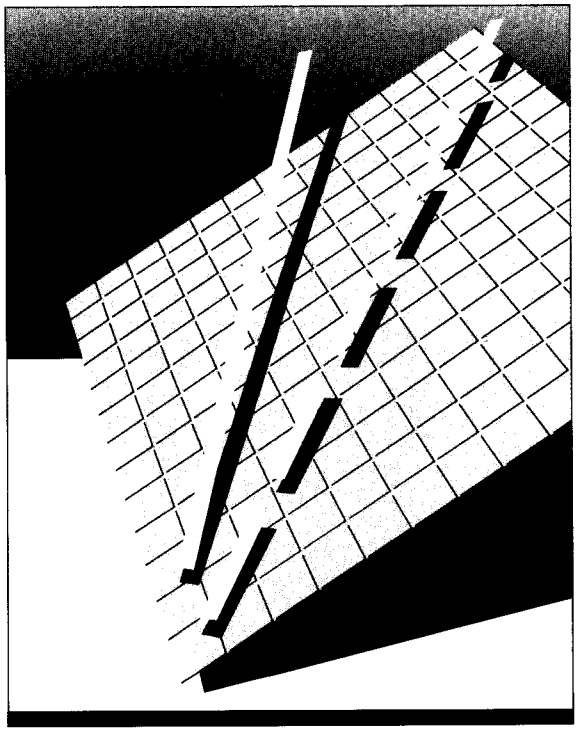
TANDEM JOURNAL

VOLUME 1, NUMBER 1 FALL 1983



PATHFINDER—An Aid for Application Development

Performance Characteristics of Tandem NonStop Systems

TMF and the Multi-Threaded Requester

INTRODUCTION

With the Tandem Journal, we are pleased to offer you a unique source of technical information about Tandem products. The purpose of the Journal is to bring you the perspectives of Tandem developers, engineers, and support analysts on Tandem hardware and software. Quarterly, the Journal will present in-depth articles on product research, design, and implementation.

In this first issue, analysts in our Customer Application Support and Product Management groups bring you their insights into system performance and architecture, and the implementation of various software products. In future issues, Tandem hardware and software developers will also discuss their research and the design of Tandem's newest products.

Our goal is to provide you with interesting and useful information, unavailable elsewhere, that will help you to use your Tandem system most effectively. We welcome your comments and suggestions for future issues.

Carolyn Turnbull White
Editor

T A N D E M J O U R N A L

Volume 1, Number 1, Fall 1983

Editor
Carolyn Turnbull White

Associate Editors
Nick Franks
Kent Madsen

Production Editors
Laurie Souza
Anita Van Auken

Technical Advisor
Geary Arceneaux

Art Director
Gary Bolen

Design
Craig Frazier Design

Cover Art
Craig Frazier

The Tandem Journal is published quarterly by Tandem Computers Incorporated, 19333 Vallco Parkway, Cupertino, CA 95014.

Purpose: The purpose of the Tandem Journal is to bring to Tandem users the perspectives of Tandem software developers, engineers, and support analysts on Tandem software and hardware.

Subscriptions: The Tandem Journal is offered with the Tandem Application Monograph Series in one subscription. The annual rate for the subscription is $100.00. Send subscription orders to Tandem Computers Inc., Sales Administration, Subscription Dept., 19191 Vallco Parkway, Cupertino, CA 95014. Tandem will bill you. Please address all subscription problems or questions to your local Tandem sales office.

Change of address: Send all changes of address to Sales Administration (address listed above).

Comments: We welcome comments and suggestions about content and format. Please send them to Carolyn Turnbull White, Editor, Tandem Journal, Tandem Computers Inc., 1309 So. Mary, Sunnyvale, CA 94087.

PATHFINDER— An Aid for Application Development

PATHFINDER is an efficient new tool for the design of screens for PATHWAY™ applications. It allows designers and users to define and create the screens together on-line, and to simulate the interaction of the screens before a single line of code is written. Created by Tandem analysts and used in the development of a variety of applications at Tandem, it is also available in the International Tandem User's Group (ITUG) library. This article discusses how PATHFINDER solves one of the major problems of effective screen design and presents an overview of how it works.

The Problem

A recurring problem in the development of on-line computer applications is the lack of user input in the development process. Typically, hands-on involvement for users starts after the application has been designed and coded. At that time, the users invariably request changes, but the designers and programmers may resist making extensive modifications because of the significant effort required to rewrite the code. The changes are often compromised to such a degree that the resulting screens may not resemble what the users want at all.

For successful application design, it is essential for designers to maintain a dialogue with the primary users of the proposed system so that the application requirements can be accurately defined. In addition to this, the users must be able to test the application's user interface before coding, so that they can see and feel how the finished product will operate. User dissatisfaction with an application can essentially be eliminated if the users are actively involved in the creation and testing of the application design.

Currently, the most common method of demonstrating screen layouts and the logical flow of the application is to present the screens to the users on paper layouts. The users must then simulate the proposed application in their minds while shuffling through stacks of paper. Clearly, this technique does not represent effective design or simulation.

The Solution

PATHFINDER makes it easy for the users and designers to define the user interface to the system at the beginning of the application development cycle. It also allows them to create screens on-line and to demonstrate exactly how the screens will interact during the execution of the application. Since this is all done before any application code is written, modifications can be made quickly and easily.

The time line in Figure 1 shows how, with PATHFINDER, users are actively involved in both the requirement specifications phase and the screen design phase of the development project.

How PATHFINDER Works

PATHFINDER is a PATHWAY application program consisting of five requesters and two servers. It can be incorporated into an existing PATHWAY system (the recommended method) or run as a separate PATHWAY system. PATHFINDER has self-contained documentation, and each screen has supporting HELP screens to facilitate its use.

PATHFINDER works on edit files, each file containing an image of one of the application screens. Each edit file consists of a single screen of information, and all of them must reside on the same volume and subvolume. The designer uses PATHFINDER to build a relational data base that links together the screen image files. The relational data base consists of a set of files that define the *navigation logic* among the screen images. PATHFINDER allows the designer to build and update the data base, edit files, and simulate user interaction with the application. Figure 2 depicts the relationship of PATHFINDER to the Editor, the screen image files, and the navigation data base.

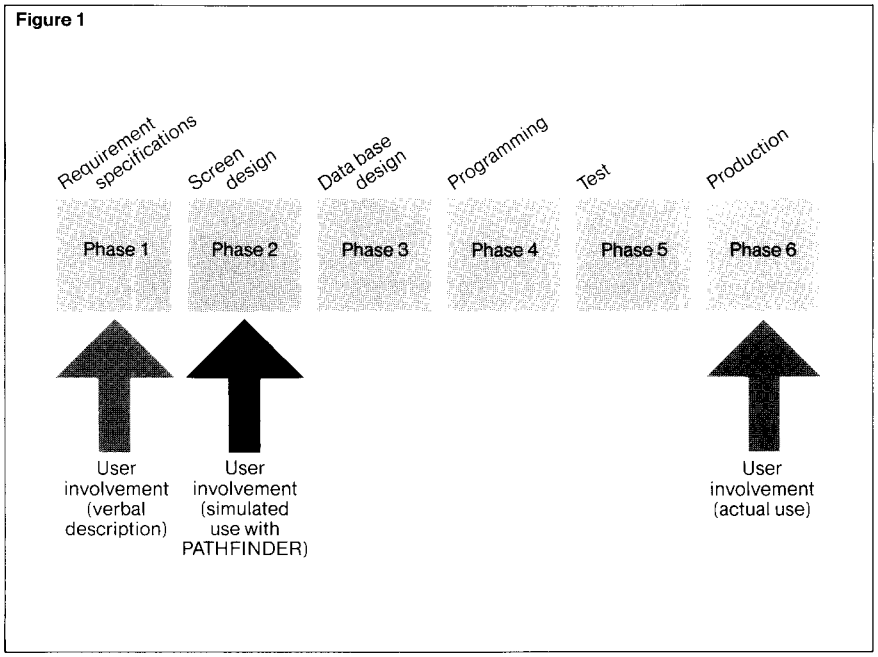


Figure 1

Application development time line when PATHFINDER is used. PATHFINDER contributes to successful application design by involving users in the screen design phase of the development.

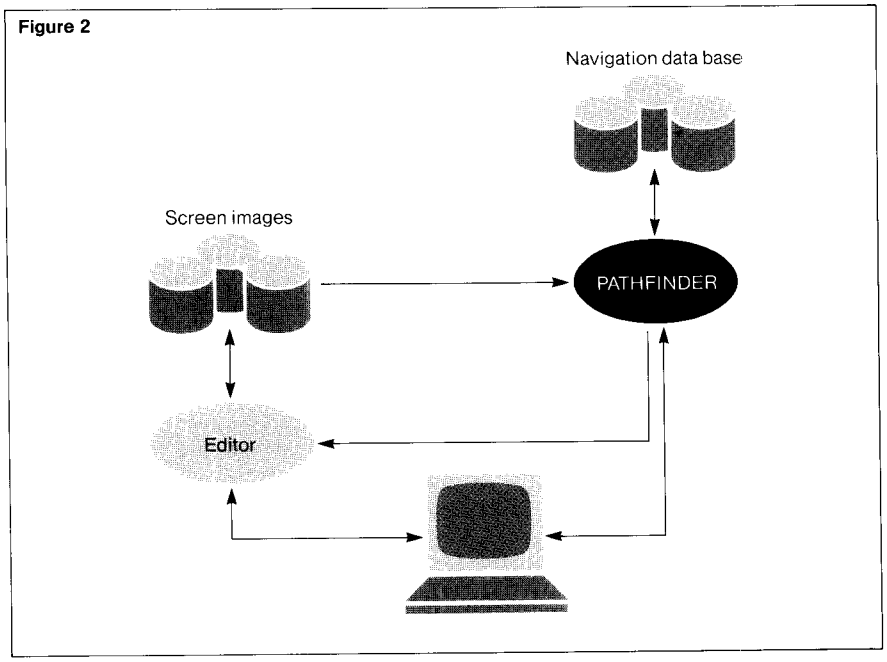


Figure 2
PATHFINDER allows the designer to create screen images in edit files, to define the navigation among the screens in a relational data base, and then to simulate screen navigation as it would occur in the finished application.

Navigation Data Base

The designer uses the ADD OR UPDATE DATA BASE facility of PATHFINDER to build the navigation data base. The interaction between the user and the simulated application is defined exactly as the live application will be enacted.

PATHFINDER presents a series of screens for defining or updating the actions associated with each application screen image. In most on-line PATHWAY applications, either function keys or entered data drive an application. To accommodate this, PATHFINDER allows *actions* to be assigned to any of the 32 function keys. These actions include:

- Moving to the next screen.
- Remaining on the current screen.
- Returning to a previously displayed screen.
- Terminating the application.
- Performing one of several possible actions on data that is entered with that function key.

The last action, the MULTIPLE ALTERNATIVE function, allows simulation of data entry. Instead of entering data, the user selects a description of the data that is to be entered.

Construction of the navigation data base can be accomplished easily in a matter of hours. (For example, the navigation data base for the student registration system at Tandem's Corporate Education Center was created and demonstrated in approximately three hours. The system consists of 54 screens.)

Simulation can be performed on a navigation data base that is either partially or totally complete by entering the NAVIGATION OF APPLICATION SCREENS facility. During navigation, PATHFINDER behaves exactly as if it were the application; selection of a function key produces the appropriate application response. When there are multiple alternatives for a selected function key, PATHFINDER displays a screen containing descriptions of the alternatives, and the user selects one.

