traditionally an extendible array of bytes. Newly created files can grow from an initial size of zero, so in a single-address-space system, a new file must be allocated space with room to grow. These "file segments" can never be re-used or moved, because a pointer into a deleted file's segment may be stored in another file, or because the file may be restored from a backup tape. With this limitation in mind, we considered several policies (note that a higher-level interface could provide a conventional read/write file abstraction on top of any of these file-system policies):

exact: Each file was allocated exactly as much space as its own lifetime-maximum size (in pages). This unrealistic policy was useful for comparison.

fixed: A fixed 4 GB segment was allocated for each file when it was created (as in Hemlock [Garrett *et al.* 1992]). Any extraneous space was never recovered.

chunked: Growing files were allocated virtual-address space in chunks, beginning with a one-page chunk for a new file. Once the latest chunk was full, a new chunk of twice the size was allocated. When the file was closed, any unused pages at the end of the last chunk were reserved for future growth. This reservation strategy limited the number of chunks, and hence the amount of metadata needed to represent a file, by doubling the size of each chunk as the file grew, but did cause some fragmentation.

Distributed allocation. When a single address space spans multiple machines there must be a coordinated mechanism for allocating addresses. The dynamic allocation of space by a centralized allocation server is clearly inadequate, for both performance and reliability reasons. The other extreme, a static division of the entire address space among all machines, does not allow the addition of new machines to the system, or for any one machine to allocate more than its original allotment. A compromise policy seems feasible, in which a centralized (or perhaps hierarchical) allocation system allocates medium-sized chunks of address space to machines, from which the machines allocate space for individual requests. When the current chunk is consumed, another chunk is requested. Careful selection of the chunk size would limit fragmentation. If, for example, every machine requested as much space as it might need for one week, the centralized service would not be overly busy, and the resulting fragmentation would reduce the overall lifetime of the system by only a week or two.

To compute the current rates, we played back our trace data through a simulator that kept track of all allocation. We used a different version of the simulator for each combination of policies.

4.1.2 Extrapolating to the future

Any attempt to extrapolate computing trends by more than a few years is naturally speculative. Previous speculations have been crude at best: most of the back-of-the-envelope calculations in Section 2 extrapolate address-space usage by assuming that the yearly address-consumption rate remains constant. A constant rate seems unlikely, given continuing increases in available technology (faster CPUs, larger primary and secondary memory), sophistication of software, usage of computers, and number of computers. A simple linear extrapolation based on the current usage rate would overestimate the lifetime of single-address-space systems.

On the other hand, it is not clear that we could extrapolate based on the assumption that usage increases directly in proportion to the technology. Figure 2 shows that the address-space usage was not correlated with CPU usage, so a doubling of CPU speed (as happens every few years) does not imply a doubling of address-consumption rate on a per-process basis. Of course, a faster CPU presumably would allow more processes to run in the same time, increasing consumption, but our data cannot say by how much. Acceleration in the rate of address-space consumption is likely to depend significantly on changing user habits (for example, the advent of multimedia applications may encourage larger processes and larger files). This phenomenon was also noticed in a retrospective study of file-system throughput requirements [Baker *et al.* 1991]: "The net result is an increase in computing power per user by a factor of 200 to 500, but the throughput requirements only increased by about a factor of 20 to 30. ... Users seem to have used their additional computing resources to decrease the response time to access data more than they have used it to increase the overall amount of data that they use." These uncertainties make it impossible to extrapolate with accuracy, but we can nevertheless examine a range of simple acceleration models that bound the likely possibilities.

Disks have been doubling in capacity every three years, and DRAMs have been quadrupling in capacity every three years, while per-process (physical) memory usage doubles about every one to two years

[Patterson and Hennessy 1990]. It seems reasonable to expect the rate of address-space consumption to grow exponentially as well, though perhaps at a different rate. Suppose a is the acceleration factor per year; for example, $a = 1$ models linear growth, and $a = 2$ models an exponential growth exceeding even the growth rate of disk capacity ($a = 1.26$) or DRAM capacity ($a = 1.59$). If r is the current rate of address-space consumption (in bytes per year per machine), and n is the number of machines, then the number of bytes consumed in year y (year 0 being the first year) is

$$u(y) = nra^y \tag{1}$$

and the total address-space usage after year y (i.e., after $y + 1$ years) is

$$\begin{aligned} T(y) &= \sum_{i=0}^{y} u(i) && (2) \\ &= nr \sum_{i=0}^{y} a^i && (3) \\ &= \begin{cases} nr\frac{a^{y+1}-1}{a-1} & \text{if } a \neq 1 \\ nry & \text{if } a = 1 \end{cases} && (4) \end{aligned}$$

