Technical Report

UCAM-CL-TR-14
ISSN 1476-2986

Number 14

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Store to store swapping for TSO under OS/MVT

J.S. Powers

June 1980

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

SUMMARY

A system of store-to-store swapping incorporated into TSO on the Cambridge IBM 370/165 is described. Unoccupied store in the dynamic area is used as the first stage of a two-stage backing store for swapping time-sharing sessions; a fixed-head disc provides the second stage. The performance and costs of the system are evaluated.

CONTENTS

FIGURES

1. INTRODUCTION

When the IBM 370/165 was installed in January 1972 the Computing Service planned to provide time-sharing facilities based on the locally produced Phoenix command language; the hope was that IBM's Time-Sharing Option (TSO) would provide adequate support for Phoenix in the way of software for swapping, session management and so forth. Experience soon curbed this optimism. The first time-sharing service, supporting a maximum of eight simultaneous sessions, was introduced in January 1973. After the installation of a second megabyte of memory and a 2305-II fixed-head disc with its own block multiplexor channel, up to 34 simultaneous sessions were supported. By September 1975 the limit had been raised to 64 simultaneous sessions; this was made possible by local changes to software, including major replacements for the IBM mechanisms for disc space management and dynamic allocation. (Technical Reports 3 and 4 describe this new software.) At this point the addressing limit of the communications hardware had been reached, and it was only the introduction of local communications software at the end of 1975 that made it possible to circumvent this limit. (This software is described in Technical Report 5.) In December 1975 the maximum was raised to 80 sessions. A year later, after the installation of a third megabyte of memory and in response to greater demand, the session limit was raised to 90.

There now seemed to be little hope of providing for further growth merely by improving the efficiency of software components; quite simply, the cost of swapping sessions in and out of store already led to frustratingly bad response times when the number of simultaneous sessions was near the limit. Yet there was a need for the session limit to be raised further.

The absence of any kind of mapped or virtual memory in the 370/165 imposes the chief limitation on the performance of the swapping system. As a result of this architectural feature, every data structure or program is bound to the position in memory in which it is first created or loaded. This implies that a time-sharing session must throughout its life execute in that region in which it is established at logon. Hence the store for sessions must be allocated in one or more fixed regions. The Computing Service currently allows five such time-sharing user regions. In effect, from the point of view of swapping and related contention the time-sharing system is equivalent to five single-region machines.

Increasing the number of time-sharing user regions would reduce the number of sessions competing for each region and increase the multi-programming level for time-sharing. Unfortunately the small number (16) of storage protection keys available on a 370/165 imposes a limitation. Each time-sharing user region must have its own protection key, and under present conditions on the Cambridge 370/165 only five keys can be spared.

The major disadvantage of the fixed region organisation is that time spent in swapping is time lost from the use of the region. One would like to keep a pool of sessions ready to run, and have the swapping I/O proceed as a parallel activity. The use of fixed regions makes this impossible;

from the start of a swap-out for a given region up to the completion of the next swap-in, the region is unavailable for useful work and the multi-programming capacity of TSO is thereby reduced. This need not be a great embarrassment if the amount of time spent in swapping is very small. However measurements have shown that the swap time is significant.

At this point two definitions are in order. The term 'swap time' is used to denote the time spent in swapping a session one way, either out or in; it is measured from when the session is queued for the swap up to when the swap is complete. The time a session spends in a region between swap-in and swap-out is called the 'residence time'.

The swapping device on the 370/165 is a 2305-II fixed-head disc. Measurements made under TSO before the introduction of store-to-store swapping, when all swapping was onto the fixed-head disc, showed that the mean swap time was 30 msec. Hence the swap time for a change of session in a region was on average 60 msec.