Beginning of Application Implementation

Once PATHFINDER has been used to design and test the user interface with the planned application, the screens in the edit files can be used in the coding phase of application development. With minor modifications, the edit files can be used as input to PATHAID, the PATHWAY screen builder, to generate SCREEN COBOL code for the requesters.

Training Tool and Documentation Aid

As an additional benefit, PATHFINDER is an excellent training tool. It allows users to begin training on the simulated application as soon as screen design and testing are complete, long before the application is ready for production.

Also, since the screen images are stored in edit files, they can easily be incorporated into user documentation as illustrations.

Conclusion

PATHFINDER provides users of Tandem NonStop and NonStop II systems with the ability to define and demonstrate proposed user interaction with an on-line application system at the beginning of the development cycle. Among its benefits are:

- User involvement at the beginning of the development cycle, not at the end, as is customary. The application is up sooner with little or no user dissatisfaction.
- Simulation of user interaction with the application.
- User acceptance and enthusiasm for the finished product because of involvement in the design stage.
- Input of the screen edit files to PATHAID to generate SCREEN COBOL code for the application.
- User training on the simulated application during the development phase.
- Screen illustrations for documentation.

Sandy Benett has been a senior systems analyst in Marketing Technical Support's Program Development Group since joining Tandem in August, 1982. Currently, he is involved in enhancing and simplifying the application development cycle, working on PATHFINDER, and teaching PATHWAY. He has also been involved in the performance and tuning aspects of Tandem systems and the enhancement of user interaction with computer systems. Sandy was a systems analyst before joining Tandem, and also taught computer science at the junior college level.

The Performance Characteristics of Tandem NonStop Systems

In 1982, the Tandem Design Support group ran a series of experiments designed to measure the performance of NonStop™ and NonStop II™ systems (in terms of transaction throughput under PATHWAY and TMF). The purpose of this article is to look at the results that were obtained and to draw some preliminary conclusions about what the observed performance characteristics mean.

The Hardware and Software Configuration

Shown in Figure 1 is the hardware and software configuration used in the experiments. Each system had eight CPUs, eight primary paths to disc (one on each CPU), eight primary paths to X.25 high-speed lines (one on each CPU), and eight virtual circuits (terminals) per line. In each case, the PATHWAY system consisted of: one Terminal Control Process (TCP) in each CPU, controlling all the terminals on the line physically primaried to that CPU, and seven general-purpose servers in each CPU, all linked to the TCP in that CPU. The experimental systems were hard/soft configured in such a way that there was no TCP context swapping and no page faulting. Furthermore, the TCP user logic (procedure division) had merely to select the transaction types at the correct frequencies.

Our tests show a strict linearity in the relationship between CPU load and throughput.

The disc accessing was genuinely random across all eight paths. That is, although the initial processing of a given transaction (from the line handler to the TCP to the server) was handled entirely within a single CPU, disc activity for that transaction was multiplexed across all CPUs at random.

We provided enough servers (and memory) to ensure that 56 terminals could be in transaction state at the same time without forcing the TCP to wait for links to free up. This was definitely overkill because the probability that all terminals would be active at the same time at the defined arrival rates was very close to zero. In fact, even at the peak workload, there was a 95% certainty that a network of several hundred terminals generating that workload could be handled without 56 of them being in transaction state at the same time.

The priority scheme adopted for the various software components was as follows: disc process highest, line handler next highest, servers next highest, and TCPs lowest.

The Application

A simple data base and three different transactions were defined. The relative arrival rate of the three transactions remained constant (at fixed percentages of the total load), and the incoming and outgoing message size was the same for each. However, the atomic CPU (or dedicated CPU cycles) cost for each differed, as did the number of logical disc I/Os and the amount of physical (elapsed or processor-overlappable) disc time required.

Particular transaction types within any application have a profile that averages out over time, i.e. a certain number of communications I/Os of certain sizes, a certain amount of processing, and a certain amount of disc and cache I/O. (The nature of the profile depends on such things as line speed, line protocol, data base structure, number and type of file system calls, etc.) At various times during the day, the various transactions arrive at different rates. However, for any one of those time periods, it is possible to define an "average transaction" whose profile is made up of the profiles of its constituent transactions weighted by their arrival rates.

Figure 2 shows how the resource consumption of a particular transaction can be summarized; Figure 3 shows how the properties of three transactions similar to those used in our experiment can be summarized in a single average transaction; and in Figure 4, that average transaction is diagrammed. Values used are for illustration only and are not necessarily those obtained in the experiments.

The test application was developed and tuned on the NonStop II system and then transported to the NonStop system. Although two minor changes had to be made, it was possible to effect this transfer without recompiling or modifying application control files.

The Experiment

A terminal simulator running on a separate system generated messages into the PATHWAY system described above. The arrival rate of incoming messages was known and could be varied. Each incoming message started a transaction, and a reply back out to the (simulated) terminal terminated the transaction.

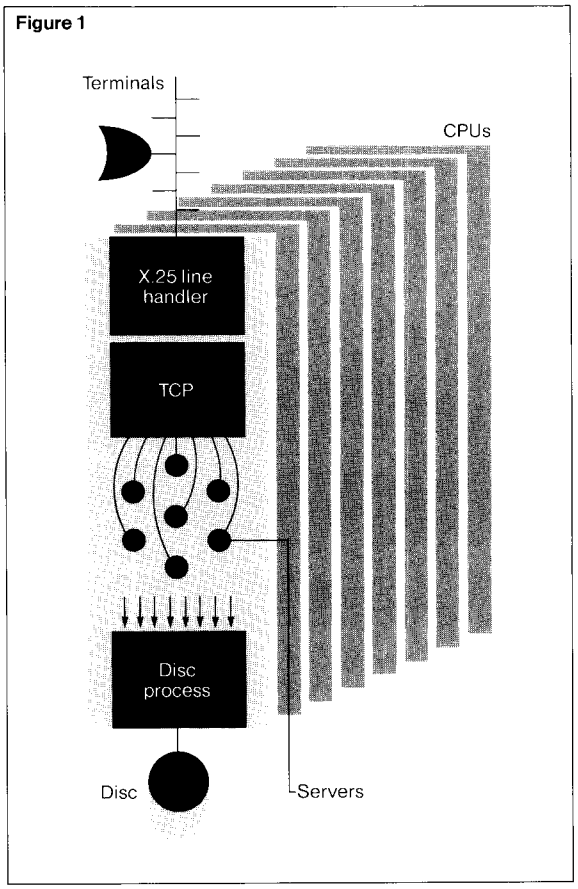


Figure 1

Hard/soft configuration used in experiments designed to measure the performance of NonStop and NonStop II systems. Seven static servers per CPU were linked to the TCP on the same CPU; each server made disc process requests randomly across all eight CPUs. One TCP per CPU controlled the terminals physically connected to the same CPU. All CPUs were hard/soft configured the same way.

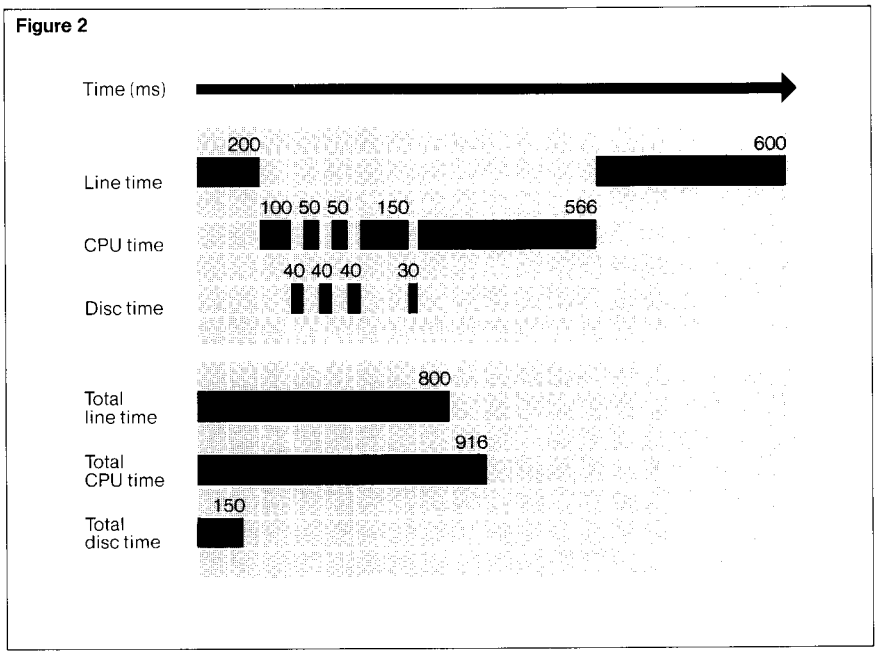


Figure 2

The average resource consumption of a particular transaction. Above: resource consumption in time sequence through the life of the transaction. Below: summary of resource consumption.

Figure 3

Transaction	Atomic CPU time (ms)	Elapsed disc time (ms)	Load
1	916	150	61%
2	458	75	26%
3	1832	300	13%
Average transaction:	916	150	100%

Atomic CPU time for the average transaction =

$$\left(\frac{\%\text{ load of tran 1} * 916}{100}\right) + \left(\frac{\%\text{ tran 2} * 458}{100}\right) + \left(\frac{\%\text{ tran 3} * 1832}{100}\right)$$

Elapsed disc time for the average transaction =

$$\left(\frac{\%\text{ load of tran 1} * 150}{100}\right) + \left(\frac{\%\text{ tran 2} * 75}{100}\right) + \left(\frac{\%\text{ tran 3} * 300}{100}\right)$$

Figure 3
Derivation of the average transaction from the actual transactions. In the experiment, there were three different transactions; the relative arrival rate of the three remained constant within the overall arrival rate. In the experiment, the incoming and outgoing message size was the same for each transaction, but the atomic CPU cost for each differed, as did the number of logical disc I/Os, and thus the amount of elapsed physical disc time. (The minimum response time = pure line time + atomic CPU time + elapsed disc time. The elapsed disc time is time that can be overlapped with processing.)

For each terminal, the full response-time distribution was maintained in the simulator. The response time was defined as the time between the first WRITE or WRITEREAD initiation of the transaction (equivalent to XMIT or first function key) and the final READ or WRITEREAD completion of the transaction (equivalent to reply being displayed, and the keyboard unlocking).

The terminal simulator imposed a variance on the selected arrival rate to replicate the real-world randomness of transaction initiation, while keeping the long-term average arrival rate at that selected. In other words, the selected arrival rate may have been 60 transactions per hour from a terminal. However, in any five-minute period, that terminal may have submitted several more (or less) than five transactions.

Each simulated terminal had the same transaction-generation rate so that, over an extended period, the same amount of communications processing was handled by each CPU. Similarly, the disc accessing was spread equally across all CPUs (i.e., over an extended period, disc process activity in each CPU was the same). Thus, the eight-CPU system was well balanced, and, as explained above, it was well tuned (e.g., all unnecessary context swapping and page faulting were eliminated). These laboratory conditions were established and carefully controlled so that all tests would be repeatable.

Results

The detailed results of this experiment have been reported in Tandem internal publications. Overall, as shown in Figure 5, we observed an approximate 22% improvement in throughput with the NonStop II system (as compared to the NonStop system) given by the relative slopes of the graphs of throughput versus CPU consumption. Raw response-time data are summarized in Figure 6; sample simulator response-time probability distributions for the average transaction at three different system loads (on the NonStop II system) are given in Figure 7;

and sample response-time probability distributions for the three transaction types at the same system load (on the NonStop II) are given in Figure 8.

