Bolt Beranek and Newman Inc.

(12)

Report No. 5285



# Development of a Voice Funnel System

Quarterly Technical Report No. 18
1 November 1982—31 January 1983



April 1983

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>Development of a Voice Funnel System<br>Quarterly Technical Report No. 18 | | 5. TYPE OF REPORT & PERIOD COVERED<br>Quarterly Technical<br>1 Nov. 82 - 31 Jan. 1983<br>6. PERFORMING ORG. REPORT NUMBER<br>5285 |
| 7. AUTHOR(s)<br>R. D. Rettberg | | 8. CONTRACT OR GRANT NUMBER(s)<br>MDA903-78-C-0356 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Bolt Beranek and Newman Inc.<br>10 Moulton Street<br>Cambridge, MA 02238 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>ARPA Order No. 3653 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>DARPA<br>1400 Wilson Boulevard<br>Arlington, VA 22209 | | 12. REPORT DATE<br>April 1983<br>13. NUMBER OF PAGES<br>36 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited



17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 5285 Bolt Beranek and Newman Inc.

# DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 18
1 November 1982 to 31 January 1983

April 1983

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 30 April 1983
Principal investigator: R. D. Rettberg



Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

# TABLES

## 1. Introduction

This Quarterly Technical Report, Number 18, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 November 1982 to 31 January 1983. This is the eighteenth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

The process scheduler is a central part of the Chrysalis operating system. In the course of the Butterfly software development effort, the scheduling algorithm and its implementation have undergone several stages of testing and refinement. This report describes the algorithm in its final form. The discussion assumes some familiarity with the operation of the scheduler for some other operating system, such as UNIX, TOPS-20, or OS. In keeping with the Chrysalis convention that resources are managed locally whenever it is practical to do so, there is a separate instance of the process scheduler on every Processor Node. Each instance of the scheduler manages the processes that are local to that node.

The first section of this report reviews certain concepts and terminology that are essential to a clear understanding of the scheduler. In the next section, the current implementation is described in detail. This is followed by a section that presents performance measurement data. The final section makes some generalizations about the structure of the scheduler, and

briefly discusses the problem of Global Scheduling.

The experimental data presented in section four was gathered and analyzed by the BSAT project as part of their effort to understand the behavior of the Butterfly system and the BSAT application. Their results are presented in this report because they give interesting insights into the behavior of the Chrysalis scheduler.

## 2. Related Concepts

This section reviews the concepts on which the Chrysalis scheduler is based. Occasional references are made to 'Chrysalis functions.' These subroutines are available to the application software from the Chrysalis Protected Library. They are described in detail in the Chrysalis Operating System Manual, which is currently available in draft form.

### 2.1 Processes

Chrysalis programs are composed of processes. Each process is linked independently and has its own address space which can be manipulated independently of other processes. Processes are created in two stages: loading of the code and initialized data segments for a given process type, and creation of a specific instance of the process. The Chrysalis function Make_Template loads the code and data segments of a process into main memory on a specific Processor Node. When this operation is complete, Make_Template returns the Object Handle of a data structure called a process template. The process template includes various process parameters along with Object Handles for the code and data segments.

Once a process template has been loaded, the Chrysalis function Make_Process can be used to create one or more instances of the process. Make_Process allocates the necessary resources

on whatever processor the caller designates, starts the process up, and returns the Object Handle of the process that it has created.

## 2.2 Priorities

Under Chrysalis, every process runs at one of four priority levels.

Priority zero, the lowest available priority, is intended for processes which are in no hurry to operate and which do not mind being preempted by ordinary processes.

Priority one is intended for the majority of ordinary processes, including most processes that service I/O devices.

Priority two is intended for processes which must preempt ordinary processes in order to operate correctly. An example might be a disk controller process. The intent is that by providing enough buffering, all ordinary telecommunications I/O processes will be able to run at level one.

Priority three is reserved for vital, crash-priority processes such as debuggers and certain operating system functions.

Process priority is initially established during Make_Process. Each of the four priorities has an associated

queue on which runnable processes are placed by the post microcode. In addition to these four 'normal' priority queues, there is another set of four 'low' priority queues (priorities -4 through -1); If a process is using more than its fair share of time, the scheduler may temporarily move it down exactly four notches to the corresponding low-priority queue.