The mean residence time on the former system used to be about 600 msec; under the current system, with store-to-store swapping, the mean is smaller, becoming as low as 300 msec at busy times. However even this value is larger than might be expected. Three factors contributing to the size of this mean are common to the current and former systems. First there is the effect of multi-programming; for part of a period of residence in its region a session does nothing. Second, while any disc input or output is in progress for a session, the session must stay in its region; this feature of the IBM software would be very difficult to change within the constraints of the real-memory architecture of the 370/165. Finally, the distribution of residence times is highly skewed and the mean is far greater than the median. The distribution is related to that for transactions illustrated in Figure 4.

The mean swap time, then, used to be about 10% of mean residence time. More significant was the relationship that swap time bore to transaction time. A transaction is here taken to be the career of a session from its becoming ready to run after waiting for terminal input or output up to its next return to a wait state for terminal I/O. By the execution time of a transaction we shall mean the sum of the cpu time and disc I/O time used by the transaction. It turns out that about 60% of transactions have execution times less than 60 msec. Furthermore some transactions, even short ones, have more than one period of residence and so involve more than one swap cycle. The current mean is about 1.75 swap cycles per transaction; formerly the value was somewhat lower, though it was still greater than 1. Hence formerly over 60% of transactions had execution times shorter than the time they spent in swapping.

The restricted amount of space on the fixed-head disc presented another obstacle to the raising of the session limit. Up to 89 sessions could be held in the swap file on the fixed-head disc; beyond this number, sessions were swapped onto a back-up swap file on a 3330 moving-head disc. On occasions when the fixed-head disc was broken and all swapping was onto a moving-head disc, mean one-way swap times were measured at 80 to 100 msec. In addition to the swap file the fixed-head disc holds some HASP and OS overlays and other system components; the swap file could not be extended

without detriment to other parts of the system. Hence performance would deteriorate at a higher rate than hitherto if the number of simultaneous sessions rose above 90.

Clearly there was scope for improvement. However this was unlikely to come from changes in hardware. Neither the fixed-head disc nor its channel were overloaded, and the transfer time for disc swaps could not have been cut substantially. Furthermore the cost of additional hardware was prohibitive. The solution lay elsewhere.

There is a further important consequence of the real-memory architecture of the 370/165, namely the inefficiency of the use of store in the dynamic area (the area of store in which offline jobs run). Offline jobs are not swapped, nor can they be moved around in store during execution; the result is storage fragmentation. Formerly quite large contiguous blocks of store remained unused for periods of 10 seconds and more; the total amount of store unused at any one time could exceed 1 Mbyte. This was inevitable given the high level of multi-programming and the heterogeneous and unpredictable job mix supported on the Cambridge 370/165. Thus for much of the time there was unused store sufficient to hold many time-sharing sessions (the mean session size was, and still is, about 22 Kbytes). Furthermore this unused store consisted of blocks long-lived by comparison with the swapping rate (currently about ten swaps-out a second across all regions for heavy loads).

It was realised that this store could be used for the first stage of swapping, constituting as it were the middle level of a three-level storage hierarchy. A store-to-store move is a relatively fast operation: to move 22 Kbytes within the main memory of the 370/165 takes about 2.5 msec, in contrast with the mean swap time between store and disc of 30 msec. In such a system, as many as possible of the swaps to and from regions should be from and to store, that is, should be store-to-store swaps. Some swapped sessions would have to migrate to disc; these could be brought back into store in anticipation of swaps into their regions.

In 1978 the memory of the 370/165 was expanded to 4 Mbytes and a speed-up feature was installed which gave an increase in cpu speed of up to 20%. With the additional capacity now available, a store-to-store swapping system was implemented at Cambridge at the beginning of 1979. The rest of this document describes the system and evaluates its effectiveness. Section 2 gives some background information on system software before the introduction of store-to-store swapping.