In Table 1 and Figure 5, CPU utilization means total XRAY-measured CPU utilization (divided by eight), and the transaction rate is that of the total (eight-CPU) system. Response times (shown in Table 1 and Figure 6) were also given by XRAY. These response times do not include line time.

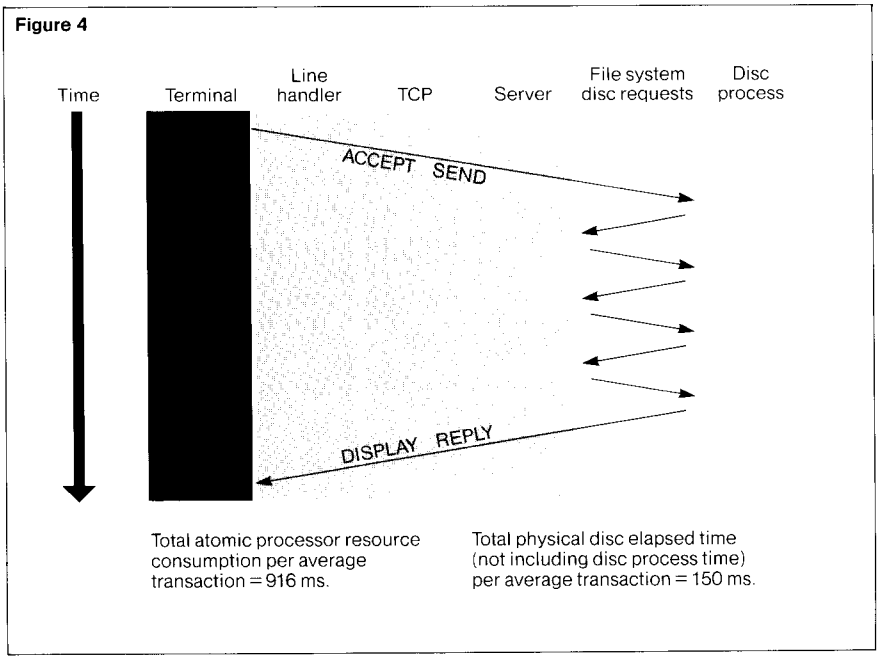
Conclusions

As mentioned above, the purpose of this article is to look at the results that were obtained for the NonStop II system (see Table 1 and Figures 4 through 7) and to draw some preliminary conclusions about what the observed performance characteristics mean.

The most fascinating aspect of the results shown in Figure 5 is the strict linearity of the relationship between CPU load and throughput. It would appear that in Tandem systems the atomic CPU cost per transaction stays the same at loads ranging from as low as 35% up to as high as 76%.

This would indicate that under controlled loads, (loads for which the various queues in the system do not exceed the memory assigned to hold them), the GUARDIAN™ operating system is extremely efficient in maintaining the real-time, multiprogramming, multiprocessing environment. It also indicates that this low-level operating system cost is small compared to the application-level costs (PATHWAY, TMF) per transaction, as one might expect.

Other experiments need to be conducted to confirm that the linear relationship (i.e., throughput proportional to CPU consumption) applies from two CPUs up through sixteen. Several less rigorous experiments and benchmarks have already suggested that this is the case.



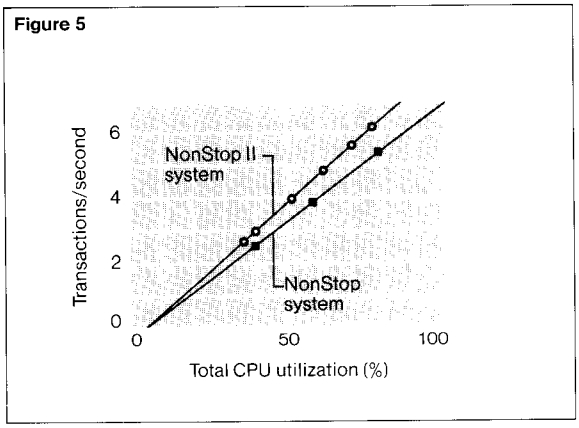


Figure 4
Graphic representation of the average transaction. Four random logical disc accesses per average transaction were multiplexed equally (over time) across all eight CPUs.

Figure 5
Measured performance of NonStop and NonStop II systems.

Figure 6

Measured response times for the NonStop II system were very close to the curve obtained by applying simple queuing theory to "average transactions."

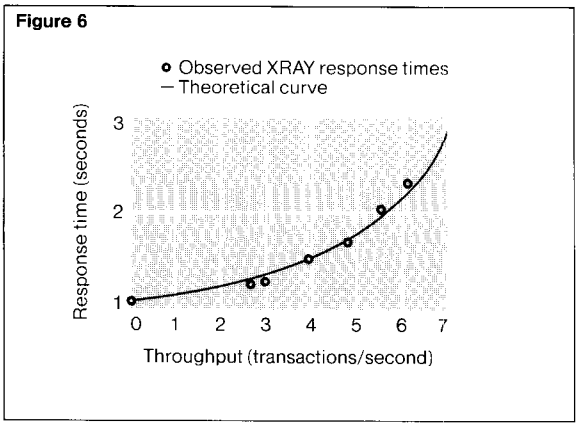


Table 1.
Experimental results for the NonStop II system.

CPU utilization (%)	Transaction rate (transactions/second)	Average transaction response time (seconds)
35.90	2.66	1.25
39.63	2.98	1.28
51.00	3.95	1.53
61.00	4.80	1.70
69.90	5.57	2.05
76.34	6.12	2.31

The fact that the plot of CPU load versus response time (Figure 6) has an easily recognizable form indicates that the response times themselves are predictable. Other properties revealed in Figures 6, 7, and 8 are as follows:

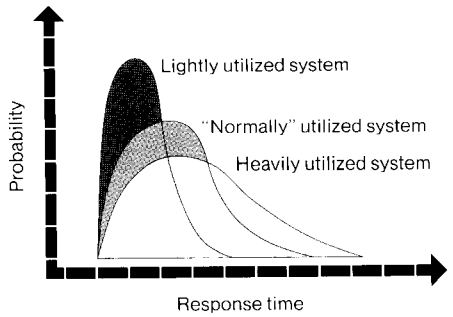
- The greater the resource consumption of a transaction, the greater its minimum response time (see Transaction 3 in Figure 8).
- The greater the resource consumption of a transaction, the greater the queuing delay experienced by that transaction under load (see Transaction 3 in Figure 8).
- The relationship between system load and the average response time for a transaction is not linear. For example, if a transaction averages a response time of one second at 45% load, it is unlikely to attain a response time of two seconds at 90% load on the same system.
- Response-time distributions are usually skewed. The average response time is usually greater than the most probable response time, and the 95% and 99% certainty response times are several times the best-case response times.

If it becomes realistic to describe applications in terms of average transaction profiles and specified arrival rates, perhaps it is also realistic to model such applications assuming laboratory conditions (i.e., that communications, disc, and processing loads are spread equally across all hard components and that the soft configuration avoids all unnecessary thrashing or processing, as in our experiment). Naturally, the throughput/response times predicted by such a model would have to be viewed as a target to aim for, not as a guarantee, because anything less than perfect balance and tuning would result in inferior performance.

Such a model would be easy to use. The only input required would be the application transaction definitions, the expected transaction rates, and the required response-time distributions. The output would be a summary of the hardware required to meet the user requirements (assuming good hardware/software design).

A second conclusion to be drawn from the experiments is suggested by the fact that, if we extend the straight line formed by the data points in Figure 5, it cuts the CPU utilization axis at about 5%. We assume that this number represents the XRAY and statistics-gathering software overhead (i.e., that in this particular case the GO-interval statistics-gathering parameters were such that the overall cost over an extended period was 5%).

Figure 7



Note that the XRAY cost does not vary with throughput. The GO interval stays the same whether the system is operating with a 10% or an 80% load.

It is not possible to balance or tune a system without XRAY, and although it is only necessary to run XRAY at certain times in the live system, most often, those will be peak-load times. Thus, the cost of running XRAY cannot be ignored. However, the experiment showed that the cost of an extremely detailed XRAY analysis of the running system, supplying all the data required to tune and balance, was minimal (and predictable).

Summary

Tandem has produced a hardware and software architecture that allows real-time business problems to be solved with real-time applications quickly, and practically. Experiments seem to indicate that given an application's transaction profiles and projected loads, Tandem will be able to predict hardware requirements for user-selected performance criteria.

Figure 8

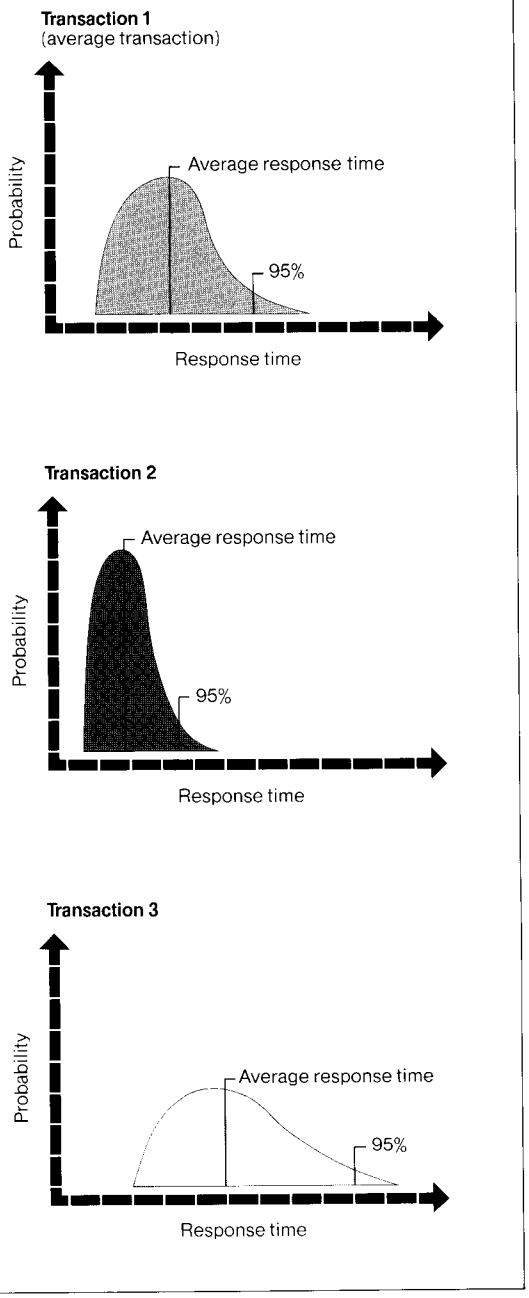


Figure 7

Sample simulator response-time probability distributions for the average transaction at three different system loads (on the NonStop II system).

Figure 8

Sample response-time probability distributions for the three transaction types at the same system load (on the NonStop II). The total area under each curve is the same and represents a probability of 1 (i.e., certainty). The average response time for a transaction is given by:

*Average response time = Σ (probability of R.T.*R.T.) where R.T. = average response time.*

Since these response-time curves are nearly always skewed, the average is not normally at the 50% cumulative probability point. The response time at which the cumulative probability for a transaction is equal to 0.95 gives the 95% certainty point for that transaction. The shape of the curves is dependent on the load on the system; the lighter the load, the taller and slimmer the curve; the heavier the load, the flatter the curve (and the longer the response times).

