AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

The Evolution of *UNIX* System Performance

By J. FEDER*

(Manuscript received October 25, 1983)

Performance has motivated much of the change in the *UNIX*™ operating system over the years. This paper gives the results of measurements of system performance taken over time and links the measured improvements to the algorithmic changes that gave rise to them. The most notable improvements have occurred in methods for performing table searches, disk input/output, and terminal handling; these have been driven heavily by the release from address space and memory restrictions in recent 32-bit hardware. Overall, the changes on 32-bit machines have yielded a more than 25-percent improvement in the system's ability to support time-sharing users.

I. INTRODUCTION

This paper presents a historical perspective on the improvements in *UNIX* operating system performance over the years and highlights the major algorithmic changes that are responsible. The movement of people, supplemented by communication by means of mail and news networks, has spread key improvements rapidly. Although all measurements in this paper were obtained from AT&T Bell Laboratories *UNIX* system versions, most of the algorithmic changes described have similar counterparts on other *UNIX* system derivatives being

* AT&T Bell Laboratories.

Copyright © 1984 AT&T. Photo reproduction for noncommercial use is permitted without payment of royalty provided that each reproduction is done without alteration and that the Journal reference and copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free by computer-based and other information-service systems without further permission. Permission to reproduce or republish any other portion of this paper must be obtained from the Editor.

run at universities* and industry throughout the world. No attempt is made here to credit specific individuals for any of the changes; similar changes have often evolved independently at different sites.

1.1 Strategy for benchmarking and performance analysis

This paper emphasizes system changes related to performance; however, to put the results in context we should say a few words on the benchmarking and analysis practices used. The term *performance*, as used here, refers to the ability to accomplish tasks with minimum consumption of resources, notably processor and disk, and thus to do more work per unit time. At a given applied load, this usually translates into faster system response. Different application work loads exercise different system components and apply different stresses; knowledge of work load is necessary to talk precisely about overall system performance. Since it is impossible to benchmark all work loads, our strategy is to measure individual system components and to use the results in conjunction with knowledge of specific applications to estimate the impact of improvements. Benchmarks modeling several applications are used to provide further, more precise, overall performance numbers. One application in particular, that of providing program development services (including documentation) in a time-sharing environment, is viewed as especially important and is emphasized in this paper.

Overall performance, regardless of application, is a composite of the performance of:

1. Hardware and microcode
2. Compiler (object-code quality)
3. Kernel
4. C libraries
5. Commands.

Each of these components exercises those preceding in the list and is measured in conjunction with them. This paper is organized according to the list above; successive sections describe measurements and improvements to the components mentioned. Items (1) and (2) are grouped together under C language performance in Section II. To show the combined effect of the changes in various areas, Section VI presents results for a simulated time-sharing work load modeling the activities of a program development community.

Our measurement technology places a premium on automated measurements and other practical considerations. Kernel measurements are performed without code modification or external instrumentation.

* The University of California at Berkeley has been notable in gathering together and instituting new developments.

Although details on the component benchmarks will not be given, the general goal of each is to measure a specific function or operation while minimally involving any others. Our benchmarks have shortcomings (to be pointed out in coming sections) but nevertheless furnish useful information. Formal benchmarks for the C library and commands have not yet been completed; only limited measurement information for these components is available.

1.2 Improving UNIX *system performance*

Recent years have seen substantial performance improvements in *UNIX* systems, especially on 32-bit machines, as a result of the application of a wide range of techniques. Extensive profiling has identified critical code segments, and tuning practices similar to those described by Bentley[1] have been used to improve efficiency. Some of the more dramatic gains, however, have come from more fundamental adaptations of the system to new hardware and to the change in relative costs of various computing factors. Large word-size minicomputers have been introduced that allow more memory to be addressed, and memory prices have fallen steadily.[2] Disks have grown larger and storage costs have fallen. Instruction rates (at least for some key *UNIX* system machines) have not kept pace. This has created an impetus to trade memory and disk space for improved performance. Other hardware developments, such as terminal-handling front-end processors and improved peripheral functionality, have also contributed.

Some potential trades for performance have been avoided. Assembler encoding, machine-specific code tuning, and use of special algorithms to take advantage of features of particular machines, can improve performance but sacrifice long-term goals of portability and maintainability.

1.3 System versions and results

The performance results presented here were accumulated from efforts to monitor system performance during development, as well as to characterize performance to *UNIX* system-based applications. The common practice of instituting a group of changes at once has, in many instances, precluded quantification of the improvements offered by each individual change. The machines for which performance results spanning an interval of time are available are the AT&T 3B20S computer and the VAX* and PDP-11* models.

In tracing performance changes over time, it is most instructive to

* Trademark of Digital Equipment Corporation.

Table I—*UNIX* system versions

Internal Version	External Equivalent	Date Developed	Machines
PG 1C-300		1977	PDP-11
3.0	System III	1980	PDP-11, VAX
4.0		1981	PDP-11, VAX
4.1.1		1981	AT&T 3B20S
4.2		1981	PDP-11, VAX, 3B20S
5.0	System V	1982	PDP-11, VAX, 3B20S

associate results with the times at which the development of the respective *UNIX* systems was completed, which typically coincide with the times at which the measurements were made. This allows comparison with unofficial prototype 3B20S *UNIX* system versions that illustrate the effect of performance tuning during the period immediately following a port to a new machine.

The system versions measured are listed in chronological order in Table I; all but the first were issued by AT&T Technologies, Inc. PDP-11/70 computer results prior to 1980 are for the Generic 3 (PG 1C-300) *UNIX* system version, which was at the time available from the *UNIX* Support Group at AT&T Bell Laboratories for use in operating company support system applications.* The 3.0 and 5.0 releases described here are especially significant since they are very close to the System III and System V releases, respectively, licensed (for the VAX and PDP-11 computers) outside of AT&T and the Bell operating companies.

II. C LANGUAGE PERFORMANCE

C is the major *UNIX* system language and the one in which the bulk of the kernel is written. Unfortunately, performance, as determined by the speed of the object code produced by AT&T Bell Laboratories C compilers, has remained relatively static.

We made the measurements of relative rates in executing C code using a collection of small C language programs that do not reference either the operating system or the C library. They bunch together the performance of machine and C compiler, and are used to determine the effect of compiler changes, as well as to provide approximate estimates of machine speed. The benchmarks do not use floating-point arithmetic and make only light use of multiplication and division operations. The object code produced contains a mixture of procedure

* The *UNIX* System Support Group Generic 3 system is a derivative of AT&T Bell Laboratories Research Version 6. AT&T Technologies Release 3.0 is a derivative of Research Version 7 and 32V systems.

