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

David Kotz and Preston Crow

Dartmouth College
Department of Mathematics and Computer Science
Hanover, NH 03755-3510
{dfk,crow}@cs.dartmouth.edu

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. Typical proposals require the single address space to contain all process-private data, shared data, and stored files. To simplify management of an address space where stale pointers make it difficult to re-use addresses, 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 in our department, and discuss the implications for single-address-space operating systems. We found that single-address-space systems will 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

Operating systems are evolving under the influence of many architectural trends. One is the collection of many processors into a distributed or parallel system. Another is the use of a shared-memory programming model, even when the physical memory is distributed. Another is the growing size of physical memories (due to denser RAM chips) and of virtual memories (with the advent of 64-bit CPUs like the MIPS R4000 [5], the HP PA-RISC [7], and the DEC ALPHA [14]). Finally, main memory and secondary storage are increasingly unified through the use of virtual memory and "memory-mapped" files.

These trends make it possible to reconsider some of the basic assumptions in operating system design. Most current operating systems provide a separate address space for each process, which makes protection rather easy but makes sharing memory rather awkward. Many researchers propose to unify the memory hierarchy of several machines and disk systems into a single, "flat" virtual-address space [2, 3, 4, 10]. (These systems are often called "single-address-space" systems.) This unification makes it easier to share data structures between processes, even when the data may contain pointers or be physically located on different machines or disk systems. It also makes it easier to build persistent pointer-based data structures, avoiding the cost of translating to and from linear representations. Finally, it may improve performance by avoiding message-packaging overhead and some kernel traps.

One of the most convenient aspects of a single address space, the universality of pointers, also makes management of the address space especially difficult. Stale pointers, stored in persistent data structures, make re-use of the address range of a deleted object highly undesirable. Some claim that a 64-bit address

This research was supported in part by a NASA Graduate Student Research Assistantship, and by Digital Equipment Corporation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMETRICS 94- 5/94 Santa Clara, CA. USA
©1994 ACM 0-89791-659-x/94/0005..$3.50

Copyright 1994 by ACM. doi:10.1145/183019.183036.
Appeared in SIGMETRICS '94, pp. 161-170.
THIS COPY IS THE AUTHORS' PRE-PUBLICATION VERSION; it may differ slightly from the official published version.

space is *so large* that re-use would never be necessary [2]. These claims are not based on any real data, and have thus been the subject of much debate. In particular, back-of-the-envelope calculations often ignore fragmentation losses or growth in the rate of address-space consumption over the years. In this paper we provide the necessary data and analyze the prospects for single-address-space operating systems. We found that single-address-space systems will 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.

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

There are many advantages and disadvantages of an operating system with a single common address space, which are summarized by Mullender [8, pages 391–392] and by Chase et al [3].

The **MONADS-PC** project [10, 11] 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, the **MONADS-MM** [6], uses 128-bit addresses, extending the address-space numbers to 96 bits and the offsets to 32 bits.

Hemlock [4] proposes a single 64-bit 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 [3] 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. They concede that conserving and re-using address space is probably necessary.

In contrast, Bartoli et al. believe that "if ten machines create objects at a rate of ten gigabytes a minute, the [64-bit] address space will last 300 years" [2]. Using their numbers, 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 [9, page 16]. At this rate, an expansion from 32 bits to 64 bits would only last 32–64 years, and a single-address-space operating system would run out sooner.

It is clear that 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. In particular, we recorded creations, expansions, and deletions of each process's data and stack segments, all files, and all shared-data 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 allocated at compile time.[1] Table 1 summarizes the traces we collected.