Every process is always assigned to a particular priority queue, either its normal-queue or its low-queue. However, it is only enqueued on its current priority queue if it is ready to run.

## 2.3 Events

Process scheduling is based on the concept of an 'event.' To illustrate the behavior of events under Chrysalis, it is useful to consider the example of a process that has requested some service from another process and needs to know when the server process has finished. Before the first process makes its request, it uses the Chrysalis function Make_Event to allocate and initialize an Event Block. Make_Event returns an Event Handle (the Object Handle of the Event Block) which the requesting process supplies along with parameters describing its request. When the operation is completed, the server process 'posts' the event, using the Chrysalis function Post_Event. In the meantime, the requesting process may continue to run, or it

may choose to wait for the event by declaring itself non-runnable and asking the scheduler to wake it up when the event is posted. If the process continues to run, it may establish other events and test outstanding events for completion. If it decides to wait, it may wait for any outstanding event or for only one of a specific set of events.

Although a process may wait only for events that it owns, it may post any event handle, including handles for events on other Processor Nodes. The automatic posting of events by the operating system is triggered by such things as timer completion and dual queue data availability. Also, the high speed synchronous I/O interface currently supported by the Butterfly is designed to post events at packet or buffer boundaries rather than to interrupt on a character-by-character basis.

The operations invoked by calling Post_Event are implemented primarily in microcode and are tightly coupled to the process scheduler. When an unposted event is first posted, it is added to a queue of posted Event Blocks associated with the process which owns it. If that process is waiting, the event posting microcode puts the process at the end of the appropriate scheduler queue, depending on its current priority. If the posted process is of higher priority than the process that is currently running, the microcode activates the process scheduler.

If the event is owned by a process on a remote Processor Node, the local Processor Node Controller sends a special switch message to the remote node. On receipt of this message, the remote microcode executes the post. Unless an error of some sort occurs, this interaction is transparent to the application level. The high-speed I/O system is restricted to posting local events. This is consistent with the general restriction that an I/O device may only interact directly with application software on the local node.

## 2.4 Epoch Scheduling

In real-time systems, as opposed to ordinary time-sharing or batch systems, it may be vital that a process get a certain minimum amount of CPU time every little while. Otherwise the job it is trying to perform may be neglected to such an extent that the system as whole may fail gracelessly.

The technique of epoch scheduling attacks this problem by defining (as a system parameter) a scheduling period called an epoch. Currently, the Voice Funnel uses an epoch of 100 milliseconds. Within this epoch, CPU time is allocated to processes based in part on a declared need. For example, when a process is started by Make_Process, a need for 20 milliseconds out of each epoch may be declared. Assuming that this process is ready to run at the beginning of the epoch and remains ready

throughout the epoch, the scheduler guarantees that the process will be allowed to run for at least 20 milliseconds.

The sum of the declared needs of all processes on a given Processor Node is called committed time. Obviously, the scheduler cannot satisfy the needs of every process unless the amount of committed time is less than the total time available in an epoch. Time which is not promised is called uncommitted time. Committed time which is not used is called windfall. The Chrysalis scheduler is designed so that any uncommitted or windfall time is available to any process which needs it on a first-come-first-served basis (with preference given to higher priority processes).

Processes using uncommitted time are run at normal priority, while those using windfall time are run at low priority. In order to make its scheduling decisions, the operating system maintains a record of how much committed time is left in the current epoch and how much time has been allocated to each process so far. At the end of each epoch, a system process called the Epoch Scheduler restores these variables to their initial state and moves each process to its normal-priority queue. The use of this information is discussed further in Section 3.

## 2.5 Time-Slicing

In real-time systems, most processes normally run in relatively brief bursts, then wait for some event to trigger further activity. An epoch scheduler is well suited to this sort of behavior. Occasionally, some processes will want to run for extended periods, either due to some unusual occurrence or due to a greater-than-normal influx of data or service requests. In other applications, some processes will want to run continuously for relatively long periods.