Table II—Normalized C object code execution speed

Machine	Relative C Execution Speed
3B20S	1.00
VAX-11/780	0.97
VAX-11/750	0.61
PDP-11/70	0.83

calls, memory, and register operations roughly typical of the larger body of *UNIX* system programs.* (In fact, the benchmarks were extracted from existing system programs.) The grouping of machine with compiler performance is unfortunate, but in general, there is no way to separate these two without resorting to hand coding of assembler benchmarks, a procedure that inserts an uncontrolled and undesirable variable.

Table II shows the relative speeds of several machines in executing C code for Version 5.0 compilers as obtained by normalizing individual benchmark results to the corresponding result for the 3B20S computer and then averaging. Larger numbers indicate better performance. All results are for "peephole" optimized code. The peephole optimizers typically reduce program text space by 5 to 15 percent and execution time by about 5 percent. The error tolerance on these results, due to timing granularity and machine variations, is a few percent.† Except for the 3B20S computer, this error tolerance is sufficiently large to cover all of the observed speed differences since 1979. (The VAX compiler is actually known to have become marginally slower as a result of changes to bring the handling of sub-word-size register quantities into conformance with the C language specification.) The 3B20S compiler and microcode performance improved about 12 percent between its first release, 4.1.1, and Version 5.0.

The VAX-11/750‡ computer runs essentially the same system software as the VAX-11/780‡ computer but at 60 to 65 percent of its speed. In Table I, the VAX-11/780 computer shows only about a 15-percent advantage relative to its predecessor, the PDP-11/70‡ computer. This difference is small, especially considering the number of

* The benchmark programs used are small and thus run with atypically high cache hit ratios. They also suffer from other problems arising from the process of extracting them from larger code segments.

† Measurements were made on the same machine sample but at different times, and thus do not account for minor performance changes due to field service updates and machine peripheral modifications.

‡ Trademark of Digital Equipment Corporation.

years involved. This small difference is misleading, however. As we noted in Section III, architectural differences between the two machines, most notably the larger VAX computer word size and addressability, yield markedly higher VAX computer performance when running the *UNIX* system. Pure C language speed can be misleading when comparing low-end 16-bit microcomputers with larger word-size machines possessing special features to help support operating systems.

The times to compile the benchmark program present an interesting sidelight. As a result of the combined effect of improvements to the kernel, C libraries, and software involved in program compilation, VAX programs compile on System V more than 25 percent faster and 3B20S programs compile more than twice as fast relative to 4.0 systems. PDP-11/70 compilation speed is essentially unchanged since System III.

III. KERNEL

The kernel comprises only a small fraction of the total system in terms of source lines, but typically consumes half or more of the execution time. It has thus been the focus of much tuning effort over the years. This effort has yielded improved throughput as well as a steady decline in the proportion of central processing unit (CPU) time spent in the kernel. In the following, the approximate importance of some key operations has been indicated by giving the percentage of total CPU time consumed in a program development environment, as calculated from the occurrence frequency and CPU time for the operation. A range of values is needed to cover different machines and the effect of improvements affecting time and frequency. Although program development CPU percentages are cited, the items mentioned are likely to be important in other applications that spend significant time in the kernel.

3.1 System call overhead

UNIX system calls all incur some common overhead in transferring control to and from the operating system. This overhead consumes 4 to 7 percent of the CPU in a program development environment. System call overhead is measured by executing a `getpid` (return process *id*) system call, which essentially fetches a small amount of information from the kernel; `getpid` CPU time is mostly taken up by the system call mechanism.

Figure 1 shows the change in system call times with release. [Due to the relatively short (<1-ms) time for the `getpid` call, memory cache transients comprise a substantial fraction of the total time; the times shown are for the typical situation of nothing useful in the memory

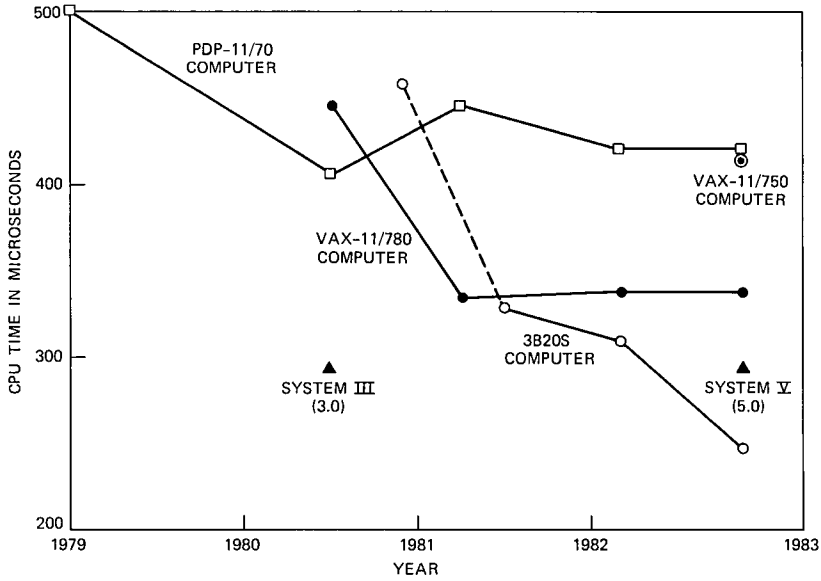

Fig. 1—System call overhead – `getpid` (invalid cache).

cache at the time of system call invocation.] In this and Figs. 2, 3, and 5, the dotted curve portions for the 3B20S computer indicate measurements of unofficial laboratory operating systems prior to initial release. These show the relatively large improvements that occur during the time interval following a new *UNIX* system first becoming operational, as the more obvious and important steps to improve performance are taken. Performance gains become more difficult to achieve as the system matures, as evidenced by the ultimate leveling off of the curves in Fig. 1. Note that the 30-percent improvements due to tuning of the C and assembler code for the VAX line actually exceed in magnitude the differences in performance of adjacent machine family members, the VAX-11/780 and VAX-11/750. PDP-11/70 Release 4.0 performance was slightly worse than that of its predecessor as a result of inadvertent change in some highly tuned code segments during a functional enhancement; this was subsequently fixed.

3.2 Context switch

A key measure of kernel performance is the CPU time it takes to transfer control between user processes, referred to here as the *context-switch* time. Context switches are performed whenever a program has to wait for data to arrive from the disk or terminal; the state of the process is saved and a new process is set up to run so as to keep the CPU as busy as possible. (The term "context switch" is sometimes

used to describe the transfer of control between a user process and the kernel. In this paper, control transfers between user and kernel are treated as system call overhead and covered in Section 3.1.)