John Day is a member of Product Management's Performance Group. Since joining Tandem early in 1978, John has held various corporate and field positions in systems recovery, network design and systems modeling. Before joining Tandem, John was an analyst in the on-line data processing field for another major main frame manufacturer. He received an Hons. degree in Mathematics from the University of Manchester, England in 1965.

TMF and the Multi-Threaded Requester

This article describes the major considerations involved in writing a TAL multi-threaded requester that utilizes the Transaction Monitoring Facility (TMF). It assumes that the requester is to run as part of a process pair (a primary and its backup). The concepts discussed have already been addressed by Tandem and are incorporated into existing system software (namely, the PATHWAY Terminal Control Process). Therefore, this discussion is intended for analysts and system programmers who write or work with specialized system-level software rather than application programs.

The major considerations addressed include:

- The correct placement of CHECKPOINTs with regard to the TMF procedures BEGIN-TRANSACTION and ENDTRANSACTION.
- Checkpointing TFILE information.
- The establishment of restart points for each individual thread.
- The effect of backup creation on the TFILE.
- The need for an active backup process.

CHECKPOINT Placement

A TMF transaction is a multistep operation that will transform a data base from one consistent state to another. The TMF transaction is initiated by the application process when it calls the procedure BEGIN-TRANSACTION. The application process terminates the transaction with a call to either ENDTRANSACTION or ABORTTRANS-ACTION. If a failure of the application process calling BEGINTRANSACTION occurs before the call to ENDTRANSACTION, TMF will completely back out all audited data base changes made on behalf of that TRANSID. In the case of the multi-threaded requester, it is possible that multiple transactions were in progress at the time of the failure. TMF will back out each of those transactions.

Because the multi-threaded requester may issue multiple BEGINTRANSACTION calls before an intervening ABORTTRANSAC-TION or ENDTRANSACTION, the transaction pseudofile (TFILE) must be used to keep track of the various TRANSIDs involved. A process can open the TFILE, which is not a physical file, by opening the Transaction Monitor Process.

The TFILE provides a mechanism through which the requester can manipulate its current TRANSID and maintain a history of each TRANSID's completion status (for use by the backup requester in the event of a failure). When the requester successfully issues a call to BEGINTRANSACTION, a tag identifying the new TRANSID is returned to the requester process, and a new TRANSID is placed in the TFILE of the primary requester process. When the requester successfully

calls ENDTRANSACTION or ABORTTRANSACTION, the entry in the TFILE for that TRANSID is removed. Therefore, it is necessary to CHECKPOINT the TFILE to the backup process at certain strategic points. The tag returned by BEGINTRANSACTION allows the process to change its current TRANSID by passing the tag to the procedure RESUMETRANSACTION.

As mentioned, a call to BEGINTRANSACTION will place a new TRANSID in the TFILE of the primary process, and a call to ENDTRANSACTION will remove the TRANSID. When the TFILE is CHECKPOINTed after a new TRANSID has been created, the TRANSID will be placed in the backup's TFILE. A CHECKPOINT of the TFILE after the TRANSID has ended or aborted will cause the TRANSID to be removed from the backup's TFILE.

In the case of a multi-threaded requester (with a backup) that does not utilize TMF, CHECKPOINTs are placed before and after each WRITEREAD to a server. In the event of a failure, the backup requester must take over and continue the transaction until it completes. With TMF, however, if a failure occurs during a transaction, TMF will back out all audited changes made to the data base. Therefore, the backup requester is not concerned with continuing a transaction, but rather with restarting a transaction that has been backed out. In this case, the CHECKPOINT will be placed near the calls to BEGINTRANSACTION and ENDTRANSACTION or ABORTTRANSACTION.

The following examples illustrate the incorrect and correct placement of CHECKPOINTs in relation to calls to BEGINTRANSACTION and ENDTRANSACTION. Consider first the example of incorrect CHECKPOINT placement shown in Figure 1.

When the failure occurs, the backup process will take over at the last CHECKPOINT. The backup process's TFILE will have an entry for the TRANSID because of the CHECKPOINT of the TFILE. However, the transaction identified by this TRANSID has been backed out by TMF. The program will issue the SEND to the server but there is no current TRANSID. Therefore, the actions by the server will fail, the transaction will be lost (because the program will prompt the terminal for the next transaction), and the TRANSID will still occupy space in the TFILE.

Figure 2 shows the correct placement of the CHECKPOINT, just before the call to BEGINTRANSACTION. If a failure occurs at point 1, TMF will back out all audited changes to the data base made by the server, and the backup requester will take over at the CHECKPOINT. Since the TFILE was CHECKPOINTed before the call to BEGINTRANSACTION, the backup doesn't know about the transaction and can take the original transaction data and successfully call BEGINTRANSACTION to re-initiate the transaction.

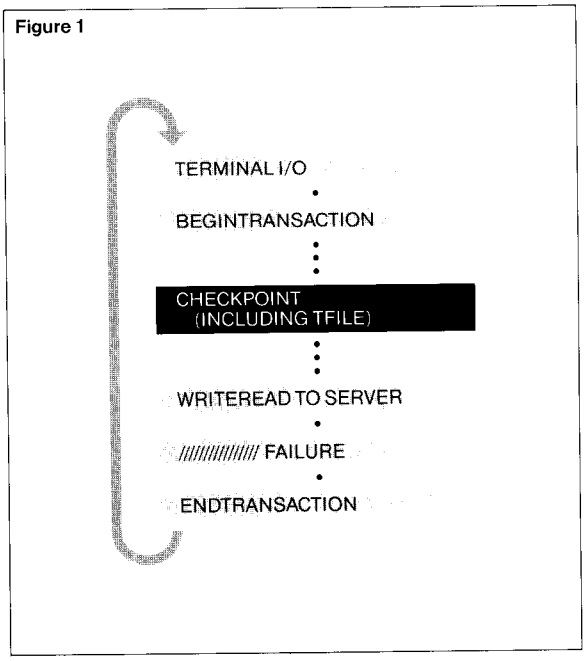


Figure 1
Incorrect CHECKPOINT after BEGINTRANSACTION.

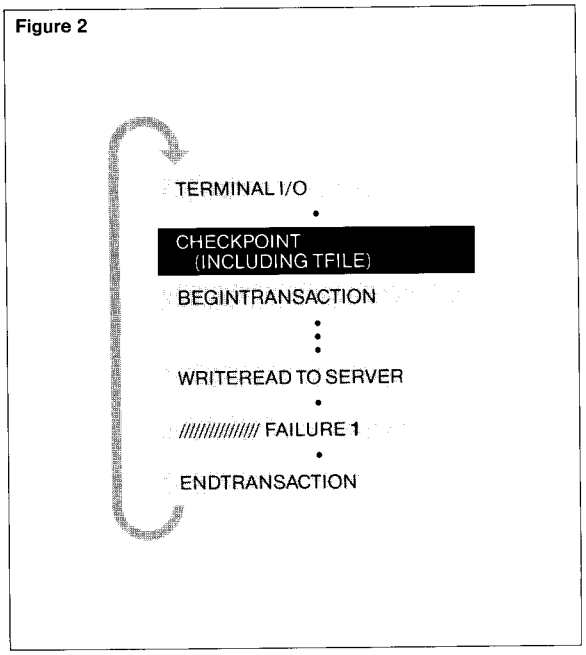


Figure 2
Correct CHECKPOINT just before BEGINTRANSACTION.

Another example of the incorrect placement of a CHECKPOINT is shown in Figure 3. If a failure occurs between ENDTRANSACTION and the CHECKPOINT, TMF will not back out the transaction because the call to ENDTRANSACTION has already been performed. The backup process will take over at the previously executed CHECKPOINT. A new TRANSID will be created by BEGINTRANSACTION, and the intended transaction will be executed twice. Also, the new primary's TFILE will still contain the old TRANSID because the CHECKPOINT after the call to ENDTRANSACTION was not performed by the failed primary requester.

Figure 4 illustrates the correct placement of the CHECKPOINT before the call to ENDTRANSACTION. Consider the actions the backup must perform upon take-over at CHECKPOINT A and CHECKPOINT B. If a failure occurs before CHECKPOINT B, the backup will take over at CHECKPOINT A. TMF will have backed out the transaction that was in progress when the failure occurred, and the backup must restore the transaction context. BEGINTRANSACTION will then be called to re-initiate the transaction.

If a failure occurs after CHECKPOINT B but before CHECKPOINT A, the backup process must determine if the previous transaction has completed, i.e., whether the failure occurred at point 2 or at point 3. If the failure was at point 2, the transaction has not completed, and the backup must restore the original transaction context so that the transaction can be re-initiated. If the failure occurred at point 3, the transaction has completed, and the backup can then prepare for a new transaction. The process must always call RESUMETRANSACTION after completing CHECKPOINT B to determine whether a failure has occurred and where that failure occurred. If an error 90 is returned by RESUMETRANSACTION, we know that a failure of the primary process occurred while a transaction was in progress. We can then set up to restart the transaction.

A failure at point 3 will cause the backup process to take over at CHECKPOINT B, but in this case the transaction has completed. The new primary requester process (i.e., the former backup) will issue a call to RESUMETRANSACTION, which will return an error not equal to 90. It will then call ENDTRANSACTION (to clear out the TFILE contents of the transaction that just completed), and the requester will then continue processing on behalf of this thread.

Consider next a failure at point 1. TMF will back out the transaction, and the backup will take over at CHECKPOINT A. At this point, the backup TFILE does not know about the TRANSID created by the primary process. Thus, BEGINTRANSACTION will be called, a new TRANSID will be created, and the transaction will proceed normally.

Figure 3
Incorrect CHECKPOINT after ENDTRANSACTION.

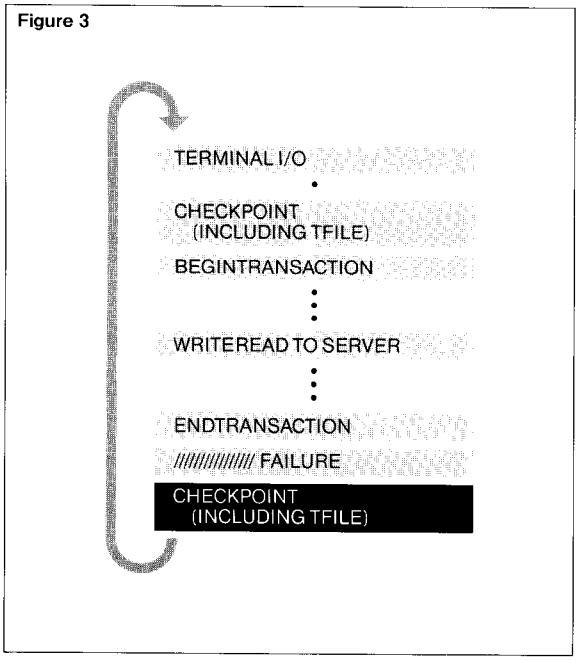
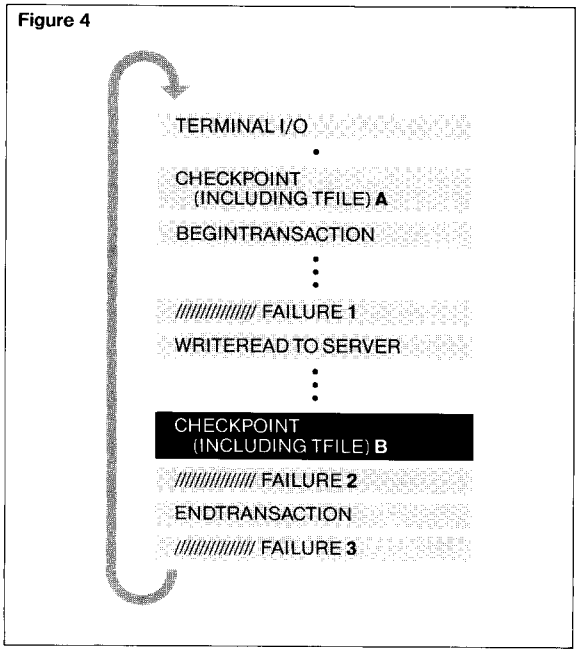


Figure 4
Correct CHECKPOINT just before ENDTRANSACTION.