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

Group	Days	Records	Lost records
Server 1	7.8	11392000	2 (0.00%)
Server 2	25.3	6595110	61709 (0.94%)
Cluster 1	22.9	915718	614 (0.07%)
	22.9	3667000	6 (0.00%)
	22.9	378430	1409 (0.37%)
	22.9	3293680	19351 (0.59%)
	22.9	417550	26 (0.01%)
	23.0	884393	2 (0.00%)
	22.9	1402850	132692 (9.46%)
	22.9	1343890	3180 (0.24%)
	23.0	849289	5995 (0.71%)
	22.1	601798	2100 (0.35%)
	23.0	1850030	0 (0.00%)
	22.9	605955	88 (0.02%)
	Total	16210583	165463 (1.01%)
Cluster 2	29.4	9792880	175785 (1.80%)
	29.4	1082960	16144 (1.49%)
	29.4	610202	6051 (0.99%)
	29.4	486763	5458 (1.12%)
	Total	11972805	203438 (1.67%)

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 includes general-use workstations in the computer science department, most located in faculty offices. Cluster 2 contains workstations used primarily by a compute-intensive signal-processing research group. All workstations are DECstation 5000s running Ultrix 4.3. A small fraction of records were lost in the collection process (see Section 3.1 for details).

3.1 Methods

To collect this data, we modified the DEC Ultrix 4.3 kernel[2] to generate a trace record for all relevant activities. 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 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 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 mechanism for other trace-collection efforts.

To measure the performance overhead of our tracing activity, we ran 25 trials of the Andrew benchmark [12] on the standard Ultrix 4.3 kernel and on our instrumented kernel. The Andrew benchmark exercises both files and processes, by creating, searching, and deleting files, and compiling programs. We discarded the first trial in each case, due to a cold file cache. An unpaired t-test showed the difference to be insignificant at the 99% confidence level, implying that our tracing apparently had no significant effect on performance. This matches our qualitative experience (no users perceived any difference).

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. This 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. Fortunately, fewer than two percent of the records were missing from any trace group, indicating that the effect on the usage rates should be quite small, perhaps underestimating usage by 1–2%.

3.2 Results

In Figure 1, we show the raw amount of address space (in units of 4 KB pages) allocated 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. Daily and weekly rhythms are clearly visible. Server 1, heavily used for timesharing, used four times as much space in one third the time. Cluster 2, used by a signal-processing research group, occasionally saw large bursts

[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.