Figure 2 shows the change in context-switch overhead over the years, as measured using a benchmark program that forces control transfers between two processes by passing a byte of data back and forth between them. The times to perform equivalent I/O without context switches have been subtracted to obtain the values plotted. The overall pattern is similar to that of Fig. 1; substantial improvements take place early during the development cycle, followed by a stabilization in performance as the system matures. Again, the 25- to 30-percent improvement in VAX performance over time rival the differences in performance between machine family members.

The time spent in context-switch operations has fallen dramatically. VAX-11/780 machines that used System III for program development performed about 100 context switches per second, consuming about 10 percent of the total CPU time. As a result of the efficiency improvements just described and changes to reduce frequency described in Section 3.6, VAX-11/780 systems doing the same kind of work with System V perform about 40 context switches per second, consuming only about 3 percent of the total CPU time.

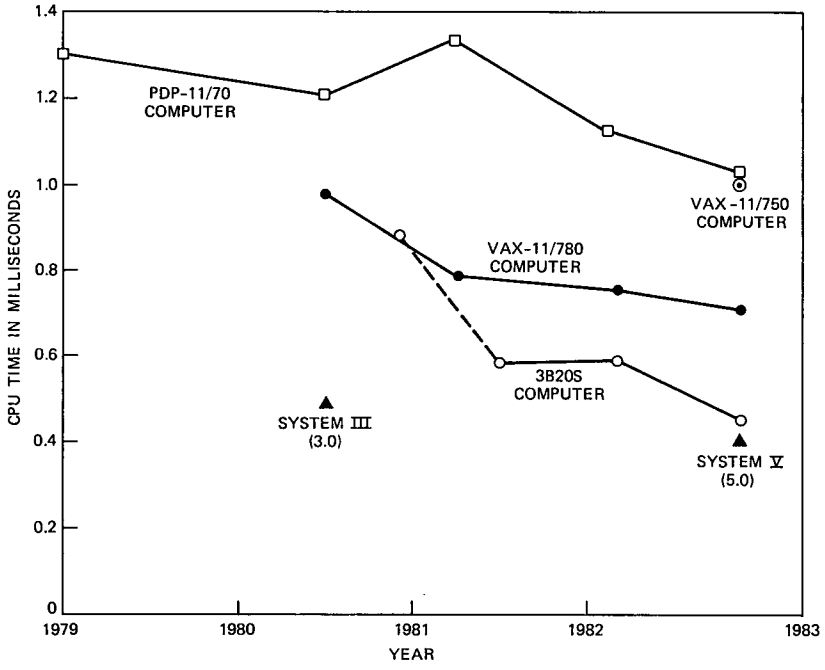

Fig. 2—Context switch.

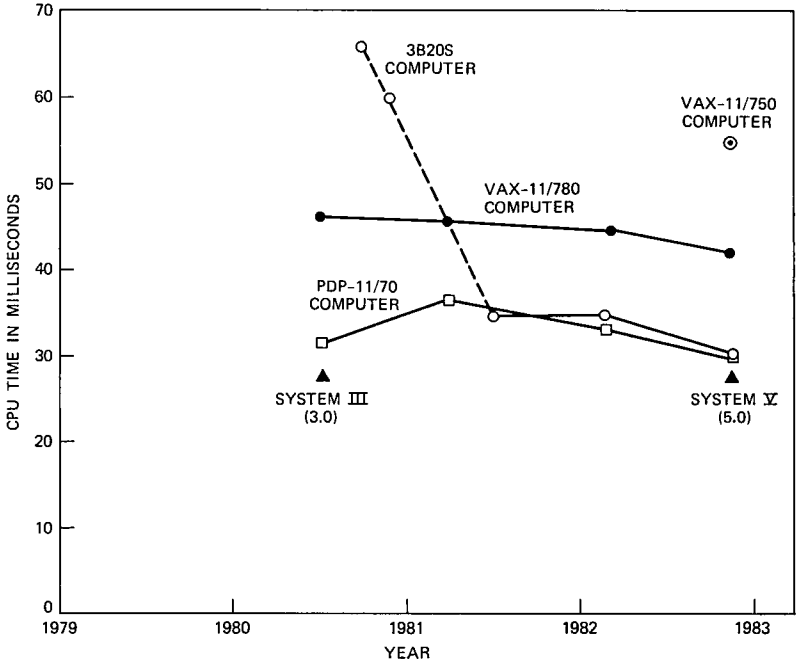

Fig. 3—`Fork/exit` + 32K-byte process.

3.3 Fork

Figure 3 shows the change in CPU time to `fork` (create) a new process and then `exit` (terminate it). For the *UNIX* system releases in this paper, the `fork` implementation requires the duplication of the data portion of the parent process; the time required is a function of the size of this data portion. The numbers in Fig. 3 are for a very small benchmark program to which 32 kilobytes of data have been artificially added.*

The strong improvements for the 3B20S computer in Fig. 3 are due largely to improvements in the kernel facility for copying data, supplemented by related microcode improvements. The unusually good performance of the PDP-11/70 computer on `forks` is due to the use of a different algorithm to replicate process data; the data part is copied to disk using a Direct Memory Access (DMA) transfer followed by a second DMA transfer of this data region into a different region of memory. The total CPU overhead for the two DMA transfers is

* This is done by having the benchmark program request more memory by means of an `sbrk` system call.

well below that of a comparable single memory-to-memory copy by the CPU.

`Fork` system calls are time-consuming, but their low rate of occurrence on program development systems (about one per second) keeps the total CPU consumption under 4 percent. The frequency of `fork`, however, is very dependent on application design.

3.4 Table searches

The original *UNIX* systems were implemented with linear table searches. These were well matched to the scarce memory and addressability, as well as smaller user communities supported by the low-end PDP-11 machines available at the time. Address space and memory are commonly no longer scarce, and user communities have grown larger. As a result, the key linear table searches have, one by one, been replaced by higher-performance ones. The *UNIX* operating system for the IBM 370[3] has been a leader at AT&T Bell Laboratories in this regard. The table search revisions have been a main factor in improved kernel performance.

First altered (done prior to System III) was the search to determine the presence of a particular disk block in the in-memory cache of disk buffers used to reduce disk accesses. Between Systems III and V the following additional search improvements were implemented:

1. Faster location of free slots in the in-memory *file-table* used to track current file transactions. This was done by maintaining a list of free entries.
2. Faster searches of the *process table* for releasing process roadblocks.
3. Faster searches of the in-memory *i-node table* used to track current activity on files and devices.