To help with these cases, the scheduler provides each process with a time-slicing parameter. This parameter limits the time that a process can run without giving up control to other processes of the same priority. When the time slice of a running process ends, another process of equal priority will be scheduled on a round-robin basis. If this parameter is set too large, other processes may have to wait a long time for service and uncommitted and/or windfall time may not be fairly shared; if it is set too small, excessive scheduling overhead may be incurred.

## 2.6 Process State

The state of a process includes its 68000 registers (including the status register), the state of its address space, and a kernel mode flag. Saving and restoring this state is the most expensive part of the scheduler. The scheduler is organized

to save the state of the currently running process only when it switches processes or goes into the idle state (when no process is runnable). The state of a process is restored only when a process switch occurs.

## 2.7 Interrupt Routines

The current Butterfly hardware uses conventional interrupts only for terminal character handling and real-time clock event signalling. The time used by interrupt routines is charged to whatever process is active at the time of the interrupt; although this policy is unfair to the active process, the time involved should be negligible.

The scheduler itself operates at interrupt level one. Since this is the lowest-priority 68000 interrupt, the state saved at the time of interrupt is that of the running application program, not the state of some other interrupt routine.

## 2.8 Timer Interrupts

The Processor Node Controller maintains an interval timer that is used to support the time-slicing functions of the scheduler. This timer is also used by the Chrysalis timer demon, which is a service that allows processes to request that wakeup events be posted at arbitrary times in the future. When an

interval timer interrupt occurs, the level two interrupt service routine first posts any wakeup events that have come due, and then checks to see if the currently running process has reached the end of its time slice. If it has, the interrupt service routine requests a level one interrupt, causing the scheduler to run.

The exact details of how the interval timer works and how the timer demon provides its service are not important here. What is important is the idea that the interval timer interrupt service routine performs two functions. As a result, a context switch at the end of a time-slice takes about 100 microseconds, which is somewhat longer than one might first expect. The alternative to this structure would involve maintaining a second interval timer. We did not do this for several reasons. First, maintaining an interval timer is expensive in terms of Processor Node Controller bandwidth and resources. Second, it is expected that in the Voice Funnel application at least, properly tuned processes will normally dismiss voluntarily before their time slices end, so the impact of this extra overhead should be small in most cases.

## 3. The Scheduler Algorithm

This section describes the detailed operation of the scheduler. In addition to switching between runnable processes

at appropriate times, it is up to the scheduler to ensure that each process gets its share of runtime, as dictated by its need and time-slice parameters.

## 3.1 Process Switching

The scheduler is entered only when some occurrence triggers a level one interrupt. If the machine is idle, posting any event will trigger the scheduler. Otherwise, posting an event owned by a process of higher priority than the currently running process will trigger the scheduler and preempt the current process. If the current time slice is exceeded, the timer interrupt routine will trigger the scheduler. Finally, if the process calls the Chrysalis functions Wait or MWait (directly or indirectly) and an appropriate event is not yet available, the process will be marked not-runnable and the scheduler will be triggered.

If the machine was running a process when the level one interrupt occurred, the scheduler accounts for the time just used by the process, allocates a new time slice for the process (if necessary), removes the process from the front of its current queue, and if the process is still runnable, places it at the end of the same or the corresponding low-priority queue. Next, the priority queues are scanned to select the highest priority runnable process; since the queues contain only processes which are runnable, the first process on the first non-empty queue is

selected. Finally, the scheduler decides whether it needs to save or restore any state information. There are three possibilities: no process is ready to run, some new process is selected, or the same process as was previously running is selected. In the first case, the state of the current process is saved and the machine goes idle. In the second case, the scheduler saves the state of the current process, restores the state of the new process and debreaks to the new process. In the third case, the scheduler simply debreaks back to the old process.

If the machine was idle when the level one interrupt occurred, the scheduler immediately scans for the highest priority runnable process. If none is found, the machine goes idle again. If the most recently active process has become runnable again, its state is still in the registers of the 68000 and the scheduler can debreak immediately to that process without restoring any state information. Otherwise the scheduler restores the state of a new process and debreaks to it.