We extend this model by assuming that the number of machines, n, is not constant but rather a function of y. Here, a linear function seems reasonable. For simplicity we choose $n(y) = my$, i.e., there are m machines added each year. We can further extend this model by adding in a k-byte allocation for each machine's kernel and initial file set. This extension adds km to $u(y)$.

$$\begin{aligned} u(y) &= km + myra^y && (5) \\ T(y) &= \sum_{i=0}^{y} u(i) && (6) \\ &= kmy + mr \sum_{i=0}^{y} ia^i && (7) \\ &= \begin{cases} kmy + mr\frac{ya^{y+2}-(y+1)a^{y+1}+a}{(a-1)^2} & a \neq 1 \\ kmy + mr\frac{y(y+1)}{2} & a = 1 \end{cases} && (8) \end{aligned}$$

In the next section we compare equation 8, for a variety of parameters, to the available address space. It is reasonable to assume that the size of the address space will also increase with time. Siewiorek *et al*

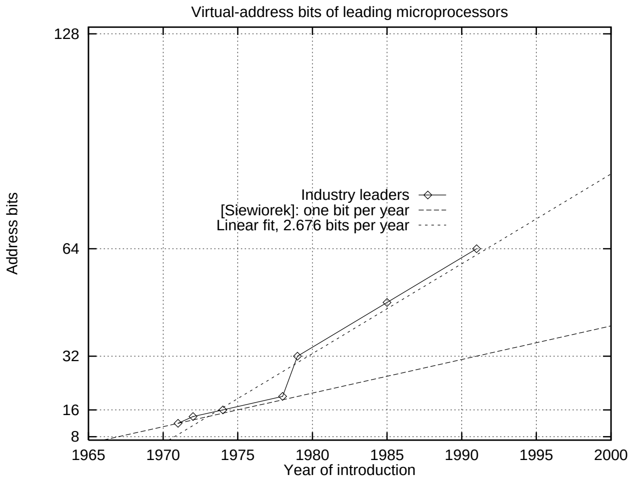

Figure 3: The number of address bits supported by various CPUs, and two curves fit to the data. The points represent the Intel 4004 (12 bits), Intel 8008 (14 bits), Intel 8080 (16 bits), Intel 8086 (20 bits), Motorola 68000 (32 bits), Intel 80386 (48 bits), and MIPS R-4000 and HP 9000/700 (64 bits). The data come from [Siewiorek *et al.* 1982, page 5], [Tanenbaum 1990], and [Glass 1991].

noticed that available virtual address space has grown by about one bit per year [Siewiorek *et al.* 1982], but their conclusions are based on old data. In Figure 3, we plot the virtual-address-bit count of microprocessor chips against the first year of introduction, for those chips that set a new maximum virtual address space among commercial, general-purpose microprocessors. We also plot two possible growth curves: the original from [Siewiorek *et al.* 1982] (one bit per year), and a new a linear regression fit (2.676 bits per year, with correlation coefficient 0.9824):

$$\text{address bits(year)} = 2.676 \times (\text{year} - 1967) - 2.048$$

Address bits generally become available in increments, every few years, rather than continuously. So, for increments of b bits,

$$\text{available address bits(year)} = b \times \lfloor \frac{\text{address bits(year)}}{b} \rfloor$$

This is the formula we use in Section 4.2.2 below.

4.2 Results

4.2.1 Allocation policies

Figure 4 shows the cumulative address space consumed by hypothetical single-address-space operating systems operating under each of the policies described above (except the "fixed" policies, which used orders of

