# THE DECOMPOSITION OF A MULTIPROCESSOR EVENT TRACE INTO USER PROGRAM RESOURCE DEMAND PATTERNS *

by

A.S. Noetzel and J.R. Haley

May, 1975 TR-49

* Research reported in this paper was sponsored by NSF grant GJ-39658

Technical Report 49

THE UNIVERSITY OF TEXAS AT AUSTIN

DEPARTMENT OF COMPUTER SCIENCES

## 1. Introduction

Statistical data such as the utilization factors of various resources is often extracted from computer systems through the use of both hardware and software monitors. But this data, summary in nature, is insufficient for several kinds of analysis. For example, it is not possible to observe or compute the factor of simultaneous utilization of two processors if the monitor has collected the utilization data of each processor independently. Generally, statistical data has only limited value in simulation modeling, because a model constructed to resolve the most specific and detailed questions concerning the system's performance will require the joint distributions of all relevant interdependent operational parameters. The value of the reduced data is limited even if it is extracted only to be observed, to gain insight or to resolve general questions concerning the system's operation, because only detailed data organized in a logical and convenient manner, can show both the values of the operational parameters and the reasons for those values.

This report concerns the problems of extracting, condensing and organizing a useful representation of the internal activity of a multiprogrammed computer system. The representation is a set of traces of the processor invocations and busy periods caused by each of the individual user jobs. The job traces are obtained from a trace of all the events in the multiprogrammed system called the system event trace. The job traces have greater utility than the system event trace in several applications.

### Motivation for the recording of job event traces

Traces of user jobs that show the demand for the system's resources made by the job are useful in the following applications:

1) Insight into the interaction of the processors, from the point of view of the degree or quantity of such interaction as it is actually determined by jobs, as opposed to potential or theoretical interactions.

2) Input for a simulation model. Use of actual user program traces will provide more accurate simulations than the technique of constructing approximate probability distributions from condensed measurement data and randomly generating jobs characteristics from them, since this latter technique wrongly assumes the job characteristics to be statistically independent of each other.

3) Extracting summary statistics describing the system's workload characteristics.

4) Measurement of the degree of duplicated processing in the system. As the individual job traces are recorded, it will be possible to determine the degree to which processing sequences are duplicated due to multiple submissions of a single job or variant versions of the job. If the degree of duplicated activity is significant, it will raise the possibility of recording detailed representations of the job's resource demand pattern (similar to the job traces), and making them available to the scheduler for predictive scheduling. This is the primary motivation for the job trace recording that is reported here.

Section two of this report is a general description of the hardware and software processors of the CDC 6600 computer system for which the trace extraction is implemented, and the interprocessor communications mechanisms. The events of the system event trace are those of interprocessor invocations, hence, this section also affords a description of the system event trace. Section three of this report contains a description of the user job trace, and summarizes the general problems that must be resolved by a program that produces the job trace from the system event trace. These two sections define and use terminology specific to the CDC 6600 system, yet the aspects of the system that are presented, and the problems that are described, are those that are common to

multiprogrammed, multiprocessor computer systems. Section four of the report describes the program, called the event trace decomposition program which produces the job traces from the system event trace. This description deals with details specific to the 6600 system, as well as the general problems. Section five presents several results of the extraction of job traces. These results show that the job traces are consistent, in that the same trace is obtained for identical runs of a program, and that they are rather insensitive to minor changes in the program run.

## 2. Description of the Multiprocessor System

The CDC 6600 system is described as a multiprocessor system since it allows several general-purpose processors to be simultaneously engaged in processing a single job. However, only one processor (the CPU) executes user-generated code. Ten peripheral processing units (PPU's) execute system routines ancillary to the user program. The diversity of system functions performed by the PPU's, including but not limited to the I/O functions, give the CPU-PPU complex the aspects of a general multiprocessor system. The facilities for communication (signaling and blocking) are general in that they have not been implemented to achieve a particular user function (such as physical I/O performed by a channel) or to be dependent on a particular state of the system. The implementation of the software interprocessor communication mechanisms is not independent of the characteristics of the hardware. The CDC 6600 hardware will be reviewed before the description of the software is presented.

### Hardware Background

Figure 2-1 shows the basic hardware organization of the CDC 6600 system. The CPU is a large fast processor that can execute out of main memory (128K of 60-bit words) and can execute block data transfers between the main and backup (ECS) memory units. It cannot directly access the channels, nor the memories of the PPU's.

Each PPU executes out of its own memory (4K of 12-bit words), and can access any channel, read and store into central memory, and interrupt the CPU if the CPU is executing in user mode. The PPU's are not only suited for I/O operations (using their own memories as buffers for data exchanges between the channels and main memory) but are more capable than the CPU of performing the control functions of the system. The PPU's cannot directly access the ECS unit.

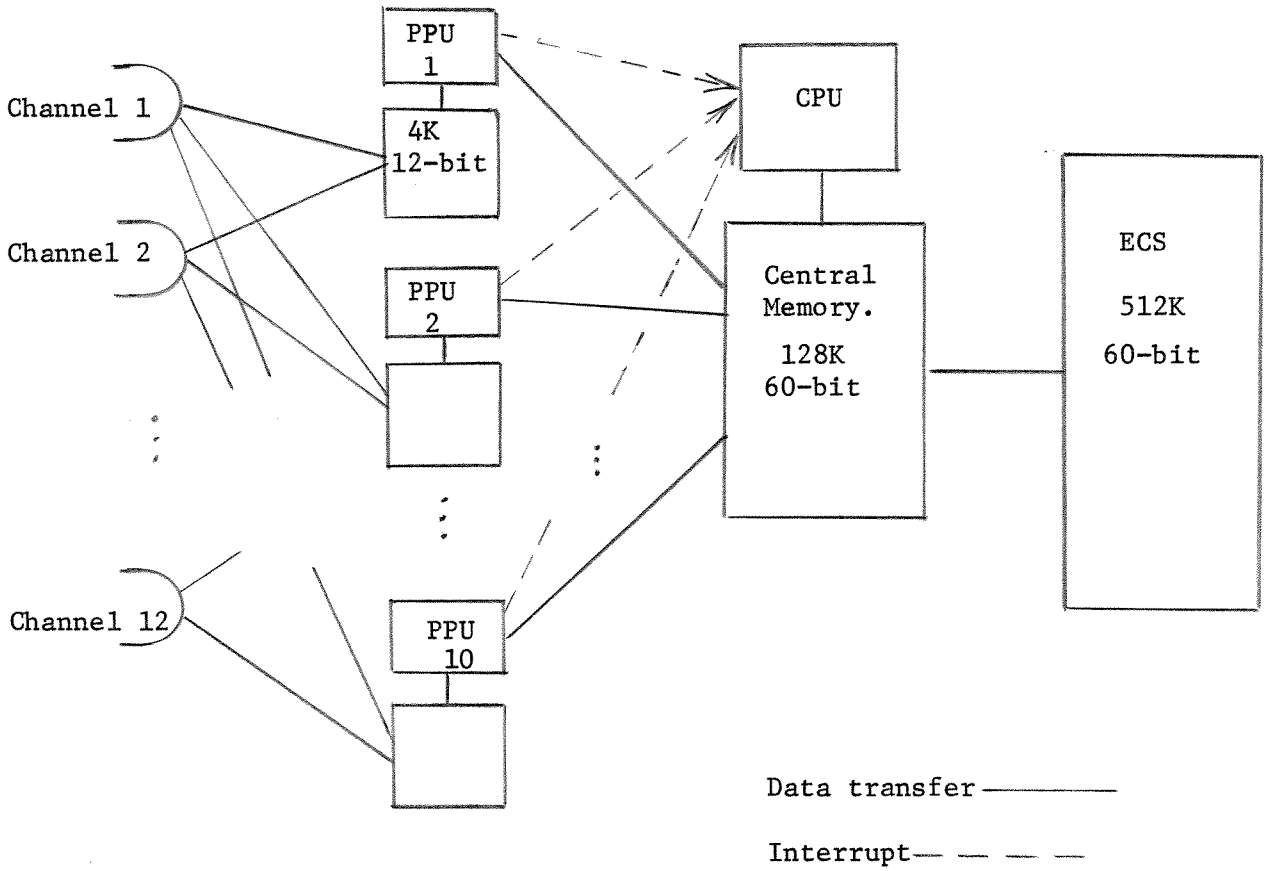



Figure 2-1

Communication paths in the CDC 6600 System

The Multiprocessor Communication Facilities

One PPU, called MTR, contains the main controlling routine of the operating system. It directs the operations of the other (pool) PPU's. These pool PPU's have no identity other than the program they are currently executing, hence they will often be called peripheral programs (PP's). A PP requires an available PPU, and communication to the PP will be identified by the PPU number. MTR also controls the sequencing of the CPU among the user programs multiprogrammed in main memory.