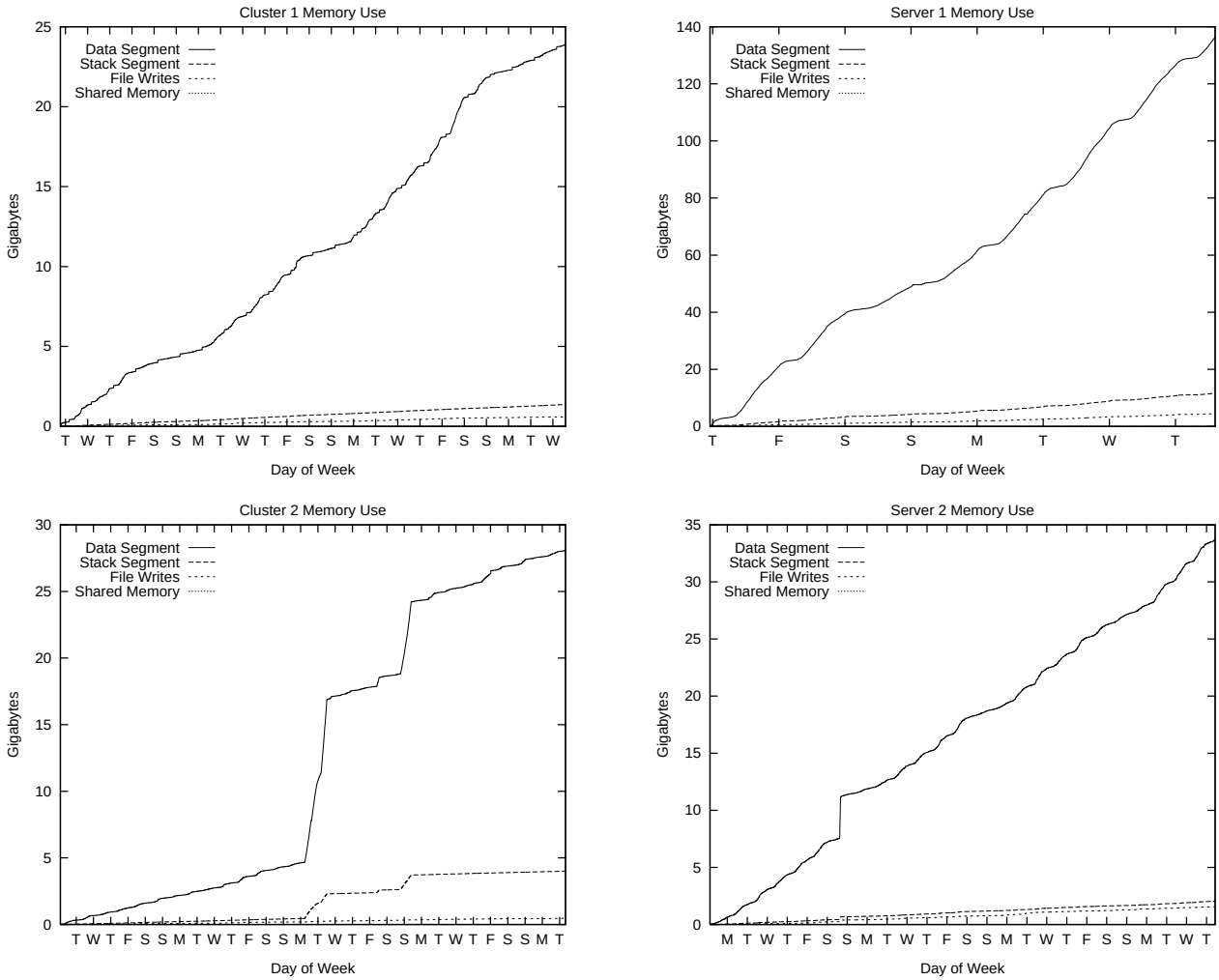

Figure 1: Cumulative address-space usage for all workstations in each trace group, separated by category of memory usage. Curves for Cluster 1 and Cluster 2 are scaled down by the number of machines in each cluster, for easier comparison. Shared Memory is indistinguishable from zero. x-axis tic-marks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

of activity caused by applications with large data segments.

To discover the nature of the significant address-space users, we compiled a list of all programs by address-space allocated. 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, pieces of the C compiler, and periodic background processes. Only two programs in the top 30 (a signal-processing application and an image-processing application) were user-written applications; all of the others were common applications used by many users. Only one could be called a large application (56MB of address space consumed per instance). This data makes it clear that policies which statically allocate (and never re-use) a large region to every process would waste a lot of virtual-address space on many small but common applications.

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 account for the costs of fragmentation in 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 units of 4 KB pages, called a segment.[3] We were concerned with the internal fragmentation caused by allocating too many pages to a segment, and the external fragmentation caused by holes left from freed segments, 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 conservative 32-bit (4 GB) subspace to the kernel and its data structures.[4] 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. A shared-data segment could never be re-used, because pointers into a shared data segment may have been stored in a private data segment elsewhere. We also assumed that shared-data segments were not extendible.[5]

[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] The alternative was to use the same 32-bit (private) subspace for all processors. This alternative, however, neither fits the general ideal of one common address space, nor allows kernels to access the kernel data structures of other processors (which may be considered useful by some designers).

[5] The actual policy choice made essentially no difference in our simulations, because our trace data contained only a tiny amount of shared data.

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 (memory 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. (Although the *exact* policy is unimplementable, it was useful for comparison purposes.) 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 other private-data or stack segments. Note that, of the four possible combinations, the two re-use policies are similar, in that neither cause any space to be lost from external or internal fragmentation *over the long term*. (Note that the 32-bit subspace of [4] is also similar to the fixed re-use policy.) Thus, we measured only **re-use**, **exact no-reuse**, and **fixed no-reuse**.