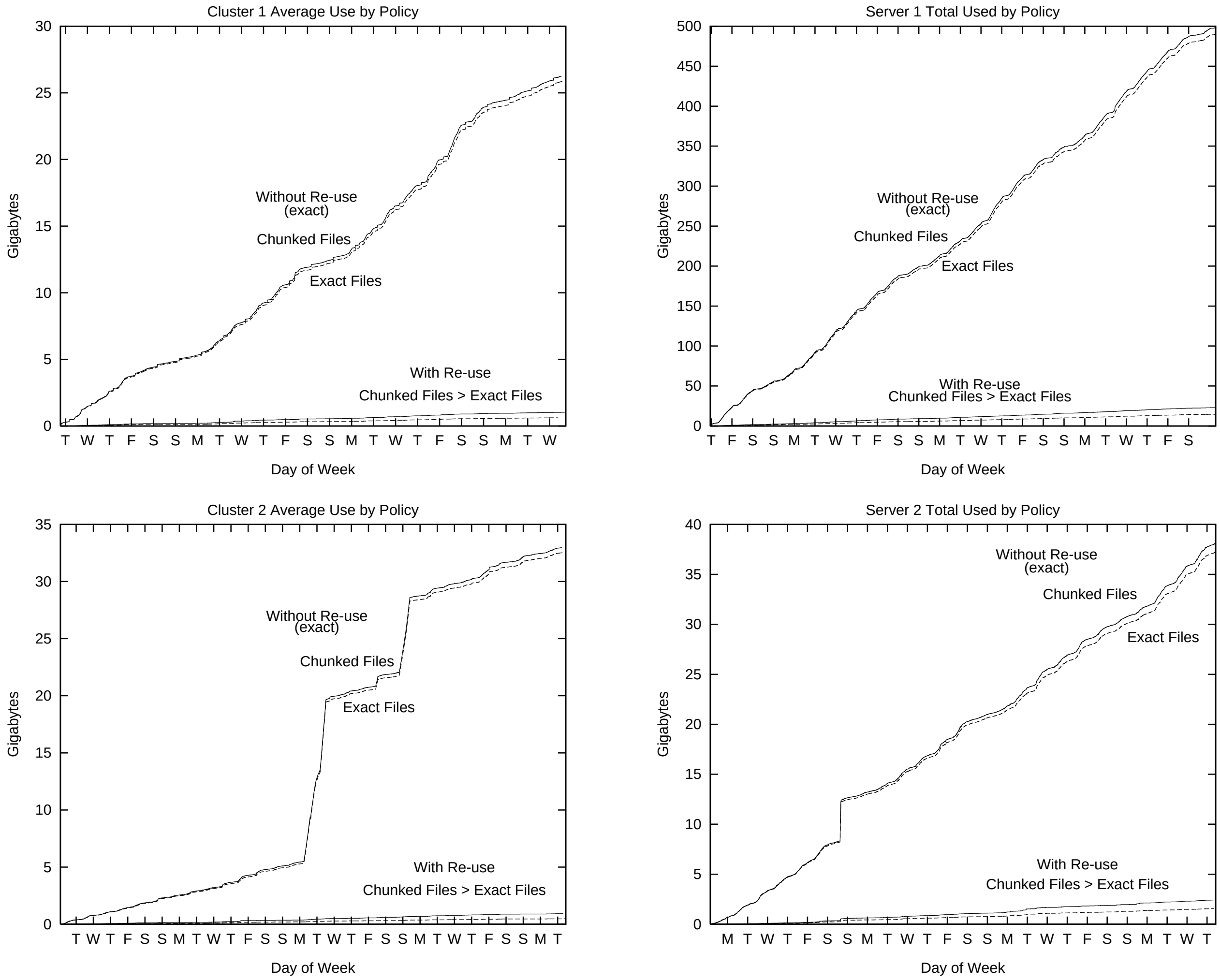

Figure 4: Cumulative address space consumed under different management policies, for each tracing group, over the interval traced. Curves for Cluster 1 and Cluster 2 are scaled down by the number of machines in each cluster, for easier comparison. x-axis tic-marks represent midnight before the given day of the week. The "fixed" file and process policies were so much worse that they are not shown (see Table 4).

magnitude more space, and hence are not shown), for each tracing group. Clearly, those that re-use data segments consume address space much more slowly. Also, the "chunked" file policy is remarkably close to the (unattainable) "exact" file policy.