If a failure occurs at point 2, again TMF will back out the transaction, and the backup process will take over at CHECKPOINT B. The requester will then call RESUMETRANSACTION, which will return an error 90, indicating that the transaction did not complete due to a failure of the primary process. The process must then call ABORTTRANSACTION to clear out the TFILE, restore the transaction context data, and return control back to BEGINTRANSACTION.

CHECKPOINTing TFILE Information

In this section we will examine the correct handling of TFILE sync information in CHECKPOINTs to the backup process.

The CHECKPOINTing of the TFILE is a very special situation, as illustrated in Figure 5. Whenever a program issues a call to CHECKPOINT, the information in the TFILE relative to the current TRANSID is sent to the backup. The checkpointing facility automatically performs a call to GETSYNCINFO for the primary's TFILE and a call to SETSYNCINFO for the backup's TFILE.

However, in the case of a multi-threaded requester, each time the requester is ready to service a thread, a call to RESUMETRANSACTION is performed to set the current TRANSID. Therefore, any CHECKPOINTs performed during the processing of the thread will CHECKPOINT the TFILE contents for that TRANSID, not the TFILE contents for the TRANSID that previously initiated or terminated a transaction. This means that the backup's TFILE for the previous TRANSID will not be up to date with the primary's TFILE, and if a failure occurs, the TFILE in the backup could potentially contain entries for transactions that were completed by the primary before the failure. The backup (now the primary) could then experience a BEGINTRANSACTION error 83 (process has begun more transactions than can be handled) because there would be no space available in the TFILE to record a new TRANSID.

The solution to this problem, shown in Figure 6, is to have each thread call GETSYNCINFO (on the TFILE) after calls to BEGINTRANSACTION, ENDTRANSACTION, or ABORTTRANSACTION, but before the next call to RESUMETRANSACTION.

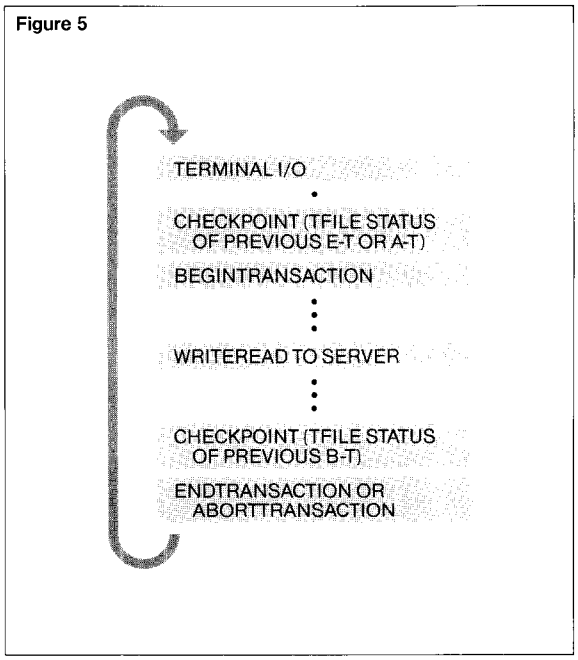


Figure 5
Incorrect omission of GETSYNCINFO and SETSYNCINFO.

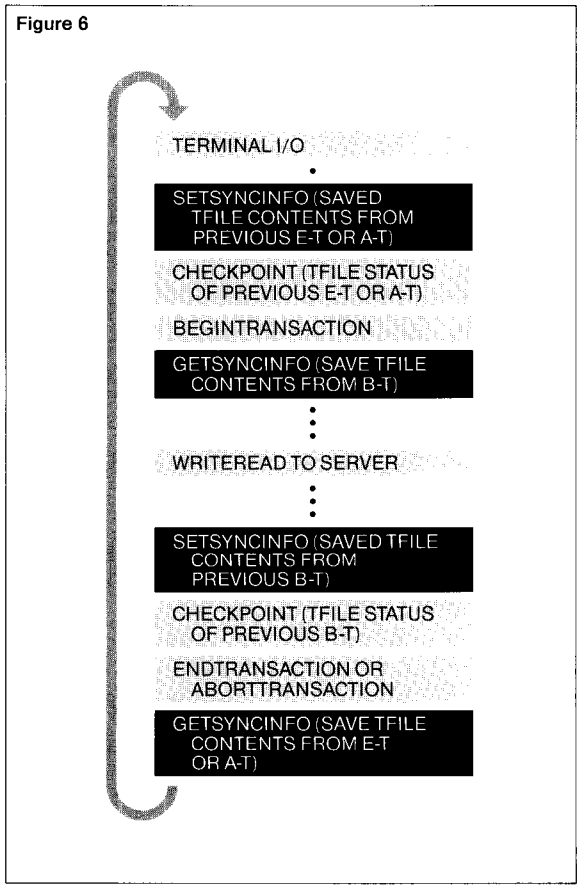


Figure 6
Correct use of GETSYNCINFO and SETSYNCINFO.

Figure 7
Incorrect WRITEs to the swap file after CHECKPOINTs.

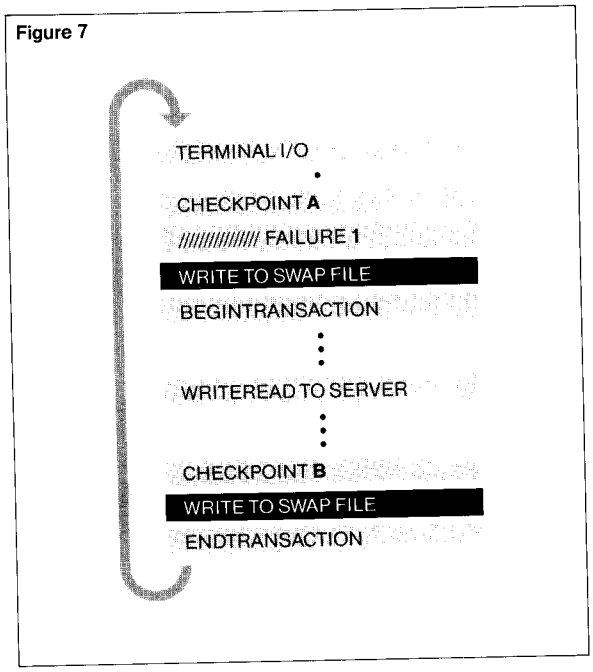
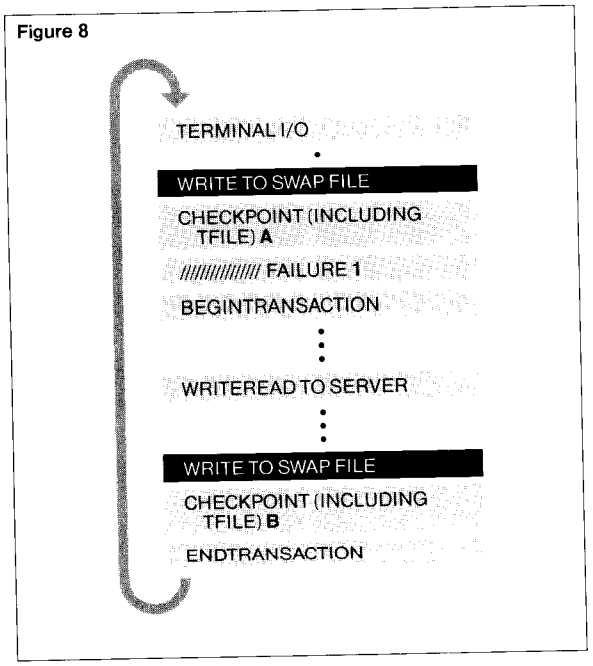


Figure 8
Correct WRITEs to the swap file before CHECKPOINTs.



Each thread must save its own TFILE information. Then, before any CHECKPOINT issued on behalf of that thread, a call to SETSYNCINFO must be performed to reinitialize the TFILE to the information that was saved by the above-mentioned call to GETSYNCINFO. The CHECKPOINT will then correctly send the information to the backup, and the backup's TFILE will be up to date with the primary's TFILE.

Establishing Restart Points for Each Thread

As the requester process is executing, each individual thread will be in a different state. For example, one thread might be waiting for a server reply, while another thread might be waiting for terminal I/O completion. The requester process must identify, for each individual thread, valid states that can be used to restart the threads in the event of a failure, and the backup process must receive this information. It is possible, when a restart point is reached for a thread, that no backup process exists. Yet, when the backup is created, it needs to know each thread's restart point. This requires that the requester safe-store the context area for each thread.

The context data that is safe-stored by the primary could be located in the primary's data area and CHECKPOINTed to the backup. However, the data area is of finite size, and such a strategy could limit the number of threads the requester could service. As an alternative, the primary requester process can safe-store context data for each thread in a disc file (swap file). Upon takeover, the backup could restart each thread from the last known restart point by reading the context data from the swap file.

The following examples illustrate the correct placement of the WRITE to the swap file in relation to the calls to CHECKPOINT.

Consider first Figure 7, in which the WRITE to the swap file is placed after the call to CHECKPOINT. If a failure occurs at point 1,

the primary process will have completed the terminal I/O, but that context information will not yet be written to the swap file. The backup will takeover at CHECKPOINT A and attempt to start a transaction. However, because the primary process has not informed the backup of the new context for this thread, the original transaction will be lost.

By placing the WRITE to the swap file before the call to CHECKPOINT as in Figure 8, you can ensure that the primary requester will have recorded the current thread's context in the swap file. The first WRITE to the swap file will record the results of the terminal I/O, and the second WRITE to the swap file will record the results of the server I/O. Now if a failure occurs at point 1, the backup process can take over, restore the context for the thread, and successfully begin the new transaction.

Figure 9 illustrates a problem that could occur with the context area of the swap file. If a failure occurs at point 1, the WRITE to the swap file at B will have overwritten the context data written to the swap file at A. TMF will back out the transaction, and the backup will take over at CHECKPOINT A and read the context area from the WRITE to the swap file at B. However, the SEND to the server has caused the original context area to be altered. It is this altered version of the context that was written to the swap file at B. When the backup requester takes over, it will resume execution at A, but with the context as it appeared at B. This inconsistency may cause the restarted transaction to behave differently.

Figure 10 provides a solution to the above problem, which involves the establishment (within the swap file) of two areas for context data for each thread. Failure 1 will cause the backup to take over at CHECKPOINT A after TMF has backed out the transaction. Because the location of the context area has been CHECKPOINTed, the backup can now read the context that was written to the swap file at A and restart the transaction normally.