Under the current scheduler implementation, the level one interrupt is intercepted by the Processor Node Controller (PNC). The PNC makes all of the necessary scheduling decisions, and presents one of six interrupt vectors to the 68000, depending on what kind of context switch is required. The level one interrupt routine performs the appropriate saving and restoring of process state (if needed), updates a small set of scheduler statistics,

and returns. In order to allow the testing of new variations on the scheduling algorithm without rewriting the PNC microcode, there is a software switch that inhibits the scheduler microcode and allows the scheduler to run entirely on the 68000 as the level-one interrupt routine.

## 3.2 Time Slices and Priorities

Whenever the scheduler starts a new process, it sets a timer. If the process is still running at the end of its allocated time slice, the timer will fire, causing the scheduler to be invoked. This mechanism enforces the time-slicing discipline introduced in Section 2. To help it decide how long the timer interval should be, the scheduler maintains a "time to end of current slice" (TEOCS) variable for every process. Since the time-slice parameter of a process is generally smaller than its declared need, the value of TEOCS for a given process will usually go to zero at least once before the need of that process has been satisfied. Thus, the scheduler may occasionally restore TEOCS to its initial value. This section describes the criteria used to decide when to restore TEOCS.

If a process has more than a few hundred microseconds left in its current slice, and it is runnable, we assume that it is being preempted. We therefore reduce its current slice by the time it has just used, and leave it at the head of its queue. In

effect, this allows a higher priority process to interrupt lower priority processes without disrupting the lower priority processes, in much the same way as an ordinary interrupt routine.

In those cases where the process has less than a few hundred microseconds left in its current slice (indicating that it has exhausted its slice), or the process has become non-runnable and is waiting for an event, we dequeue the process from its current queue, account for the time used so far, and allocate a new time slice. If we cannot allocate enough time to run the process on its normal queue without encroaching on time committed to some other process, we switch the process to use its low priority queue. Note that if the process is not runnable, and it does not wake up again during the current epoch, the allocation of a new time slice will turn out to be superfluous. However, if the post microcode does try to make this process runnable again, it must know which priority queue to use. In any event, the time that it takes to do the allocation is negligible.

If the process is still marked runnable (or has been posted recently) we return it to the end of the appropriate queue. Otherwise, we mark it off-queue so that the post microcode will put it on the appropriate queue later.

## 3.3 Allocating Time Slices

In addition to deciding whether it should renew TEOCS, the scheduler must decide what value to assign when it does the renewal. Related to this is the requirement that the scheduler lower the priority of a process whose declared need has been satisfied within the current epoch. A somewhat complicated set of criteria for making these decisions is required in order to make the behavior of the scheduler reasonable under all combinations of circumstances.

If a process has been assigned to low-queue, it is always allocated its full time slice when TEOCS is renewed. (Of course, a low-queue process can be preempted by any normal-queue process.) At the end of each epoch, all processes return to normal-queue and have their time-needed intervals renewed. No process goes to low-queue until all uncommitted time has either elapsed or been allocated to specific processes.

The principal complication in this scheduler is the method by which uncommitted and windfall time is determined and allocated. At the beginning of each epoch the total declared need for all processes is determined. This time interval is subtracted from the time at which the epoch will end to determine when the 'critical scheduling period' (CSP) will begin; the period up to this time is called uncommitted time -- even if it is wasted, we can still theoretically satisfy the declared need

of every process before the end of the epoch.

If a process with a declared need runs before the critical scheduling period begins, the size of the CSP is reduced. Therefore, whenever a normal-queue process is rescheduled, the time it has used so far is added to the time at which the CSP begins, and subtracted from TEOCS.

Before a new time slice is allocated for a process assigned to normal-queue, the amount of available uncommitted time is calculated by subtracting the current time from the time at which the CSP begins. The amount of time available for the next slice is then calculated by adding the remaining need of the process to the amount of available uncommitted time. Finally a new value for TEOCS is obtained by taking the minimum of the calculated value and the requested time slice for this process. If the calculation yields a slice that is too small to be useful, we drop the process to the corresponding low-priority queue and give it the full slice requested. Otherwise the process continues at normal priority. If we have allocated some uncommitted time to the process, we subtract that time from the time at which the CSP begins. This keeps us from allocating the same time to some other process later.

When a time slice ends for a process assigned to normal-queue, we reduce its time-needed variable by the time it has just used, and add that time interval to the time at which the CSP

begins. If the allocated time slice does not match the time used, any allocated but unused time is also added to CSP, while any time used in excess of the allocated time and the total time-needed is subtracted from CSP.

