Developing Benchmarks to Measure the Performance of the Mach Operating System *

David Finkel Robert E. Kinicki Aju John
Bradford Nichols Somesh Rao

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609

Abstract

We have developed a series of benchmark programs to test the performance of the Mach operating system. A discussion of the philosophy of the benchmarks is given, and the individual benchmark programs are described. Results of the benchmarks on 386-based machines running Mach 2.5, Mach 3.0, and SCO System V/386 are given.

Appeared in: Proceedings of the USENIX Mach Workshop, 1990, 83-100.

1 Introduction

As Mach progresses towards its micro-kernel version, important questions about its performance need to be answered. The micro-kernel version will have some clear advantages over the monolithic kernel version: removal of contention points, improved portability, scalability, and ease of maintenance. However, an important question is how much these changes will affect performance. The micro-kernel version of Mach must demonstrate good performance if Mach is to gain acceptance outside the research community.

*This research is supported by a grant from the Research Institute of the Open Software Foundation.

In order to understand the performance implications of this micro-kernel design, we have undertaken a performance study of Mach. The overall goal of the study is to develop a series of benchmarking programs which simulate system usage at a user level, and will allow system developers to evaluate the performance of the evolving versions of Mach.

We are in the process of developing a suite of benchmark programs to test the performance of Mach on a variety of platforms. In the following sections, we describe our approach to developing system benchmark programs, describe the benchmark programs we have developed so far, and give some results for these benchmarks run on identical 386-based systems running Mach 2.5, Mach 3.0 and SCO System V/386. In the last section we present some conclusions and outline our plan for future work.

2 Previous Work

Of course, a large number of benchmarking studies have been published. Mach-specific benchmark results have appeared in [BLA89], [FOR89], [GOL90], and [TEV87]. For the most part, these performance studies were low-level benchmarks, repeatedly exercising a single system call or system service. While such low-level benchmarks are important to system developers to test the efficiency of their implementations, the results may be difficult for users to interpret in terms of the kinds of applications they are interested in running.

We identified a large number of benchmarking programs and suites, for example [SPE90], [CUR76], [DON87], [SMI90], and [WEI84]. For the most part, these benchmarks emphasize CPU intensive applications, and did not specifically target operating system performance. An especially thorough set of benchmarks for Unix systems is given in [ACE89]. This report describes ten benchmark suites, and gives the results of running them on 47 Unix systems. Again, most of the benchmarks are either CPU-bound, or low-level tests of system functions.

3 Developing User-Level Benchmarks

Our goal is to create user-level benchmarks of system services. We wish to create benchmarks that make extensive use of significant system services in

a mix reflecting the usage by actual user applications. We could have accomplished this either by adopting actual user code, as in the SPEC benchmarks [SPE90], or by writing synthetic programs with the desired properties. We have adopted the latter strategy, for the following reasons.

First, writing synthetic programs allowed us to understand in detail what system services are used by the benchmark programs. This in turn enables us to create low-level benchmarks to test individual system functions to understand the reasons for the differences between results on different systems, and to provide guidance to system developers about areas of the system needing improvement. Second, synthetic programs permitted us to parametrize the benchmarks, to allow the same benchmark program to run on small scale and large scale systems (with different parameter settings). Third, writing synthetic programs allows us to make sure that our suite of benchmarks collectively covers the entire range of important systems services. Finally, we can ensure that our synthetic benchmarks are easily portable and available free of cumbersome licensing requirements.

We used several methods to ensure that our synthetic benchmark programs reflected the usage pattern of system services representative of actual user application programs. One approach was to run actual user applications under the control of a profiler, such as *gprof* [GRA82]. This allowed us to identify the particular system calls used by the program, and the number of times each call was made in the application. We also used statistical utilities, such as *vmstat* and *iostat*, to track the use of other system resources. This then gave us some guidelines in constructing our programs and in tuning them to match the resource utilizations of the original programs.

Another approach to constructing our benchmarks was to examine the source code of the user application, and identify the key system calls. This, together with the information provided by the statistical utilities, allowed us to construct our benchmarks. In addition, one of our benchmarks does not emulate a user program, but is a specially constructed test program designed to test specific operating system features.

