

October 1992

Context Switch Time Scalability: Benchmark Results

J.L. Sloan
jsloan@ncar.ucar.edu

SCIENTIFIC COMPUTING DIVISION

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO

Abstract

The work a computer system performs to temporarily suspend execution of one process and resume execution of another process is called a context switch. The time it takes for a computer system to perform a context switch is often cited as an important performance metric. Lengthy context switch time can result in poor performance in process-intensive applications such as distributed servers or interactive systems. However, context switch time is usually assumed to be constant for a particular system. This document describes a benchmark program which shows that for many systems this is not the case. The program was run on a variety of UNIX[®]-based systems at NCAR, and the results are given in both tabular and graphical form. They show that for many systems, context switch time increases, sometimes dramatically, with the number of concurrent processes running in the system. This non-scalability of context switch times leads to diminishing returns when, for example, upgrading to a faster processor in order to support more interactive users.

Introduction

A *process* is a thread of execution inside a software program. A program which contains, for example, three processes, has at any one point in time three different instructions which may be executed by the central processing unit. In a single processor system, only a single instruction can be executed at a time, and execution is switched between processes as each process waits for I/O or exceeds its allotted CPU time quantum. In a multiprocessor system, multiple instructions may be executed simultaneously. Still, whenever there are more process running on a computer system than there are central processing units, the processes must share the CPUs. The work that a computer system must perform to temporarily suspend the execution of one process and resume the execution of another is called a *context switch*.

The *context* of a process consists of all the information unique to the process that resides in portions of the computer hardware that cannot be shared concurrently among processes. Examples are the contents of general purpose and floating point registers and the contents of memory management registers. When a new process gains control of the CPU, all the context of the prior process must be saved, and the context of the new process must be reloaded. Reduced Instruction Set Computer (RISC) architectures are particularly susceptible to long context switch times, despite their faster CPUs, because part of their performance often comes from keeping more information in many high-speed registers rather than in memory, information which must be saved and restored during a context switch.

Some architectures reduce context switch time by having several banks of registers between which the CPU can switch. As long as the number of running processes is no more than the number of register banks, context switching is performed very quickly since no data is actually moved between the registers and main memory. Other architectures have a high speed context cache; a context switch moves data between the CPU registers and a particular context in the cache. Again, this accelerates the context switch as long as the contexts of all processes fits within the context cache.

There are several applications for which supermicrocomputer-class UNIX[®] systems are typically used that are process-intensive. For multi-user interactive use, each user may have tens of processes running simultaneously. For server use in a distributed computing environment, each client request may spawn off a new process which runs until the request has been serviced. Also, I/O bound systems will context switch more frequently, regardless of the number of processes, than CPU bound systems, since I/O bound processes spend more time waiting for I/O to complete.

Although context switch times for computer systems are often measured and published, frequently the time to do a context switch is assumed to be constant. However, the benchmark results discussed in this document show that not only may context switch time vary with the number of processes running on the system, but the increase in context switch time may not be linear.

NCAR uses hundreds of supermicro-class computers for a variety of purposes. Most of them are employed as compute servers, multi-user interactive front ends, and to provide other distributed services, all of which may be process-intensive. Poor context switch performance of a computer system could have serious consequences for its use in these roles.

Methodology

A goal of this project was to devise a context switch benchmark in which the number of processes could be easily altered, whose source code would be easily ported with few changes to a wide variety of UNIX[®] platforms, and which would incur a minimum of CPU time not related to context switching. The need for common source code across all platforms was deemed critical, since any substantial changes in the benchmark program would make meaningful comparison of the results between systems problematic. This requirement placed severe restrictions on how complex the benchmark could be. The program was reduced to the lowest common denominator in terms of available system facilities for timing and resource measurement. There was also a concern to minimize the effect that measurement itself might have on the results. Finally, the program had to be nearly correct the first time it was used; since many of the systems that would be benchmarked were loaners, there might not be an opportunity to rerun a modified benchmark on the same machine at a later date.

To measure context switch time, a method is needed to force a context switch under controlled circumstances with a minimum of extraneous overhead. This is harder than it sounds. Some operating systems have a **short wait** system call which just causes a context switch without any other action. The running process invokes the **short wait** system call, which causes the process' context to be saved and placed at the end of the list of processes waiting to run, to be restored and run when the operating system's process scheduler permits.

UNIX[®] however lacks a standard user-callable mechanism to do just a context switch, although such a facility is frequently available to functions internal to the UNIX[®] kernel. However, there are system calls which can force a context switch as a side effect. Unfortunately, these calls impose additional overhead performing their intended function while also causing a context switch. This overhead is measured along with the context switch time, so it is always possible that what is actually being measured is some non-scalable behavior other than context switch overhead. No matter what mechanism is used to cause a context switch, the benchmark program which uses it must run long enough that the overhead of for process creation and termination imposed at the beginning and ending of the benchmark be small relative to the total context switch time.

Waiting for I/O is a common reason for a context switch. The **read(2)** system call causes a context switch if the file descriptor being read does not have data available (for example, from an interprocess communication link to another process). This is the approach taken in the context switch benchmark program.

pipeline is a program, written in C, which can be used to determine how system performance degrades as the number of processes running on the system increases. This simple program was originally described in a discussion on the computer architecture newsgroup on USENET. It creates a ring of processes, the number of which is determined by a command line argument, all circularly linked by interprocess communication pipes. By passing a one byte token around through the pipes, the processes can explicitly and sequentially pass control of the CPU to one another. By running the benchmark several times, varying the number of processes each time, and measuring the execution time, a graph can be generated. By running the benchmark when a system is mostly idle, effects of other

processes in the system can be minimized. The source code for **pipeline** is shown in *appendix 2*.

It was necessary to develop two different C programs for measuring the CPU time used by the benchmark. **getrusage** was used on BSD-derived systems, while **times** was used on SVID-derived systems. The source code for **getrusage** and **times** are shown in *appendix 3* and *appendix 4* respectively.

In each case, the benchmark was run for one million (1,000,000) iterations with the indicated process counts on the following systems at NCAR as shown in the table below. In some cases the operating system limited the number of processes a single user could create, which reduced the upper range of the test, and there was not time, particularly on the loaner systems, to reconfigure the kernel for more processes. The process count was varied by not only powers of two (2, 4, 8, etc.) but also by multiples of ten (10, 20, etc.) so that artifacts in the design of the operating system or in the architecture of the context switch hardware, which might be keyed to powers of two, would not bias the results. The total number of context switches generated is the number of iterations multiplied by the number of processes. All operating systems were some variant of UNIX®.

Pipeline Benchmark				
Computer Vendor	Computer Model	CPUs Used	Operating System	Process Counts
Data General	AViiON AV400	1	DG/UX 5.4	2,4,8,10,16,20,32,48,64
Hewlett-Packard	HP3000/720	1	HP-UX A.B8.05	2,4,8,10,16,20,32,48
IBM	370/4381	1	VM AIX/B370 1.2	2,4,8,10,16,20,32
IBM	RS6000/530	1	AIX 3.1	2,4,8,10,16,20,32,48,64
IBM	RS6000/530	1	AIX 3.2	2,4,8,10,16,20,32,48,64
Solbourne	702/E	2	SunOS 4.1.1	2,4,8,10,16,20,32,48,64
Sun Microsystems	Sun-4/280	1	SunOS 4.1.1	2,4,8,10,16,20,32,48,64
Sun Microsystems	Sun-4/470	1	SunOS 4.1_PSR	2,4,8,10,16,20,32,48,64
Sun Microsystems	Sun-4/630	2	SunOS 4.1.2	2,4,8,10,16,20,32,48,64

Results

The results of the **pipeline** benchmark are shown in *appendix 1* in both tabular and graphical form. The graphs were generated using the **xgraph** program. All graphs are to the same scale so that they can be overlaid and compared. The following data are shown: is the number of processes created by the benchmark program. Note that in all cases there will invariably be other processes active in the system, so this number does not represent the total number of processes running during the benchmark. Furthermore, this number will vary from system to system. However, because the benchmark was always run for an extended period of time on an otherwise idle machine, the number of additional processes should be mostly constant while testing a particular system. The presence of the additional processes should not unduly affect the shape of the resulting performance curve, although it does make direct comparison of the raw data for different systems more difficult. is the portion of CPU time spent in user state. (*Sys*) is the portion of CPU time spent in system state. The relative differences between system CPU time and user CPU time may only

indicate the resolution to which a system's kernel can discriminate between system and user CPU usage. (*Cpu*) is the total CPU time. It should be no less than the sum of system and user times. It may be more if the system accounts for CPU seconds spent in some other unreported state. is the wall clock time. This can be less than the reported CPU time for systems with multiple CPUs, since for each real second there may be more than one available CPU second. (*Proj*) is the projected CPU time if the test case for sixty-four processes took exactly thirty-two times as long as the test case for two processes.