Some notes on terminology are included here; a glossary will be found in Appendix A. To each user logged onto the system there corresponds an active session; a ready session (at a given moment) is an active session which is ready to execute or is executing, that is, a session which is not in a wait state. To observe the distinction between task and routine on all occasions would lead to prolixity; where no confusion might result the distinction is blurred. The verb 'migrate' is sometimes treated in this document as transitive in the active voice; the excuse is that here it is jargon.

2. BACKGROUND

Early versions of OS/MVT had no time-sharing facilities. TSO was added later, and so is not an integral part of the operating system but rather, like HASP, a permanently running job with a complex task structure. To TSO belong the time-sharing user regions (currently five of them) and the time-sharing control region, which contains most of TSO's routines and resident data structures.

The top level of control of TSO is vested in the time-sharing control task. The main job of this task is to handle swapping activity; the routine used for this will be called the swapper in this document. The time-sharing control task's other activities are not relevant to this discussion. The time-sharing control task begets region control tasks, one for each time-sharing user region, the Parrot task and the TSO STOP/MODIFY task; these last two will not be discussed further. The subtasks of a region control task belong to a session and are subject to swapping - the task-related data structures lie inside the swapped region. A region control task manages the disconnection of a session from the system environment prior to a swap-out, and the reconnection of the session after a swap-in.

The swapper routine handles the actual swapping of sessions; the scheduling of swaps is managed elsewhere, in the time-sharing driver, here called the driver for short. The driver is a routine not running under a specific task but rather entered via an SVC known as TSEVENT. (In fact the driver is frequently called from system routines by a simple branch rather than by an SVC call; this is commonly the case for other SVCs also.) Diverse parts of MVT and TSO use TSEVENT to signal significant events, such as WAITs and POSTs of session tasks, stages in swapping and terminal input/output, and so on. This embedding of TSEVENT calls constitutes the large part of MVT's knowledge of TSO. The driver maintains a private set of data structures describing sessions, regions and swap scheduling queues; indeed the driver is remarkable among MVT and TSO system modules for its narrow and well-defined interface with the rest of the system. On each TSEVENT call the driver inspects and modifies its data structures as appropriate. If necessary it initiates swapping activity by setting values in a small communications area called the time-sharing interface area; subsequently the MVT dispatcher inspects the time-sharing interface area and POSTs the time-sharing control task and the region control tasks as required. We can in fact regard the driver as performing the POSTs directly; the direct route is avoided in practice only because POST itself calls the driver, and to have the POST routine sort out the matter itself would be to introduce complexity and thereby inefficiency into a central part of the system.

The central data structure in TSO is the time-sharing communication vector table (TSCVT), resident in the time-sharing control region. Many other data structures and chains are pointed to by the TSCVT, in particular the time-sharing job blocks (TJB), also resident in the time-sharing control region and therefore not swapped. The TJB is the basic

data structure describing a session; there are as many TJBs as there may be simultaneously logged-on users.

The allocation of regions in the dynamic area is in the hands of the store manager, an important Cambridge extension of OS/MVT comprising a separate system task and some SVC routines. Changes to the store manager were necessary for the introduction of store-to-store swapping.

Let us now consider the sequence of operations involved in swapping. A session, we may suppose, is resident in one of the time-sharing user regions. A critical event occurs relating to that region, TSEVENT is called with appropriate parameters and the driver is entered. The event may be for example the resident session's going into a terminal I/O wait, the expiration of the session's time-slice or the end of a wait condition for some other swapped-out session belonging to the region. The driver adjusts its data structures and, we will suppose, makes the decision to swap the session out. In the case of another session's becoming ready after a wait, various conditions determine whether the resident session is swapped forthwith. The driver signals the swap by setting a flag in the part of the time-sharing interface area appertaining to the region. Some time after the driver has exited, the OS dispatcher recognises the flag in the time-sharing interface area and POSTs the relevant region control task. The region control task disconnects the session from the OS environment, unlinking its data structures from the OS chains, and in turn POSTs the time-sharing control task with appropriate parameters. As the request is for a swap the time-sharing control task enters the swapper, which then handles the business of actually swapping the session out. The region control task and the swapper keep the driver informed of progress with a series of TSEVENTs. On receiving the 'swap-out complete' TSEVENT, the driver selects a session to swap into the region and again signals its decision through the time-sharing interface area. On this occasion the time-sharing control task is POSTed first; the swapper sees to it that the session is swapped in and POSTs the region control task, which in turn reconnects the session and TSEVENTs the driver. It is then up to the driver to set up the session's residence time-slice and allocate its share of the total time-sharing processing time.