At present our benchmark programs all issue only Unix system calls. By avoiding Mach-specific system calls, the benchmarks are directly portable to Unix systems and the timing comparisons reflect differences in implementations and not differences in system calls.

4 Descriptions of Benchmarks Programs

At this writing we have completed three benchmark programs. They are 1) the synthetic dump benchmark, which is intended to emulate the activities of the Unix *dump* program, 2) the synthetic compilation benchmark, which emulates the activities of the *gcc* compiler compiling a large program, and 3) the jigsaw puzzle program, which tests memory management efficiency. We describe each of them in detail below. In addition, in Section 6 we discuss additional benchmarks which are under development.

4.1 Synthetic Dump Benchmark (*sdump*)

sdump is a synthetic benchmark program that functionally simulates the working of the Unix *dump* program, which is used to backup a file system onto tape. Executing *dump* requires root privileges and it is not possible to backup a part of the file system or a sub-directory. The *dump* program creates child processes to write the data on the tape unit and these processes communicate to the parent through pipes to obtain data and control information. *dump* speeds up the process of file transfer by forking up to a maximum of twenty processes that independently transfer data from disk to tape.

The key issue considered during the design of *sdump* was to avoid dumping a file-system. Instead, *sdump* uses a directory containing a fixed number of files in place of a file-system. It was done this way for the following reasons: 1) It is easier to transport a directory to systems, than an entire file-system. 2) This enables any user to run this benchmark, as opposed to the *dump* program, which requires super-user privileges. 3) A fixed number of files of known sizes can be used to test and compare a set of machines. 4) Varying the number of files permits the benchmark to be scaled for testing a variety of machine types.

A file that can be locked was used to simulate the tape backup unit to avoid making the test results depend on the speed of a particular tape unit. File locks were used so that at any instant, only one process could write to it, thus simulating the sequential nature of a tape device. Other processes will be blocked while the simulated tape is in use.

As in the *dump* program, *sdump* forks a child process that writes data onto the simulated tape. The parent sends the data through a pipe, to the child in blocks of 4 kilobytes (KB), which is the buffer size of the pipe. The

child reads the pipe to gather data and writes it in 4KB blocks onto the file emulating the tape. The child then checks a control pipe to see if more data is to come from the parent. A rudimentary protocol was established within *sdump* to synchronize the data flow. When the parent has finished sending the last packet of data, it sends a special character to indicate the end of transmission. This is recognized by the child which closes the file emulating the tape and exits.

Some aspects of *dump* are not simulated by the *sdump* benchmark. *sdump* has no operation in it to simulate the process of traversing the inodes in a file system, checking for dates and bit-mapping the pattern of inodes that need to be dumped, onto an array. *dump* depends on this array to determine which inode blocks need to be dumped. *sdump* also does not synthesize the *dump*'s feature, since tape unit failures will not occur in *sdump*.

sdump has the optional ability to fork a separate process for each file in the directory, each of these can then independently read the data from its file from the disk. Each of these processes can in turn create its own child process to dump the data on the tape. Multiple processes are created and each of them competes for the file that emulates the tape. The process that gains control first locks the unit for its use, while others wait for the resource. In addition, the size of the data packets written to pipes can be increased and the rate of reading the pipes can be reduced to stress the system to higher limits. The results shown are for the case when *sdump* does not fork individual porcesses for each file.

Depending on the operating system under test, time is measured by using the `ftime` or `gettimeofday`.

4.2 Synthetic Compilation Benchmark (*scomp*)

The purpose of this benchmark is to utilize system resources in a manner similar to a compiler. The approach taken in creating the benchmark was to use monitors and profilers to obtain information about system resources utilized as the compiler operated on a well-defined set of input files. The data collected was used to write a program called *scomp* which mimics the actual compiler in measured usage of system resources.

The compiler used was gnu's *gcc*. System resource utilization was measured for *gcc* compiling itself. This combination was chosen because *gcc* compiling *gcc* is a commonly used benchmark and because we had access to

the source code, which was necessary for using the profiler.