The i-node table searches were improved by instituting a "hashed" search strategy. Figure 4 demonstrates the improvement resulting from the faster i-node searches, by plotting the CPU time to locate a particular table entry as a function of its position. The actual operation measured is a `chdir "."`, that is, change directory to the directory where the program resides. This minimal operation does not accomplish anything useful; it does, however, entail a search for the i-node representing `"."` (The position of `"."` is controlled by starting with an empty i-node table and then opening a prespecified number of files. We then cause `"."` to be brought into the desired location of the i-node table by transferring into it as a directory.)

In Fig. 4, the systems with linear search strategies (PDP-11/70 computer; VAX-11/780 computer, Version 4.0) are shown with dashed lines; those with high-speed searches (VAX-11/780 and 3B20S computers) are denoted with solid lines. Version 5.0 results for VAX-11/

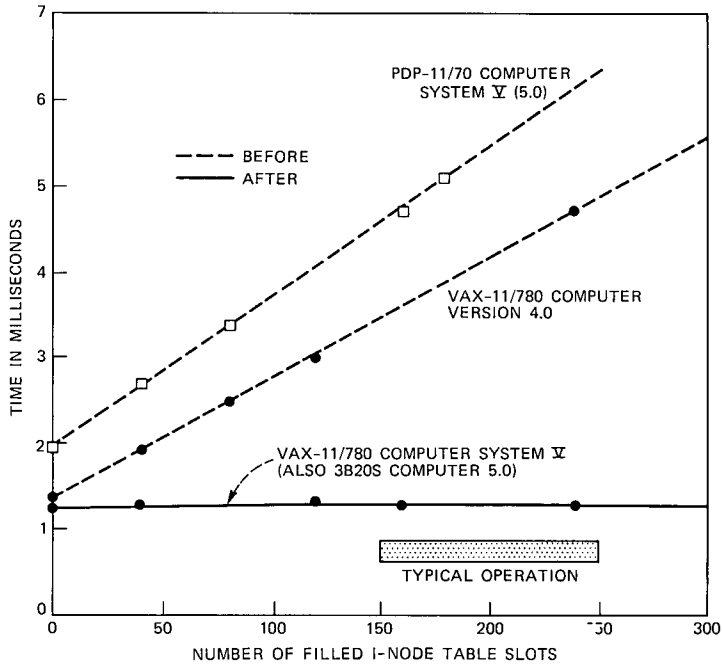

Fig. 4—Effect of modified i-node search (`chdir "."`).

780 and 3B20S computers are close, and are shown as one line; data points are for the VAX computer. Typical table fill levels for time-sharing use are indicated at the bottom of Fig. 4. PDP-11/70 and VAX-11/750 computers tend to operate at the lower end of the region shown (~160 slots in use); and VAX-11/780 and 3B20S computers at the upper end (~240 slots in use). The CPU time saving for this table search change on the VAX-11/780 computer is given by the distance between the respective solid and dashed curves. This saving depends on whether the desired entry is in the table, and on its position. Entries present in the table are located in a linear search, on the average, about halfway through the table. Entries not present result in searches through to the end of the table. One consequence of the old linear search strategy is that, if kernel tables were configured larger, failed searches would take longer, causing the operating system to run more slowly. Note that with the improved "hashed" search, search times are nearly constant. Furthermore (using the VAX computer as an example), measured search times are essentially equal to those for linear searches of nearly empty tables. (Theoretically, "hashed" searches of full tables should take slightly longer due to collisions; in practice, however, this effect is small enough to be difficult to measure.)

3.5 Data movement via pipes

UNIX system *pipes* transfer data between processes. They are implemented by copying data from the sender process into kernel buffers and then from these buffers into the address space of the receiver. Pipe measurements are important, because pipes are used a lot, and because they exercise operating system data copy and other mechanisms used more generally in reading and writing files; good performance here is especially important for applications that transfer large amounts of data.

Thus far we have looked at performance in terms of the time it takes to perform an operation; for pipes we view the work accomplished per unit time, which has the effect of reversing the ordinate direction representing good performance in the figures. Figure 5 shows the maximum rate at which data can be transferred between two processes using a pipe. This rate depends on the size of the chunks of data that are transferred. For the time being, let us direct attention to performance at 512-byte transfers. Several things are worth noting. First, for

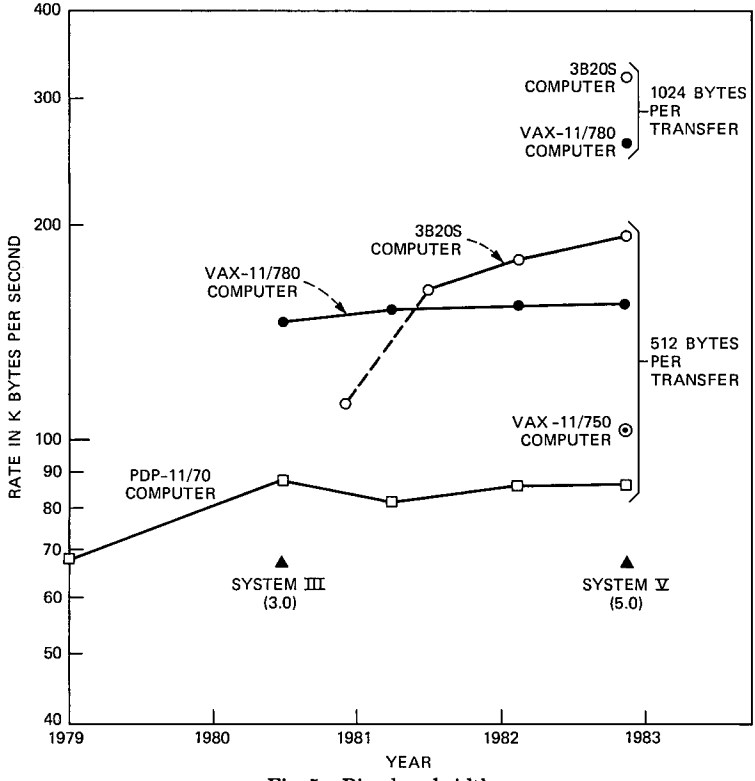

Fig. 5—Pipe bandwidth.

512-byte transfers, the 32-bit 3B20S and VAX-11/780 computers outperform the 16-bit PDP-11/70 system by almost a factor of two. This contrasts with the approximately 15-percent difference between these machines on general C language programs noted in Section II. The strong performance of the 3B20S and VAX computers is due to the greater efficiency of copying data for larger word-size machines. In fact, even the VAX-11/750 computer, which is notably slower than the PDP-11/70 computer in C language instruction rate (Table II), easily outperforms it on an operation such as piping, which involves moving data.