To understand the burstiness of address-space usage, we computed each policy's usage for each five-minute interval on each machine. Figure 5 shows the distribution of this "instantaneous" usage across all 5-minute intervals on all workstations in each trace group, for each policy, on a logarithmic scale. Several interesting results appear. First, the "re-use" policies reduce the consumption by an order of magnitude or more. Second, the "chunked" file policy is not much worse than the (unattainable) "exact" policy. Third,

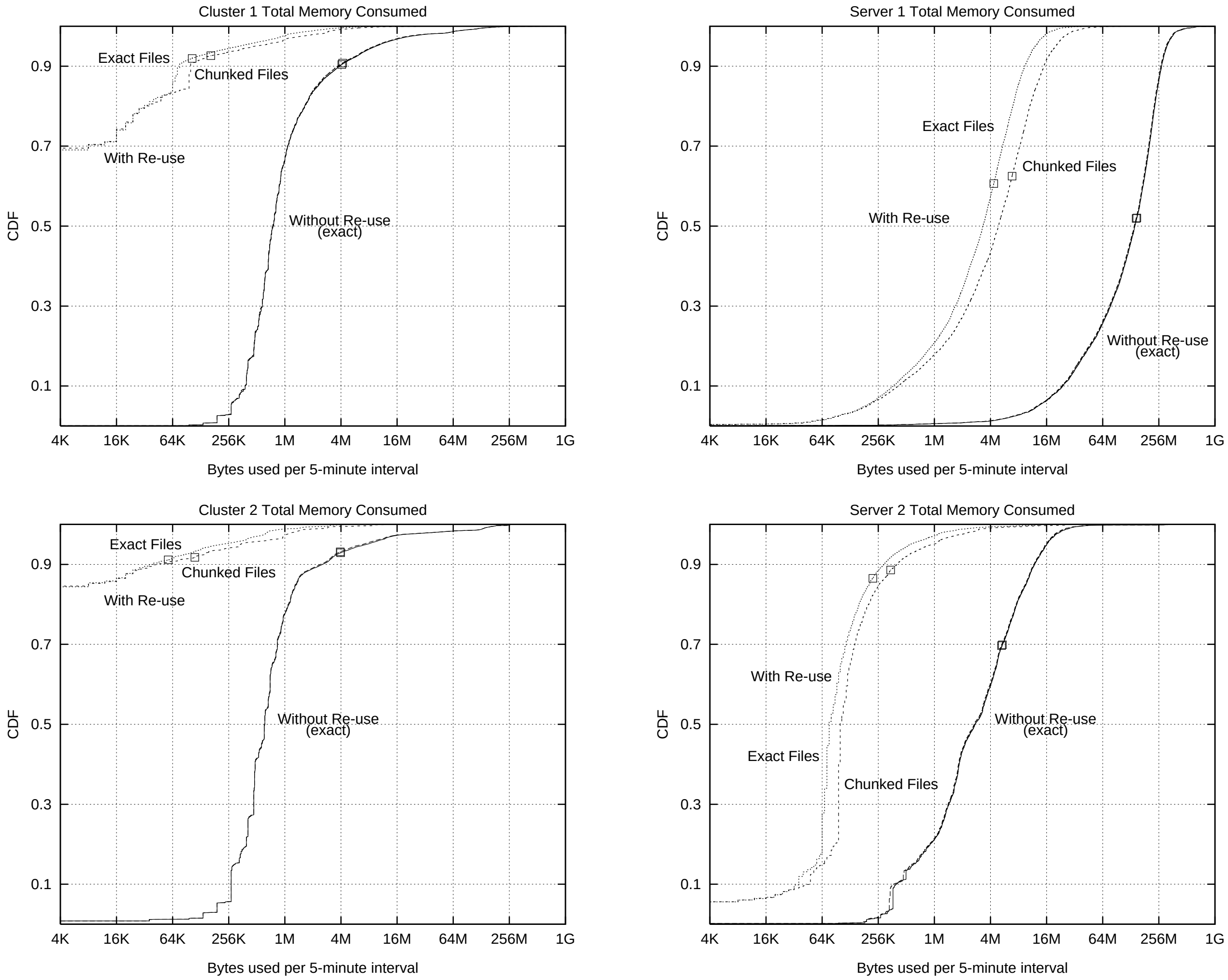

Figure 5: The cumulative distribution function (CDF) for the distribution of "instantaneous" address-usage rates across all 5-minute intervals on all workstations in each trace group, for each policy, for each trace group. Note the logarithmic scale. The mean rates are indicated by the box markers. Mean values significantly larger than median values indicate many intervals where little or no address space was consumed. Though both the "chunked" and "exact" file policies were plotted for the "exact, no re-use" process policy, there is no significant difference. The "fixed" file and process policies were so much worse that they are not shown (see Table 4).

in the clusters, the machines were frequently idle, as implied by the 69–84% of intervals where the reuse policies consumed at most one page.