But not all of the operating systems functions are performed by MTR and the PP's. Because of a requirement of speed, availability of large working memory, or communication with ECS, some functions will be performed by the CPU. This part of the operating system will be called Central-Memory-Resident (CMR).

A user job resident in main memory is called a control point (CP) and is uniquely identified by a control point number (CP number). Jobs on backup storage have no CP number. The CP number is used as an index to a table resident in main memory, the Control Point Status Table (CPST). The CPST provides the relocation address (RA) and field length (FL) of the main memory occupied by the job.

Each of the virtual processors--MTR, CMR, PP or CP can communicate with any other through table entries in central memory. The CPST is one such table. Another important table is called the PP communications area. This table consists of the following entries for each PPU: one word, called the input register (IR), which contains the name of the PP being executed in the PPU; an output register (OR) that contains the PP's requests for service and their responses; and a message buffer (MB) that contains data needed to amplify the other entries.

With this background, it is possible to review all of the possible processor invocation types in the system. Then, as the trace facility is described, it will be seen to provide a complete record of processor invocations during the trace period. Last, the job trace can be understood to be a detailed record of processor invocations resulting from one job. The mechanisms for invocation are distinguished as primary or secondary, depending on whether they operate in a conceptually single step, or whether they are a composite of other invocation types. Both polling and the interrupt are identified as primary mechanisms.

The possible processor invocation types are shown in figure 2-2.

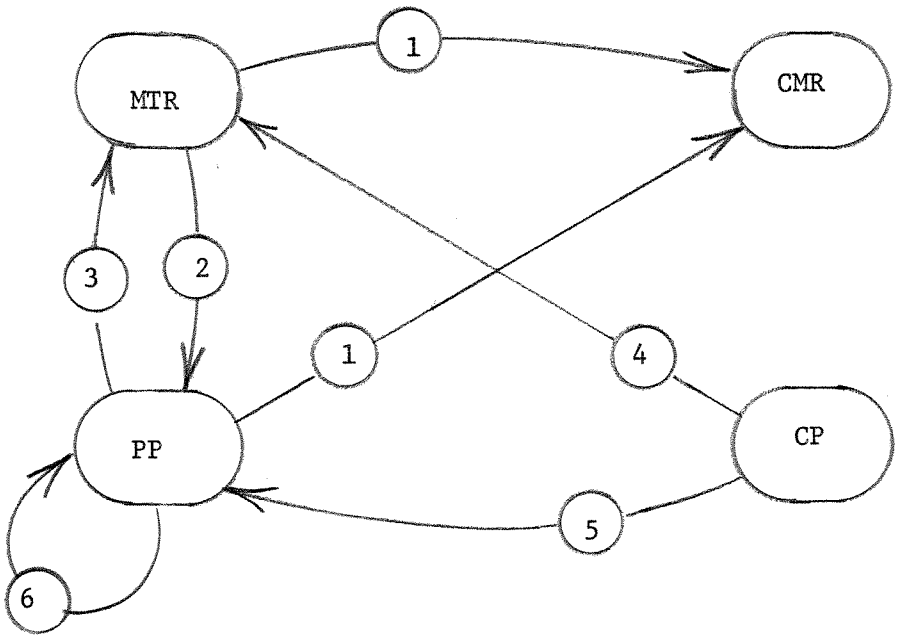
Primary mechanisms

1. PP-CMR and MTR-CMR invocations. The PPU's can interrupt the CPU, causing it to execute CMR. When any PP requires a CMR function, it places the code and parameters of the function in its OR and signals an interrupt. CMR scans the OR's of all the PPU's and performs any CMR function found.

2. MTR-PP invocation. An idle PPU polls its IR. MTR places a PP name in the IR; the PPU locates the PP, loads and executes it.

3. PP-MTR invocation. A PP requiring service places the call for service in its OR. MTR, in its main loop, scans the OR's of all PP's. On finding a function request, it either dispatches it to CMR or another PP, or performs it itself. This is called the PP-function request, or PP-MTR function request.

4. CP-MTR invocation. A control point places a call for service in address 1 relative to the relocation address of the CP. MTR polls the RA+1 location of the active CP in its main loop. MTR either performs the function or dispatches it to another processor, as before. This is called the RA+1 call.




1. Interrupt
2. PP-assignment
3. PP-MTR function
4. RA+1 call
5. RA+1 call and PP assignment
6. a) PP-MTR function (RPP)
   b) PP self-assignment
   c) PP-MTR function (EDR)

Figure 2-2

Processor Invocation Mechanisms

Secondary Mechanisms

MTR spends most of its time polling the OR's of PP's and the RA+1 location of the active CP. It performs simple functions itself, and passes the larger functions on to other processors. The secondary invocations involve a request that MTR picks up and passes on. These are also shown in figure 2-2.

5. CP-PP invocation. The active CP calls a PP by means of the RA+1 call. MTR recognizes it as a PP name, and places it in the IR of an available PPU.

6. PP-PP invocation. A PP calls for another PP by one of three methods.

a) A PP specifies its need for an auxiliary PP by placing the PP name and the 'request PP' (RPP) function code in its OR. MTR picks this up and assigns a PPU for the PP.

b) A PP can cause sequential execution of another PP in its own PPU by placing the new PP name in the IR of the PPU, and transferring to the PPU idle loop. The new PP will execute without MTR being appraised of the change of PP in the PPU. This will be called PP self-assignment.

c) A PP may request that another PP begin after a specified time delay. It requests the 'Enter Delayed Request' function (EDR, which is performed by MTR), specifying the desired PP and delay. MTR places the PP call and name on the 'Delayed Request' stack, and assigns the PP when the delay expires. A PP may make an immediate request by specifying a delay of zero.

In the hierarchy of processes that constitute the system, CMR is at the lowest level. CMR never invokes MTR or a PP, nor does it modify the status of a CP. Thus the CMR-MTR, CMR-PP, and CP-CMR processor invocations do not exist.

The only processor invocations that are missing from figure 2-2, then, are invocations of a CP. CMR, following periodic orders by MTR, sequentially transfers control among the control points that are not blocked for a signal from another processor. But this is allocation of the CPU as a resource, and is not to be confused with the logical invocation of a process, or virtual processor. Strictly speaking, a CP is not invoked by another process, because it is at the top of the hierarchy of processes. However, MTR enables processing at a CP by setting the status of control point to 'ready'.

Thus, the following two communication paths should be considered as forms of invocation.

7) MTR-CP. MTR sets the status of the CP to 'ready for CPU'. If the CPU is idle, it is interrupted to begin processing at the CP.