### 3.4 Windfall

Once we are past the beginning of the CSP, any process whose time slice ends will be placed on low queue if its need for the current epoch has been satisfied. When a process on low queue is allowed to run, it is making use of windfall time. If any process has used windfall time, the scheduler cannot guarantee that all processes with unsatisfied need can be satisfied during the current epoch. Looking at it another way, if a process is not ready to claim its stated need until after the beginning of the CSP, the scheduler will give time away to other processes rather than letting it go to waste. Note that if an unsatisfied process becomes ready to run during the CSP, it will preempt any processes using windfall time. Note also that a process which comes ready just at the end of the epoch will make good use of its time in the next epoch. What's past is lost.

There is another, less desirable, mechanism by which a process may get more than its share of normal-queue time when others may fail to get their declared need. This can happen if one process runs during uncommitted time and gets its entire need

before the beginning of the CSP. After the need of such a process has been satisfied, sufficient uncommitted time may remain to allocate another time slice at normal priority. If the process waits a while and then uses this time near the end of the epoch, it may block other normal priority processes which are ready to run at the same (or lower) priority, and which have not been run enough to satisfy their declared needs.

Although it is not the best possible outcome, this situation is by no means fatal, as the unsatisfied processes will get their time in the next epoch, if they are ready soon enough. Note that this condition will occur only if the unsatisfied processes are not ready to run until after the start of the CSP.

It may seem unreasonable to allocate, and thus commit, uncommitted time to a process which has used up its declared need early in the epoch. However, if the process is runnable, it must have a time slice allocated to it, and it is the policy of the scheduler to demote processes to low queue only during the CSP. Furthermore, we expect that if a process has been busy enough to consume its total need at the beginning of the epoch, the extra time allocated is likely to be put to good use.

## 4. Scheduler Performance Measurement

Given the central role of the scheduler, it is important that its operation be both efficient and understandable. To

achieve maximum performance, much of the scheduler and many related functions are implemented in microcode. This raises the additional concern that the scheduler be coded correctly, because microcode PROM updates are expensive. We have therefore run a fairly extensive set of tests in order to characterize the performance of the scheduler under various sets of circumstances. The results of these experiments are presented in this section.

## 4.1 Measurement Tools

In order to observe the behavior of the scheduler and the application code, we installed a certain amount of instrumentation: the Process Control Block (PCB) was extended to make room for a set of per-process counters; the supervisor data segment (Segment F8) was changed to make room for a set of per-processor-node counters; and a small amount of code was added to the scheduler to maintain these counts. Since the impact of this instrumentation on performance appears to be minimal, we have left it in place to support future experiments.

The per-node counters include one counter for each of the six scheduler interrupt paths so that we can determine how often each path is taken. In the special path where the scheduler switches from one running process to a another, we installed an additional counter that records how often the switch is caused by a high queue process interrupting a low queue process. All of

these counts are cumulative from the time Chrysalis first begins running.

In the Process Control Block definition, there was already a record of the total run time of each process being maintained by the Epoch Scheduler. We added to this a count of the number of times the process has been scheduled to run. Both of these counts are cumulative from the time that the process was created.

A collection of software procedures was written by the Butterfly Satellite IMP project, part of the Wideband Satellite Network contract, to sample all of the counters in the system, and present a readable summary of their contents. The numbers are scaled to convenient units, such as milliseconds of run time per second of real time, or events per second, before being displayed. The procedures also allow the sample interval to be varied easily so that the experimenter can see both short term and longer term behavior of the system.

## 4.2 Method of Measurement

The Butterfly Satellite IMP program (BSAT) contains a Message Generator, two types of Message Sink, and a Delivery (routing) process which can be easily controlled and moved from processor to processor. The operation of these processes was well understood and so could be modelled easily. The BSAT monitoring and display process was run in a separate processor

from the processes being measured so that it would not interfere with the measurements.

The principal method of analysis was one of making incremental changes to the system under test and observing the resulting changes in its behavior. The changes consisted of varying the frequency and type of scheduling performed and the amount of uncommitted time in the epoch. The behavior consisted of the number of packets sent through the system (a measure of useful work done), the percentage of the CPU used by each process, and the number and type of schedulings that occurred.