Over the time interval shown in Fig. 5, the DEC machines show little change in performance for 512-byte transfers. 3B20S computer performance has improved, owing to the data-copy microcode revisions described earlier.

For Version 5.0, the internal block size for the 32-bit 3B20S and VAX computers was changed to 1024 bytes, and the C library was changed to cause programs to read and write in 1024-byte chunks. Note in Fig. 5 that these changes combine to yield a factor of 1.5 to 2 improvement in overall throughput relative to 512-byte transfers. The PDP-11/70 computer retained the 512-byte size due to space limitations. As a result, there is a factor of three to four difference in System V pipe performance between the PDP-11/70 computer and the other machines.

3.6 Disk interaction

Until now, this paper has focused on improvements arising from doing things more quickly. Another way to gain performance is to do things less often. Disk accesses are a main consumer of *UNIX* system resources, affecting two critical areas:

1. The disk—The accesses create a load on the disk subsystem, most notably contention for the moving arm on each disk drive, which must be directed from place to place to fetch blocks from different cylinders of the disk.
2. The CPU—There is overhead on the CPU due to the need to queue disk transfers, service interrupts when disk transactions complete, and context switch so as to keep busy while waiting for data to arrive.

Disk and CPU overhead are each incurred on a per-transfer basis, and (for transfers of the sizes discussed here) are largely independent of transfer size. This creates a strong incentive to reduce the number of disk transfers that take place.

One technique has been to increase the file system block size. This has the effect of cutting almost in half the number of accesses for sequential reads and writes of large files. (Transfers to access small

pieces of data such as file system i-nodes, small files, and directories are not helped by this change.) An unpleasant performance side effect of the large block size is that a given size memory cache is able to hold fewer buffers; this reduces effectiveness, since some blocks are retained that hold only a small amount of useful data. There are also adverse disk space side effects, but these have been alleviated by the availability of higher-capacity disks as technology advances.

Reduced buffer-cache effectiveness was helped by a second major step taken to reduce the need for disk transfers: the use of a larger buffer cache. This reduces disk interaction by increasing the likelihood that desired data will be retained in memory. Main driving forces here were inexpensive memory and the release from size restrictions in moving from 16-bit to 32-bit addressability, creating an incentive to use large amounts of memory effectively.

Figure 6 shows the evolution in the number of buffers used by *UNIX* systems. For early systems, disk buffers were part of the kernel data address space; operating systems were configured by allocating to

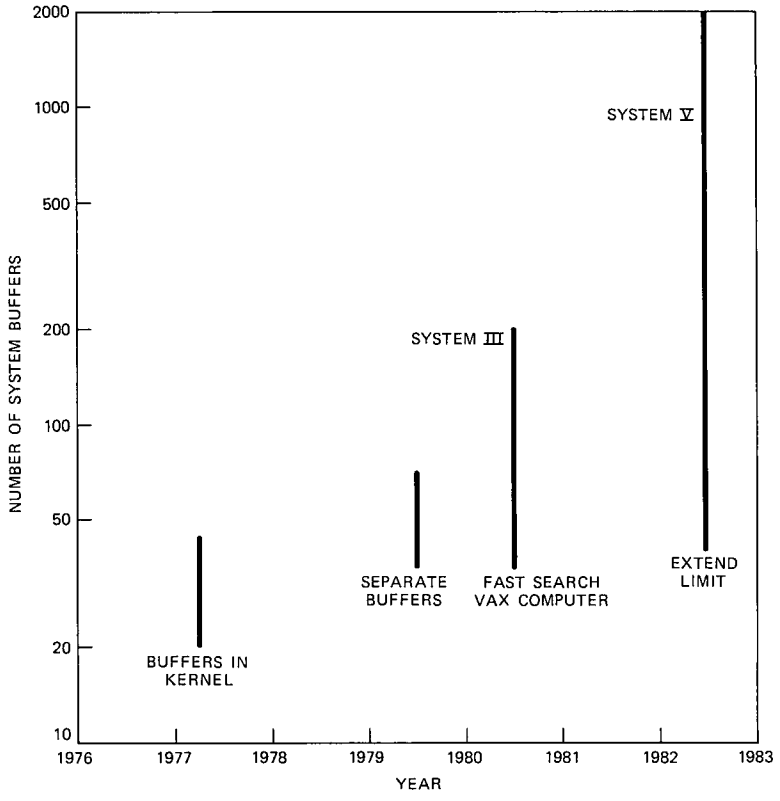

Fig. 6—Number of system buffers.

buffers whatever space was left over by the rest of the kernel. This typically left room for twenty to thirty-five 512-byte buffers.

The first significant change was to place kernel buffers in their own separate address space on PDP-11 computers. This allowed on the order of 100 buffers. When attempts were made to configure with this many buffers, however, performance got worse due to the increased search time to determine whether a buffer was in memory (using the then extant linear search strategy).

The next major change, which occurred for System III, was the use of the "hashed" buffer-cache search scheme. This coincided roughly with the initial *UNIX* system release for the VAX computer line. At this point in time, VAX systems using 150 to 200 buffers became common. Most recently, led by enthusiastic reports on experiments by AT&T Bell Laboratories computation centers, changes were made to relax remaining size restrictions, leading to systems where more than two thousand 1024-byte buffers can be configured on 3B20S and VAX computer installations carrying large amounts of memory. [Unfortunately, when run with this many buffers, our time-sharing benchmark (Section VI) operates with an unrepresentatively high buffer-cache hit ratio; we have not yet quantified the improvement offered by running with lots of buffers.]

The reductions in disk accesses especially help disk-limited applications. By lowering disk loads, they have also made less critical the tuning and distribution of disk activity for other applications. The resultant CPU savings played a large role in the time-sharing throughput gains described in Section VI.

3.7 Comparisons of Systems III and V

Table III gives times for some System V operations, along with improvements calculated by dividing System III operation times (t_{III})

Table III—System V kernel operations

Operation	3B20S Computer Time (ms)	VAX-11/780 Computer Time (t_v) (ms)	VAX-11/780 Computer t_m/t_v	PDP-11/70 Computer Time (t_v) (ms)	PDP-11/70 Computer t_m/t_v
1. * `Chdir "."`	1.2	1.2	(2.5)	3.4	(1.0)
2. * `Open/close` "file"	1.9	2.5	(2.8)	6.8	(1.0)
3. * Search path 3rd level	4.1	5.9	(2.9)	17.1	(1.0)
4. * Search dir 32nd position	2.2	3.0	(2.5)	11.1	(1.0)
5. Access disk block	3.1	3.1	(1.0)	4.2	(0.9)
6. `Read` 4K file	16.	19.	(2.1)	66.	(0.9)
7. `Fork/exit` 8K data	17.	22.	(1.1)	24.	(1.0)
8. `Exec` 8K BSS	14.	19.	(1.2)	35.	(1.1)

* I-node table entries: VAX = 120; PDP = 80

by the respective System V operation times (t_V). Version 5.0 results for the 3B20S computer have also been included for comparison.