Profiling was done using the *gprof* program on a Sun 3/60 workstation. Using *gprof* required that *gcc* be built first with a -g option during compilation to create a profiled version and then making *gcc* again using the profiled version of itself as the compiler. The profiled version of *gcc* creates data files used by the *gprof* program to tally procedure call tables and a call graph. From the procedure call tables the system calls were identified.

Additional information about the amount of memory utilized and the amount of paging activity was collected using the *vmstat* monitor. The average number bytes for read and write calls was not readily available so benchmark values were chosen based on the number of files accessed, the number of calls and the average size of source and intermediate files (.i, .s, .o) used. The relative portion of time spent inside system space to inside user space was measured using the time program.

When the *gcc* compiles *gcc*, a make program executes *gcc* in sequence to compile each source file and then to link them. Each time *gcc* is called it forks a child process and executes in sequence the precompiler, compiler, assembler and the linker, depending on the input flags or the type of files supplied.

In order to maintain the environment of process forking and program execution occurring in the actual compilation process, the benchmark program was modeled along the lines of the actual compilation. A separate synthetic program is used to represent the make program, *gcc*, precompiler and compiler called *scomp*, *sgcc*, *scpp* and *scc1* respectively. The synthetic versions of assembler and linker programs were not created because we felt their basic nature in terms of system resource utilization was already captured by the synthetic precompiler and compiler. Calls to the assembler and linker are therefore replaced in the benchmark with calls to the compiler and precompiler respectively.

Each synthetic program is divided into two parts, one that exercises system resources and another that generates a proportionally correct amount of user time. The system resource code is a series of system calls as identified by profiling. Arguments for the system calls were chosen to generate the amount of utilization seen by the monitoring or files accessed. We provide the appropriate amount of user time by embedding the dhrystone benchmark code [WEI84] as a loop directly into each synthetic program and executing it in order to create the desired amount of user time.

Some Unix system calls identified by the profiling process were not available in all the operating systems that were to be tested. Most noticeably `vfork` was not available in the SCO System V/386. In order to maintain a common benchmark, we changed the `vfork` calls to `fork` in all the versions of this benchmark. All other system calls that were not common to all the operating systems did not contribute significantly to the execution time of the benchmark and were removed.

In order to isolate the performance of operating systems in providing specific types of system services used by the the compilation process, the benchmark can execute portions of the code related to either system utilization or user time. Within the system utilization code, execution can be limited to those system calls and activities associated with file operations, file reads and writes, memory operations and process management.

Time was measured in this program using `ftime`.

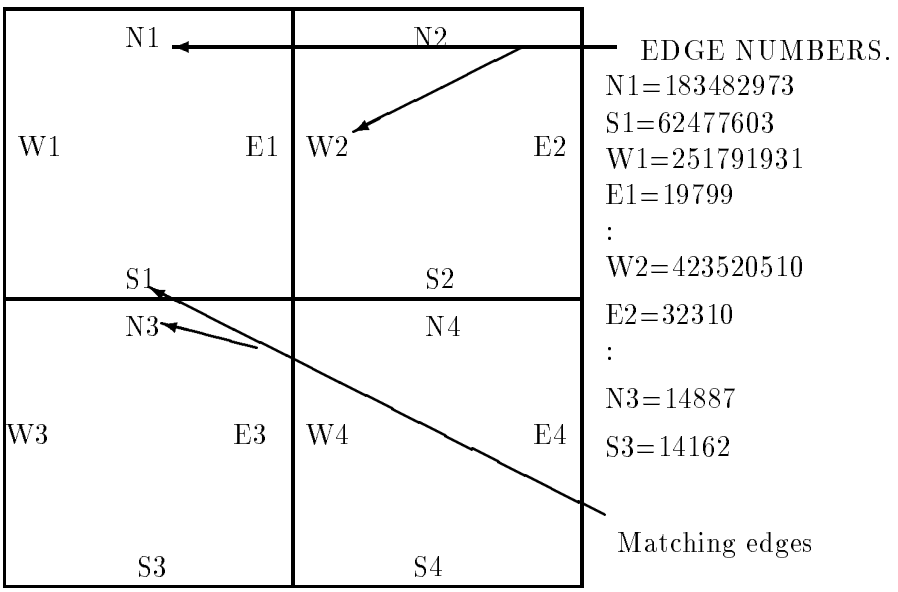
4.3 Jigsaw Puzzle Benchmark (*jigsaw*)