A process was modelled as having two components: time spent executing the application task, and time spent doing system overhead functions. Timer interrupts, for example, are charged to the then-running process as far as scheduling is concerned, though the interrupted process gets no useful work done during the interrupt. Similarly, time spent saving and restoring the process state is charged to a process, even though the application process is not progressing during that time. Observing changes in the number of packets transmitted was used as a measure of time spent executing the application task. Changes in the charged run time of a process, coupled with knowledge of time spent executing the application work, may be used to measure time spent in a system overhead operation.

The foregoing may be summed up in the following equation:

$$t = M * B + n * s \qquad (1)$$

where:

t = total run time needed (msec. CPU time/1 sec. clock time)

M = messages/second flowing through the system (messages/sec)

B = time needed to process one message (msec./message)

n = number of times the process is scheduled (1/second)

s = overhead time per scheduling (msec.).

This equation may be solved for "s" to yield:

$$s = (t - M * B) / n \qquad (2)$$

This says that the average time spent per scheduling is the total run time of the process minus the time spent doing the application's work, spread over the total number of times that the process was scheduled.

From the experimental data we can get values for t, M and n directly. This leaves the problem of determining B and s. This problem is solved by considering the equation as having the unknowns B and s, and declaring that these values must be constant from one set of experimental data to the next. Using this assumption, we can take two sets of measurements and use the following formula (derived from equation 1) to estimate B:

$$B = (n2 * t1 - n1 * t2) / (n2 * M1 - n1 * M2) \qquad (3)$$

where t1, M1, and n1 are from one experiment and t2, M2, and n2 are from a second measurement. The value of B is milliseconds of run time per message processed.

It turns out that calculation of "s" using the formula above is very sensitive to small changes in B. We were thus able to determine the value of B with good accuracy.

It should be noted that the time per scheduling(s) that we compute from these measurements is NOT the same as the microcode scheduling time, it is the total of anything that occurs when the process is scheduled and may include timer interrupts, Chrysalis system code invoked only when a process stops or starts, or user process code invoked only when the process awakens or goes quiescent. In interpreting the measurements, it was necessary to identify all the various causes of once-per-scheduling time spent before we could believe that we understood the behavior of the scheduler.

## 4.3 Experimental Results

In the experiments that we conducted, we quickly determined that the Chrysalis system processes behaved in a constant and experimentally uninteresting fashion. In brief, the Epoch Scheduler caused 10 preemptive process swaps per second, and another 10 process swaps as it returned to running the lower priority process it had preempted. The Remote Demon, responsible

for garbage collecting memory, also caused 20 process swaps per second. Together, they took a relatively constant 3.1% of CPU time. Most of this is taken by the Remote Demon, whose run time is a parameter that we can change.

Because these two processes are so constant, the experiments below are described as if they were not present. However, each processor always had an Epoch Scheduler and a Remote Demon running on it.

The other near-constant we found is that 0.8% of the CPU time is not charged to any process. We believe that most of it is lost between the time one process ceases to be charged and the next process starts to be charged. Some of it may also be lost because of the manner in which the Epoch Scheduler attempts to update its own run time while it is running.

In the first experiment we determined the time needed to reschedule a process. To do this, the Message Generator was started with its parameters set so that it would try to send to an illegal destination. In this mode it will run constantly. The Message Generator was the only process trying to run on its processor.

The process requested 100% of the CPU time, so the processor was overcommitted. By doing this, the Message Generator was guaranteed never to be placed on low-queue. The experiment consisted of varying the scheduling slice parameter from 96

milliseconds per slice (one scheduling per epoch) to 1 millisecond (approximately 96 schedulings per epoch).

The time to re-schedule a process whose time slice has run out was measured at 129 microseconds. This includes the time spent in the timer interrupt code, which also implements a wakeup service that is not directly related to the scheduler. In later experiments we found that the timer interrupt code seems to take about 100 microseconds to run. This means that the microcode time for stopping and restarting a process is approximately 29 microseconds.