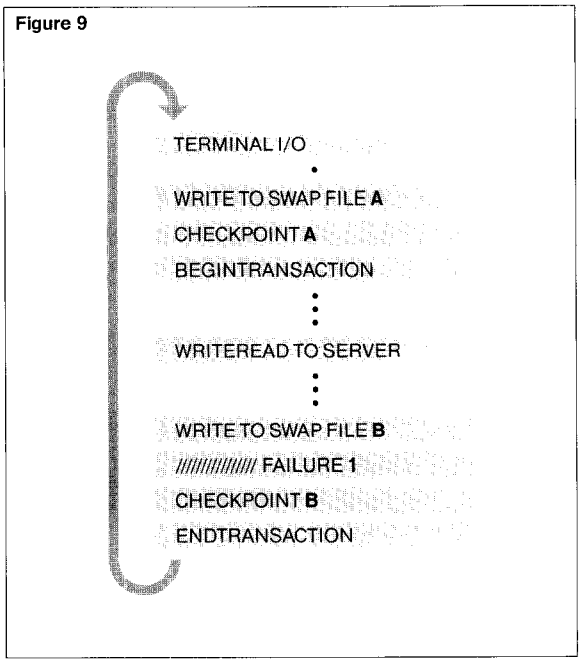


Figure 9
Incorrect WRITEs to only one area in the swap file.

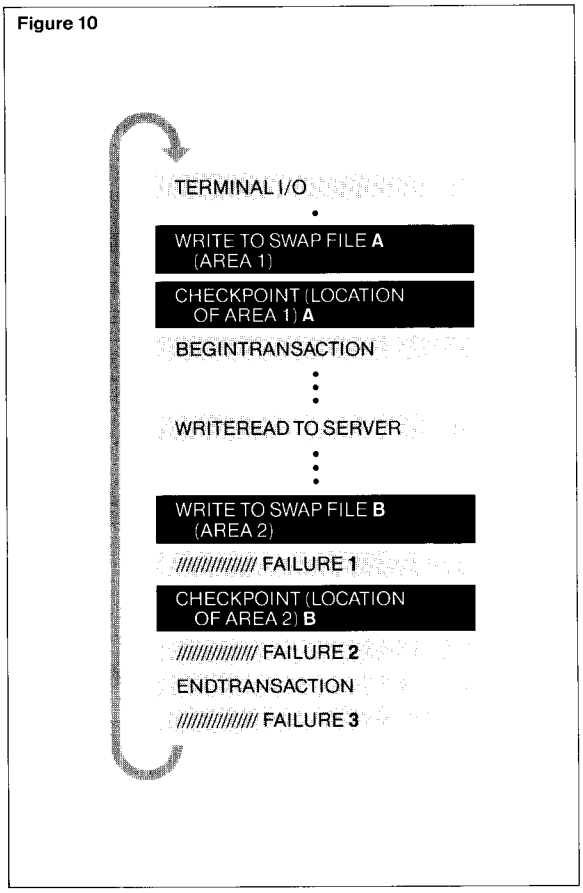


Figure 10
Correct WRITEs to two areas in the swap file.

Figure 11
Incorrect CHECKPOINT of swap file sync information.

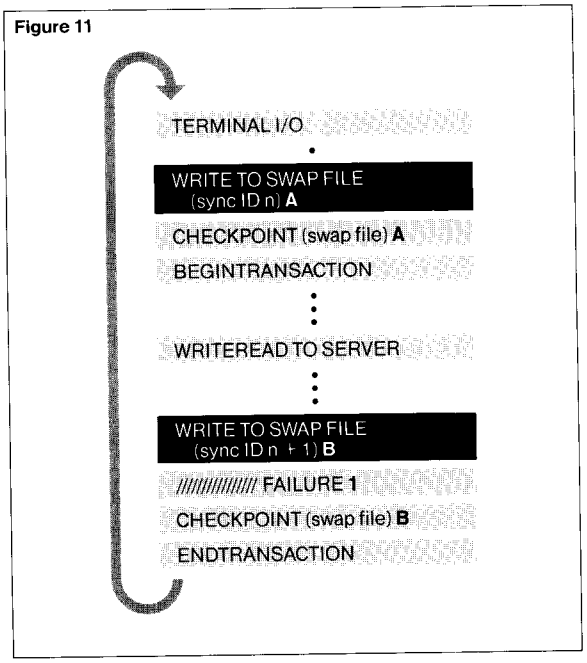
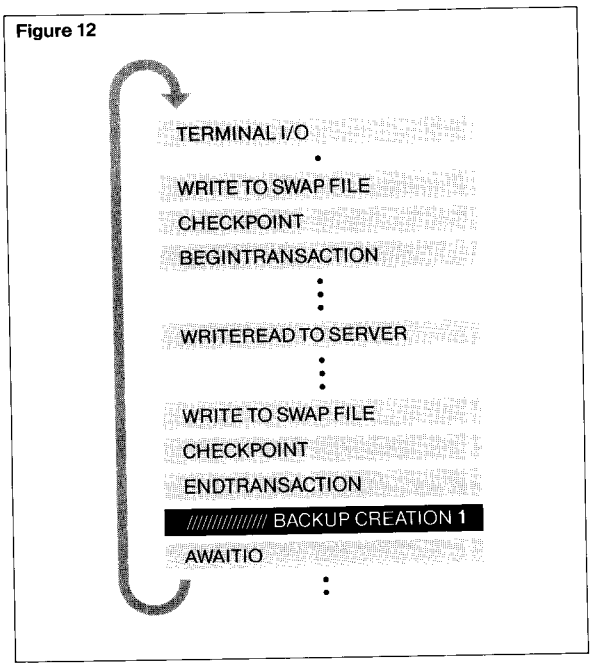


Figure 12
Incorrect backup creation when a thread is in ENDTRANSACTION state.



If a failure occurs after the ENDTRANSACTION, the backup can take over at CHECKPOINT B (the transaction has completed and been committed by TMF) and restore the context area from the WRITE to the swap file B. The requester can then prepare for a new transaction from the thread.

If the requester fails at point 2, the transaction will be backed out, and the backup will take over at CHECKPOINT B. RESUMETRANSACTION will be called, and an error 90 will be returned. The process must then call ABORTTRANSACTION to clear the TFILE, restore the transaction context data from the WRITE to the swap file at A, and return control to BEGINTRANSACTION.

One last concern related to the swap file has to do with synchronization information. The requester must open the swap file with a sync depth of 1 (so that the file system and disc process will ensure that the data from the WRITE request will, in fact, be written to the swap file), and the swap file's synchronization information may be included in the CHECKPOINT.

In Figure 11, just before the failure, the second WRITE to the swap file B has been completed, but the new sync ID has not been CHECKPOINTed to the backup. After the transaction has been backed out, the backup will take over at CHECKPOINT A. A new transaction will then be started. If another process has changed records to be accessed by this transaction, the context data from the WRITE to the swap file B will be different from that CHECKPOINTed by the old primary process before it failed. However, when the WRITE to the swap file B is executed, the file system will increment the sync ID (which is sync ID n from the last CHECKPOINT) to sync ID n+1, and the disc process will assume that this is a duplicate request. Thus, the WRITE will not be performed. This will yield incorrect context data for the thread. For this reason, the primary process should not CHECKPOINT swap file sync information to the backup process.

The Effect of Backup Creation on the TFILE

When the primary requester creates its backup, one of the functions that must be performed is the synchronization of the TFILE for the backup. This involves issuing a call to RESUMETRANSACTION followed by the CHECKPOINT of the TFILE for each thread that the requester can service. One major concern is the synchronization of the TFILE in the event of a failure and subsequent backup creation. It should be noted that if a process is using the TFILE, the call to ENDTRANSACTION is a no-waited operation, which must be completed with a call to AWAITIO. The time between calls to ENDTRANSACTION and AWAITIO is termed ENDTRANSACTION state. The primary process must not attempt to create a backup and synchronize the TFILE if any thread is in ENDTRANSACTION state.

Whenever ENDTRANSACTION is called by a process, information is sent to each CPU, which will indicate the change of state for the currently ending transaction. The BUSRECEIVE interrupt handler in each CPU takes this information and places it in the backup's TFILE if it exists in that CPU.

A problem could arise if, in the situation illustrated in Figure 12, the primary process attempts to create a backup at point 1. Two operations are occurring asynchronously. First, the state change from the call to ENDTRANSACTION is being processed, and second, the primary process is attempting to synchronize the backup's TFILE. If the primary process has not had time to CHECKOPEN the backup's TFILE before the change-of-state information is processed, that information will be thrown away, and the backup's TFILE will not be in sync with the primary's TFILE. The solution to this is to make sure that the primary process does not create a backup if any one of its threads is in ENDTRANSACTION state.

Also, if it is ready to create a backup, the primary process should not put any new threads into ENDTRANSACTION state, until all previous threads have completed ENDTRANSACTION, and a backup process has been created.

The Need for an Active Backup

The call to CHECKPOINT is a WAITed operation. This means that the process issuing the CHECKPOINT is suspended until the completion of that operation. The multi-threaded requester should only suspend activity for an individual thread when that thread is willing to suspend; the process itself should not suspend completely (this will suspend all threads) unless it has no more work to complete. Also, as the number of threads the requester is servicing increases, the number of CHECKPOINTs increases, causing the requester to suspend itself more often. This will significantly retard processing by the requester. The solution is to write an active backup process allowing the primary process to issue NOWAITed WRITEREADs to the backup process, suspend only that thread, and continue executing.

The primary process should not create a backup if any of its threads is in ENDTRANSACTION state.

Tony Lemberger is a senior systems analyst in Marketing Technical Support's Data Base Group. Since joining Tandem two and a half years ago, Tony has done research in the programming of process pairs and developed a version of the TAL Programming course. Currently, he is involved in design reviews and performance analysis.

NonStop II Memory Organization and Extended Addressing

Still maintaining a large degree of compatibility with the Tandem NonStop system, memory addressing and access on the NonStop II system have been extended to provide a logical address space of one gigabyte per processor, supported by a physical memory of up to 16 megabytes. This article describes the forms of logical addresses used by programs and how they are transformed into physical addresses in a processor.

Address Translation Requirements

All instructions, except those that initiate processor cold load or reload, reside in memory. Consequently, any meaningful instruction execution requires the processor to translate logical addresses into physical addresses, both to locate and interpret the instructions and to operate on their data. The processor performs this translation using internal registers and tables that define the correspondence between logical and physical memory addresses.

Definitions

For the purposes of this article, we will adopt the following definitions:

Physical Memory: Electronic circuitry which stores information. It is organized into words of 16 bits each, and only whole words can be read from or written to physical memory. Each processor has from 256 thousand to four million words of physical memory. It is private to the processor and not shared with any other processor in the system. A word is stored in or retrieved from physical memory when the control circuitry is presented with a 23-bit address identifying the word to be accessed. Access to individual bytes requires that software or firmware perform the operations necessary to extract or insert bytes in words.

Physical Memory Address (or "physical address"): A 23-bit number identifying a word of physical memory.

Logical Memory: Memory as perceived by a program. Depending on the instruction being executed, a program views logical memory as either words or bytes, having either 16- or 32-bit addresses. By utilizing information stored in registers dedicated to the purpose (map registers) and algorithms embodied in microcode, the memory references required for executing an instruction are transformed from logical addresses to physical addresses. The total amount of logical memory available to be shared by all the programs in a processor is 1,073,741,824 bytes (one gigabyte). This limit is independent of the amount of physical memory in the processor.

Logical Memory Address (or "logical address"): A 16- or 32-bit number used within a program to identify a word or byte of memory.

Virtual Memory: A technique for permitting the amount of logical memory currently allocated and in use to exceed the amount of physical memory in the processor. Images of the contents of logical memory are usually maintained in disc storage and brought into physical memory as required by program execution. Information no longer required for a program's execution may be returned to disc so that the physical memory can be reused.