jigsaw is a benchmark designed to evaluate the performance of the memory management features of the operating system. It uses a mathematical model of a puzzle [GRE86] made up of tiles that are arranged in a square matrix. The entire puzzle can be regarded as a 2-dimensional array of tiles. Figure 1 shows a model of a simple puzzle of size 2x2. Each tile fits into a specific slot in the matrix, and this location is determined by comparing the edge numbers of the tile being considered, with those of its neighbors. The type of puzzle that is modeled is torroidal, where the left edge of the puzzle wraps around to the right edge and the bottom edge wraps around to the top edge.

4.3.1 Overview

jigsaw has 3 stages of execution:

- The puzzle generation stage: During this stage, the program accepts the size of the puzzle to be generated and dynamically allocates the memory required for the puzzle, and makes each tile of the puzzle by assigning its edge numbers and identity. The tiles are then connected to form a linked-list.



S1,N3 forms a simple pair of matching edges, while N1,S3 forms a torroidal match. Two edges match if the remainder of dividing the higher value with the lower equal a KEY value (101).

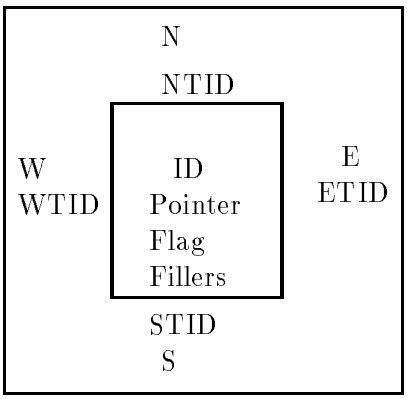
Figure 1: Mathematical model of a 2x2 puzzle

- The puzzle scrambling stage: The links of the tiles are re-arranged so that the sequence of the tiles gets jumbled. A fixed re-arrangement sequence is used in place of the suggested random sequence [GRE86], to ensure that the amount of computation for different runs are the same.

- The solution stage: The scrambled puzzle which is now in memory, is solved during this stage. In order to eliminate the effects of the delays caused by the secondary storage devices, only the solution process of the program is timed, using system calls `ftime` or `gettimeofday`, right before and after the solution stage.

4.3.2 Structure of a tile

Figure 2 shows the structure of a tile in the puzzle.



Structure:

N,S,E,W = edge numbers

N
S
E
W

TID= ID of the tile that matches that side

ID = Tile Identification number
Pointer= Pointer to next tile
Flag = to indicate solved status.

Fillers are used to increase the memory requirement of the tile.

Figure 2: Structure of a tile

It is assumed that each tile has four sides. A tile is represented as a record with fields for numbers to represent the four sides. These numbers are all unique and are used for finding the side of another tile that matches with it. For the sake of simplicity, only the left side of any tile can match with the right side of another, south side of one with the north side of another and so on. Each tile also has a field to hold its identifier and four fields (one for each side), initially blank, to record the identifier of the matching tile to that

side found while solving the puzzle. In addition, the tile structure has fillers to increase the memory occupied by the tile. By adjusting the parameters of the puzzle, such as the number of tiles in the puzzle and the size of each tile, the amount of computation and the memory requirement can be varied.

4.3.3 Solving the puzzle

The solution of the puzzle begins by picking the first unmatched tile in the sequence and comparing its edges with those of the remaining tiles until any side matches with any side of another tile. At this point, the IDs of the matching tiles are stored in the corresponding fields of the two tiles. When all four sides of the tile have been matched, the next tile is taken, to match the remaining sides. The solution continues until all tiles have been have been placed in the correct position.