Two things to look for in the graphs: whether the CPU time increases linearly, and how closely the reported CPU time follows the projected time. The more the reported CPU time diverges from the projected CPU time, the less scalable the system is when adding processes. If the CPU time does not increase linearly, then there are serious scalability problems as more processes are run, and users may experience sudden and catastrophic changes in qualities such as interactive response time.

Discussion

This study illustrates how casual benchmarking of CPU performance can lead to misleading performance assessments. Many of the non-scalable systems benchmarked in this study have good context switch times for small numbers of processes. It is these good numbers that vendors advertise.

Increasing processor power on systems which exhibit non-scalable context switch time in order to support more concurrent processes (for example, to support more interactive users) can yield diminishing returns. Context switch overhead can increase dramatically and catastrophically once some kernel or hardware imposed threshold is passed. Even for those systems in which context switch time increases linearly (i.e. those which do not have "knees" in their performance graphs), context switch overhead may eventually dominate the available processor power as the number of concurrent processes is increased.

Although this paper does not pretend to identify the cause of non-scalable context switch performance in the various UNIX kernels, we can suggest what to look for. One example would be algorithms used to search the list of runnable processes. A naive approach would be to keep the information on processes in a list that is searched linearly from top to bottom. This would be workable on computers which could support only a few tens of processes, but it would present a scalability problem for more powerful machines which would be expected to support many more processes. A search for 1 through n processes using a perfect hashing scheme would result in a total search cost of n , while a simple linear search would result in a total search cost of $\sum_{i=1}^n i$ which approximates an $o(n^2)$ algorithm.

This establishes some rational boundaries on the search time.

Recent UNIX[®] kernels optimize the linear approach for searching for processes in the process table. Berkeley 4.3BSD [BSD86] uses a simple hash scheme to generate a first approximation of the process' position in the list, then linear searches from there. System V Release 2 [SYSV84] performs a linear search, but always begins the search where it left off in the list last time, so that consecutive searches for the same process return immediately.

The graphs of the benchmark results fall for the most part between the n and n^2 curves. This is consistent with the fact that the benchmarked systems are based on implementation of System V or Berkeley UNIX[®]. However, the results suggest that perhaps the search algorithms used are still not scalable to large numbers of processes.

Mogul and Borg [Mogul90] suggest that context switch performance may be affected by the presence of a memory data cache. Context switching violates the locality of reference upon which cache architectures depend to accelerate processor performance. Mogul and Borg's results suggest that cache performance can produce context switch times on the order of tens to hundreds of microseconds for the microprocessors they studied.

Acknowledgements

Much of the inspiration for this project came from a lively discussion on context switch measurements in the computer architecture newsgroup *comp.arch* on the computer network USENET. I am indebted to the discussion participants, particularly Peter Van Epp of Simon Fraser University (Burnaby, British Columbia) who pointed me in the right direction, and Peter Lamb of the Institut fuer Integrierte Systeme (Zuerich, Switzerland) who shared with me his own results from a similar benchmark which he ran on a variety of Sun Microsystems SUN-3 platforms. I am also grateful to Dennis Hunter, of the University Corporation for Atmospheric Research, for providing me with a real-life example of the problems with unscalable context switch performance. Finally, thanks go to Paul Rotar, who reviewed this paper and made many helpful comments.

The sources for the software described in this document, along with helpful shell scripts and raw data, are available in machine readable form from the author.

UNIX[®] is a trademark of AT&T.

References

- [BSD86] Licensed source code, *Berkeley Software Distribution 4.3*, University of California at Berkeley, 1986
- [Mogul90] Mogul, J., and A. Borg, *The Effect of Context Switches on Cache Performance*, WRL Technical Note TN-16, December 1990
- [SYSV84] Licensed source code, *UNIX System V Release 2.0 3B2 Version 1*, AT&T Technologies, 1984

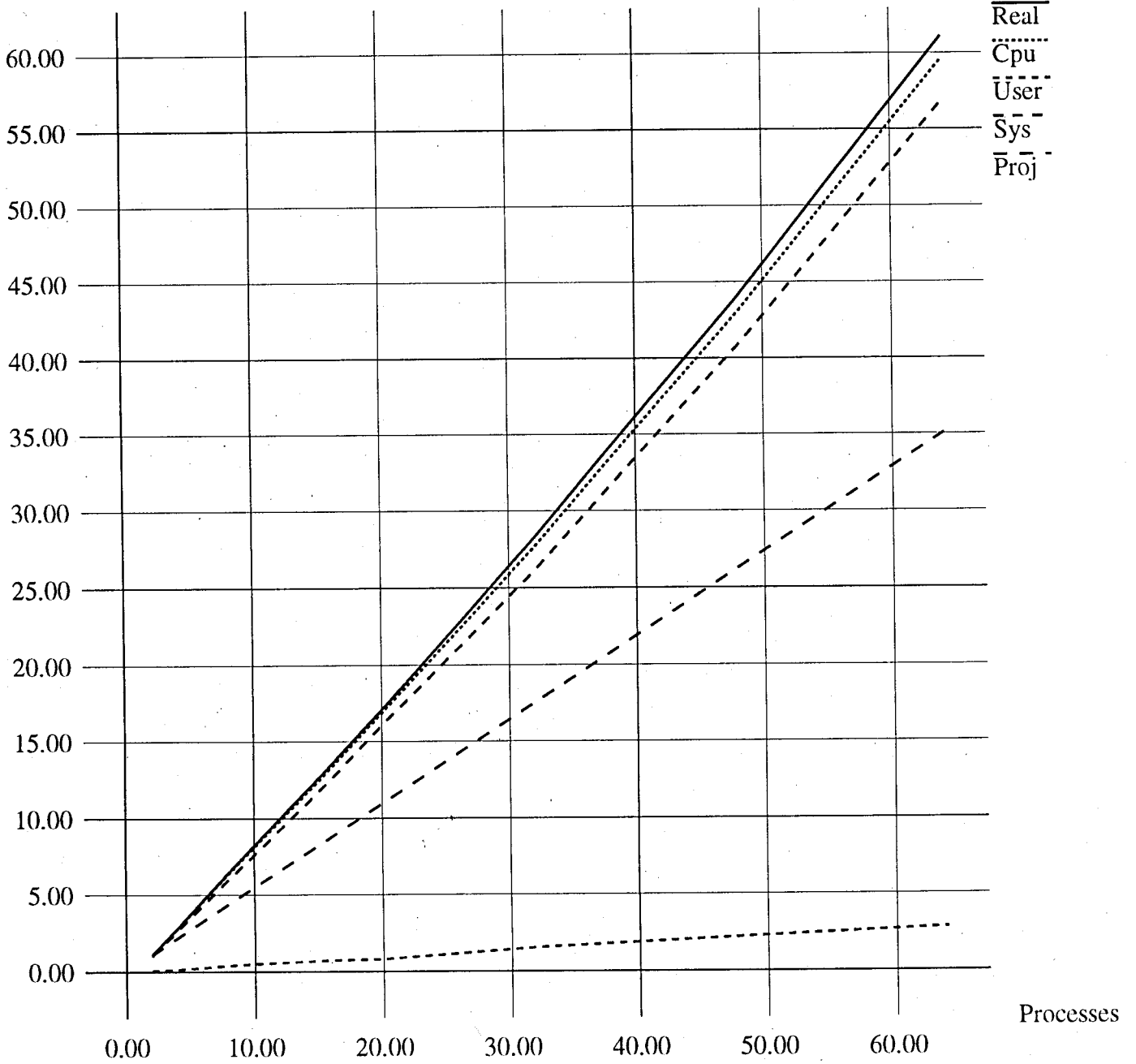
Appendix 1
Graphical and Tabular Results

Data General AViiON AV400, DG/UX 5.4

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	1108	1096	58	1039	1096
4	2816	2786	152	2635	2192
8	6439	6368	395	5974	4384
10	8215	8091	500	7592	5480
16	13471	13265	734	12532	8767
20	17076	16865	808	16057	10959
32	28234	27636	1548	26089	17535
48	44038	43024	2206	40819	26302
64	61135	59610	2808	56803	35069