Based on these results, we estimate the yearly rate of address-space consumption for each policy, given the current workload. Table 4 shows two rates for each tracing group, and for each policy: the first is the mean consumption rate (representing the situation where some machines are idle some of the time, as they were in our trace) computed by a linear extrapolation of the observed rates, and the second is the 95th percentile consumption rate (representing the situation where all machines are heavily used) taken from the

Process Policy	File Policy		bytes/year/machine Mean	95th %ile
exact no re-use	chunked	S1	8.2×10^{12}	1.7×10^{13}
		S2	5.9×10^{11}	1.8×10^{12}
		C1	4.5×10^{11}	1.0×10^{12}
		C2	4.4×10^{11}	8.3×10^{11}
exact no re-use	exact	S1	8.1×10^{12}	1.6×10^{13}
		S2	5.8×10^{11}	1.7×10^{12}
		C1	4.6×10^{11}	1.0×10^{12}
		C2	4.3×10^{11}	7.6×10^{11}
reuse	chunked	S1	3.8×10^{11}	1.1×10^{12}
		S2	3.7×10^{10}	1.1×10^{11}
		C1	1.8×10^{10}	5.3×10^{10}
		C2	1.2×10^{10}	3.7×10^{10}
reuse	exact	S1	2.4×10^{11}	6.7×10^{11}
		S2	2.4×10^{10}	6.1×10^{10}
		C1	1.1×10^{10}	3.6×10^{10}
		C2	6.1×10^{09}	2.3×10^{10}
reuse	fixed	S1	7.7×10^{16}	1.8×10^{17}
		S2	6.7×10^{15}	1.9×10^{16}
		C1	1.5×10^{15}	5.9×10^{15}
		C2	8.9×10^{14}	4.1×10^{15}
fixed no re-use	exact	S1	1.7×10^{15}	3.3×10^{15}
		S2	1.1×10^{14}	3.1×10^{14}
		C1	7.5×10^{13}	1.5×10^{14}
		C2	1.2×10^{14}	1.1×10^{14}

Table 4: Address-space consumption rate of various policies, given the current workload, in bytes per year per machine. We include both the mean rate, across all times on all machines in each group, and the 95th percentile rate, across all 5-minute intervals on all machines in each group. The other "fixed"-policy combinations, not shown, had worse usage than anything shown, and were not considered further.

distributions in Figure 5. The table makes it clear that both the "fixed" process policy and the "fixed" file policy were, as expected, consuming space extremely fast. The table reconfirms that re-using private-data and stack segments cut about one to one and a half orders of magnitude off the consumption rate, and that there was little difference between the "exact" and "chunked" file policies. Also, the 95th percentile rate was about one-half order of magnitude larger than the mean rate, and Server 1 was about an order of magnitude larger than the other machines, due to its heavy multi-user load.

4.2.2 Extrapolating to the future

We can compare the growth of available address space with the consumption of a single-address-space system that began in 1994. It is difficult to choose an appropriate value for parameters a and m, but by examining a

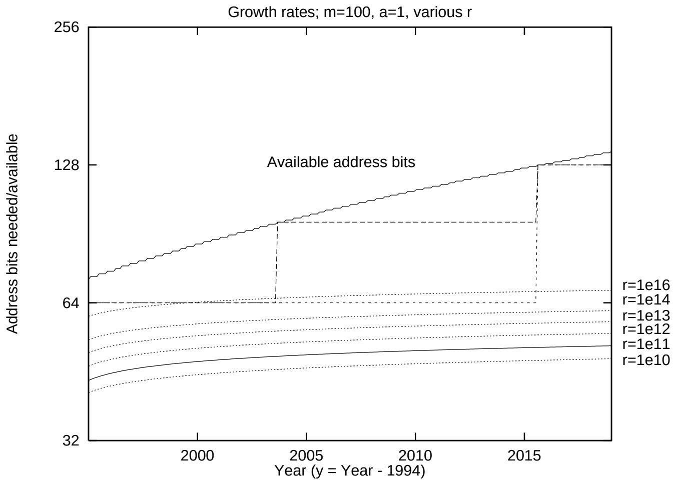

Figure 6: Comparison of available address bits with the consumption of address space for a variety of current rates, r, assuming no acceleration ($a = 1$) and $m = 100$. The solid consumption curve indicates the r value used in the other graphs. The available address bits grow in increments of 1, 32, or 64 bits.