4.3.4 Paging activity during solution

The movement of the tile of the puzzle is a logical one. It happens in two parts. At first, when jumbling the tiles of the puzzle, every tile is accessed to reposition the pointers of each tile, to point to another random tile, instead of the next logical tile in sequence. Later, while solving puzzle, the program accesses each tile of the jumbled puzzle, using the pointers and compares each tile with the sides of others for matches. At this point, each tile, represented by at least a KB of memory or more, is accessed excessively for comparing sides, and four write operations are done on each tile. For example, in order to complete the solution for a 32x32 puzzle, *jigsaw* has to perform 4096 writes and an average over 2 million reads to tiles in memory. If each tile, represented by 4096 bytes, form a page, *jigsaw* would make over 2 million page requests!

5 Results of Operating System Comparison Tests.

5.1 Equipment Configurations

Our tests were run on Hewlett-Packard Vectra RS/25C Model 304e machines. These machines are connected to the campus wide Ethernet Local Area Network. They have been set up so that it is possible to run these machines on the campus network or connected together in a stand-alone network.

One of the machines is running the Mach 2.5 operating system (release i386m, July 10 1990), one is running Mach 3.0 (release #6.0, June 19/20, 1990), and the third is running Santa Cruz Operations (SCO) Unix System V/386, Release 3.2. To compile our benchmark programs, we used the *gcc* compiler on the Mach machines, and the *cc* compiler provided with the distribution on the SCO System V/386 machine.

Each of these machines has the following configuration:

- 25 MHz Intel 80386 CPU
- 310 Mbyte ESDI hard disk drive,
- 5.25inch/1.2 Mbyte floppy drive,
- Flexible disk and ESDI hard disk controller
- 16 Mbytes of RAM
- Enhanced HP Vectra PC keyboard
- High resolution Display controller card.
- 16" High resolution Monitor (1280x1024 pixels)
- HP 27210A ThinLAN Interface Card (Ethernet)
- Serial Port
- Parallel Port

5.2 Numerical Results and Discussion

5.2.1 Synthetic Dump Tests

Results of *sdump*, given in Table 1 shows that Mach 2.5 has the best performance, compared to SCO System V/386 and Mach 3.0. Tests were done for data sizes varying from 1MB to 14MB. The entire range of tests showed that Mach 2.5 performed approximately 6% better for lower data sizes and about 10% better for large data sizes than SCO on a 386 machine.

This program stresses the system in terms of memory requirement and the management of virtual memory. Mach 3.0 failed to handle data files that were larger than 5MB. The failure is believed to have taken place when an attempt is made to create a file that is greater than 5MB. Other programs showed similar effects on Mach 3.0. For example, a simple attempt to catenate multiple files into a large one, greater than 5MB failed.

The figures for the benchmarks were obtained after multiple runs of these programs. The standard pattern that was observed was that, the first run took about 10% more time to run than subsequent runs for smaller data sizes. This pattern was observed on all the test beds. However, for higher data sizes, this effect was insignificant. Successive executions yielded similar results, with less than 1% difference between them. The first 4 runs were recorded and their average are reported. This was observed due to file caching that happens when all of the data can be fit into the available memory. As memory requirement equals or exceeds the memory available, effects of file-caching are less felt.

Table 1: **Synthetic Dump** *(sdump)* **Results**

Av data size MBytes	No of files	Mach 2.5 KB/sec	Mach 3.0 KB/sec	SCO System V/386 KB/sec
4.38	3	175	52	164
7.14	5	174	*	158

* Mach 3.0 crashes for large data sizes with an error:
"Trap type 3, Bus Panic"

5.2.2 Synthetic Compilation Tests

The results of running the synthetic compilation program on the test systems are shown in Table 2. Results are reported for running the complete benchmark, the benchmark with only the system utilization code (no dhrystone loops to generate user time) and with only the code representing the specific system utilization areas of file operations, file reads and writes, memory management and process management. The code for process management is executed in all versions.