Note that swapping is scheduled by individual region, although there may be interference between regions in queueing for the single fixed-head disc; the analogy of five logical machines is applicable. The driver takes the regions together only when allocating slices of processing time to the currently resident sessions.

3. STORE-TO-STORE SWAPPING

3.1. An overview

Figure 3 may be found helpful in this and the following subsections; the percentages marked on the Figure will not be used until section 4.

The aim of the store-to-store swapping system is to ensure that as many as possible of swaps to and from time-sharing user regions should be from and to store. The term 'swap' is used in the rest of this document to refer to the move of a session to or from its time-sharing user region, either between region and store or between region and disc. When a session is to be swapped out, the swapper tries to acquire a swap buffer of sufficient size, consisting of one or more blocks from the dynamic area. If the attempt is successful the session is moved to the swap buffer, otherwise it is swapped directly to disc.

Usually the store available for swap buffers is insufficient to hold all active sessions; in addition the store manager may wish to claim for another purpose store occupied by swap buffers. For either reason sessions may migrate from their swap buffers to disc.

For each region the driver attempts to predict which ready session will be the next to be swapped in. Whenever such a prediction can be made, the system tries to ensure that the chosen session is in store in readiness. If the last move of the session was a swap-out to store it will still be in a swap buffer. If, however, the session is on disc, the system must acquire a swap buffer and transfer the session from the disc to the buffer. In either case the system is said to preswap the session; a preswapped session cannot be migrated.

When a session is to be swapped in it is simply transferred from disc or swap buffer as appropriate.

3.2. Swap buffers

When the driver demands a swap-out or a preswap for a session on disc, the swapper must first try to get a swap buffer. This is done by a call to the granule manager, a part of the swapper responsible for the management of swap buffers.

A swap buffer consists of one or more granules; a granule is a block of store in the dynamic area starting on a 2 Kbyte boundary and a multiple of 2 Kbytes in extent. In the implementation active at the time of writing, the granules are all of a fixed size, but the external interfaces of the granule manager do not assume that this is so. The granule size must be chosen with care: if larger granules are used fewer are needed and the overheads are lower; on the other hand a small size reduces the wastage of store both inside and outside granules. When a session is swapped out only those parts of the time-sharing user region which the session is using are moved. Hence the swap sizes of different sessions, and of a single session at different times, vary considerably. The distribution of swap sizes is

uneven: certain sizes, for example the size common when a session is running the Phoenix command program alone, occur more frequently than others. As a result the relationship between granule size and efficiency in the use of store is not linear; in fact it is not monotonic. Experiments have shown that both 14 Kbytes and 26 Kbytes are good choices; the former is marginally more efficient in use of store, whereas housekeeping for the latter size is considerably cheaper. In the current implementation the granule size is 26 Kbytes.

Granules are not permanently allocated; the number and positions of granules change as more or less store becomes available and is needed by the swapper.

Each granule owned by the swapper has an associated data structure, a swap granule element (SGE). The SGE contains the address and size of its granule, three link fields and miscellaneous flags and data. The active SGEs are held on a two-way chain ordered by the addresses of their associated granules. This SGE chain is the only description of allocated granules held by the system, and therefore is used by the store manager as well as by the swapper; the two-way chaining is entirely for the benefit of the store manager. There is a pool of SGEs in the time-sharing control region; more can be obtained dynamically. Figure 1 illustrates the chaining of SGEs.