8) PP-CP. A PP requests reactivation of a CP via the PP-MTR function 'recall central program' (RCP). MTR changes the CP status.

A CP may request blocked status for a specified period, or until the completion of a PP, by means of the RA+1 call 'Recall' (RCL). If a specific delay was requested, MTR places a request for its Recall Central Program function on the delayed request stack.

The Event Trace

The choice of the processor invocations to be recorded as events and the amount of data to be recorded with each depends upon the level of detail desired in the analysis. More detailed analysis requires more voluminous data to be recorded and reduced; and as the expense is increased, the incremental benefit of the analysis is correspondingly decreased. (This effect is exacerbated by the increasingly large amount of artifact generated by the recording process.) For analysis of the efficiency of operation of the machine, a desirable operating point is the recording of enough data to determine the utilization of each process in each system function; that is,

events signalling the entry and exit time of each identifiable system program. Because of the cost constraints, the implementation of the UT-2 event trace makes significant approximations to this ideal.

Specifically, the events recorded, in terms of the invocation types shown in figure two, are the following:

a) All RA+1 calls, recorded when the RA+1 call takes place. (Invocation types 4 and 5.)

b) The PP-MTR functions, usually recorded at the time the function is completed. (This is invocation type 3; the events here will serve as a record of other types as well.)

c) Events providing the CMR entry and exit times. (The times of invocation type 1.)

d) Events describing the completion of CMR functions. Since CMR functions are completed quickly, the time of these events will be essentially that of the CMR initiation, hence, this completes recording of invocation type 1. If the CMR function cannot be completed immediately (due to queueing delays) an event is recorded at the time the function is first attempted, and another when the function is completed.

e) Clock update events, recorded when CMR is active for an extended period. This provides an estimate to the time of interrupt events when the CPU was in supervisor (non-interruptible) state, as well as indicating the duration of CMR functions.

f) Events provided by the scheduling routines describing the status changes of the OS.

The PP-function events (b, above) provide sufficient data for the complete specification of all other processor invocations. When an idle PP first reads a program name in its input register it immediately calls for

the PP-function 'LPP' to find the location of the required PP. Hence, invocation type 2 is in the trace. When a PP completes its operation, it so signals MTR by means of the PP-function 'Drop PP', hence completion time of the PP is available.

Both 'Request PP' (RPP) and 'Enter Delayed Request' (EDR) are PP-MTR functions. Hence, invocation types 6a and 6c appear in the trace. The presence of PP self-assignment (invocation type 6b) is implicit in the trace, in the form of an LPP function without a corresponding DPP. A PP completing a function for a CP will issue the PP-MTR function RCP to change the CP status to ready. This is invocation type 7.

In summary, not all of the data associated with the event of each invocation type is recorded at the time the event takes place. Timing information in particular may be recorded in events surrounding any particular invocation. However, the events of each invocation type are recorded in some form in the system event trace, and each is tagged with the identification of the PP or CP making the call.

## 3. Decomposition of the Event Trace to Job Traces

The two qualities most desired of the individual job traces are completeness of the representation of the user job's demand for system resources, and independence of the scheduling effects of the operating system. Completeness of the resource demand data is achieved in the job trace chosen for the UT-2 operating system by including all the events that can be related to that user that are not scheduling events. From the preceding description of the event trace, this is seen to provide a detailed record of processing periods, memory requirements, channel holding time, etc.

The degree to which the job traces are independent of the scheduling effects is indicated by the reproducibility of the results of the tracing and decomposition processes. If a single program is run several times with identical data, the job traces obtained in every case should be the same.

### Format of the Job Trace

A job trace consists of a set of sequences of (time, event) pairs. There is one such sequence for each processor run. The events are those produced by the processor, and the times are relative to the beginning of the processor run. A simple example of a job trace is shown in figure 3-1. Each sequence beginning with an 'lpp' event and ending with a 'dpp' event represents a peripheral processor run. The events of these sequences are those of invocation types 1, 3, and 6 recorded in the system event trace. The right most trace of figure three is the trace of the central processor activity. These CP events are the RA+1 calls (invocation types 4 and 5) recorded in the system event trace. Those that are calls for PP service (invocation type 5) include pointers to the PP runs providing the service.

The leftmost PP sequences in figure 3-1, which are not called by the CP, represent the initialization sequence of the user job. These PP runs occurred

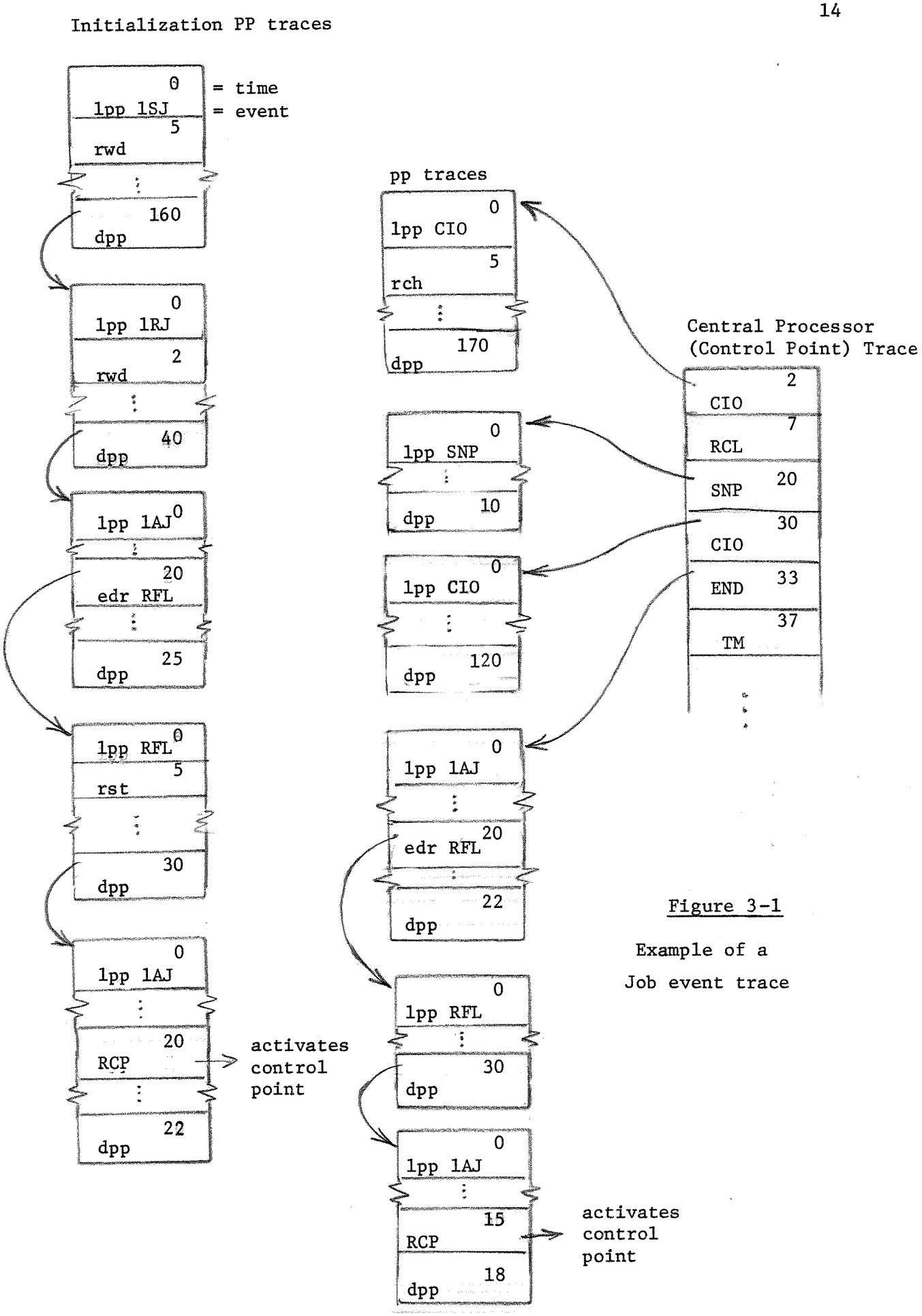



Figure 3-1

Example of a

Job event trace

for the job before the CP was ever activated. The 'Recall Central Processor' event in a PP sequence (invocation type 8) represents the enabling of the central processor. It does not include a link to the central processor in the job trace, because due to lack of synchronization of the PP and CP, this PP event cannot specify the particular point at which CP processing should resume.