Page: A block of memory 1024 words in length, beginning at an address that is a multiple of 1024 words.

Organization of Logical Memory

A program views memory as consisting of segments within which its instructions and data reside. It uses logical addresses to identify segments and words or bytes within them. Different types of segments and correspondingly different rules of access make it possible to share or not share segments and provide different levels of independence between the program and the memory it uses. There are three different, but related, types of segments.

Absolute Segments

Logical memory consists of up to 8192 absolute segments. Each segment consists of up to 64 pages. Absolute segments are numbered from 0 to 8191, and within a processor there can be only one absolute segment having a given number. Thus, the 8192 segments constitute a resource that must be shared by all programs (processes and interrupt handlers) that can potentially be in execution within the processor. Some absolute segments are allocated by SYSGEN to the operating system. The remainder are managed by the operating system both for its own use and to satisfy the memory needs of application processes. GUARDIAN sets the length (in pages) of segments according to the needs of the programs using them, and the length of an absolute segment will change from time to time. Unused absolute segments have a length of zero. The pages in a segment are numbered sequentially starting with 0 and continuing up to the last page in the segment. The number of a page within a segment is called the *relative page number,* since it is relative to the beginning of the segment.

Only a privileged program can address memory directly as an absolute segment. To do so it uses an *absolute extended address.* An absolute extended address consists of a 32-bit doubleword having the format illustrated in Figure 1. It is often convenient to think of bits 15 through 31 as a 17-bit byte address within the segment.

The NonStop II system provides logical address space of one gigabyte per processor.

Relative Segments

If all programs always used absolute extended addresses for all their memory references, the instructions making up a program would vary depending on which absolute segments contained its code and data. This situation, similar to that which prevailed on most computers in the 1950's, can be improved greatly by introducing a scheme of automatic "address relocation." Address relocation permits all programs to be written as if the code and data reside at the same conventional logical addresses. When the program is executed, these relative addresses are automatically translated, by the computer, to the absolute addresses actually being used.

Figure 1

Bits	Contents
0	1 (absolute bit)
1	0 (reserved)
2-14	Absolute segment number
15-20	Relative page number
21-30	Word number
31	Byte number

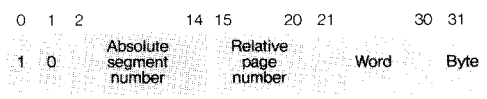


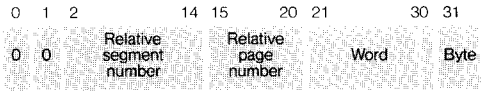
Figure 1
Absolute extended address.

Thus, on the NonStop II system, programs can view memory as relative segments. This feature permits each program to begin numbering its segments with segment number 0 and to utilize the same addresses regardless of which absolute segments the program is actually using. The maximum permitted relative segment number is 1027. The operating system assigns absolute segments to the program for its use, and the Tandem NonStop II hardware and microcode convert the relative memory references to the form needed to access the assigned absolute segments. Both privileged and non-privileged programs may use relative addresses.

Figure 2
Relative extended address.

Figure 2

Bits	Contents
0	0 (absolute bit)
1	0 (reserved)
2-14	Relative segment number
15-20	Relative page number
21-30	Word number
31	Byte number



Relative segments 0, 1, 2, and 3 always refer to particular absolute segments accessible to a program. They correspond to the program's current data, system data, current code, and user code segments. The size and boundaries of these relative segments coincide with the size and boundaries of the corresponding absolute segments.

Relative segments 4 through 1027 have a different relationship to the underlying absolute segments. Most of the time they will correspond one to one with absolute segments, but not always. The relocation mechanism permits a relative segment in this range to begin at any location within some absolute segment, and it permits the highest-numbered relative segment used (greater than three) to end at any byte. Thus, that highest numbered relative segment need not contain an integral number of pages. These segments constitute the program's current "extended data segment," which is discussed in the next section.

All relative segments can be addressed with 32-bit *relative extended addresses.* A relative extended address is identical to an absolute extended address, except that bit 0 contains 0 and the segment number field contains a relative segment number instead of an absolute segment number. The format is illustrated in Figure 2. In this case also, bits 15 through 31 can be thought of as a 17-bit byte address within the segment.

As mentioned above, the first four relative segments (0, 1, 2, and 3) correspond to the program's current data, system data, current code, and user code segments. A program can address these relative segments using either relative extended addresses or 16-bit addresses (which are also relative). The 16-bit addresses are essentially the same as 16-bit addresses on the Tandem NonStop system. The Instruction Processing Unit (IPU) interprets 16-bit addresses depending upon the address use (i.e., code access or data access), Environment Register bits, and instruction word. To the extent that the two systems have code compatibility, they interpret 16-bit addresses identically.

Depending on the value of the PRIV, DS, CS, and LS bits in the ENV register, a program may be able to access up to four code segments (user code, user library, system code, and system code extension) and two data segments (user data and system data), but not all at the same time. No more than four of these segments can ever be accessible simultaneously, using either 16-bit addresses or relative extended addresses.

The segments accessible via 16-bit addresses are selected as follows:

- *Current data (relative segment 0):* Normal program data references are in this segment. It is either user data or system data, depending on the DS bit in the ENV register (0: user data, 1: system data). In the NonStop II kernel, only interrupt handlers execute with the DS bit set to 1.

- *System data (relative segment 1):* Privileged programs can execute instructions that access system data regardless of the setting of the DS bit. Non-privileged programs executing the same instructions access current data (with no error indication).

- *Current code (relative segment 2):* A program's instructions always come from the current code segment, which may be user code, user library, system code, or system code extension, depending on the values of the CS and LS bits in the ENV register. The selection rule is:

CS	LS	Code Segment Used
0	0	user code
0	1	user library
1	0	system code
1	1	system code extension

- *User code (relative segment 3):* In order to permit procedures executing in other code segments access to read-only arrays stored in the user code segment, certain non-privileged instructions are able to specify the user code segment as a data source, using 16-bit addresses.

Note: If the DS bit is a 1, relative segments 0 and 1 both refer to the same absolute segment (system data). If CS and LS are both 0, relative segments 2 and 3 both refer to the same absolute segment (user code).

Using relative extended addresses to access relative segments 0, 1, 2, and 3 is an alternative to addressing the same memory locations using 16-bit addresses. A 16-bit address is always faster. Often there is no compensating advantage to be gained by using the extended address. Occasionally, however, the features listed below make extended addressing preferable.

- Uniform byte addressing over the whole segment (instead of the restricted byte addressing available using 16-bit addresses, and the restricted range of SG mode addressing).

- Ability to use the entire set of extended address instructions with any segment. (Note, however, that an attempt to store into relative segments 2 or 3 will fail with an instruction failure.)

Neither 16-bit addresses nor relative extended addresses permit a program to read a code segment other than user code or the current code segment at any given instant; therefore, at least two (and often three) code segments are inaccessible via these types of address. All four code segments can be accessed at any given time, however, by using absolute extended addresses.

Extended Data Segments

Relative extended addresses in relative segments four and above are always within an *extended data segment.* An extended data segment is a contiguous block of logical memory having a length of from 1 to 134,217,728 bytes. Its boundaries need not be on either absolute segment or page boundaries. It may consist of part of an absolute segment or up to 1024 full absolute segments. If it contains more than one absolute segment, however, the absolute segments must be contiguous.

With absolute extended addresses, all four code segments can be accessed at any given time.

A process may have many extended data segments defined and allocated at any instant, but it can access only one at a time, since the segment relocation mechanism provides for only one base address. Since there are only 8192 absolute data segments in a processor's logical memory, a process that allocates several very large extended data segments may reduce the number of absolute data segments available to other processes in the same processor to a level that interferes with their operation. Excessive use of large extended data segments can interfere with GUARDIAN operations in other ways as well.

GUARDIAN provides facilities allowing processes to allocate, deallocate, and identify extended data segments. It also provides facilities for switching from one to another. When a program uses a relative extended address having a value of %2000000 or more (segment four and above), it refers to the currently selected extended data segment. Extended data segments can be accessed only via extended addresses; there is no way to access them via 16-bit addresses.

The terminology can be somewhat confusing. We have defined three different kinds of segments. They are summarized below.

1. *Absolute Segment:* A block of memory containing 0 to 64 pages. An absolute segment always contains some integral number of whole pages. Each processor in a system can have up to 8192 absolute segments.
2. *Relative Segment:* A block of memory containing 1 to 131,072 bytes. The use of relative segments permits programs to be written without concern for the identity or allocation of the absolute segments they utilize. Each process can access up to 1028 relative segments at a given time.
3. *Extended Data Segment:* A block of memory containing 1 to 134,217,728 bytes. An extended data segment begins at relative segment 4, byte 0, and extends through however many relative segments are needed to make up its total size. Extended data segments are composed of one or more relative segments. They provide a way for a program to organize and manage large blocks of data storage.

Translating Logical Addresses into Physical Addresses

When an IPU needs to read its next instruction or to read or write data in memory, it starts with a logical memory address which must be transformed into a physical memory address. The transformation is performed in two stages. The IPU converts the logical address, 16-bit or 32-bit, to a mapped address. The memory mapping unit then converts the mapped address into a physical memory address.

Memory Mapping

All access to physical memory, both for the IPU and the I/O channel, is performed via memory mapping registers. The GUARDIAN Memory Manager (or SYSGEN) assigns pages of physical memory to an absolute segment as needed for program execution. They may or may not be contiguous in physical memory, but by converting the relative page number (within a segment) to a physical page number (in physical memory), the mapping unit makes them appear, to a program, to be a contiguous block of memory.

The memory mapping unit consists of 16 *maps*. A map is a set of 64 registers capable of containing the 64 physical page numbers that would be required by a segment of maximum size. The registers within a map are numbered from 0 to 63 and correspond to the relative page numbers within a segment. The maps, like the physical memory, deal only with word addresses, not bytes. The IPU, or the channel, presents the memory mapping circuitry with a four-bit map number and a 16-bit word address. The map number picks one of the sixteen maps. The leftmost six bits of the 16-bit address are used to select one of the 64 registers within the map. This register then supplies 13 bits (a physical page number) which are joined with the rightmost ten bits of the original address to form the 23-bit physical address.

Regardless of its original form, the logical address must, in the final stage of translation, go through this mapping step. The combination of the map number and 16-bit word address is a *mapped address*. When the physical page numbers corresponding to a particular absolute segment are contained in some map, the segment is said to be "mapped."

The remainder of this article describes how the various forms of logical addresses are transformed into mapped addresses.

Translating an Absolute Extended Address to a Mapped Address

Since there are 8192 absolute segments, each of which could require a map to identify its physical pages, and there are only 16 maps, we must have ways of managing the use of maps, and we must be prepared for the case in which a segment is not mapped. In general, therefore, to translate an absolute extended address to a mapped address we must determine whether the segment is mapped, and if so, by which map. If the page we wish to access is not mapped, then its absolute page number must be placed into a map register before the memory access can be completed.

To keep track of whether each absolute segment is mapped, and if so by which map, each processor has a table of 8192 elements, one for each segment. This table, called the *Segment Table,* is stored in memory. To access other parts of memory, we must first be able to read the Segment Table. Thus, it must remain mapped at all times. For reasons explained later, the amount of information we need to keep about each segment is sufficient to fill two words. Therefore, the Segment Table is defined to consist of 8192 doubleword entries, and can thus occupy up to 16 pages of memory. So that we can always reach it when we need to, 16 registers from map 14 are reserved for mapping this table. (Registers 28 through 43 in map 14 are used for this purpose.)