Data General AViiON AV400, DG/UX 5.4

Seconds x 10³

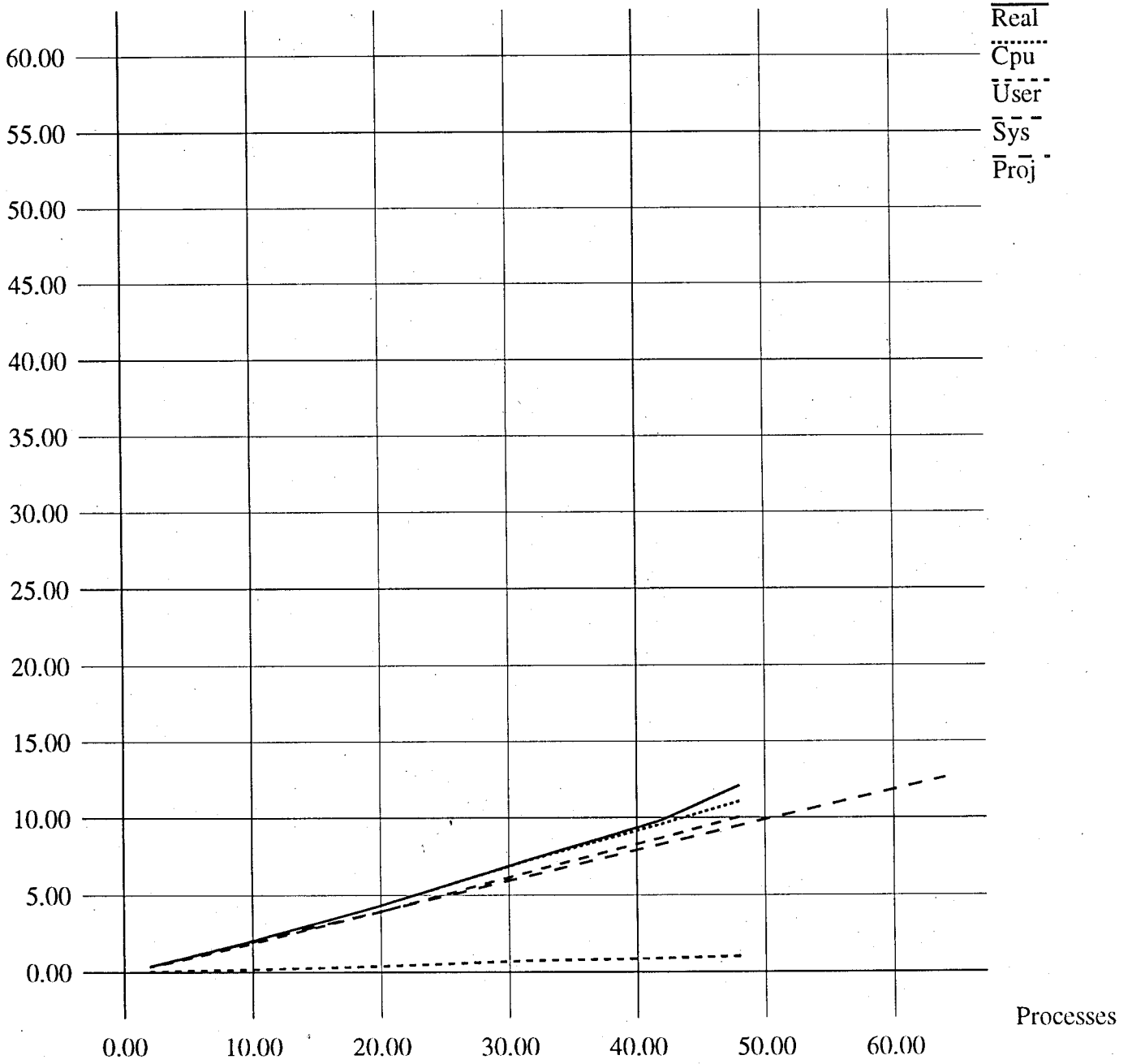


Hewlett-Packard 9000/720, HP-UX A.B8.05

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	395	395	36	360	395
4	790	790	73	717	790
8	1624	1624	152	1473	1580
10	2048	2048	182	1867	1976
16	3394	3394	299	3096	3161
20	4328	4327	391	3937	3951
32	7400	7346	751	6595	6322
42	9839	9588	886	8703	8297
48	12108	11066	1028	10039	9483

Hewlett-Packard 9000/720, HP-UX A.B8.05

Seconds x 10³

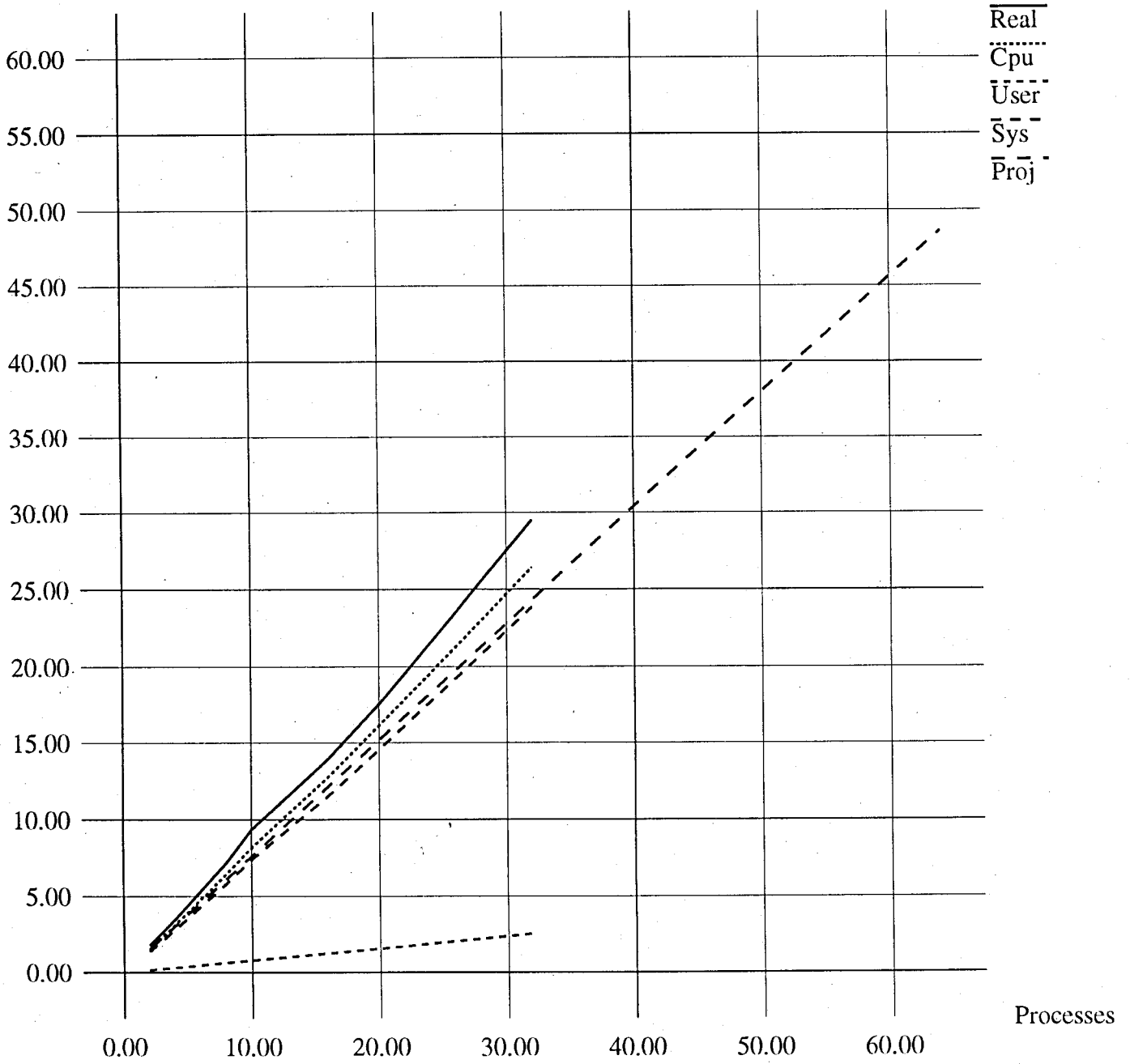


IBM 370/4381 (1), VM AIX/370 1.2

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	1774	1520	158	1363	1520
4	3463	3116	313	2804	3039
8	7139	6412	632	5780	6079
10	9340	8160	793	7367	7598
16	13947	12777	1251	11527	12157
20	17527	16123	1560	14564	15197
32	29488	26407	2544	23863	24315