File allocation. 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 library, such as *stdio*, 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. 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, contiguous to the previous chunk if possible. 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.

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 improving technology, the increasing sophistication of software, the increasing usage of computers, and the increasing 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. We found that the address-space usage was not correlated with CPU usage (correlation coefficient 0.0238), so a doubling of CPU speed would not imply a doubling of address consumption on a per-process basis. Instead, 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 recent study of file-system throughput requirements [1]: "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 [9, pages 16–17]. It seems reasonable to expect the rate of address-space consumption to grow exponentially as well, though perhaps not as quickly. If r is the current rate of address-space consumption (in bytes per year per machine), a is the acceleration factor per year (e.g., $a = 2$ implies doubling the rate every year), and n is the number of machines, then the number of bytes consumed in year y is

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

and the total address-space usage after y years is

$$T(y) = \sum_{i=0}^{y} u(i) \tag{2}$$

$$= nr \sum_{i=0}^{y} a^i \tag{3}$$

$$= \begin{cases} nr\frac{a^{y+1}-1}{a-1} & \text{if } a \neq 1 \\ nry & \text{if } a = 1 \end{cases} \tag{4}$$

Note that $a = 1$ models linear growth, and that $a = 2$ models an exponential growth exceeding even the growth rate of disk capacity ($a = 1.26$) or DRAM capacity ($a = 1.59$). We extend this model by adding in a k-byte allocation for each machine's kernel and initial file set (which grows with n, but not with y, as we describe above in Section 4.1.1). We can further 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.

$$u(y) = n(y)ra^y \tag{5}$$

$$T(y) = kmy + \sum_{i=0}^{y} u(i) \tag{6}$$

$$= kmy + mr \sum_{i=0}^{y} ia^i \tag{7}$$

$$= \begin{cases} kmy + mr\frac{ya^{y+2}-(y+1)a^{y+1}+a}{(a-1)^2} & a \neq 1 \\ kmy + mr\frac{y(y+1)}{2} & a = 1 \end{cases} \tag{8}$$

In the next section we compare equation 8 to the available address space. It is reasonable to assume that the size of the address space will also increase with time.

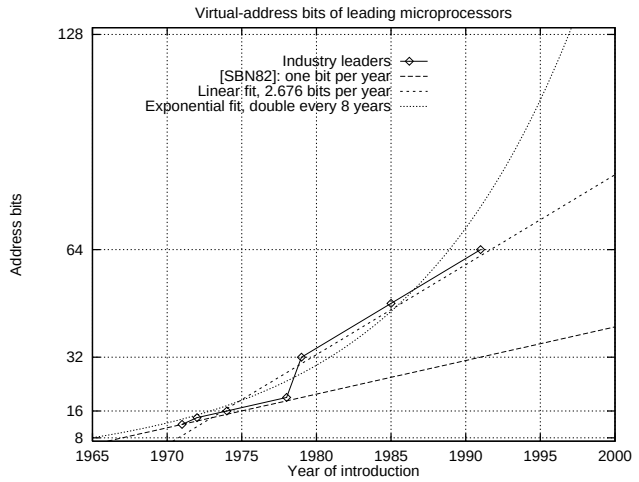

Figure 2: The number of address bits supported by various CPUs, and three curves fit to the data. The points represent the Intel 4004 (12 bits), Intel 8008 (14 bits), Intel 8080 (16 bits), Intel 8086 (20 bits), Motorola 68000 (32 bits), Intel 80386 (48 bits), and MIPS R-4000 and HP 9000/700 (64 bits). The data come from [13, page 5], [15, page 27], and [5].