In the next experiment the Message Generator was set up to send messages via a Dual Queue to a Message Sink process in the same processor. Both processes always ran on high-queue. The Message Generator ran until its time slice had run out, while the Message Sink voluntarily dismissed when it found it had no more messages to discard. Both processes ran at the same priority and so were round-robin scheduled by the Butterfly scheduler. Again the scheduling slices were varied.

We found that the time to switch from the Message Sink to the Message Generator was 119 microseconds. The time to switch from the Message Generator to the Message Sink was 219 microseconds. The difference between these two types of context switch is the 100 microsecond runtime of the timer interrupt routine.

In another experiment we varied the amount of committed time requested by the Message Generator as well. This meant that during part of each epoch the Message Generator ran on low-queue. Since the Message Sink always ran on high-queue, and since Enq_DualQ to a higher priority process results in an immediate preemption, we were able to cause the scheduler to be invoked for every message and to measure the time spent during preemptive scheduling.

We found that the total time for a preemptive process swap was 135 microseconds. However, this measurement includes the time that it takes to post the Event that triggers the preemption. Thus, we can separate this result into 119 microseconds for the swap and 16 microseconds for the Posting of the event. Our experiments are not sufficiently complete to be sure that this division of the 135 microseconds is the correct one, but it corresponds closely to what we expected.

The results of these experiments are summarized in Table 1. These numbers appear to be consistent with what we know about the operation of the Butterfly hardware. For example, the difference in time between switching processes and restarting a process is mostly the time to save and restore the registers when switching. The difference of 119 - 29 = 90 microseconds is slightly (and within experimental error) larger than the time a 68000 takes to perform a register save and restore.

**SCHEDULER TIMINGS**

| OPERATION | TIME | COMPONENTS |
|---|---|---|
| RESTART PROCESS | 129 μsec | 29 μusec MICROCODE<br>100 μsec TIMER INTERRUPT |
| ROUND-ROBIN SWAP AT END OF SLICE | 219 μsec | 119 μsec REGISTER SWAP & MICROCODE<br>100 μsec TIMER INTERRUPT |
| ROUND-ROBIN SWAP ON VOLUNTARY DISMISS | 119 μsec | REGISTER SWAP & MICROCODE |
| PREEMPTIVE SWAP | 135 μsec | 16 μsec EVENT POSTING<br>119 μsec REGISTER SWAP & MICROCODE |

Scheduler Performance Measurement Results
Table 1

Incidental to these experiments, we were able to make some preliminary measurements on the BSAT processes that were used as test vehicles. Four types of process were used. There was a Message Generator Process which composed messages and put them onto a Dual Queue. There were two kinds of Message Sink process. The "simple" Message Sink received message IDs on a Dual Queue and discarded them. When a message ID appeared, the Simple Message Sink removed it from the Queue, mapped in the message, incremented a counter, and freed the message buffers. The "complex" Message Sink simulated part of the activity of a BSAT host I/O process, and its processing rate was much lower (in fact, the results of these measurements prompted a rewrite of part of the corresponding BSAT process). We also used the BSAT Local Delivery process. The function of this process in the BSAT is to route messages that are destined for locally connected hosts. When a message ID arrives, the Local Delivery Process removes the ID from a Dual Queue, maps in the message, uses the header to make a routing decision, enqueues the message ID to the appropriate host output queue, and posts an event.

Some preliminary throughput measurements for these processes are given in Table 2. The first part of the table shows the processing rate of the Message Generator, the Simple Message Sink, and the Local Delivery process, running on separate Processor Nodes. For the Message Generator, results are given for two cases. In one case, the Message Generator output queue

was local to the process. In the other case, the queue was on a remote node and the enqueue operation typically involved Posting an event to a process waiting on the queue. This accounts for the difference in throughput. The second part of Table 2 gives the measured throughput for the Message Generator, both types of Message Sink, and the Local Delivery Process running together on the same Processor Node. As expected, these numbers indicate lower throughput.

Since these measurements were made, some of the BSAT code has been rewritten to be faster. Measurements of Chrysalis routines have changed our understanding of which operations are "expensive" and which are "cheap," and will be a basis for tuning Chrysalis, the Voice Funnel and the BSAT for better performance.