IBM 370/4381 (1), VM AIX/370 1.2

Seconds x 10³



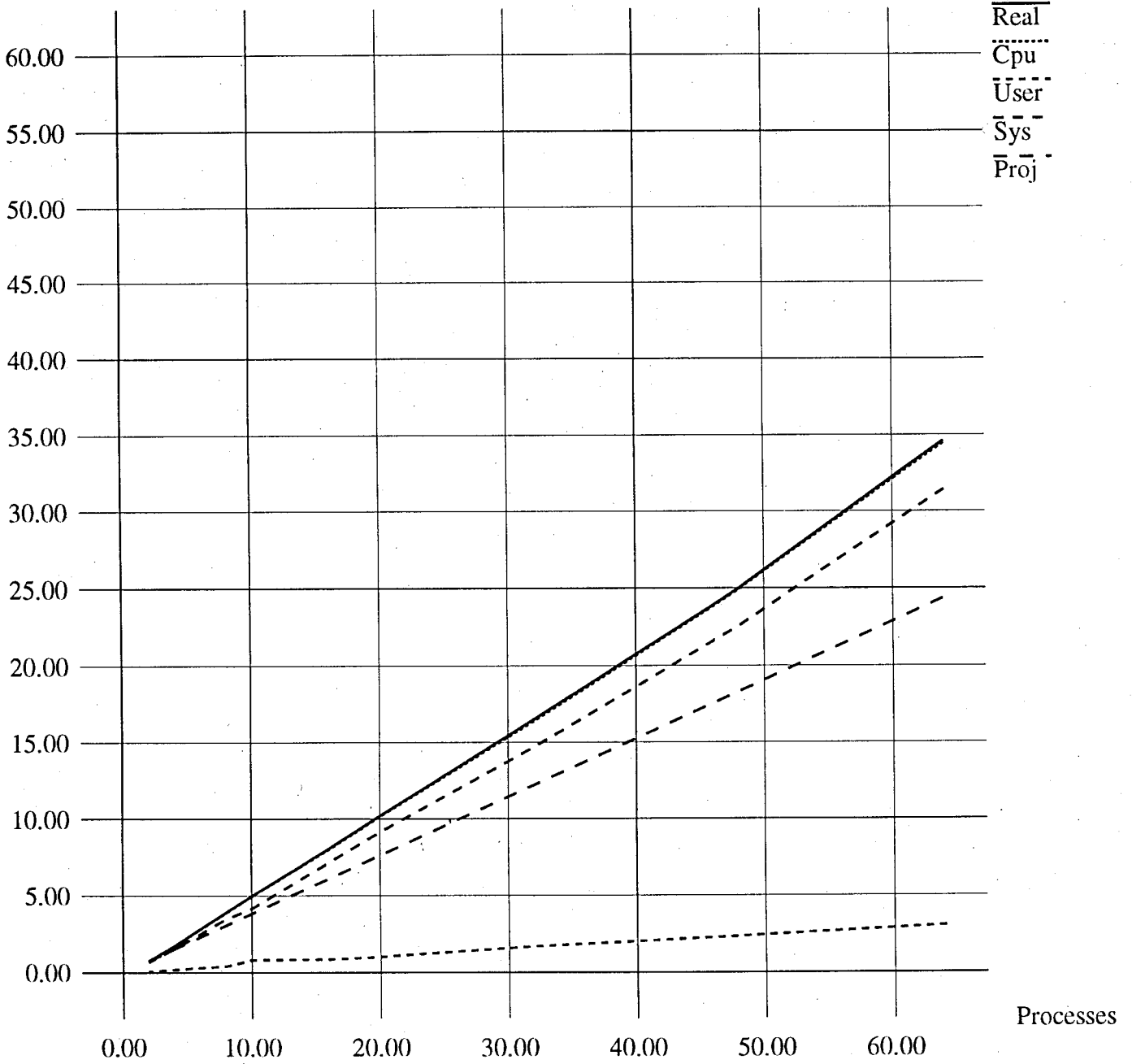
IBM RS6000/530, AIX 3.1

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	766	763	77	686	763
4	1737	1724	198	1527	1526
8	3884	3861	405	3456	3051
10	4955	4942	811	4131	3814
16	8052	7997	853	7144	6102
20	10196	10150	1014	9136	7628
32	16494	16365	1679	14687	12204
48	24969	24897	2336	22561	18306
64	34609	34456	3038	31418	24408

Comparing the IBM RS6000/530 running AIX 3.1 with the HP9000/720 shows how measuring raw CPU speed with small numbers of processes can be misleading. The two machines are quite close in performance with two processes, the RS being 1.93 times slower than the H-P. As the number of processes increases, the H-P remains close to the line of perfect scalability, while the RS diverges. As processes are added incrementally, the RS running AIX 3.1 is increasingly expensive. At the forty-eight process data point, the RS running AIX 3.1 is 2.25 times slower than the H-P.

IBM RS6000/530, AIX 3.1

Seconds x 10³



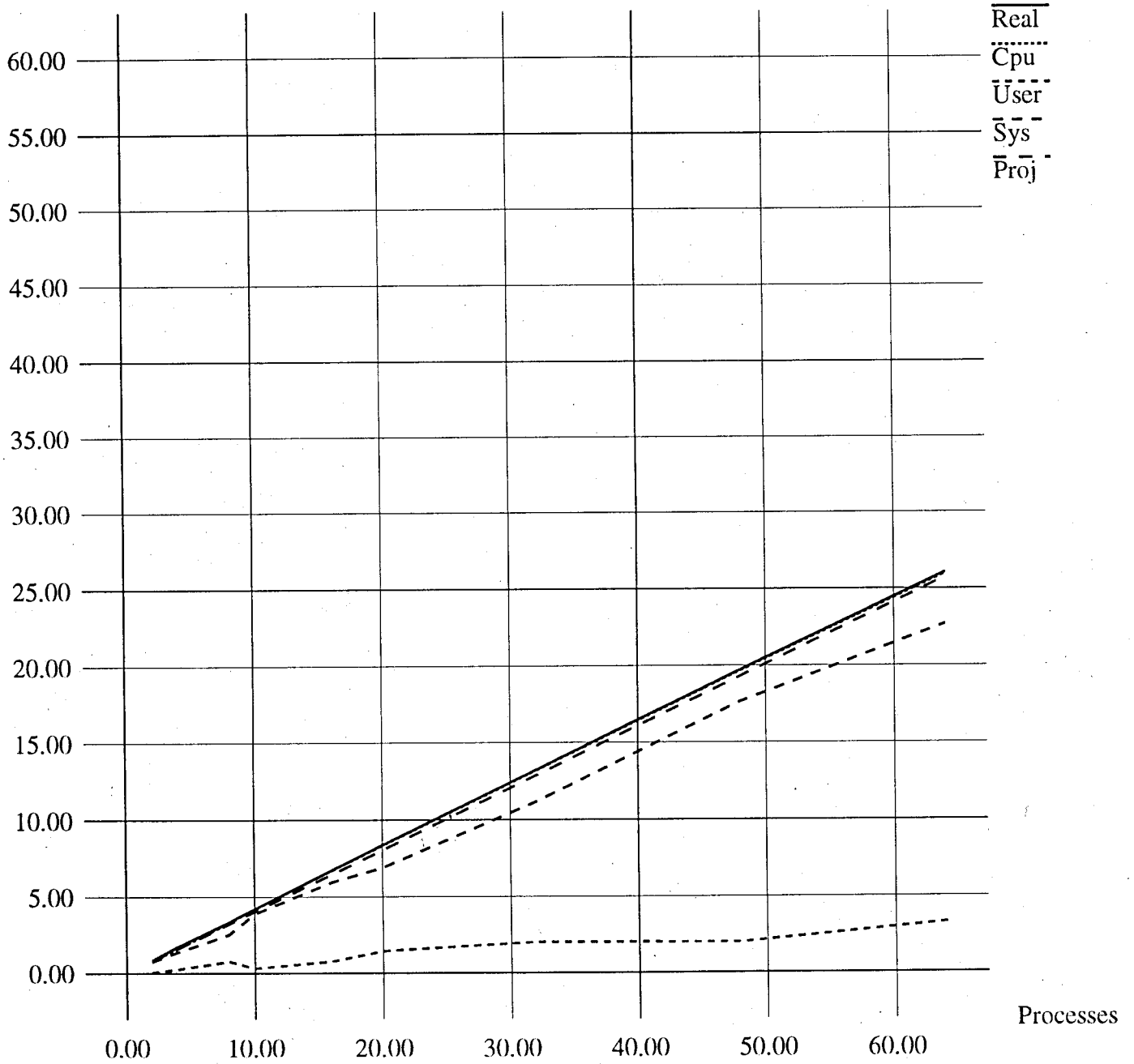
IBM RS6000/530, AIX 3.2

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	840	804	53	751	804
4	1751	1622	251	1371	1608
8	3334	3295	763	2532	3215
10	4197	4175	315	3860	4019
16	6718	6700	763	5938	6430
20	8370	8332	1435	6898	8038
32	13220	13178	2020	11159	12860
48	19703	19645	2002	17644	19291
64	26061	25979	3277	22703	25721

A comparison of AIX 3.2 with AIX 3.1 on the same machine show how tweaks in kernel design, and not just hardware architecture, can dramatically affect system performance. While the cost for two processes in AIX 3.1 was actually less than in AIX 3.2, the cost in 3.1 is not scalable, while in 3.2 it is. Hence for larger numbers of processes, AIX 3.2 has less total overhead for context switching.