The first four lines of Table III show the improvements for some representative file operations: `chdir "."` (as previously described); `open` then `close` a file that is not already open by another process; search (via access system call) to a third-level directory, and search to the 32nd position in a large directory. The data were taken with target entries at the halfway point with typical i-node table fill levels, and show improvements for the VAX computer by factors of 2.5 or more due to the faster table searches previously described.

As we see from lines five and six, the VAX CPU time to access a disk block has changed relatively little. (The time given includes disk management overhead and context switches, but not system call overhead or the time to copy the data into the address space of the user program.) However, since the System V blocks are twice as big, the respective CPU overhead to read a 4K-byte file is improved by more than a factor of two. The last two lines of Table III also show some modest VAX improvements in `fork/exit` and `exec` time.

In contrast to the VAX computer, PDP-11/70 kernel performance has been, across the board, relatively static. Many of the changes (particularly the block size and table search) involved trades of space for performance that were unattractive on a machine that was already pushing the limits of its 16-bit address space.

IV. TERMINAL HANDLING

The terminal-handling portion of the *UNIX* system performs a variety of services to make life easy for users at terminals. Terminal ports are also used for networking connections to other machines by means of `cu` and `uucp`. The general trend towards higher-speed lines, screen editors, new kinds of terminals such as the *Teletype*® terminal DMD 5620 (Blit),[4] and networking, have resulted in ever-increasing demands on terminal-handling software and hardware. Terminal handling is an area in which performance has improved most dramatically. This section addresses kernel overhead; there have also been C library improvements related to terminal handling, which we will discuss later.

Figure 7 depicts the change in terminal-handling overhead over time by showing the maximum achievable output traffic levels for *cooked* (characters processed) and *raw* (transparent) modes, assuming that the CPU is involved with nothing else but character output. The measurements were made while data were being outputted simultaneously on some twenty 9600-baud outgoing terminal lines. For some recent *UNIX* systems, even this very highly stressful situation is

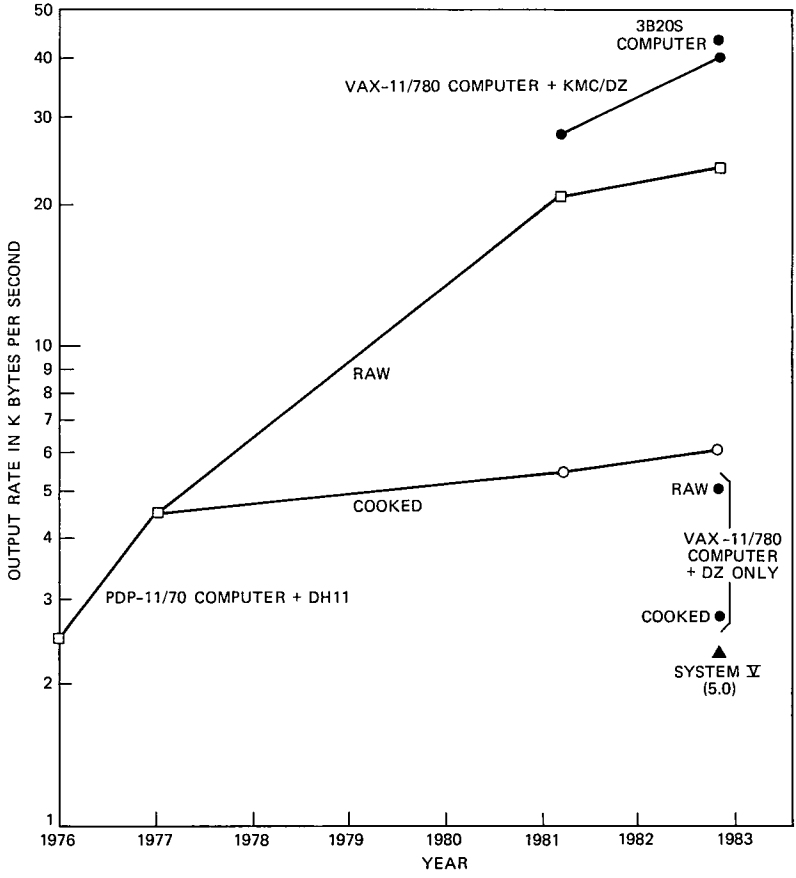

Fig. 7—Terminal output at 100-percent CPU use.

insufficient to load the CPU fully; ultimate capacity was then projected based on the traffic level and leftover CPU with 20 lines driven. (CPU consumption is approximately linear with traffic level.) The terminal-handling capacity measured in this fashion depends on the size of the data chunk that is written to the terminal. To stress the terminal handling maximally as opposed to other kernel parts, relatively large (256-byte) chunks were used for Fig. 7. Unfortunately, early (prior to 1977) data for the PDP-11/70 computer are unavailable; the first data point shown in Fig. 7 is an approximate projection of PDP-11/70 capability based on measurements made on the PDP-11/45 computer. (A 2.5:1 ratio in CPU power between the two machines was assumed for this.) Figure 7 shows that more than an order of magnitude reduction in terminal-handling overhead has occurred over time.

The original *UNIX* system terminal-handling algorithms had several design properties that severely limited the traffic levels that could be achieved. They were:

1. Interrupt for each outgoing character
2. Slow buffering mechanism (the original *clist*), involving a subroutine call to enqueue and dequeue each character
3. Poor (inefficient) provision for bypassing character processing for transparent output (raw mode). Transparency is especially needed in communicating with other machines.

The first major change, which occurred in PDP-11 systems released around 1977, was to take advantage of the DMA output capability of the DEC DH11 peripheral, then in heavy use. This removed the need for an interrupt for each character, substituting instead one every eight characters, and effectively halving total output overhead.