Another aspect of the scheduler that was tested but which is difficult to describe easily is the handling of committed, uncommitted, and windfall time. We performed tests that determined that the scheduler does indeed run processes on high-queue during uncommitted time, moves them to low-queue at the proper point after committed time has begun, and will properly preempt lower priority processes when a higher priority process becomes runnable. These changes were seen as increases in both the number of times selected processes were scheduled, and in the throughput numbers. By modelling the processes' behavior during each phase of the epoch (uncommitted, committed, windfall), we were able to compare our composite numbers with the observed

values. There was excellent agreement.

Processing Rate

| Process | Messages/second | msec/message |
|---|---|---|
| Message Generator | | |
| queue on same processor | 1811 | 0.5522 |
| queue on remote processor | 1633 | 0.612 |
| Simple Message Sink | 6580 | 0.1520 |
| Local Delivery | 2095 | 0.478 |

Combined Throughput - Single Processor Node

| Processes | Messages/second | msec/message |
|---|---|---|
| Message Generator/ Simple Message Sink | 1345 | 0.743 |
| Message Generator/ Local Delivery/ Complex Message Sink | 688 | 1.453 |

Table 2. Preliminary BSAT Throughput Measurements

To complete these results, it would be useful to count the instructions in the appropriate macrocode and microcode routines to make sure that the measurements correspond to the instruction times and to give us a better feel for such things as what impact the instrumentation software has on the performance of the scheduler. This is an activity that we have not yet undertaken.

## 5. Conclusions

The concept of an epoch scheduler was developed by the Butterfly group at BBN, based in part on the TENEX/TOPS-20 idea of a pie-slice scheduler. A pie-slice scheduler allows a user to buy a fixed percentage of a shared computer for a fixed price. If all users are always ready to run and the machine is fully committed, the goals of the two schedulers are quite similar (although the implementations no doubt differ). The major differences involve the idea of an explicit epoch (a specific pie-slice is only guaranteed over some indefinite period) and the treatment of uncommitted and windfall time.

Since pie-slice schedulers do not ordinarily guarantee to provide service within any specified interval, needed time may not be made available soon enough. Real-time applications may need time several times a second or a minute, while time-sharing or batch systems may provide pie-slices only when averaged over periods of minutes or hours.

A secondary problem is the distribution of uncommitted and windfall time. For example, assume that groups A and B own pie-slices, but that group A fails to use any time for a few days, while group B is using the entire machine. If group A then also attempts to use the entire machine, it may well be assigned its pie-slice plus all available uncommitted and windfall time, in an attempt to equalize the situation between the two groups.

When allocating uncommitted time, pie-slice schedulers generally try to provide fairness over an interval of perhaps several months. This sort of policy may be good from a billings point of view, but is not necessarily suitable for real-time applications. The current Butterfly epoch scheduler uses a first-come-first-served approach (with time-slicing) for distributing such time; once an epoch is over, all processes are restored to an equal footing. Thus, performance under heavy load is proportional to need, independent of past history.

We expect that for some applications it may be necessary to monitor system performance on a longer-term basis, and to tune the need and time-slice parameters of various processes dynamically in order to provide desired performance under varying types of load. This could be done either manually or automatically. In the Butterfly multiprocessor, this monitoring and tuning is closely related to the problem of load balancing among the individual processors and the same set of programs would probably perform both tasks.

The epoch scheduler described above is fast and efficient, and seems to provide the facilities needed for real-time telecommunications applications. Coding much of the posting and scheduling algorithms in microcode has allowed us to switch processes in times on the order of a hundred microseconds. To speed this up further would require a CPU with the ability to switch between sets of internal registers without dumping its

current set into main memory and restoring a new set. This would allow process switching times in the tens of microseconds.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
~~Dr. Vinton Cerf (1)~~

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Danny Cohen
Steve Casner

MIT/Lincoln Labs
Dr. Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center
Neil Marples - RBES (1)
Julian Gitlin - DCLD (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office (2)
S. Blumenthal
R. Bressler
R. Brooks
W. Edmond
G. Falk
J. Goodhue
S. Groff
E. Harriman
F. Heart
M. Hoffman
M. Kraley
A. Lake
W. Mann
W. Milliken
M. Nodine
R. Rettberg
P. Santos
G. Simpson
E. Starr
E. Wolf