A swapper routine requests a swap buffer of a certain size by a call to the granule manager. If the store is available, the granule manager returns one or more granules with their SGEs chained through the third link field. The chain is then attached to the TJB of the session in question. In practice the swap buffer will seldom contain more than two granules, and will frequently consist of just one.

3.3. Swapping, preswapping and migration

A session may be in one of four states:

1	in a region
2	swapped out to store, in a swap buffer
3	preswapped, in a swap buffer
4	on disc

Possible transitions between these states are:

A:	1 to 2	swap out to store
B:	1 to 4	swapout to disc
C:	2 to 4	migrate
D:	2 to 3	preswap in store
E:	4 to 3	preswap from disc
F:	2 to 1	swap in from store
G:	3 to 1	swap in from store, from preswapped state
H:	4 to 1	swap in from disc

The letters are references to Figure 3.

When a session is to be swapped out, it is swapped to store if a swap buffer is available. A session in state 2 is held on one of two queues, the keep chain or the don't-keep chain; the former is for sessions with at

least one task ready to run at the time of swap-out, the latter is for the rest. When a session is swapped out to store its TJB is placed at the head of the appropriate queue. A swap to disc occurs only if there is no swap buffer available.

A swap-in may be from disc or from store; in the latter case the swap buffer is released to the granule manager when the move is complete.

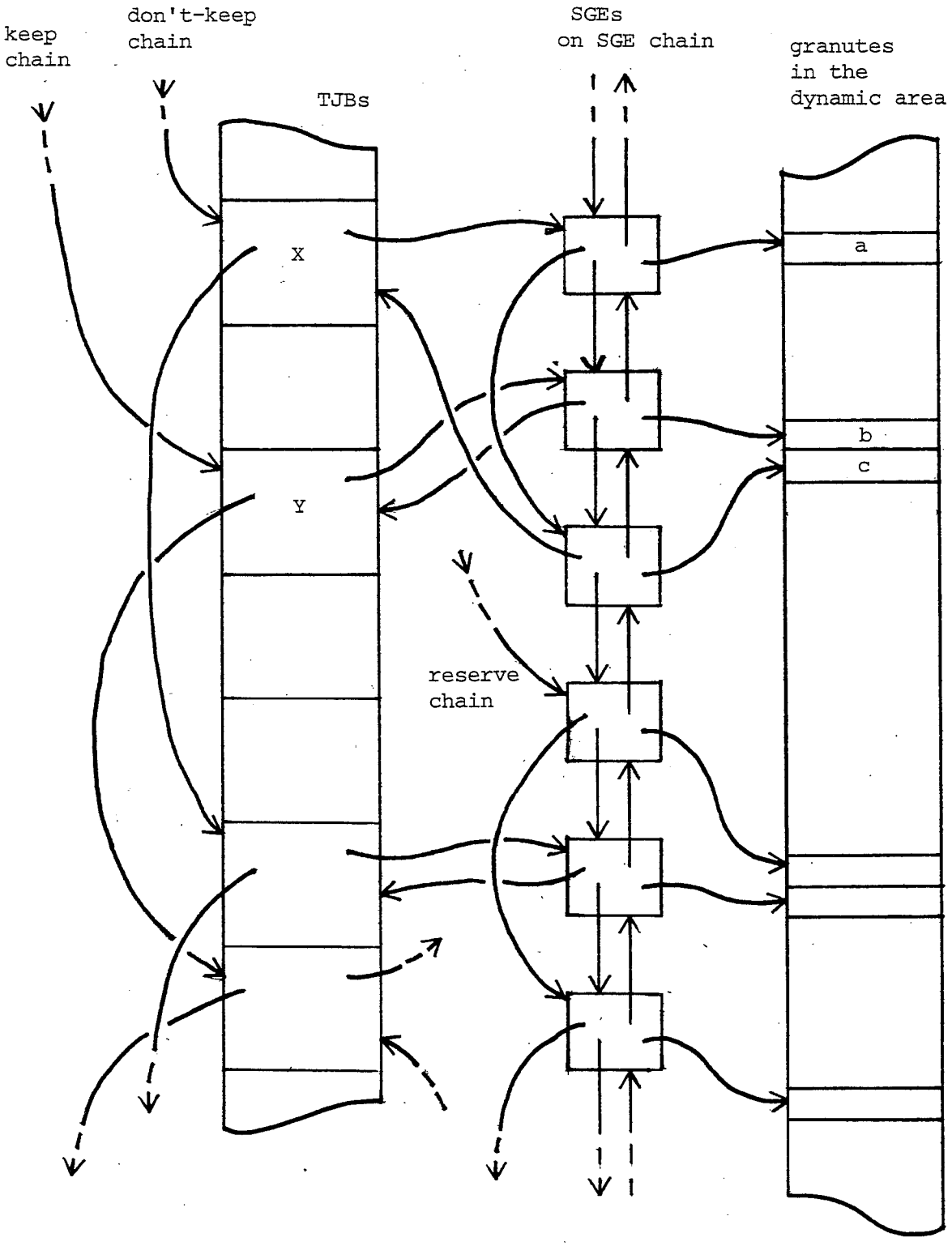
Preswaps are used to ensure that as often as possible a session is in store ready to be swapped into the region. The driver calls for preswaps on the basis of its predictions and without regard to whether sessions are in store or on disc. A session in store is removed from the keep chain or the don't-keep chain and its TJB flagged as preswapped. A session on disc is transferred into store if a swap buffer is available and flagged preswapped; the preswap attempt fails if there is no buffer. A preswapped session is expected to move into its region in the near future, so migration from preswapped state is forbidden. There may be more than one session preswapped for a given region at one time. For example an interactive session may become ready after a compute-bound session has been preswapped; as the interactive session has priority the driver will revise its prediction of the next session to run and call for the interactive session to be preswapped as well.