IBM RS6000/530, AIX 3.2

Seconds x 10³



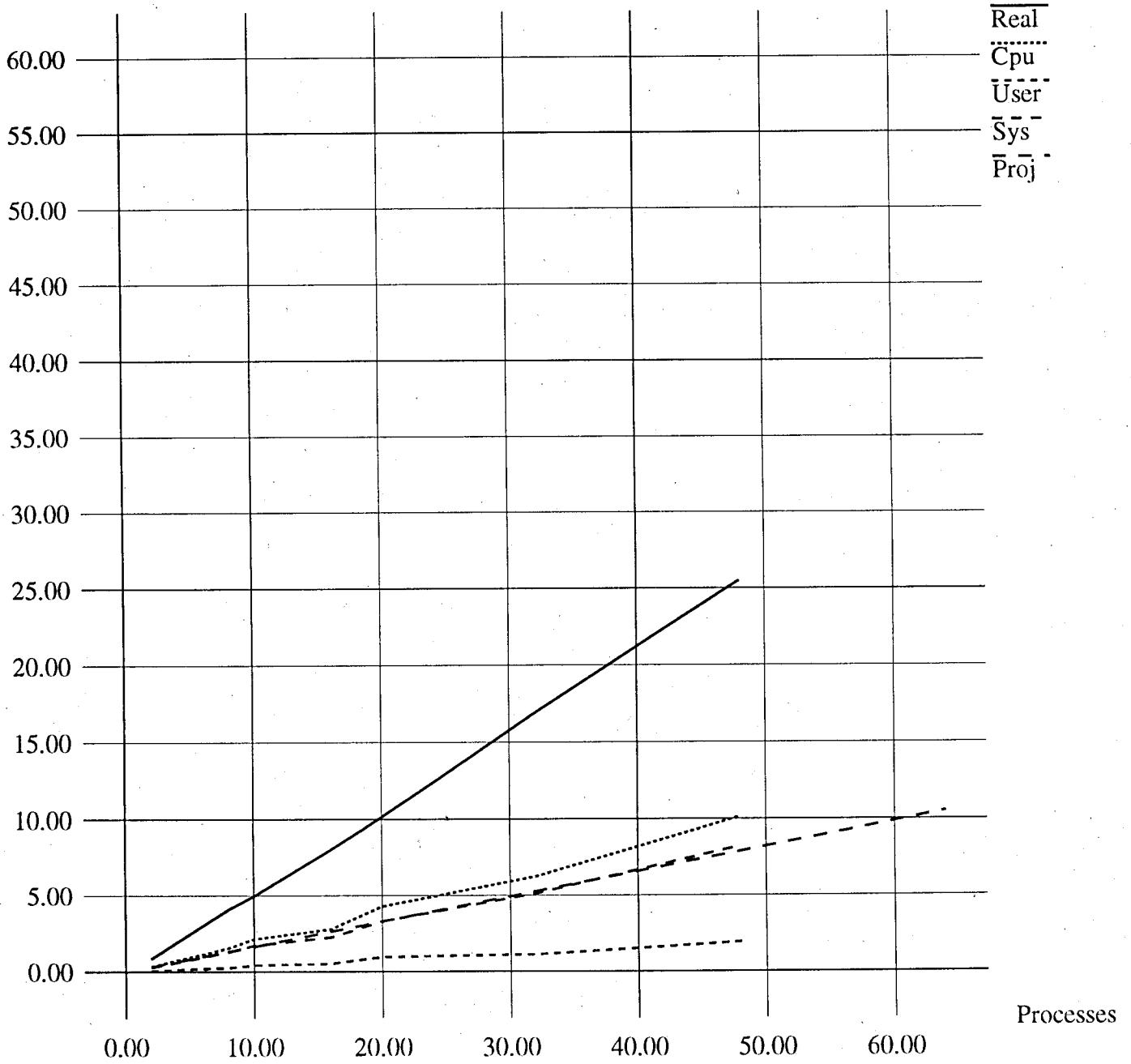
Solbourne 702/E (2), SunOS 4.1.1

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	849	328	48	281	328
4	1925	753	151	602	657
8	4044	1540	253	1288	1314
10	4940	2137	418	1720	1642
16	7995	2783	526	2257	2628
20	10141	4262	967	3296	3284
32	16883	6223	1118	5105	5255
48	25489	10144	1924	8221	7883

The real clock time reported by the dual processor Solbourne is remarkably higher than the total CPU clock time. Compare this to the dual processor Sun-4/630, in which the real clock time is *less* than the total CPU time (presumably because for each wall clock second there are two CPU seconds available). Assuming that the times reported by the Solbourne are accurate, there is no obvious explanation for this.

Solbourne 702/E (2), SunOS 4.1.1

Seconds x 10³

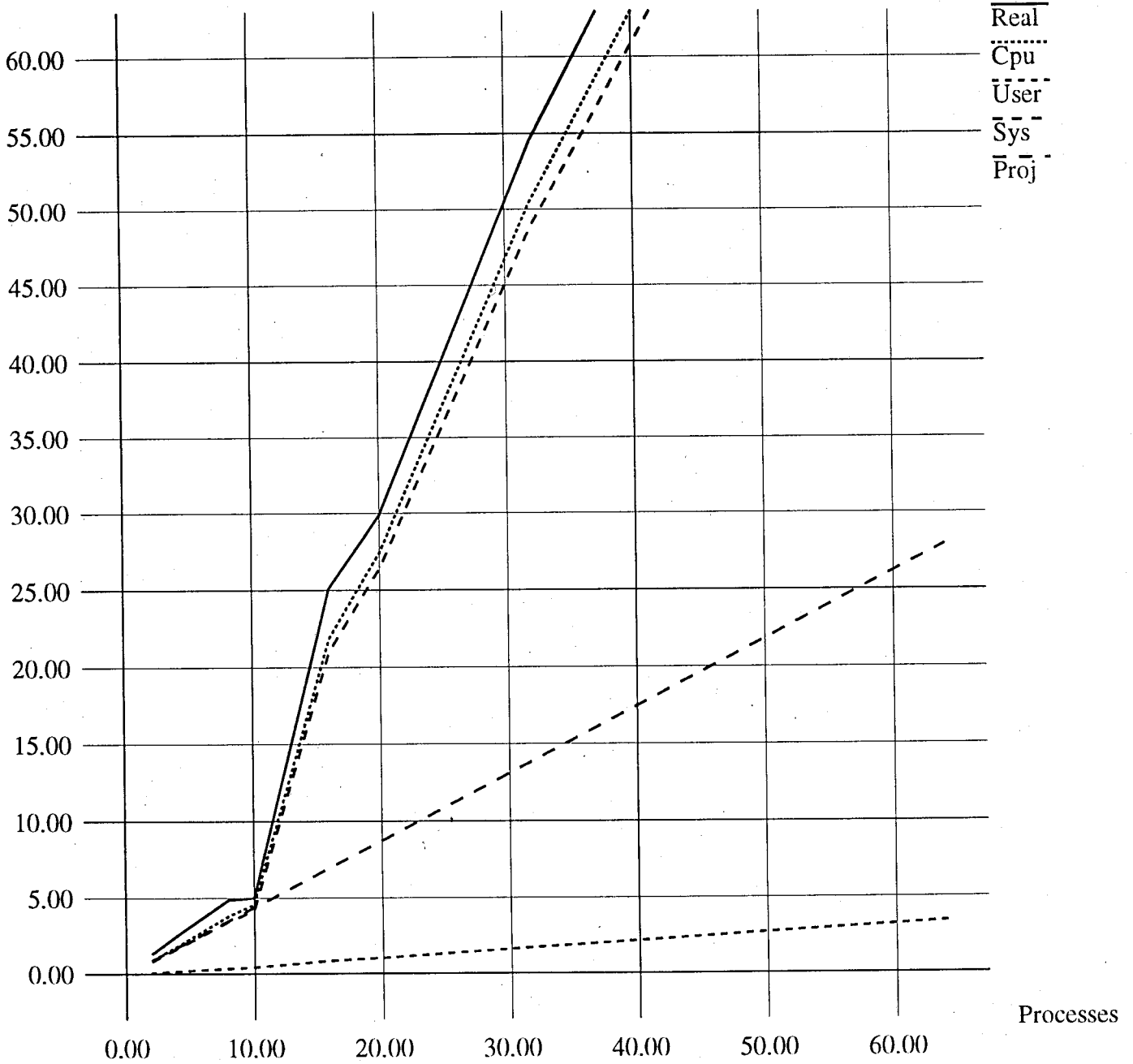


Sun Microsystems SUN-4/280, SunOS 4.1.1

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	1294	873	76	798	873
4	2532	1835	171	1665	1746
8	4859	3788	342	3447	3493
10	4985	4569	426	4143	4366
16	25097	21874	850	21025	6985
20	29898	27461	1031	26431	8732
32	54572	50551	1724	48828	13971
48	80003	75362	2589	72774	20956
64	101425	96545	3402	93143	27941