wide range of values we can bound the likely behavior of future systems. For the acceleration a, we chose 1, 1.1, 1.2, 1.6, 2, and 3, i.e., ranging from linear growth ($a = 1$) to tripling the rate every year ($a = 3$). (To put these rates in perspective, recall that DRAM capacity grows at $a = 1.59$.) We chose $m = 100$ as the growth rate for the machine population, although we show below that there was little difference when varying m from 1 to 10000. From Table 4, we selected a range of representative rates r (in bytes/year/machine), as follows:

r	Clusters	roughly representing
10^{16}	all	"fixed" file policy
10^{14}	all	"fixed" process policy
10^{13}	Server 1	"exact, no re-use" process policy
10^{12}	others	"exact, no re-use" process policy
10^{11}	Server 1	"re-use" process policy
10^{10}	others	"re-use" process policy

Note that these rates are dependent on the nature of our workload— workstations in a computer science department. We speculate that the rate of a different workload, such as scientific computing, object-oriented databases, or world-wide-web servers, may differ by perhaps 2–3 orders of magnitude, and have a similar growth rate. If so, our conclusions would be qualitatively similar for these other workloads.

Figures 6–9 display the models, using a logarithmic scale to compare address bits rather than address-space size. Note that we plot the available address space as growing in increments of 1, 32, or 64 bits (see Section 4.1.2).

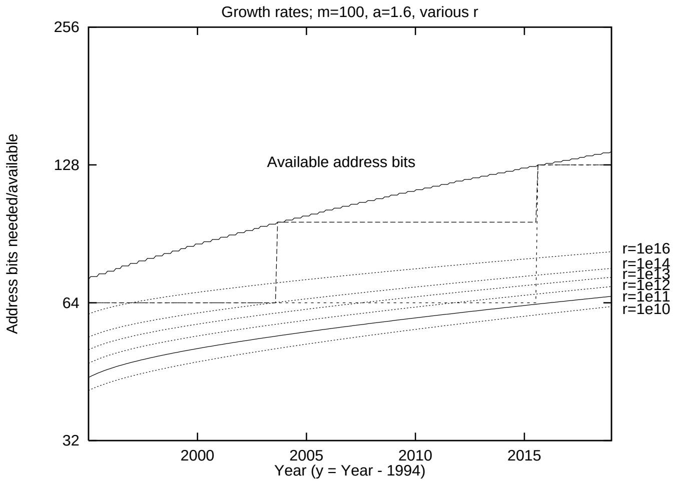

Figure 7: Comparison of available address bits with the consumption of address space for a variety of current rates, r, but with an acceleration factor of $a = 1.6$. $m = 100$. The solid consumption curve indicates the r value used in the other graphs.

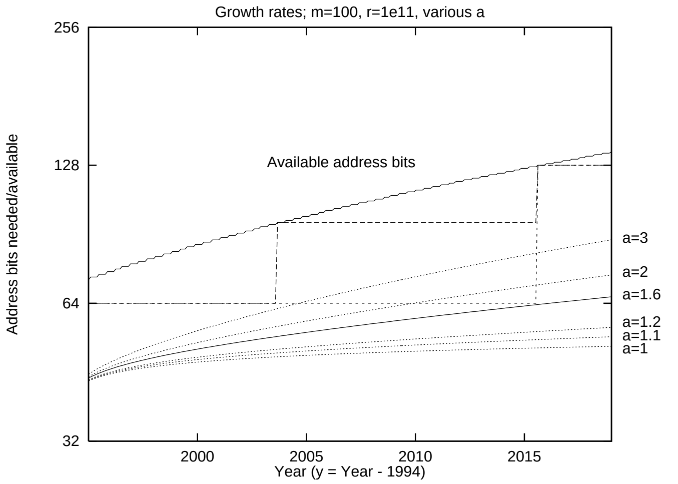

Figure 8: Comparison of available address bits with the consumption of address space for a variety of acceleration factors, a. The solid consumption curve indicates the a value used in the other graphs. Other parameters were $r = 10^{11}$ and $m = 100$.

Figure 6 examines the simple case of $a = 1$, where the yearly consumption remains constant at current levels. We see that a 64-bit address space is sufficient (that is, the "address bits needed" curve remains below the "address bits available" curve) only if the "fixed" file policy was avoided, or if a 96-bit address space were available soon. If the current consumption rate, r, accelerated (Figures 7–8) or if the number of machines grew especially fast (Figure 9), it would be even more important to avoid "fixed" policies or to migrate to a 96-bit address space soon.