Use of the Job Event Trace

The format of the job trace results from consideration of future processing requirements as well as completeness and scheduler independence. The use of the trace as input to a simulation model can easily be imagined. Each pseudo-processor of the simulation model will maintain its own pointer to its event sequence and will simulate an invocation by passing the pointer indicating another processor run (which is in its event sequence) to the pseudo processor being invoked.

General Considerations of the Decomposition Process

The decomposition program creates the individual job traces in a single scan of the event trace. The program maintains counters of accumulated processing time for each program run.

In most instances, the number of the processor (CP or PPU) which produced the event is present within the recorded event. Then, the process of collecting the events associated with a particular job requires maintaining the mapping from every PPU to the CP for which the PPU is operating, and from each CP to the job occupying the CP. Both of these mappings will vary during the activation period of any one job.

The techniques for eliminating the operating systems scheduling effects depend upon the particular form of these effects; some are general, in that they would appear in any trace of multiprocessing system, others specific to scheduling in the UT-2 operating system.

Removal of Scheduler-Induced Operations

In resolving the question of which processing activities should be charged to the user job (and become part of the job trace) and which considered operating system overhead functions, the concept of reproducibility of the user job trace is useful. If the processing activity appears each time the program is run (with identical data) it should be considered part of the user program trace.

In the UT-2 operating system, a particular PP program, 1AJ, is called to supervise the processing of each control card. This OS function will become part of the job trace. Memory scheduling for a job is done periodically by a scheduler in CMR. If a job is to be brought into memory (assigned a CP) the PP 1RJ (Resume Job) is called. When the job is to be removed, the PP 1SJ (Suspend Job) is activated. The 1RJ run at the beginning of a job and the 1SJ run at the end are required, and are part of the event trace.

Other 1SJ and 1RJ activations during the job's run are due to the preemptive memory scheduling algorithm, and are not included. The decomposition program sets a flag at the occurrence of beginning-of-job conditions (signalled by an overlay of the PP 1PS) and end-of-job conditions (the PP 1CJ) so that only the required 1SJ and 1RJ runs are included as part of the job trace.

The effects of the Round-Robin CPU scheduling algorithm are easily removed as the CPU burst time events are summed to become the accumulated processing time of the job.

Monitoring Job Initiation

The consideration of reproducibility dictates that a PP run that is made for a system control point will sometimes be included in a job trace. The appearance of a new job in the system is associated with the activation of the PP 1PS (which has other functions besides initiating jobs). Since the new job has not been assigned a CP at this point, 1PS operates for a system control point. After it becomes clear that the 1PS activation is introducing a new job, the entire PP run is saved in the job trace.

Because the events of the PP 1PS cannot be immediately stored in a job trace, the decomposition program accumulates complete PP traces in a buffer associated with the PPU before saving the PP run in the job trace.

Removing System Delays from the Job Traces

A requested system function that was not immediately satisfied, due to queueing delays, appears twice in the event trace; once, negated, at the time of the original call ($t_1$), and again, at the time the call is completed ($t_2$). If the function was requested by a CP, the CP will have either been blocked during the interval ($t_1$, $t_2$) or it will have been performing useful computation; it will not have been in a 'busy idle' loop

awaiting completion of the function. Hence, the delay $t_2-t_1$ has no effect on the accumulated processing time for the job.

A PP requesting a PP-MTR function does wait in a loop for the completion of the request. The delay $t_2-t_1$ must be removed from the relative processing time of the PP. The decomposition program accomplishes this by adding the delay to the apparent real time of the PP's initiation, which is the base from which the relative run times of the PP are computed.

Associating Processor Invocations with Processor Sequences

When a PP program calls for another by the RPP PP-MTR call, or when the control point calls for a PP, the called PP will not be synchronized with the remainder of the calling processor run. The called PP may be initiated and in fact may terminate either before or after the calling PP. This presents complications in the order in which PP sequences are stored in the user event trace, with the proper linkage from calling PP to called PP.

If a Control Point continues activity after an RA+1 call for a PP, it may then issue additional RA+1 call and in fact may have several requests for runs of a single PP outstanding at once. The PP runs made in response to these calls may not appear in the system event trace in the same order in which the requests were issued. The decomposition program must properly identify the PP sequence occurring for each call. These and other capabilities of the decomposition program are discussed more completely in section four.

Stability of the Decomposition Program

When the event recording and trace decomposition are applied to several runs of a single program with the same data for each run, the structure of the event traces should be identical. The only differences that should

occur are in the relative times of the events in the trace. These are due to hardware effects that are extremely difficult to measure. Examples are: a) the amount of interference with processing due to memory cycle-stealing, b) the unpredictable time spent in a busy form of waiting until the MTR polling sequence picks up the RA+1 call, and c) the unpredictable seek time in disk operations. The implemented trace recording and decomposition procedures are stable in this regard. These and other results of tests of these procedures are presented in section five.

## 4. The Event Trace Decomposition Program

The decomposition program is used to collect complete interactive user job representations with respect to the jobs' demand for the following system resources:

The central processor (CPU)
Pool peripheral processors (PPU's)
Main memory (CM)
I/O channels

User job representations (job traces) are created by the decomposition program in a single scan of an event trace tape produced by the UT-2D system event recorder. The event recorder, a software probe embedded in the system monitor control programs, records on magnetic tape detailed event traces containing a microscopic history of user program activity and monitor decisions during the recording session. The decomposition program distinguishes (in the event trace) events associated with particular user jobs and deletes from these the effects of the system's multiprogramming environment to produce individual user job traces which are both compact and complete, with respect to the resource requirements for each job.

Figure 4-1 shows the procedures required to obtain the user job traces. The job traces produced by the decomposition program are examined for resource-demand patterns, and are used as input to a trace-driven simulation model of the system to analyze predictive scheduling policies.