Siewiorek *et al* noticed that available virtual address space has grown by about one bit per year [13], but their conclusions are based on old data. In Figure 2, 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 three possible growth curves: the original from [13] (one bit per year), a linear regression fit (2.676 bits per year, with correlation coefficient 0.9824), and a linear regression fit to the logarithm of the address bit count (leading to a doubling in address bits every eight years; correlation coefficient 0.9781). The best fit is the linear growth:

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

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

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

4.2 Results

4.2.1 Allocation policies

Figure 3 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 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. In the clusters, idle intervals dominate the distributions, with 69–84% of intervals consuming at most one page under the re-use policies. Based on these results, we estimate the yearly rate of address-space consumption for each policy, given the current workload. Table 2 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), and the second is the 95th percentile consumption rate (representing the situation where all machines are heavily used), based on the busiest five-minute intervals. 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 shows 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, by choosing reasonable values for the parameters. 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 ($a = 3$). Given that DRAM capacity grows at $a = 1.59$, we suspect that 1.6 is the highest realistic a. We chose $m = 100$, as the growth rate for the machine population, although we found that there was little difference when varying m from 1 to 10000. From Table 2, we selected a range of representative rates r (in bytes/year/machine), as follows:

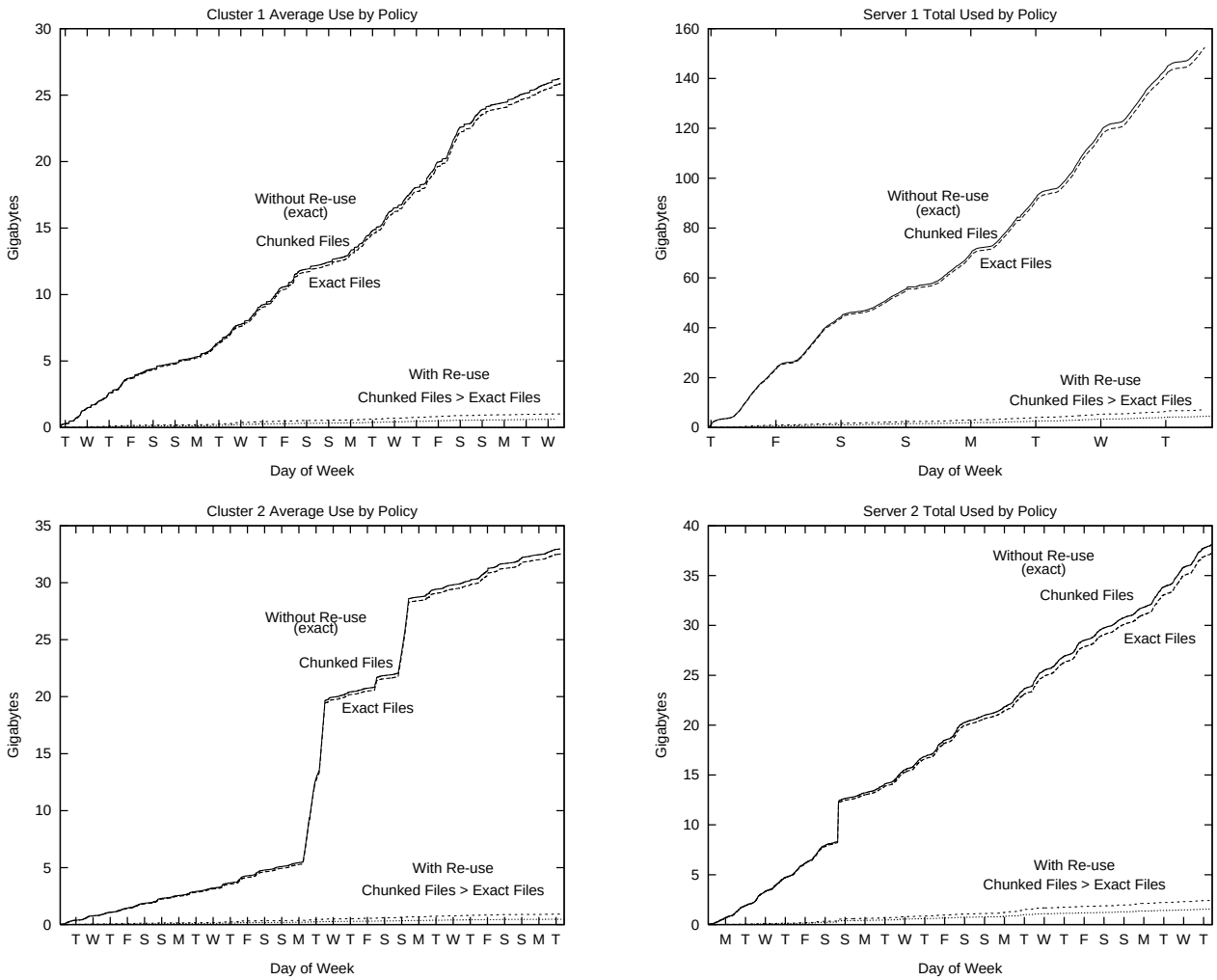

Figure 3: Cumulative address space consumed under different management policies, for each tracing group, over the interval traced. Curves for Cluster 1 and Cluster 2 are scaled down by the number of machines in each cluster, for easier comparison. x-axis tic-marks represent midnight before the given day of the week. The "fixed" file and process policies were so much worse that they are not shown (see Table 2).

r	Cluster	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 or object-oriented databases, 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 4–6 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 4.1.2).

Figure 4 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" policies were avoided, or if a 96-bit address space were available soon. If the current consumption rate, r, accelerated especially fast (Figures 5–6), the re-use policies were definitely necessary.

Process Policy	File Policy		bytes/year/machine Mean	95th %ile
exact	chunked	S1	8.2×10^{12}	1.7×10^{13}
no re-use		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	exact	S1	8.1×10^{12}	1.6×10^{13}
no re-use		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	exact	S1	1.7×10^{15}	3.3×10^{15}
no re-use		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 2: 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.

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 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.

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

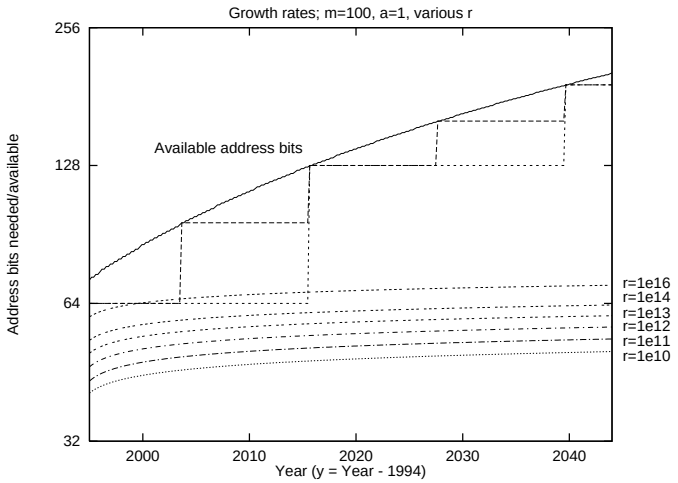

Figure 4: Comparison of available address bits with the consumption of address space for a variety of current rates, r, assuming no acceleration ($a = 1$) and $m = 100$. The available address bits grow in increments of 1, 32, or 64 bits.