A second major set of changes occurred around 1980, and was centered around the introduction of a revised clist mechanism. The new scheme retained the old byte-at-a-time interface of the original clist, but also added a new one in which characters could be placed on and removed from queues in groups of up to 64 (24 for the PDP-11 computer). In addition to saving subroutine call overhead to enqueue and dequeue characters, the new scheme made possible bulk copies of outgoing data between user and kernel address spaces, thereby bypassing another extremely slow byte-at-a-time mechanism. For transparent output, the bulk-copied 64-byte regions of data were handled directly to the device driver as DMA output areas to achieve very low overhead. These changes permitted PDP-11/70 rates of approximately 6 and 20K-bytes per second in cooked and raw modes, respectively.

The VAX computer utilized the DEC DZ11 peripheral, which unfortunately lacked the DMA output feature that enabled the high performance levels of the PDP-11/70 computer. However, the Digital Equipment Corporation made available at about this time the KMC11 front-end computer, which AT&T Bell Laboratories developers programmed to handle *UNIX* system output character processing. It was possible to formulate a means of operation whereby the KMC11 was handed large blocks of unprocessed characters, and would process and transmit them via the DZ11, but still appear to the kernel as a simple DMA device. This mode of operation permitted all of the previously discussed efficiencies of transparent mode; overhead and achievable traffic levels for raw and cooked modes were then essentially equal. Continued minor refinements have appeared since System III, so that at this point VAX-11/780 machines using the KMC11 peripheral can achieve traffic levels in excess of 40 kb/s.

Figure 7 also shows the traffic levels that can be achieved on a VAX-11/780 computer without the KMC11. As we can see, the KMC

introduces an order of magnitude improvement relative to the DZ11 used alone.

The 3B20S incorporated a terminal-handling front-end from the outset, and thus throughout has had very low terminal-handling overhead. With current software, the terminal-handling performance of the PDP-11/70, VAX-11/780 (with KMC), and 3B20S computer is sufficiently good that character processing overhead due to screen editors, new terminals, and high line speed takes no more than 1 to 2 percent of the CPU and has ceased to be an issue of concern.

Character input overhead is roughly an order of magnitude higher than output overhead. Fortunately, input traffic levels from human typists are at least an order of magnitude lower and impose no significant load. Networking connections, however, often impose CPU loads in the neighborhood of 5 percent due to terminal input; this remains as an area where some performance improvement would be worthwhile.

V. C LIBRARY AND COMMANDS

The lack of formal benchmarks and systematic measurements for the C library and commands prevents giving a detailed performance history. This section presents the highlights of what we know.

5.1 C library

The C library routines act as an interface between commands and application code running at user level, and the kernel. The following focuses on performance changes in commonly used portions of the C library dealing with file I/O, string manipulation, and conversion between ASCII and numeric quantities. For System V, used in program development, these C library components are responsible for about 10 percent of the total CPU consumption. Although there is some difference between the actual changes and respective times at which they occurred for the various machines, some general trends emerge.

1. Assembler encoding—Beginning with the portable C versions of the C library, improvements were achieved on the VAX computer by recoding in assembler language, utilizing the functionality of special VAX machine instructions. A similar approach, supplemented by some specially tailored new instructions implemented in microcode, was also subsequently taken on the 3B20S computer. New machines entering the picture and rising support costs, however, have caused this approach to be reexamined. Fortunately, a good understanding of critical areas of C library performance has made it possible to recode major portions of the library routines in C and still preserve the performance of the assembler versions.

2. Changed level of abstraction—The original C library routines for file I/O and string handling were coded using character-at-a-time primitives (`putc`, `getc`, etc.). By eliminating these, it has been possible to take advantage of the functionality of *UNIX* system `read` and `write` calls as well as special machine features for handling large blocks of data. In some cases, the performance improvement from this change alone exceeds an order of magnitude.

3. Arithmetic on integers where possible—Since floating-point operations are commonly slower than their integer equivalents, it was desirable to change routines involving conversion between floating-point numbers and ASCII strings to do as much as possible of their total work using integer quantities.

4. Larger buffer size—When the 3B20S and VAX kernels were changed from 512- to 1024-byte orientation, the C libraries were similarly changed to buffer I/O in 1024-byte quantities to reduce system call overhead.

5. Buffered output to terminals—Buffering by the C library can interfere with interactive conversations with terminals. This is because output is held in buffers without being sent; users don't see it at the point when a response is intended. The original, heavy-handed solution to this problem was to make all output to terminals unbuffered. This caused output to be written in units of a single byte, resulting in very high overhead. System V handles the problem by buffering terminal output in units of lines and flushing partial lines to the terminal when input is requested. This permits interactive terminal operation and reduces overhead to output lines of any sizable length by an order of magnitude relative to the unbuffered approach.

5.2 Commands

Overall, the rather large body of command code has not been as finely tuned as either the kernel or C library. Many commands have been modified and made faster or slower according to whether the momentary purpose involved new features, performance, maintainability, or use of the C library. However, attempts to improve command performance have often yielded sizable gains. For example, a modest effort recently resulted in a factor of three improvement to the `cat` command, and a factor of two improvement to the `who` command. (These improvements appear in System V, Release 2.)

`Nroff`, owing to its prominence in overall CPU consumption at many installations, has been the most discussed command. Unfortunately, its complexity has discouraged attempts at tuning. For some applications, there are substitutes for `nroff` that are several times faster. Some feel, however, that a complete reworking of the text package would be the best approach.

VI. PERFORMANCE ON TIME-SHARING WORK LOADS

This paper has described improvements by widely differing amounts in various portions of the *UNIX* system. Work-load modeling benchmarks are used to determine the impact of the different individual improvements on ability to support specific real-life loads. These are constructed by observing a target application for a period of time and then creating a set of programs that imitate the application with respect to usage and proportion of time spent in various commands, libraries, and the kernel, as well as amounts of I/O and swapping activity. A number of such benchmarks have been developed to model various *UNIX* system usage situations, but most focus on special-purpose telephone company operations-support systems. This section will describe results for a benchmark intended to model some typical time-sharing use. The benchmark was based originally on a 1978 study of a community of programmers using a PDP-11/70 machine to develop software for the *5ESS*™ switching equipment;[5] the actual command mix has been updated, however, to reflect more recent *UNIX* system usage. Modeling every aspect of an application, however, can be difficult in practice, and requires some compromise. Observations of resource consumption of real-life work loads, therefore, provide a useful supplement.

Our time-sharing work-load benchmark operates by running increasing numbers of scripts consisting of *UNIX* system and editor commands in parallel, so as to obtain a picture of system performance under increasing load. The order in which the commands are issued is permuted in the various scripts so as to avoid synchronization effects. Commands and editor input are read from files, thus bypassing the terminal-handling portion of the system. This should distort results minimally, however, since terminal handling does not significantly consume resources on the *UNIX* systems described in this paper when used for program development. In accord with real-life program development situations, the benchmark is CPU-limited for the applied load range of interest and does not swap except at very high applied loads.