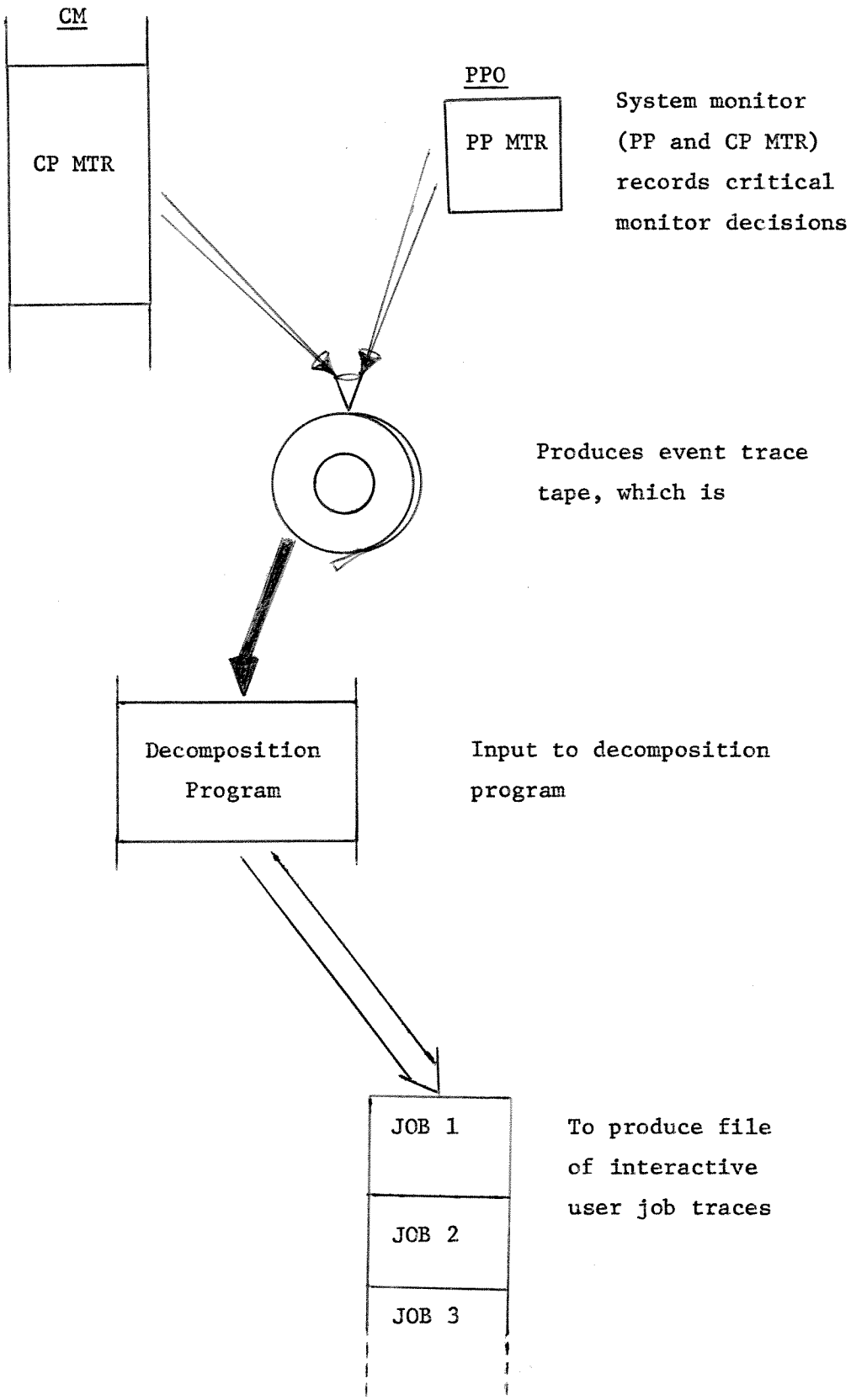

CM
CP MTR
PPO
PP MTR
System monitor (PP and CP MTR) records critical monitor decisions
Produces event trace tape, which is
Decomposition Program
Input to decomposition program
JOB 1
JOB 2
JOB 3
To produce file of interactive user job traces

Figure 4-1

Primary program capabilities

1. Since complete user job traces are desired, the decomposition program recognizes the occurence of system initialization and termination sequences for user jobs in the event trace. Jobs become known to, and leave, the system in several different ways --- for example, a job initially recognized as interactive may be switched to batch status at user discretion -- and hence recognizing when a particular job starts and when it terminates is not necessarily a straightforward task. Only those user jobs which are initialized during the recording session are traced (since only from these jobs can complete traces be obtained).

The occurence of an initialization sequence for a user job causes the decomposition program to allocate buffer space for the job trace. A termination sequence for a user job signals the program to move the accumulated trace for the job from program buffer storage to a mass storage file.

2. During its lifetime in the system, a user job may execute sequentially at several different control points as a result of scheduler-initiated memory swaps. Occasionally, a swap in sequence for a user job must be aborted because of a CM-scheduler decision conflict. The decomposition program recognizes aborted swap in sequences (punts) by maintaining what is essentially a swapping state graph, represented in the event trace as a sequence of special events, which describes the status of a control point during the implementation of a scheduler decision (see Figure 4-2). If a control point in the "rollin" state becomes vacant, a punt has occurred. If the control point becomes active, then the swap-in has succeeded, and the decomposition program notes that the swapped-in job is now executing at the specified control point.

CONTROL POINT SWAPPING STATE GRAPH

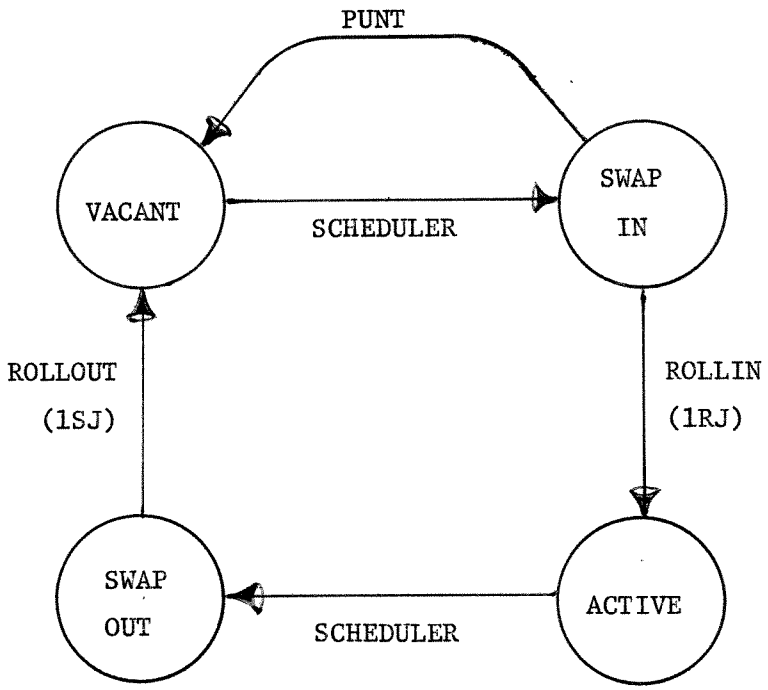



FIGURE 4-2

Memory swaps require resources, at the very least a PPU to perform the swap-in or swap-out and possibly an I/O channel if ECS has become full and the system disk must be used as a swapping device. Since these resources reflect only the multiprogramming environment of UT-2D, and not the resource requirements of the affected user jobs, the events associated with memory swapping (except for the CM of an incoming job) are not included in the job traces. The "think time" interval between a swap-out and the next swap-in for a particular user job may be recorded for use in the trace-driven simulation model, however, if the job is suspended due to a "TTY wait" condition.

3. The user job traces must reflect only the jobs' actual demand for resources. Hence the effects of the UT-2D multiprogramming environment are filtered from each job trace as these effects are recognized. Events generated by swapping activity, as described above, are discarded. In addition, the following are removed from the user job traces:

(i) time spent in queues for system resources
(ii) time spent waiting for global memory compaction
(iii) uninteresting events, such as local and global file reservations, which do not reflect a user job's use of resources

(i) and (ii) require adjustments to be made to PPU holding times -- that is, queueing or compaction times are subtracted from the accumulated processing time of a PPU working for a user job. (iii) merely requires that the decomposition program record only a specified subset of events associated with user jobs. The events that remain, then, are used to produce the user job traces and include the following:

(i) CPU bursts for user jobs
(ii) user requests for service (RA+1 events)
(iii) control command (CC) generated PPU runs
(iv) channel reservations and holding times, exclusive of queueing delays