Although the acceleration factor a of course has the most profound effect on address consumption, in the long term address-space growth should outpace even $a = 2$, and in the short term reasonable allocation

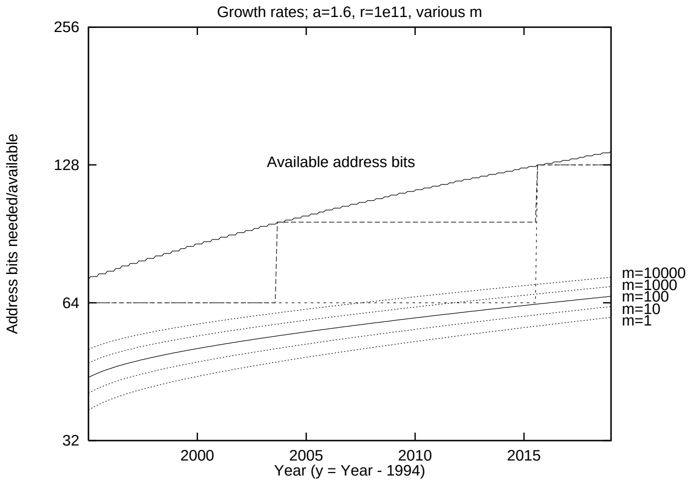

Figure 9: Comparison of available address bits with the consumption of address space for a variety of m, where the number of machines $n(y) = my$. The solid consumption curve indicates the m value used in the other graphs. Other parameters were $r = 10^{11}$ and $a = 1.6$.

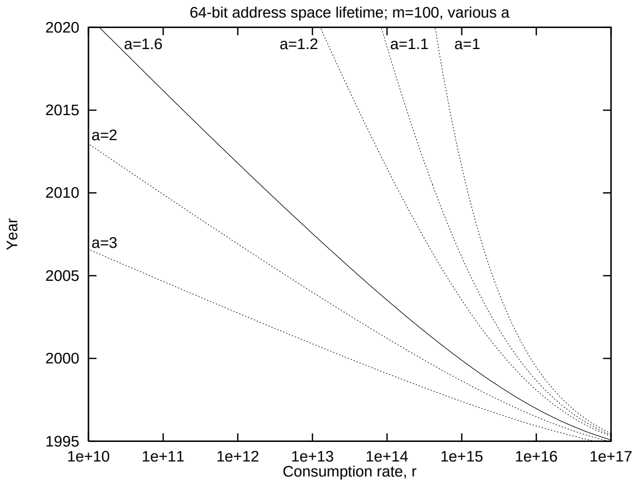

Figure 10: Comparison of various acceleration factors (a), showing the year in which a 64-bit address space will be completely consumed based on the initial rate (r). The solid curve indicates the a value used in the other graphs. We assume a 1994 start, and add $m = 100$ machines per year.

policies can keep the consumption rate low enough to last until the available address-space doubles again to 128 bits. Nevertheless, an intermediate jump to 96 bits would accommodate the most aggressive growth trends.

In short, Figures 6–9 tell us that it is possible to build a long-lived single-address-space system without complex space-allocation policies. Figure 10 presents the lifetime of a 64-bit address space for various a and r. It seems necessary only to re-use data and stack segments, and to use "chunked" file allocation, for a system to last more than 10 years. To accommodate maximum growth, however, the system should be able to adapt to larger addresses as they became available.

5 Summary

We traced several campus workstation clusters to gain an understanding of the current rate of address-space consumption, and the behavior of several likely policies under the current workload. Most of the current usage is from private-data and stack segments, with files using more than an order of magnitude less space, and shared data an essentially negligible amount. Fortunately, we found realizable allocation policies ("chunked" file allocation and "fixed, re-use" process allocation) that allowed re-use of the private-data and stack segments, leading to yearly consumption rates of 10 to 100 gigabytes per machine per year. Because of their simplicity, and low overhead, we recommend these policies.

We used an extrapolation model that assumed an exponential acceleration of the usage rate, linear growth in the number of machines involved, and linear growth in the number of virtual-address bits, to predict the future of a single-address-space system. Our model predicts that a single-address-space system would not run out of virtual-address space, as long as it used reasonable allocation policies (such as the ones we suggest) *and* adapted gracefully to larger addresses (e.g., 96 or 128 bits) as they become available. Indeed, Figure 10 shows that a system with a single 64-bit address space could add 100 machines each year, triple its usage rate each year ($a = 3$), and still last for 10 years, by re-using data and stack segments and using our "chunked" file allocation policy.