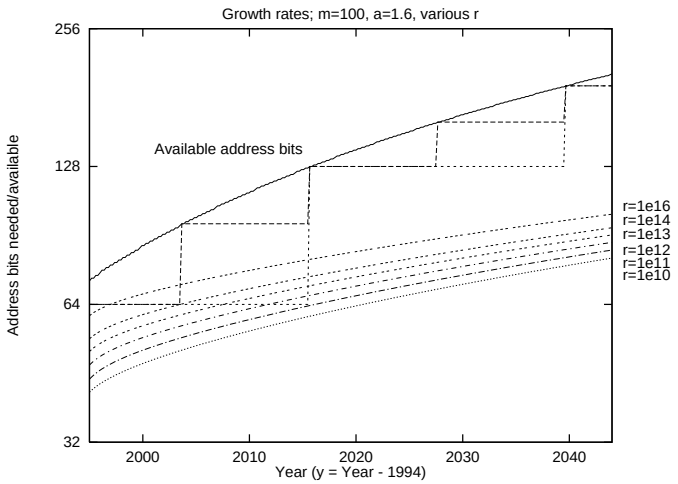

Figure 5: Comparison of available address bits with the consumption of address space for a variety of current rates, r, but with an acceleration factor of $a = 1.6$. $m = 100$.

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.

Using an extrapolation model that assumed an exponential acceleration of the usage rate, linear growth in the number of machines involved, and linear growth

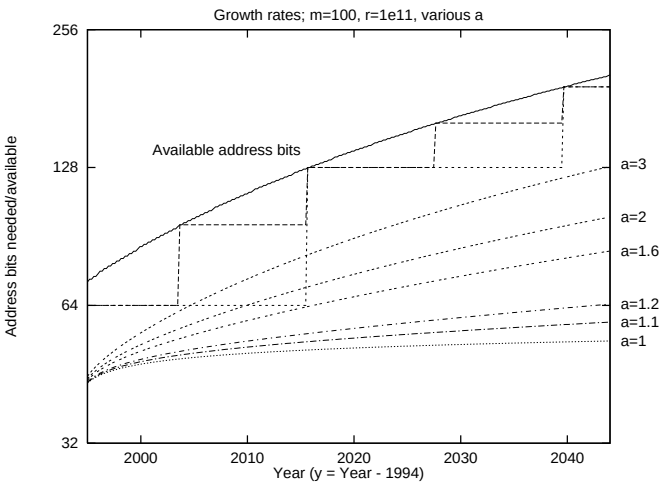

Figure 6: Comparison of available address bits with the consumption of address space for a variety of acceleration factors, a. Other parameters were $r = 10^{11}$ and $m = 100$.

in the number of virtual-address bits, we show that a single-address-space system would not outgrow the available address space. However, to accomplish this feat, any single-address-space system must re-use the private-data segments of processes, limit file-segment fragmentation, and adapt gracefully to larger addresses (e.g., 96 or 128 bits) as they become available. We emphasize that 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.

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. Finally, thanks to Wayne Cripps and Steve Campbell for their help with the tracing.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[2] Alberto Bartoli, Sape J. Mullender, and Martijn van der Valk. Wide-address spaces — exploring the design space. *ACM Operating Systems Review*, 27(1):11–17, January 1993.

[3] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, May 1994.

[4] William E. Garrett, Ricardo Bianchini, Leonidas Kontothanassis, R. Andrew McCallum, Jeffery Thomas, Robert Wisniewski, and Michael L. Scott. Dynamic sharing and backward compatibility on 64-bit machines. Technical Report 418, Univ. of Rochester Computer Science Department, April 1992.

[5] Brett Glass. The Mips R4000. *Byte Magazine*, 16(13):271–282, December 1991.

[6] David Koch and John 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.

[7] R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[8] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993. Lecture notes from the annual Advanced Course on Distributed Systems.

[9] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1990.

[10] J. Rosenberg, J. L. Reedy, and D. Abramson. Addressing mechanisms for large virtual memories. *The Computer Journal*, 35(4):24–374, 1992. Australia.

[11] John Rosenberg. Architectural and operating system support for orthogonal persistence. *Computing Systems*, 5(3):305–335, Summer 1992.

[12] M. Satyanarayanan. Andrew file system benchmark. Carnegie-Mellon University, 1989. Source code available on request.

[13] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors. *Computer Structures: principles and examples*. McGraw-Hill, second edition, 1982.

[14] Richard L. Sites. Alpha AXP architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[15] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, third edition, 1990.