(v) CM requests issued by PPU's working for user jobs

(vi) PPU runs, with holding times adjusted as described above

(vii) PPU drops

(viii) clock ticks, used to obtain channel and PPU holding times

4. Cause/effect relationships between user requests and system actions are recorded by the decomposition program in order to produce an accurate history of user job resource demands. This is accomplished as follows:

i) RA+1 requests for a PPU transient program are linked to the resulting PPU program traces. The linking process is complicated by the possible variations in the sequence in which RA+1 requests are made and satisfied. For example, a series of RA+1 requests may appear in the event trace prior to the LPP event for the first RA+1 request in the series; hence a table of outstanding requests and pointers into user job traces is maintained by the program.

ii) PPU program requests for reactivation of the PPU program after a specified time interval (EDR requests) are linked to the resulting PPU program traces. System resource requests issued by a particular PPU program for a user job are collected in an event sequence buffer associated with the PPU (the decomposition program maintains an event sequence buffer for each PPU in the system). When an executing PPU program terminates and drops the PPU, the information contained in the associated event sequence buffer is transferred to the appropriate user job trace, if one exists, or else is discarded. The decomposition program uses the table of outstanding requests and pointers to determine whether or not a calling PPU program exists for the terminated PPU. If the terminated program was indeed called by a PPU program in a different PPU, the event sequence trace of the terminated PPU must be linked into the calling PPU's event sequence trace buffer (if the calling program

has not yet terminated) or into the associated user job trace (if the calling program has terminated and its event sequence trace has been transferred). Similarly, pointers in the table must be modified whenever a PPU program with outstanding EDR requests terminates, drops the PPU, and has its associated event sequence trace moved to a user job trace.

As an illustration of the above procedure, consider a user-initiated request for additional CM. The PPU transient program RFL is loaded into a PPU and determines if the desired storage is available. If not, RFL will issue an EDR for itself and drop the PPU. This sequence is repeated until the desired CM is obtained. As each RFL terminates, its event trace is transferred to the corresponding user job trace and is linked to the previous RFL which issued the EDR (except for the initial RFL which is probably linked to an RA+1 request). (See Figure 4-3).

5. CPU bursts for individual user jobs are recognized and accumulated in a fairly straightforward manner. Since a CPU burst for a control point appears in the event trace as a single event, it is necessary only to maintain a record of job assignments to control points to be able to associate a burst with a particular user job.

RA+1 requests may be issued only during a CPU burst; hence accumulated burst times are used to mark time intervals between RA+1 events in the job traces. Figure 4-4 shows the format of a user job trace containing event codes, links, and accumulated times. Note that the PPU times shown represent accumulated holding time for the particular PPU relative to the load of the first PPU program in a sequence.

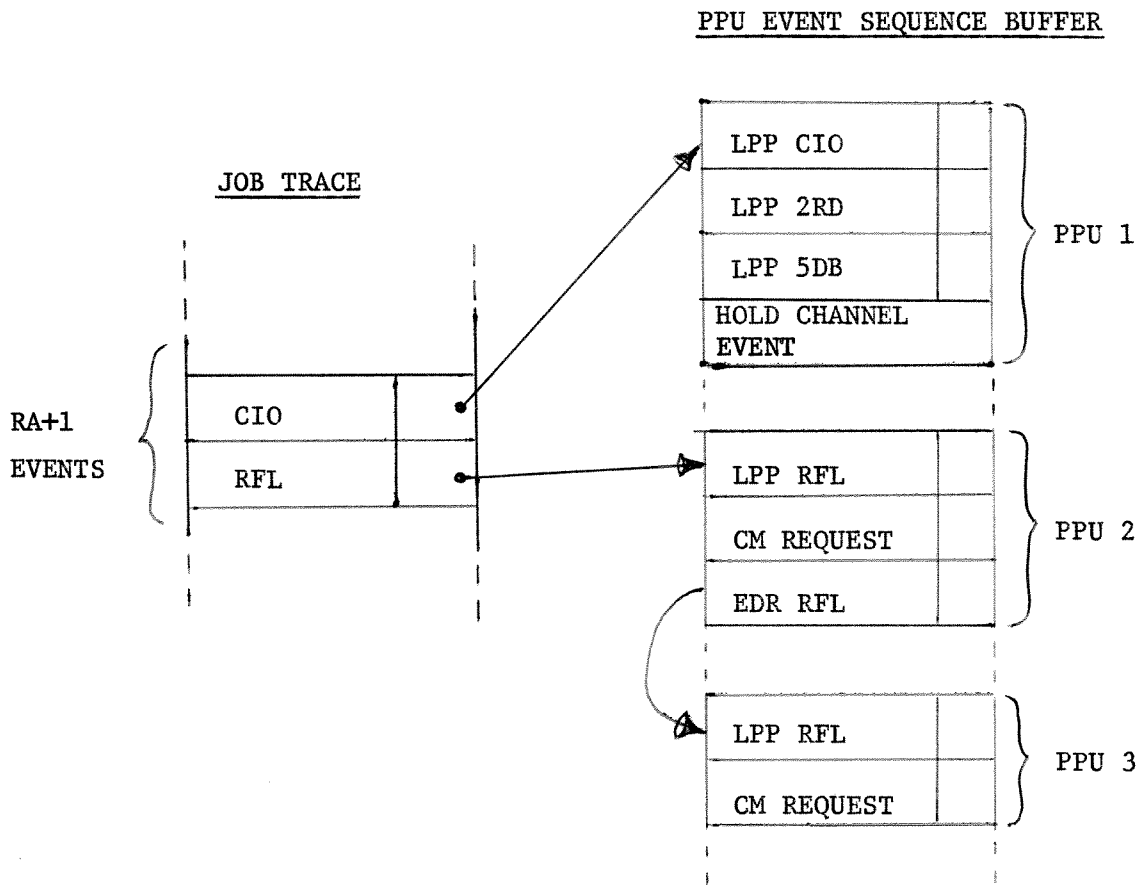

PPU EVENT SEQUENCE BUFFER
JOB TRACE
RA+1
EVENTS
CIO
RFL
LPP CIO
LPP 2RD
LPP 5DB
HOLD CHANNEL
EVENT
PPU 1
LPP RFL
CM REQUEST
EDR RFL
PPU 2
LPP RFL
CM REQUEST
PPU 3

FIGURE 4-3a

AFTER PPU'S 1 AND 2 ARE DROPPED:

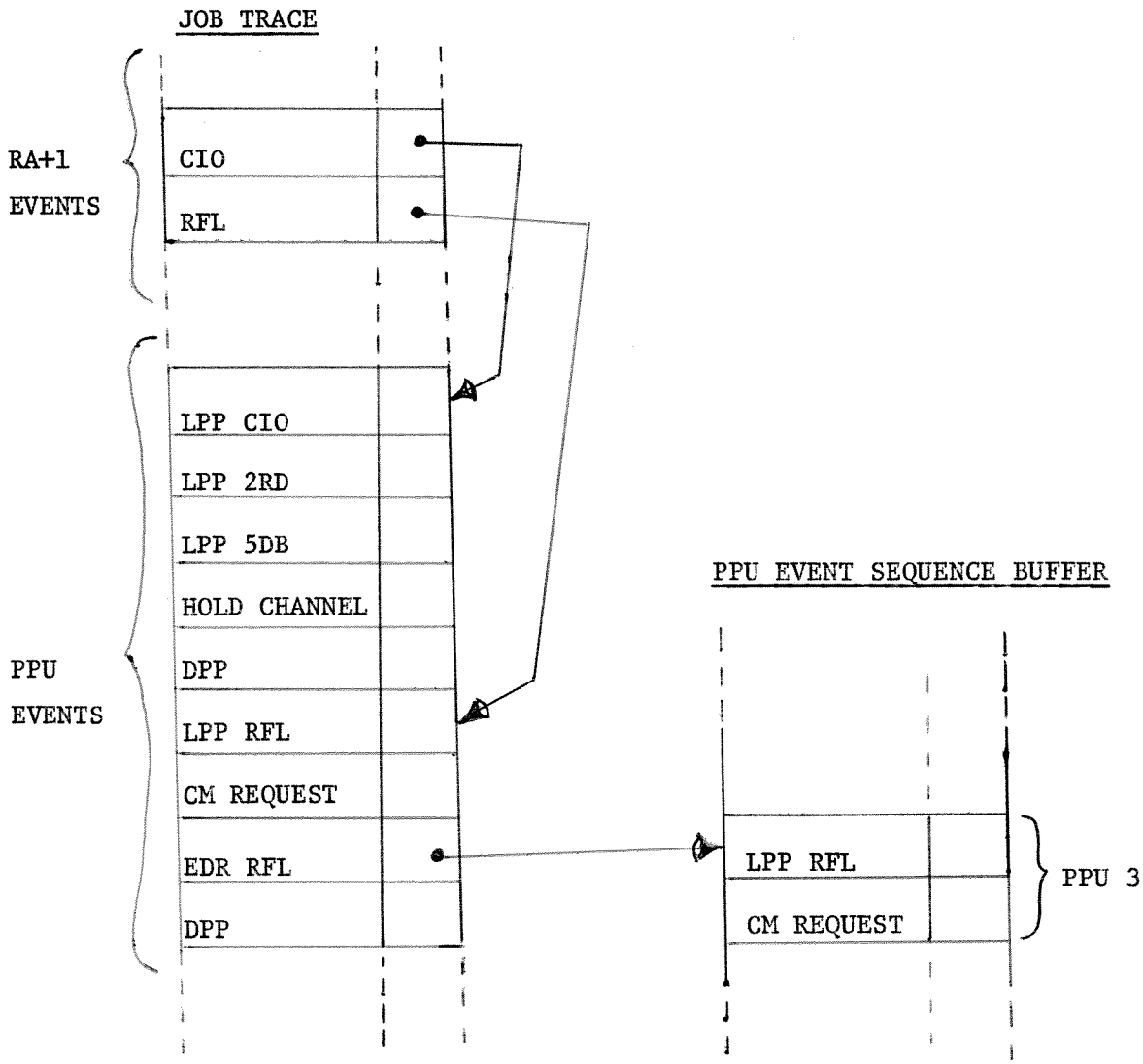



FIGURE 4-3b

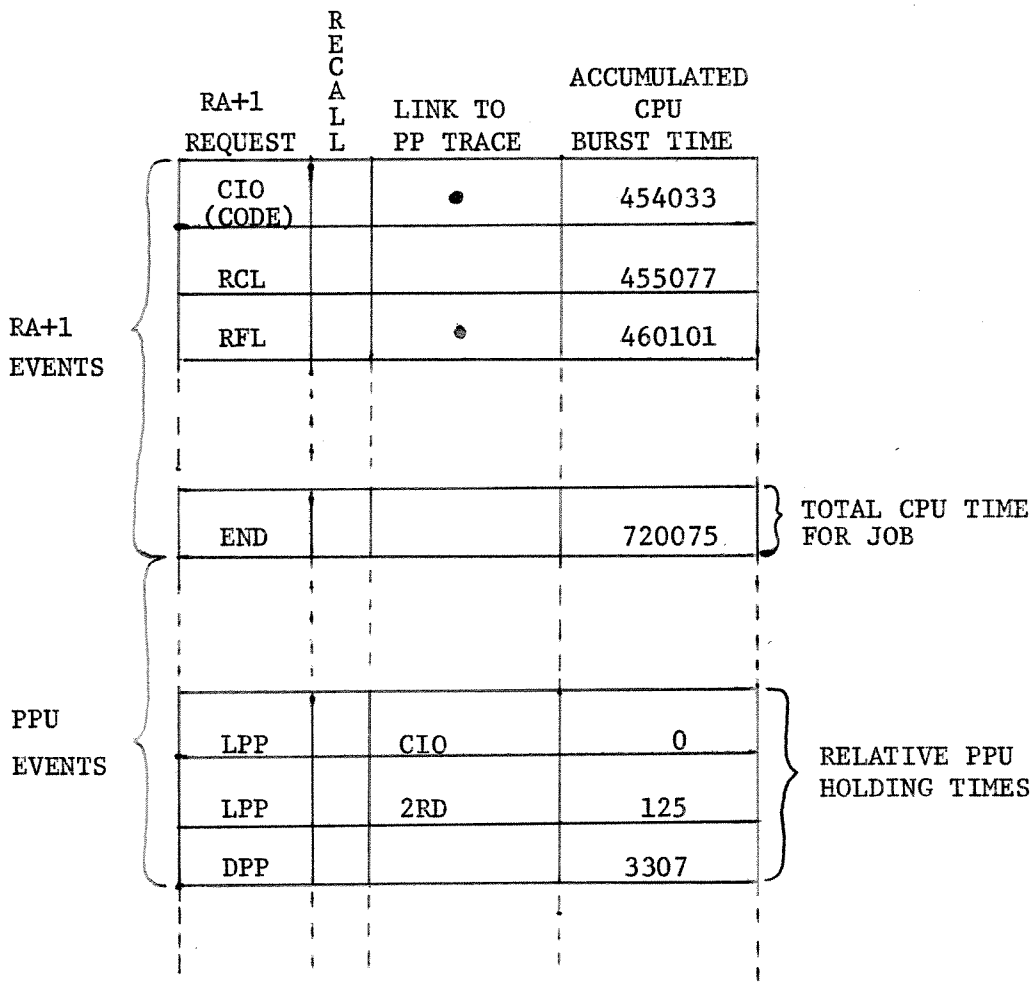

RA+1 REQUEST
RECALL
LINK TO PP TRACE
ACCUMULATED CPU BURST TIME
CIO (CODE)
454033
RCL
455077
RFL
460101
RA+1 EVENTS
END
720075
TOTAL CPU TIME FOR JOB
PPU EVENTS
LPP
CIO
0
LPP
2RD
125
DPP
3307
RELATIVE PPU HOLDING TIMES

FIGURE 4-4

Auxiliary Program Features

The decomposition program contains a storage management facility which dynamically allocates buffer storage for a given user job trace subsequent to the initialization sequence. The job trace actually consists of blocks of CM, allocated on demand and linked together in a list structure. When a user job terminates, the associated trace blocks are written to an external trace file on disk, and the CM storage for the trace is reclaimed for future use. Thus the decomposition program's data area will grow dynamically during program execution but will never greatly exceed the actual CM requirements of the program.

The UT-2D operating system is frequently modified to provide expanded services to the community of users. Occasionally such modifications affect the format or sequencing of the event trace tapes. As a result, event traces sometimes present a moving target to the would-be analysis program. An error trap facility in the decomposition program allows the output of accumulated trace information and system status when anomolies in the system event trace are recognized. Originally intended as a debugging tool, the error trap facility now keeps us abreast of system changes which affect the event traces.

Input control variables (which may be modified dynamically during program execution) allow the decomposition program to:

i) Begin or terminate job tracing at any desired point in the event trace tape (or disk file copy);

ii) Record and save the state of the program at specified points in the event trace to provide a restart capability;

iii) Output formatted details of the state of the system during the recording session continuously or at specified points during program execution;

iv) Output formatted copies of job traces as they are written to mass storage.

Program Output

The decomposition program produces job traces for all complete jobs recognized in the event trace tape within the bounds specified by the input control variables. Job traces contain one word for each event recorded by the program, and contain the following information:

i) Event type (code)
ii) Link (if any)
iii) Time (if relevant)
iv) Resource type (sometimes implicit in the event type)
v) Amount of resource used (size or holding time)

The job traces are designed primarily as input to a trace-driven simulation of the system, but have been formatted to lend thenselves to hand analysis as well.