Sun Microsystems SUN-4/280, SunOS 4.1.1

Seconds x 10³



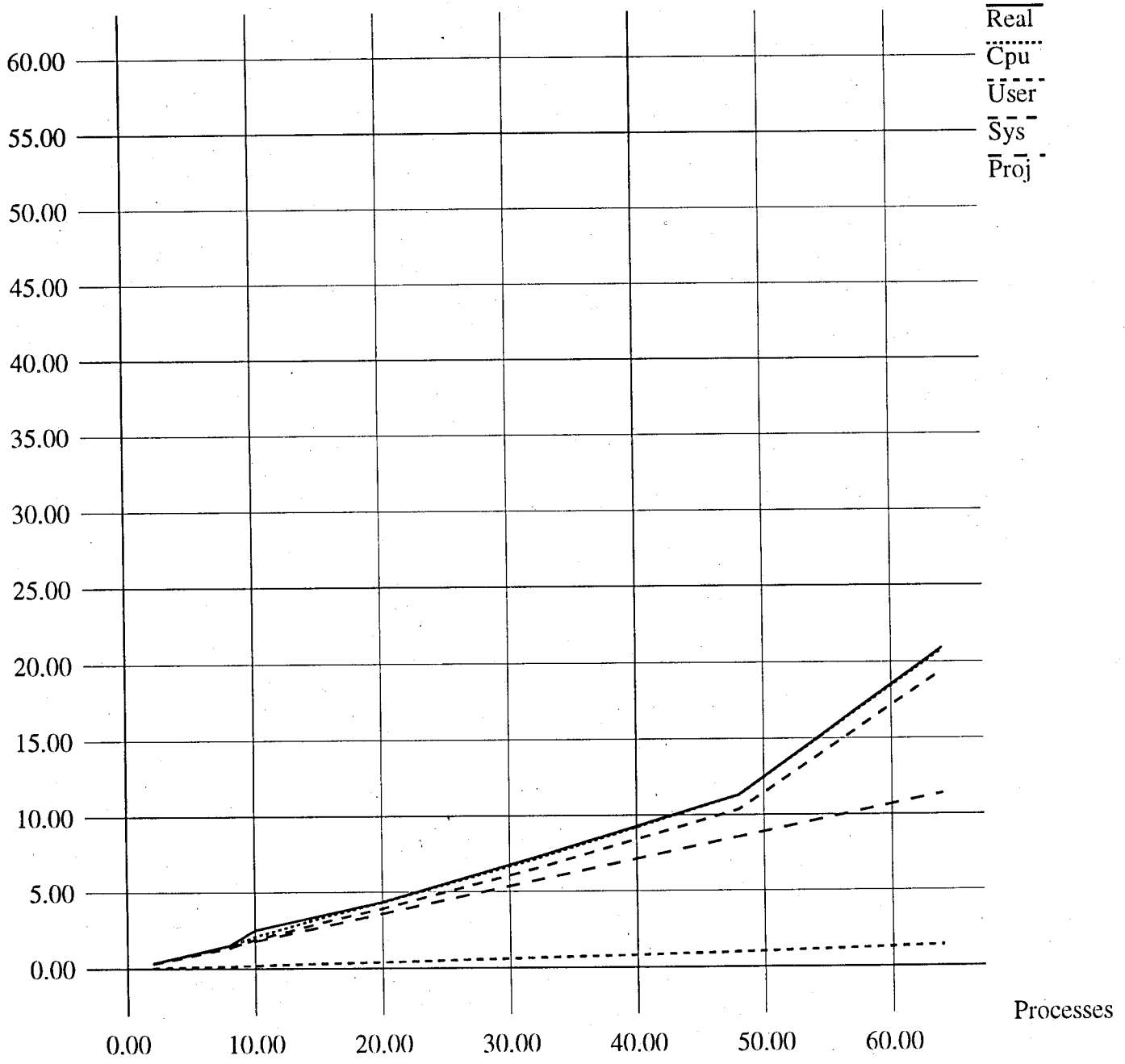
Sun Microsystems SUN-4/470, SunOS 4.1_PSR

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	358	356	32	325	356
4	734	728	83	646	712
8	1509	1489	150	1339	1424
10	2509	2088	200	1889	1780
16	3602	3475	324	3152	2848
20	4317	4284	395	3890	3560
32	7239	7115	637	6479	5695
48	11313	11290	955	10335	8543
64	20902	20792	1377	19415	11390

The "knees" in the Sun-4/280 and the Sun-4/470 performance curves graphically show the effects of the Sun-4 architecture on context switch overhead. The Sun-4/280 has sixteen high-speed register banks used to hold the contexts of runnable processes. Beyond that, process contexts must be moved in and out of main memory in the usual manner. The Sun-4/470 has thirty-two such banks. The Suns offer excellent benchmark performance up until the context banks are saturated, after which performance degrades, rapidly in the case of the 4/280, less so in the case of the 4/470. Note that the resolution of the data points on the graphs makes it difficult to determine exact position of the knees.

Sun Microsystems SUN-4/470, SunOS 4.1_PSR

Seconds x 10³



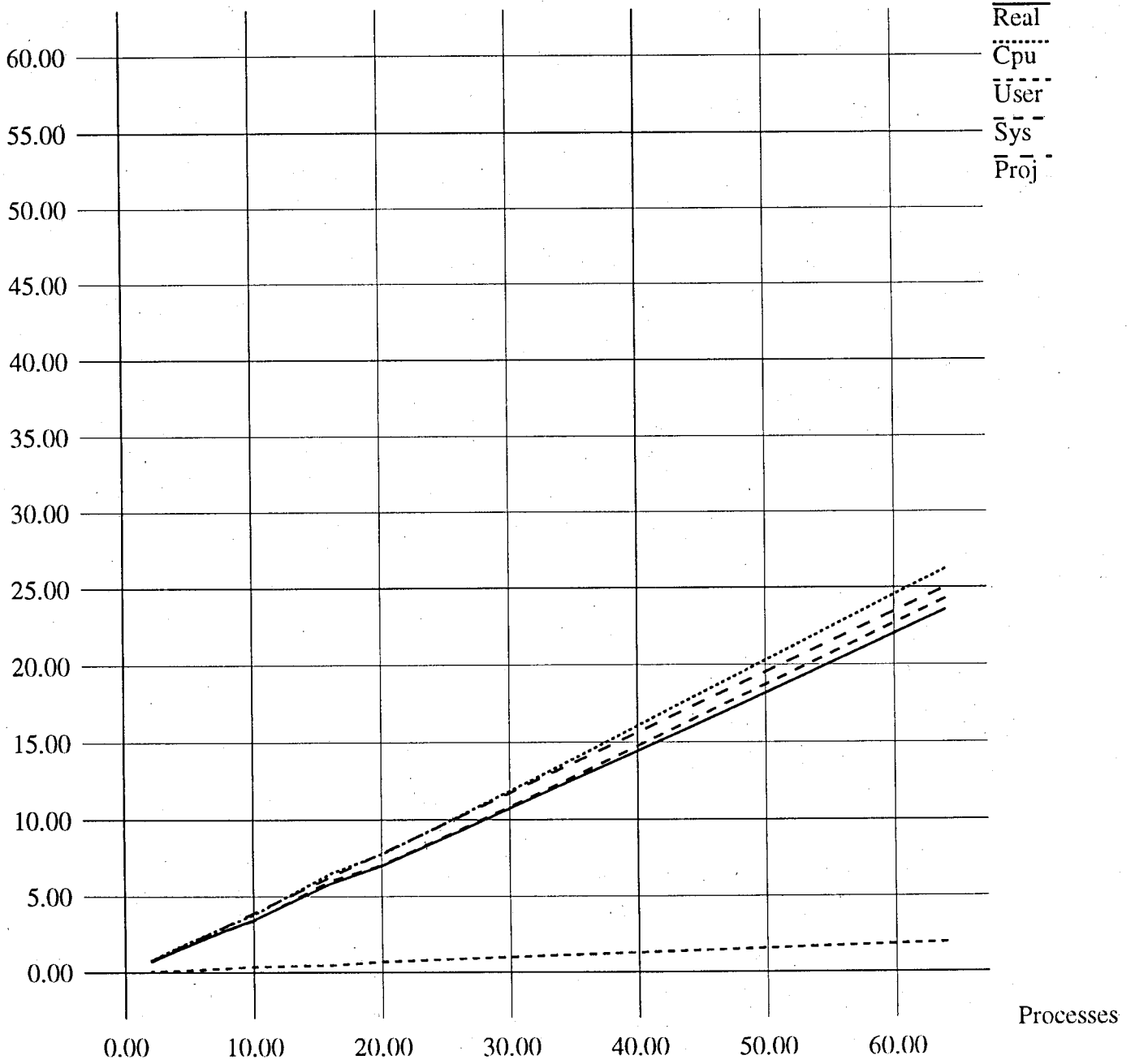
Sun Microsystems SUN-4/630 (2), SunOS 4.1.2