If an absolute segment is mapped, its entry in the Segment Table will tell us which map to use. Thus, we can give the memory mapping unit the correct map number and word address. If it is not mapped, however, we need to do two things:

1. Find that physical page that corresponds to the logical page we are trying to access.
2. Find at least one map register in which we can load this physical page number in order to execute the mapped memory access.

To satisfy our first need, we could have made each entry in the Segment Table long enough to store 64 physical page numbers so that we would have the information we need immediately at hand. Since many absolute segments are quite small, this solution would entail inefficient use of both physical storage and map registers. Instead, the Segment Table contains the address (in some other part of memory) where this information is to be found. Each absolute segment of length greater than zero has, somewhere in memory, a *Page Table* which contains the physical page number corresponding to each of its relative pages (if physical pages have been assigned to the relative pages). The segment's entry in the Segment Table tells us how to find it. The format of a Segment Table entry is illustrated in Figure 3.

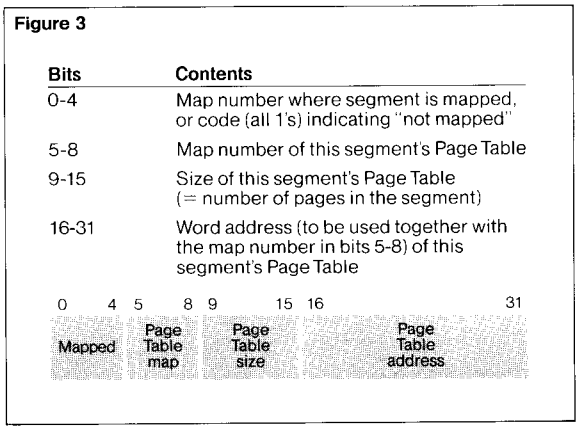


Figure 3
Segment Table entry.

Each segment of length greater than zero has a Page Table. A Page Table consists of an array of words, one for each page in the absolute segment. Each entry in the Page Table contains the information that must appear in the corresponding register of a map when the segment is mapped. Thus, a segment is mapped by moving the contents of its Page Table to a map and setting any map registers not used to indicate the page is not present.

The memory in which the Page Tables are kept must, of course, be continuously mapped. In principle any map could be used, but the conventional assignments lead to the Page Tables being stored in segments mapped by maps 6 through 13. GUARDIAN assures that the memory containing the Page Tables is always accessible.

A Page Table entry contains, in addition to the 13-bit physical page number, three bits used by the hardware and the Memory Manager to implement virtual storage. (The maps serve a dual role. They not only supply the upper 13 bits of a physical memory address, but also assist in virtual memory bookkeeping, keeping a record of whether a page is present or absent and whether it has been read from or written to by the IPU.) The format of a Page Table entry is illustrated in Figure 4.

To summarize the steps to this point, from an unmapped segment, the microcode examines the Segment Table and locates the Page Table for the segment. It has the physical page number of the page it needs, but has not selected a map in which to load it.

The allocation and control of maps 0 through 13 have been left to SYSGEN and GUARDIAN. A single map, map 15, is reserved to the IPU microcode for the exclusive purpose of giving access to pages in unmapped segments.

It would have been possible to respond to any reference to an unmapped absolute segment by mapping the entire segment in map 15 and then proceeding with the memory access in the normal way. Instead, however, the microcode maps only a single page at a time. This strategy speeds up this type of memory access and permits individual pages from different absolute segments to be mapped simultaneously. The technique used results in map 15 being used as a *Map Cache.*

Figure 4
Page Table entry.

Figure 4

Bits	Contents
0-12	Physical page number
13	Referenced bit
14	Dirty bit (written into)
15	Absent bit

One of the problems which must be solved in any cache management scheme is keeping track of the contents of the cache. In the case of the Map Cache, we must know the absolute segment number and the relative page number within the segment for each page that is mapped. The solution adopted for the Tandem NonStop II system involves using only half of map 15 for actually mapping memory and the remainder as memory space to keep track of the contents of the "working" half. Thus, in map 15, registers 0 through 31 contain normal map entries (Page Table entries) while registers 32 through 63 contain "tags"; register 32 containing the tag for register 0, register 33 containing the tag for register 1, etc.

Tags must be designed so that they fit within a map register (16 bits). Since an absolute segment number requires 13 bits and a relative page number requires 6, a map register cannot contain a number large enough to uniquely identify a given page of logical memory. Instead a kind of "half-map" register assignment is used in which the Page Table entry is placed into its corresponding map register, modulo 32. That is, map 15, register 0 maps relative pages 0 or 32, register 1 maps relative pages 1 or 33, and so on. Then the tag in the corresponding register in the upper half need only contain the absolute segment number plus one bit to indicate whether the relative page number actually mapped is less than 32, or 32 and up.

This scheme has two advantages:

1. The first word of an absolute extended address contains exactly the right information needed for the tag.
2. No searching is needed to determine whether a page is mapped. Bits 16 through 20 of an absolute extended address can be used as an index to select both the tag word for checking, and the map entry for the corresponding memory access. If a page is mapped, it will be mapped only in that one register in the lower half of map 15, and its tag will be in the corresponding register in the upper half of map 15.

The tag actually used in the Map Cache is identical to the first word of the absolute extended address except that bit 0 contains 0.

The technique of indexing into the two halves of the Map Cache is very important to the speed of operation of the IPU. In the actual implementation of extended addressing the first word of an absolute extended address (with bit 0 cleared) is compared to the appropriate tag word in the Map Cache before checking the Segment Table to see if the whole segment is mapped. If the page is not in the Map Cache, then the Segment Table is checked to see whether it is mapped in one of the other maps. Since a map register can be accessed (for comparison) much faster than a word in memory, the fastest access to memory, using an extended address, is to a page already mapped in the Map Cache.

A summary of the steps required to translate an absolute extended address to a mapped address is given below.

1. Use bits 16 through 20 of the extended address as an index into map 15. Call the index *i*.
2. Compare map 15, register i+32 to the first word of the extended address (with bit 0 cleared).
3. If the comparison shows the values are equal, map 15, register i contains the correct Page Table entry for this extended address. The mapped address consists of a map number of 15 and a word address taken from bits 16 through 30 of the extended address. (The first bit of the word address is 0.)
4. If the comparison shows the values are not equal, perform the remaining steps.
5. Using the segment number as an index, look up the correct entry in the Segment Table for this segment. (The table begins at word %70000 in map 14.)
6. If bits 0 through 4 of the Segment Table entry contain a map number, this segment is mapped. Use this map number and the word address (bits 15 through 30) from the extended address to provide the mapped address.
7. If bits 0 through 4 of the Segment Table entry contain all ones (%37), this segment is not mapped. Perform the remaining steps.
8. Read the Page Table entry from memory. (Use the relative page number from the extended address as an index into the segment's Page Table. The mapped address of the Page Table consists of the Page Table map and the Page Table address from the Segment Table entry. See Figure 3.)
9. If map 15, register i+32, contains all 1's, the position in the Map Cache is free. Place the Page Table entry in map 15, register i, and the correct tag in map 15, register i+32. The page is now mapped. Complete the memory access as in step three.
10. If map 15, register i+32, contains a valid tag for some other page, locate the Page Table for that segment and store the contents of map 15, register i, into the correct entry. Then the position in Map Cache is free; proceed as in step nine.

The fastest access to memory, using an extended address, is to a page already mapped in the Map Cache.

Converting a 16-bit Address to a Mapped Address

For 16-bit logical addresses, selecting the correct map is simple. The first six maps have been assigned specific functions corresponding to the memory segments normally used by a process. These assignments are:

Map	Segment
0	user data
1	system data
2	user code
3	system code
4	user library
5	system code extension

It is a responsibility of GUARDIAN to assure that whenever a process is allowed to run, these six maps contain the correct physical page numbers for that process. In practice all processes share the code and data mapped by maps 1, 3, and 5. The contents of maps 0, 2, and 4 may have to be changed when a process becomes active. Which maps are used is determined by the values of the CS, DS, LS, and PRIV bits of the Environment Register at the time an instruction is executed. (See the *Nonstop II System Description Manual.)*

Translating a Relative Extended Address to a Mapped Address

Relative extended addresses in relative segments 0 through 3 are converted to mapped memory addresses as follows:

- Bits 15 through 30 are used as a word address.
- A map is selected according to the table shown in Figure 5.

If the relative segment number is four or above, the IPU converts the relative extended address to an absolute extended address by adding a 32-bit base value to it. The resulting absolute extended address is then translated to a mapped address as described previously.

The base value is a 32-bit quantity kept in map 14, registers 60 and 61. Since a full 32-bit addition is performed, the relocation base can be at any byte address in any absolute segment. For the typical extended data segment the relocation base is set to the beginning of an absolute segment (the first in the extended segment).

Before adding the base, the IPU performs a bounds check on a relative extended address to detect whether a program is trying to go beyond the length of the extended data segment currently in use. Map 14, registers 62 and 63, contain a 32-bit number which, if added to a relative extended address too large to fit in the extended data segment, will produce a carry out of bit 0. If this check produces the carry, the IPU invokes the instruction failure interrupt handler.

Figure 5

Relative segment number	DS	CS	LS	PRIV	Map number	
0	0	-	-	-	0	User data
	1	-	-	-	1	System data
1	1	-	-	-	1	System data
	-	-	-	1	1	System data
	0	-	-	0	0	User data
2	-	0	0	-	2	User code
	-	1	0	-	3	System code
	-	0	1	-	4	User library
	-	1	1	-	5	System code extension
3	-	-	-	-	2	User code

Figure 5
Map Selection Table.

Access to Single Bytes

Both the physical memory and the memory maps use word addresses. When the IPU executes an instruction that reads a single byte from memory, it must read the whole word from memory and select the correct byte from the word. When it executes an instruction that writes a single byte into memory, it must first read the whole word, modify the byte within the word, and then write the whole word back into memory. These functions are performed in the IPU microcode.

Maps 6 through 13

SYSGEN and GUARDIAN are free to use maps 6 through 13 for whatever purposes will best serve the needs of system operation. The IPU contains no built-in assumptions about their assignment or use.

In practice they are used to map memory which must be kept continuously mapped for long periods of time. Primarily this memory serves two purposes:

1. It is used to store Page Tables for absolute segments. (Recall that the Page Tables must be readily accessible for the Map Cache to work properly.)
2. It is used for I/O buffers. (The I/O channel employs mapped addresses for its memory accesses. Rather than use the limited resources of system data space, GUARDIAN uses maps 6 through 13 to map the buffers.)

Dick Thomas, manager of the System Software Group, Marketing Technical Support, joined Tandem in June, 1981, as a systems analyst. He developed and taught Technical Problem Solving for Analysts and has since been involved in GUARDIAN operating system research and teaching activities. Dick has had a lengthy and varied career in the computer industry, including system design, programming, and management in both applications and systems software.

Please enroll me as a subscriber to the *Tandem Journal* and the *Tandem Application Monograph Series*. I understand that for $100.00 per year, I will receive four issues of the *Tandem Journal* and a copy of each *Application Monograph* published in the subscription year (at least six). Make check payable to Tandem Computers.

New subscription ☐ Renewal ☐ Invoice ☐ Paid ☐

Subscription questions should be referred to your local Tandem sales office.

Billing address

Company name

Address

Country

Attention

Shipping address

Your name

Company name

Address

Country

If purchase order is enclosed:

P.O. no.

P.O. date

☐☐/☐☐/☐☐

TANDEM

NonStop™ Computer Systems