Although our results necessarily depend on speculation about trends in technology and user behavior, and may or may not apply to workloads different from the typical office-workstation environment, we believe that our fundamental predictions are fairly robust. For example, we measured only one workload during one brief period, yet Figures 6–7 provide fundamentally the same conclusion for a wide range in the value of r. Similarly, Figure 9 shows that our ultimate conclusions hold for a wide range of the parameter m. Potential developers of a single-address-space system who have a better understanding of their system's workload can use our model to determine whether simple policies suffice. Only systems with unpredictable or extremely aggressive workloads should consider developing more sophisticated allocation policies.

Although there are many other issues involved in building a single-address-space operating system that are beyond the scope of this paper, it appears that address-space consumption will not be an impossible hurdle.

Acknowledgements

Many thanks to DEC for providing the Ultrix 4.3 source code and for providing a workstation and disk space for the data collection and analysis, and to our campus computer users for allowing us to run an experimental kernel and to trace their activity. Thanks also to Wayne Cripps and Steve Campbell for their help with the tracing. Finally, many thanks to the anonymous reviewers for their helpful feedback.

References

[Baker *et al.* 1991] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[Bartoli *et al.* 1993] A. Bartoli, S. Mullender, and M. van der Valk. Wide-address spaces — exploring the design space. *ACM Operating Systems Review*, 27:11–17, January 1993.

[Broessler *et al.* 1987] P. Broessler, F. Henskens, J. Keedy, and J. Rosenberg. Addressing objects in a very large distributed virtual memory. In *Distributed Processing. Proceedings of the IFIP WWG 10.3 Working Conference*, pages 105–116, 1987.

[Chase *et al.* 1992] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, March 1992.

[Chase *et al.* 1994] J. Chase, H. Levy, M. Feeley, and E. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12, May 1994.

[Garrett *et al.* 1992] W. Garrett, R. Bianchini, L. Kontothanassis, R. McCallum, J. Thomas, R. Wisniewski, and M. Scott. Dynamic sharing and backward compatibility on 64-bit machines. Technical Report 418, Univ. of Rochester Computer Science Department, April 1992.

[Glass 1991] B. Glass. The Mips R4000. *Byte Magazine*, 16:271–282, December 1991.

[Jain 1991] R. Jain. *The Art of Computer Systems Performance Analysis*, page 210. Wiley, 1991.

[Koch and Rosenberg 1990] D. Koch and J. Rosenberg. A secure RISC-based architecture supporting data persistence. In *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 188–201, 1990.

[Kotz and Crow 1993] D. Kotz and P. Crow. The expected lifetime of "single-address-space" operating systems. Technical Report PCS-TR93-198, Dept. of Math and Computer Science, Dartmouth College, October 1993. Revised in 1994 to appear in SIGMETRICS '94, and revised again on March 15, 1996.

[Lee 1989] R. Lee. Precision architecture. *IEEE Computer*, 22:78–91, January 1989.

[Mullender 1993] S. Mullender, editor. *Distributed Systems*, pages 391–392. Addison-Wesley, second edition, 1993.

[Patterson and Hennessy 1990] D. Patterson and J. Hennessy. *Computer Architecture A Quantitative Approach*, pages 16–17. Morgan Kaufmann, 1990.

[Rosenberg *et al.* 1992] J. Rosenberg, J. Reedy, and D. Abramson. Addressing mechanisms for large virtual memories. *The Computer Journal*, 35:24–374, November 1992.

[Rosenberg 1992] J. Rosenberg. Architectural and operating system support for orthogonal persistence. *Computing Systems*, 5:305–335, Summer 1992.

[Satyanarayanan 1989] M. Satyanarayanan. Andrew file system benchmark. Carnegie-Mellon University, 1989.

[Siewiorek *et al.* 1982] D. Siewiorek, C. Bell, and A. Newell, editors. *Computer Structures: principles and examples*. McGraw-Hill, second edition, 1982.

[Sites 1993] R. Sites. Alpha AXP architecture. *Communications of the ACM*, 36:33–44, February 1993.

[Tanenbaum 1990] A. Tanenbaum. *Structured Computer Organization*, page 27. Prentice Hall, third edition, 1990.

[Wakerly 1989] J. Wakerly. *Microcomputer Architecture and Programming: The 68000 Family*, page 150. John Wiley and Sons, 1989.