Processes	Real Seconds	CPU Seconds	User Seconds	System Seconds	Projected Seconds
2	714	781	67	714	781
4	1458	1640	117	1523	1562
8	2860	3107	282	2825	3124
10	3459	3792	374	3419	3905
16	5826	6485	473	6013	6248
20	6996	7751	703	7048	7811
32	11472	12662	1071	11591	12497
48	17393	19430	1521	17909	18745
64	23543	26222	1959	24264	24994

The Sun-4/630 shows linear scalability up to the limit of the benchmark. Note that the real wall clock time on the two processor Sun-4/630 is less than the reported CPU time, presumably because for every wall clock second there are two CPU seconds, even though in the *pipeline* benchmark, the only time in which two processes were active (during which two CPUs could be effectively utilized) was a narrow window in which the kernel was handling the pipeline I/O and the associated context switch between the old and new process.

Sun Microsystems SUN-4/630 (2), SunOS 4.1.2

Seconds x 10³



Appendix 2 Pipeline Source Code

```
/*
**      P I P E L I N E
**
**      Copyright 1991,1992 University Corporation for Atmospheric Research
**      All Rights Reserved
**
**      Title           Pipeline
**      Program        Pipeline
**      Project        Context Switch Benchmarking
**      Author         John Sloan
**      Email          jsloan@ncar.ucar.edu
**      Date           Tue Aug 20 09:51:11 MDT 1991
**      Organization   NCAR, P.O. Box 3000, Boulder CO 80307
**
**      Abstract
**
**      pipeline _chars_ _count_
**
**      This program sets up a circular chain of _count_ (where
**      _count_ is two or more) processes which pass _chars_ (where
**      _chars_ is one or more) characters around the circle through
**      pipes. This forces context switches to move sequentially
**      around the circle.
**
*/

static char copyright[]="Copyright 1991,1992 University Corporation for Atmospheri
c Research - All Rights Reserved";
static char sccsid[]="@(#)pipeline.c    1.4 92/07/10 jsloan@ncar.ucar.edu";

#include <stdio.h>
#include <signal.h>
#include <values.h>
#include <sys/wait.h>
#include <errno.h>

/*
**      Process: read a character from the in fd and write it to the
**      out fd count times or until EOF.
**
*/
static int
process(count,in,out)
long count;
int in;
int out;
{
    char ch;
```

```

    int rc, xit;

#ifdef DEBUG
    fprintf(stderr,"%d: count=%d in=%d out=%d\n",getpid(),count,in,out);
#endif DEBUG

#ifdef SIG_ERR
    if (signal(SIGPIPE,SIG_IGN)==SIG_ERR)
#else SIG_ERR
    if (((int)signal(SIGPIPE,SIG_IGN))==-1)
#endif SIG_ERR
    {
        perror("signal");
        return(11);
    }

    for (xit=0; count>0; count--)
    {

        if ((rc=read(in,&ch,1))<0)
        {
            xit=12;
            break;
        }
        else if (rc==0)
            break;

#ifdef DEBUG
        putchar(ch);
        (void)fflush(stdout);
        ch=(ch=='Z')?'A':(ch+1);
#endif DEBUG

        if (count>1)
            if ((rc=write(out,&ch,1))<0)
            {
                xit=13;
                break;
            }
            else if (rc==0)
            {
                xit=14;
                break;
            }

    }

    (void)close(in);
    (void)close(out);

    return(xit);
}

```

```

/*
**      Main: set up the filter chain, insert a character into the
**      pipeline circle to get things primed, participate in the
**      chain as one of the processes, and finally reap all the
**      children as they terminate.
*/
main(argc,argv)
int argc;
char **argv;
{
    long chars, count;
    pid_t pid;
    int loop[2], pipeline[2], in, out, rc, xit;
    char ch;

    if (argc!=3)
        fprintf(stderr,"usage: %s chars count\n",argv[0]), exit(1);
    chars=atol(argv[1]);
    if (chars<1)
        fprintf(stderr,"%s: chars %d<1\n",argv[0],chars), exit(2);
    count=atol(argv[2]);
    if (count<2)
        fprintf(stderr,"%s: count %d<2\n",argv[0],count), exit(3);

    if (pipe(loop)<0)
        perror("pipe"), exit(4);
    in=loop[0];
    for (count--; count>0; count--)
    {
        if (pipe(pipeline)<0)
            perror("pipe"), exit(5);
        out=pipeline[1];
        if ((pid=fork())<0)
            perror("fork"), exit(6);
        else if (pid==0)
        {
            (void)close(loop[1]);
            exit(process(MAXLONG,in,out));
        }
        (void)close(in);
        (void)close(out);
        in=pipeline[0];
    }
    out=loop[1];

    ch='A';
    if (write(out,&ch,1)<0)
        perror("write"), exit(6);

    xit=process(chars,in,out);

    do
        rc=wait((void *)0);

```

```
        while (rc>0);  
  
#ifdef DEBUG  
    putchar('\n');  
#endif DEBUG  
  
    exit(xit);  
}
```

Appendix 3 Getrusage Source Code

```
/*
**      G E T R U S A G E
**
**      Copyright 1991,1992 University Corporation for Atmospheric Research
**      All Rights Reserved
**
**      Title           Getrusage
**      Program        Getrusage
**      Project        Context Switch Benchmarking
**      Author         John Sloan
**      Email          jsloan@ncar.ucar.edu
**      Date           Tue Aug 20 10:00:39 MDT 1991
**      Organization   NCAR, P.O. Box 3000, Boulder CO 80307
**
**      Abstract
**
**      getrusage program argument [ argument ... ]
**
**      Run a program. When it is complete, display the contents of its
**      getrusage structure.
*/

static char copyright[]="Copyright 1991,1992 University Corporation for Atmospheri
c Research - All Rights Reserved";
static char sccsid[]="@(#)getrusage.c 1.2 92/02/13 jsloan@ncar.ucar.edu";

#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <errno.h>
#include <string.h>

#define MICROSECS      (1000000)

/*
**      Usec: return a character array containing the value of the
**      provided timeval structure in microseconds. This function is
**      necessary because it is required that we print in effect
**      a double long.
*/
static char *
usec(time)
struct timeval time;
{
```

```

static char buffer[(2*sizeof(long))+1];

if (time.tv_sec!=0)
{
(void)sprintf(buffer,"%ld",time.tv_sec);
if (time.tv_usec!=0)
(void)sprintf(buffer+strlen(buffer),"%6.6ld",
time.tv_usec);
else
(void)strcat(buffer,"000000");
}
else
{
if (time.tv_usec!=0)
(void)sprintf(buffer,"%ld",time.tv_usec);
else
(void)strcpy(buffer,"0");
}
return(buffer);
}

/*
** Total: return a character string containing the Usec value
** of the sum of two timeval structures. This function is
** necessary since we must effectively add two double longs.
*/
static char *
total(system,user)
struct timeval system;
struct timeval user;
{
struct timeval sum;

sum.tv_usec=system.tv_usec+user.tv_usec;
sum.tv_sec=system.tv_sec+user.tv_sec;
while (sum.tv_usec>MICROSECS)
{
sum.tv_usec-=MICROSECS;
sum.tv_sec++;
}
return(usec(sum));
}

/*
** Delta: return a character string containing the Usec value
** of the difference between two timeval structures. This
** function is necessary since we must effectively subtract
** two double longs.
*/
static char *
delta(before,after)
struct timeval before;
struct timeval after;