Figure 8 shows the throughput versus load for Systems III and V. Throughput increases as additional scripts are added during the early portion of the curves. This is because several scripts running in parallel are necessary to provide work for the CPU while I/O is taking place so as to achieve maximum throughput. There is a slight tendency of the curves to droop at high loads due to decreasing buffer-cache hit ratio and slightly higher system overhead.

Table IV summarizes the peak throughputs for System V and improvements since System III. The 32-bit VAX computer has enjoyed a 25-percent throughput improvement. Note that the VAX-11/750

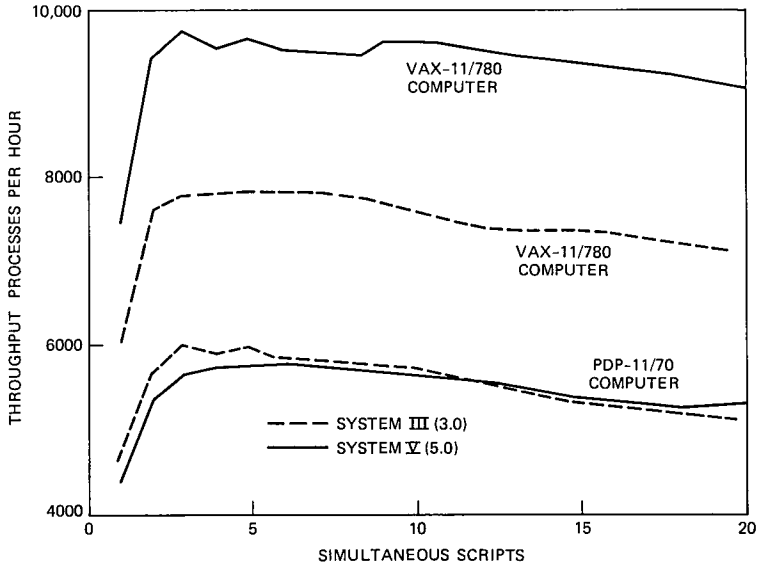

Fig. 8—Performance on time-sharing benchmark.

computer running System V outperforms the PDP-11/70 computer, whose performance has remained relatively static as the result of having been left out of key changes.

Throughput results from Fig. 8 and Table IV are supported by experience in monitoring amounts of work done in real program development environments with the systems in question. An attempt to calibrate the ordinate in Fig. 8 with the number of users capable of being supported was performed by surveying AT&T Bell Laboratories computation centers and asking how many time-sharing users would be placed on the various systems. This survey indicated that 10,000 processes/hour in Fig. 8 correspond roughly to being able to support 35 users with reasonable response.

Table IV—System V (5.0) peak benchmark throughput (processes/hour)

Machine	Throughput	Percent Change (Since 3.0)
3B20S	10,000	na
VAX-11/780	9,800	+25
VAX-11/750	6,000	na
PDP-11/70	5,800	-3

VII. SUMMARY AND CONCLUSIONS

This paper has described changes involving various portions of the *UNIX* system that have given rise to a 25-percent improvement in ability to support time-sharing users. Kernel revisions to take advantage of large address spaces and inexpensive memory have been the most significant factors, but improvements in the C library and selected commands have also helped. Kernel overhead, which in the past typically consumed 65 to 70 percent of the CPU, now consumes only about 50 percent. The most spectacular change has been a reduction by better than an order of magnitude in terminal-handling overhead, which has greatly eased the migration to higher line speeds, screen editors and networking. Performance of the object code produced by the C compiler has remained relatively static.

Kernel and C library improvements are pervasive and are likely to help any application that uses these components. On the other hand, the static picture for compiler code efficiency implies that applications that predominately execute application-specific code, and do not often use the kernel or libraries, will see no performance change.

It is difficult to compare *UNIX* system performance with that of other operating systems. Where an application makes only light use of operating system services, the comparison generally hinges on the relative efficiency of the compilers and libraries, performance of available software packages, and the suitability of the languages available on the systems to the task at hand. Where operating system services are used heavily, comparison is impeded by the difficulty of defining equivalences between operations for different operating systems and of determining the impact of missing functions and services. Efficient application architectures for the operating systems in question may be very different.

Where do we currently stand with respect to *UNIX* system performance, and what can we expect to see in the future? At this point, for the kernel and C library, we have addressed the more straightforward tuning steps and critical program areas as identified by profiling; we can obtain major improvements only by making fundamental changes and by moving functionality into hardware. As examples, new file system designs using much larger block sizes show greatly improved performance in transferring data, and systems with paged memory management can efficiently handle very large programs. The commands continue to be a fertile area for tuning and algorithmic revision. Global optimization for the C compiler also appears promising, although the extensive hand tuning that has already taken place throughout the system will reduce its impact.

Evolution towards greater functionality, such as transparent networking, will create challenges to implement new features without

hurting performance. The machines described here were originally designed without significant knowledge of hardware characteristics amenable to the *UNIX* system. Currently, as a result of experience in optimizing C, kernel and C library performance, we are in a much better position.

VIII. ACKNOWLEDGMENT

The author expresses his thanks to Jeffrey Lankford and Steven Sutor, who helped perform the measurements described in this paper, and Lawrence Rosler, who provided the material on C library performance. Jeffrey Lankford, Lawrence Rosler, and Barton Stuck provided helpful feedback on early drafts of this paper.

REFERENCES

1. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice-Hall, 1982.
2. C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering*, Chapter 2, Bedford, MA: Digital Press, Digital Equipment Corporation, 1978.
3. M. J. Bach and S. J. Buroff, "The *UNIX* System: Multiprocessor *UNIX* Systems," AT&T Bell Lab. Tech. J., this issue.
4. R. Pike, "The *UNIX* System: The Blit: A Multiplexed Graphics Terminal," AT&T Bell Lab. Tech. J., this issue.
5. S. L. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Proc. Computer Measurement Group XIII* (December 1982), pp. 62–7.

AUTHOR

Jerome Feder, B.Sc. (Electrical Engineering), 1963, Cooper Union; M.Sc. and Ph.D. (Electrical Engineering), New York University, 1969; AT&T Bell Laboratories, 1969—. Mr. Feder's early work at AT&T Bell Laboratories was in the area of machine aids and computer graphics, where he worked on the Graphic 2, XYMASK and Electron Beam Exposure System (EBES) projects. He is currently the Supervisor of the *UNIX* Performance Measurement and Analysis group. He has been engaged in software development and performance measurements involving the *UNIX* operating system since 1975.