For the benchmark in general the results show that system utilization code makes up roughly 10% of the total execution time of the benchmark. Within the system utilization code the memory management section is responsible for approximately 65% of the execution time followed by the file reads and writes, file operations and process management sections with approximately 20%, 10% and 5% respectively.

In overall system utilization Mach 3.0 is faster than Mach 2.5 which is faster than the SCO System V/386. Mach 2.5 is faster than Mach 3.0 in all system utilization areas except memory operations. This area dominates the system utilization execution time of the benchmark and the performance difference is large enough to make Mach 3.0 the overall winner. The largest proportional difference between the operating systems is in process management calls where Mach 3.0 and SCO System V/386 are from twice to three times slower than Mach 2.5.

A comparison of the complete benchmark results with the system utilization results shows that user time accounts approximately 90% of the execution time of the benchmark (and of the actual compilation on which this is based). Thus we see that compilation is not an operating system intensive application.

Each test was run three times, and the figure reported in Table 2 is the average of the three runs. The times for the individual runs varied by less than 2% from each other.

5.2.3 Jigsaw Puzzle Tests

Results of execution of the *jigsaw* for two tile sizes are presented in Table 3.

This program stresses the machine in two ways:

Table 2: **Synthetic Compilation** *(scomp)* **Results**

Test Type	Mach 2.5 sec.	Mach 3.0 sec.	SCO System V/386 sec.
Complete	810.0	795.7	942.5
System Utilization Code	82.3	74.1	87.1
File Operations	12.0	15.0	14.2
File Reads and Writes	19.4	21.4	22.8
Memory Management	56.8	50.7	64.4
Process Management	2.7	4.8	7.0

1. It dynamically allocates large amounts of memory for the entire matrix.

2. It then requests pages of memory at a very rapid rate for reading and for writing.

The puzzle size is gradually increased to a stage at which the system fails. Each run was monitored on the Mach 2.5 system to see the paging patterns. This was done by using the Mach system call, *vm_statistics*, before and after execution. This yielded among other figures, the total page faults and separate figures for soft and hard page faults. If a requested faulted page is to be obtained from the disk, it is regarded as a hard fault, else it is a soft fault.

Table 3 shows the test results run on different environments, up to failure. The performance of the virtual memory management was not as expected. It was observed that as long as the page faults that occurred were soft faults, the program's demands were met by the system. Hard faults were observed to begin for puzzle size 32x32, with 4MB tile size and sharply increased with higher puzzle sizes. The program execution failed, signaling a "Bus Error" on SCO system and "Error: Trap type 3, Bus Panic" on Mach systems, when trying to solve puzzles of large sizes. On the Mach systems, these programs were re-compiled with debug option and were run under the debugger *gdb* which reported that the crash occurred from within the kernel.

Mach 2.5 was observed to be the most stable of all the systems tested. The SCO System V/386 was very unstable and an unusual observation was that the program used to run for some values of memory requirements but not for others. The reason for this behavior is not known.

Mach 3.0 performed the best before it failed, when compared to Mach 2.5 and SCO, in most runs. Unfortunately, unlike Mach 2.5, it was not stable for higher puzzle sizes. In all cases, the failure was seen at the time of solving the puzzle, where pages are requested at a rapid rate. All the systems could handle the generation and jumbling of the puzzle, since paging was gradual in this part of the program.

The figures reported in Table 3 are the average of three runs of the *jigsaw*. It was observed that there was very little variation in the times reported by consecutive executions of the benchmark.

6 Conclusions and future work

We have presented benchmark programs which test system services in a variety of areas, and shown the results of running these tests on Mach 2.5, Mach 3.0, and SCO System V/386. At present we are working on extending these results in a number of directions.

First, we are constructing additional benchmarks. One will be based on the operation of the X window system, emphasizing interprocess communication and network traffic. In this effort we have been helped very much by some tests and performance tools presented in [DRO90]. We are also developing a benchmark based on the operation of a database system, which exhibits a very different sort of file access patterns than shown in our synthetic dump program.

We would also like to run our benchmarks on a wide variety of machines. This will require finding appropriate parameter settings for our benchmarks for different classes of machines.