Structure of the Decomposition Program

A small preprocessing routine positions the event tape past the initial low-core table dumps and, if so directed, skips to a particular buffer within the event trace. A large Fortran control subroutine then defines and initializes the various tables, pointers, and data structures used in the program, and initiates job tracing. This routine controls job recognition and event tracing, and calls on a number of compass and Fortran utility routines which perform dynamic memory allocation, state saving and program restarting, job trace output, job trace compaction, and error detection and processing functions.

## 5. System-Independence of the Event Trace Recording and Decomposition Processes.

The event trace decomposition program produces a user-program resource utilization profile that is almost entirely independent of the system's action in scheduling and monitoring the run. To obtain an indication of this independence, a set of ten jobs (described in the following section) was executed twice during a trace period, with no changes to the jobs or their data made between the runs. The order of the jobs was permuted between the runs, and the total load on the system varied. The trace for the various runs of each individual job are almost identical. Occasionally a few RA+1 calls, which are due to a user program repeating an unsatisfied request, appear in one run and not the other. Otherwise, the sequences of RA+1 calls for each pair of job traces is identical, and corresponding CPU bursts no more than a millisecond different. (Each trace CPU burst is the sum of several quanta of processing time. The quanta are variable length, and measured within one millisecond of accuracy, and with a quarter millisecond precision.)

The numbers of PP runs for each pair of job runs are likewise well correlated. The PPU run times do show some variance, due to disk positioning and the interference of another machine sharing these units. The aggregate statistics for each of the two sets of job runs, are the following.

| | Set Run 1 | Set Run 2 |
|---|---|---|
| Total CP events | 932 | 844 |
| Total CP time (seconds) | 30.807 | 30.795 |
| Total PPU runs | 526 | 524 |
| Total PPU time(seconds) | 129.706 | 129.525 |

These data show that the recording and decomposition process is stable, with only minor variations. The differences in the resource utilization patterns of two runs is therefore due to the differences in resource demand in the runs rather than operating system effect. Some examples of program resource demand patterns are presented in the next section.

Correlation of Resource Demand Patterns Between Similar Job Runs.

The event trace recording and decomposition procedures are implemented to study and measure the degree of duplication of particular processing patterns. This data may be extremely helpful in the design of schedulers which may use the prediction of jobs' resource requirements. Hence, it is of some interest to observe the differences in the extracted resource demand patterns as the control cards, program, and data of a job are modified. Therefore, programs that are basically similar but differed to the degree one might expect from two runs of a production program or two compilations of a program in development, were run during the trace period. The results show the resource utilization patterns to be rather insensitive to these changes. The results are as follows:

1. Changes of Control Card Specification.

In runs one through three, the same relocatable binary program was loaded and executed. In run one, this was specified by a single control card, but two separate control cards were used in run two. In run three, the program was read from the file system, loaded, and executed in three separate control cards. The results are as follows:

| | Run 1 | Run 2 | Run 3 |
|---|---|---|---|
| CP events | 34 | 53 | 69 |
| CP time (seconds) | .291 | .306 | .328 |
| PP runs | 27 | 30 | 46 |
| PP time (seconds) | 1.547 | 1.961 | 2.443 |

As before, the differences in the number of CP events are due to repeated requests for a function that the operating system may not be prepared to accept. This is a system influence that is not filtered out, yet any application of these traces can ignore the superflous requests. The variation in PP run time among these three runs is due to disk positioning and interference due to sharing.

2. Changes in Compilation.

Run four was a straightforward correct compilation of a Fortran program, but several errors (undefined statement number, parenthesis mismatch, illegal statement function) were introduced into the source program in run five. The results are:

| | Run 4 | Run 5 |
|---|---|---|
| CP events | 62 | 64 |
| CP time (seconds) | 3.113 | 3.142 |
| PP runs | 46 | 48 |
| PP time (seconds) | 10.611 | 11.537 |

The differences in these two runs is actually within the error of the trace and decomposition process. The sequences of supervisor calls were exactly the same (when the repeated events are eliminated) and the additional CP time for run five accumulated over the length of the program run, so the sequences of individual CPU burst times

are almost exactly the same.

3. Compilation With Many Errors.

Runs six and seven were again Fortran compilations. Run six had five Fortran errors, while several cards of the source program were scrambled in run seven, causing 24 errors. The results are:

| | Run 6 | Run 7 |
|---|---|---|
| CP events | 63 | 85 |
| CP time (seconds) | 3.335 | 3.217 |
| PP runs | 50 | 51 |
| PP time (seconds) | 12.663 | 15.426 |

These results again show similar processing patterns. The sequence of supervisor calls was identical for sixty percent of the accumulated CPU time, and the sequences differed only in the name of one supervisor call (except for repeated supervisor calls).

4. Changes in Data

The amount and type of computation occurring in runs of a program may vary over a great range, depending on the data values used. But the usual repeated occurrences of a production program probably result in similar execution patterns. Runs eight and nine are runs of a Fortran program that weaves doubly-linked lists together. Run eight executed and terminated normally, while run nine which had different data, terminated when a bad data value was read on input. The results are:

| | Run 8 | Run 9 |
|---|---|---|
| CP events | 102 | 106 |
| CP time (seconds) | 4.809 | 4.496 |
| PP runs | 72 | 74 |
| PP time (seconds) | 15.586 | 16.478 |

Summary

The system event trace decomposition process produces a job trace that is independent of the multiprogramming effects of the operating system. The traces obtained are in a form that is useful for input to simulation modelling. The traces obtained indicate that there may be a good deal of repetitive processing activity in the system. Scheduling algorithms test use predictive data collected from the system are being designed, and will be evaluated through simulation modelling, with the user job traces as input.

REFERENCES


1. Alexander, W.P.
   Analysis of Sequencing in Computer Programs and Systems, Ph.D. Dissertation, University of Texas at Austin, 1974.

2. Anderson, J.W.
   Primitive Process Level Modeling and Simulation of a Multiprocessing Computer System, Ph.D. Dissertaion, University of Texas at Austin, 1974.

3. Baskett, F., Browne, J.C., and Raike, W.M.
   "The Management of a Multi-Level Non-Paged Memory System," Proc. AFIPS 1970 SJCC, Vol. 36, Montvale, N.J., pp. 459-465.

4. Estrin, G., Muntz, R.R., and Uzgalis, R.C.
   Modelling, Measurement, and Computer Power, Proc. SJCC, 1972 pp. 725-738.

5. Howard, J.H. and Wedel, W.M.
   The UT-2 Operating System Event Recorder, University of Texas Computation Center Report, CC-TSN-37.

6. Johnson, Douglas S.
   A Process-Oriented Model of Resource Demands in Large Multiprocessing Computer Utilities, University of Texas Computation Center Report TSN-33, Austin, Texas.

7. MacDougall, M.H.
   "Simulation of an ECS-based Operating System," Proc., AFIPS 1967 SJCC Vol. 30, pp. 735-741.

8. Noe, J.D. and Nutt, G.D.
   Validation of a Trace-driven CDC 6400 Simulation, Proc. SJCC (1972) 749-757.

9. Noetzel, A.S.
   The Design of a Meta-System, Proc. SJCC, 1971, 415-424.

10. Schwetman, H.D.
    A Study of Resource Utilization and Performance Evaluation of Large-Scale Computer Systems, University of Texas Computation Center Report TSN-12 (July 1970), Austin, Texas.

11. Sherman, S., Baskett, F., and Browne, J.C.
    "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System," The University of Texas at Austin, 1970.

12. Wedel, W.M.
    An Introduction to UT-20 for Systems Programmers Use, University of Texas Computation Center Report TIMS-7.