```

```

{
struct timeval diff;

diff.tv_sec=after.tv_sec-before.tv_sec;
diff.tv_usec=after.tv_usec-before.tv_usec;
while (diff.tv_usec<0)
    {
    diff.tv_sec--;
    diff.tv_usec+=MICROSECS;
    }
return(usec(diff));
}

/*
**      Main: run the command and display its resource usage.
*/
void
main(argc,argv,envp)
int argc;
char **argv;
char **envp;
    {
    pid_t pid, npid;
    struct rusage usage;
    struct timeval before, after;
    struct timezone zone;
    struct tm *toc;
    int status, typ, xit, sig;

    if (argc<2)
        exit(127);

    if (gettimeofday(&before,&zone)<0)
        {
        perror("gettimeofday");
        exit(127);
        }

    if ((pid=fork())<0)
        {
        perror("fork");
        exit(127);
        }
    else if (pid==0)
        if (execve(argv[1],&argv[1],envp)<0)
            {
            perror("execve");
            exit(127);
            }
    fprintf(stderr,"%d: start      %s",pid,ctime(&before.tv_sec));

    do
        if ((npid=wait3(&status,0,&usage))<0)

```

```

        {
            perror("wait3");
            exit(127);
        }
while (npid!=pid);

if (gettimeofday(&after,&zone)<0)
    {
        perror("gettimeofday");
        exit(127);
    }
fprintf(stderr,"%d: stop          %s",pid,ctime(&after.tv_sec));

typ=status&0xff;
xit=(status>>8)&0xff;
sig=status&0x7f;

fprintf(stderr,"%d: %s %d\n",pid,typ?"signal":"exit ",typ?sig:xit);
fprintf(stderr,"%d: rtime          %susec\n",pid,delta(before,after));
fprintf(stderr,"%d: utime          %susec\n",pid,usec(usage.ru_utime));
fprintf(stderr,"%d: stime          %susec\n",pid,usec(usage.ru_stime));
fprintf(stderr,"%d: total          %susec\n",pid,total(usage.ru_stime,usage.
ru_utime));
fprintf(stderr,"%d: maxrss          %ld\n",pid,usage.ru_maxrss);
fprintf(stderr,"%d: ixrss          %ld\n",pid,usage.ru_ixrss);
fprintf(stderr,"%d: idrss          %ld\n",pid,usage.ru_idrss);
fprintf(stderr,"%d: isrss          %ld\n",pid,usage.ru_isrss);
fprintf(stderr,"%d: minflt          %ld\n",pid,usage.ru_minflt);
fprintf(stderr,"%d: majflt          %ld\n",pid,usage.ru_majflt);
fprintf(stderr,"%d: nswap          %ld\n",pid,usage.ru_nswap);
fprintf(stderr,"%d: inblock          %ld\n",pid,usage.ru_inblock);
fprintf(stderr,"%d: oublock          %ld\n",pid,usage.ru_oublock);
fprintf(stderr,"%d: msgsnd          %ld\n",pid,usage.ru_msgsnd);
fprintf(stderr,"%d: msgrcv          %ld\n",pid,usage.ru_msgrcv);
fprintf(stderr,"%d: nsignals          %ld\n",pid,usage.ru_nsignals);
fprintf(stderr,"%d: nvcsw          %ld\n",pid,usage.ru_nvcsw);
fprintf(stderr,"%d: nivcsw          %ld\n",pid,usage.ru_nivcsw);

exit(typ?127:xit);
}

```

Appendix 4 Times Source Code

```
/*
**      T I M E S
**
**      Copyright 1991,1992 University Corporation for Atmospheric Research
**              All Rights Reserved
**
**      Title           Times
**      Program        Times
**      Project         Context Switch Benchmarking
**      Author          John Sloan
**      Email           jsloan@ncar.ucar.edu
**      Date            Fri Aug 23 09:19:54 MDT 1991
**      Organization    NCAR, P.O. Box 3000, Boulder CO 80307
**
**      Abstract
**
**      This program does the equivalent of getrusage for those
**      systems which lack the getrusage(2) system call but
**      provide times(2) instead. The command parameter is executed
**      with the provided arguments and the resulting system
**      resources expended are printed.
*/

static char copyright[]="Copyright 1991,1992 University Corporation for Atmospheri
c Research - All Rights Reserved";
static char sccsid[]="@(#)times.c          1.5 92/07/16 jsloan@ncar.ucar.edu";

#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <errno.h>
#include <string.h>
#include <sys/times.h>

#ifdef CLK_TCK
#define CLK_TCK          (100)
static int clk_tck[]="Assuming 100 ticks/second";
#endif

/*
**      Usec: print a microsecond magnitude as a double long.
*/
static char *
```

```

usec(time)
struct timeval time;
{
    static char buffer[(2*sizeof(long))+1];

    if (time.tv_sec!=0)
    {
        (void)sprintf(buffer,"%ld",time.tv_sec);
        if (time.tv_usec!=0)
            (void)sprintf(buffer+strlen(buffer),"%6.6ld",
                time.tv_usec);
        else
            (void)strcat(buffer,"000000");
    }
    else
    {
        if (time.tv_usec!=0)
            (void)sprintf(buffer,"%ld",time.tv_usec);
        else
            (void)strcpy(buffer,"0");
    }
    return(buffer);
}

/*
**      Delta: print the difference of two timevals as a double long
**      magnitude.
**/
static char *
delta(before,after)
struct timeval before;
struct timeval after;
{
    struct timeval diff;

    diff.tv_sec=after.tv_sec-before.tv_sec;
    diff.tv_usec=after.tv_usec-before.tv_usec;
    if (diff.tv_usec<0)
    {
        diff.tv_sec--;
        diff.tv_usec+=1000000;
    }
    return(usec(diff));
}

/*
**      tic2usec: convert tics to useconds.
**/
static double
tic2usec(tics)
long tics;
{
    double tusec, ftics;

```

```

ftics=tics;
tusec=CLK_TCK;
tusec=tusec/1000000.0;
return(ftics/tusec);
}

/*
** Main: run the command, wait for it to complete, display
** it's resource consumption.
*/
void
main(argc,argv,envp)
int argc;
char **argv;
char **envp;
{
    pid_t pid, npid;
    struct timeval before, after;
    struct timezone zone;
    struct tms cputime;
    int status, typ, xit, sig;
    clock_t since;

    if (argc<2)
        exit(127);

    if (gettimeofday(&before,&zone)<0)
    {
        perror("gettimeofday");
        exit(127);
    }

    if ((pid=fork())<0)
    {
        perror("fork");
        exit(127);
    }
    else if (pid==0)
        if (execve(argv[1],&argv[1],envp)<0)
        {
            perror("execve");
            exit(127);
        }
    fprintf(stderr,"%d: start      %s",pid,ctime(&before.tv_sec));

    do
        if ((npid=wait(&status))<0)
        {
            perror("wait");
            exit(127);
        }
    while (npid!=pid);
}

```

```

if (gettimeofday(&after,&zone)<0)
{
    perror("gettimeofday");
    exit(127);
}
fprintf(stderr,"%d: stop          %s",pid,ctime(&after.tv_sec));

if ((since=times(&cputime))<0)
{
    perror("times");
    exit(127);
}

typ=status&0xff;
xit=(status>>8)&0xff;
sig=status&0x7f;

fprintf(stderr,"%d: %s %d\n",pid,typ?"signal":"exit ",typ?sig:xit);
fprintf(stderr,"%d: rtime      %susec\n",pid,delta(before,after));
fprintf(stderr,"%d: tics/sec   %ld\n",pid,CLK_TCK);
fprintf(stderr,"%d: utime      %lfusec (%ldtics)\n",
        pid,tic2usec(cputime.tms_cutime),cputime.tms_cutime);
fprintf(stderr,"%d: stime      %lfusec (%ldtics)\n",
        pid,tic2usec(cputime.tms_cstime),cputime.tms_cstime);
fprintf(stderr,"%d: total      %lfusec (%ldtics)\n",
        pid,tic2usec(cputime.tms_cstime+cputime.tms_cutime),
        cputime.tms_cstime+cputime.tms_cutime);
fprintf(stderr,"%d: since      %lfusec (%ldtics)\n",
        pid,tic2usec(since),since);

exit(typ?127:xit);
}

```