In addition, we would like to run the tests concurrently to observe the performance of the system under load. This could be accommplished by running multiple copies of the same test, or different tests concurrently.

The issue of using benchmarks to compare incompatible systems also needs to be addressed. This necessitates leaving the domain of Unix-syntax benchmarks, and using the different system calls that each operating system permits. While it is obviously more difficult to ensure the fairness of such comparisons, this approach allows each operating system to use those features

Table 3: **Jigsaw Puzzle** *(jigsaw)* **Results**

Tile size KB	Puzzle Size	Mach 2.5 msecs.	Mach 3.0 msecs.	SCO System V/386 msecs.
1	8x8	30	20	-
	12x12	100	90	116
	16x16	230	230	-
	18x18	360	360	-
	24x24	975	930	1000
	32x32	2360	2470	-
	40x40	5590	-	-
	45x45	8620	-	-
	50x50	12260	-	-
	55x55	16690		
	60x60	21660		
	62x62	-		
4	8x8	30	30	-
	12x12	105	100	124
	16x16	250	240	-
	18x18	370	-	-
	24x24	1003	-	1693
	32x32	2546	-	-
	40x40	5715	-	-
	45x45	-	-	-
	50x50	-	-	-
	55x55	-	-	-
	60x60	-	-	-

Note: A "-" indicates unsuccessful execution.
A blank space indicates that the test was not executed.

that allow it to work best.

Acknowledgements

This research is supported by a grant from the Research Institute of the Open Software Foundation. The authors wish to thank ACE (Associated Computer Experts, bv, Amsterdam) for providing a copy of their benchmarking report. The authors also wish to thank Mr. Dhruve K. Shah of Worcester Polytechnic Institute, the co-author of the *jigsaw* puzzle program described in Section 4.3.

References

[SPE90] "Benchmark Results ", *SPEC Newsletter*, Vol. 2, No. 2, Spring 1990.

[ACE89] *Benchmarking UNIX Systems* , ACE (Associated Computer Experts bv, Van Eeghenstraat 100, 1071 GL Amsterdam, The Netherlands, 1989.

[BLA89] D.L. Black, R.F. Rashid, D.B. Golub, C.R. Hill, and R.V. Baron, "Translation Look-aside Buffer Consistency: A Software Approach", *Digest of Papers, COMPCON Spring '89, Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage* , (1989), 184 - 190. [Also Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-201].

[CUR76] H.J. Curnow and B.A. Wichmann, "A Synthetic Benchmark", *The Computer Journal*, 19 (1976), 43 - 49.

[DON87] J. Dongarra, J.L. Martin and J. Worlton, "Computer Benchmarking: Paths and Pitfalls", *IEEE Spectrum*, July 1987, 38 - 43 , .

[DRO90] R.E. Droms and W.R. Dyksen, "Performance Measurements of the X Window System Communication Protocol", Bucknell University Computer Science Department Technical Report #90-9, March 1990.

[FOR89] A. Forin, J. Barrera, M. Young, and R.F. Rashid, "Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach", Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-165. [Also published as "The Shared Memory Server", USENIX Winter Conference, San Diego, 1989.]

[GOL90] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program", *Proceedings of the USENIX Summer Conference*, June, 1990, 87 - 95.

[GRA82] S.L. Graham , P.B. Kessler, and M.K. McKusick, "gprof: A Call Graph Execution Profiler", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No. 6 (June 1982) 120 - 126.

[GRE86] P.E. Green and R.J. Juels, "The Jigsaw Puzzle - A Distributed Performance Test", *Proceedings of the 6th International Conf. on Distributed Computing Systems*, May 19-23, 1986 288 - 295. .

[SMI90] B. Smith, "The Byte Unix Benchmarks", *Byte*, March 1990, 273 - 277.

[TEV87] A. Tevanian, "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", Ph.D. Thesis, Carnegie Mellon University School of Computer Science, Dec., 1987. [Also Carnegie Mellon University School of Computer Science Technical Report, CMU-CS-88-106].

[WEI84] R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark", *Comm. of the ACM*, 27 (1984), 1013 - 1030.