The causes of migration are discussed below. It is important at this point to note that migration goes on as it were in the background. The driver, which alone initiates preswaps and swaps-out and -in, is not aware of migrations. Hence the driver may well request a swap-in or a preswap for a session which is in the process of migrating. The session's swap buffer is surrendered only when the migration is complete, and until that point the swapper can preswap the session, or swap it into its region, without waiting on the migration. The session may just be queued for the disc transfer, in which case the migration is simply cancelled. If however the disc transfer is in progress it is left to finish disassociated from the session; on the transfer's completion the space in the swap file is immediately released. Thus it is possible for a session to be executing in a region while it is still nominally migrating to disc.

When a session is to be transferred between store and disc its TJB is put on the transfer queue. Requests are serviced in the order in which they are queued. The position of a new request on the queue depends on the transition involved: a swap-in from disc has the highest priority, while a migration from the keep chain has the lowest. In practice long transfer queues are rare. The only occasion on which there are likely to be many queued requests is when the store manager demands a section of the dynamic area which contains granules belonging to several different sessions.

Before the introduction of store-to-store swapping the swapper did not use a transfer queue. Without migration and preswapping the maximum number of simultaneous requests for disc transfers was equal to the number of time-sharing user regions; the swapper could provide duplicate data structures and have all the requests active at the same time. With migration and preswapping the number of outstanding requests may in principle be much higher, and a queueing mechanism becomes essential.

Figure 1: TJBs, SGEs and granules.



Examples:

TJB X is on don't-keep chain and has granutes a and c in its swap buffer

TJB Y is on keep chain and has granute b in its swap buffer

3.4. The granule manager

The granule manager is a set of subroutines in the swapper. Other routines of the swapper call the appropriate granule manager routines in order to request a swap buffer of a specified size or to return granules which are no longer occupied. The granule manager has an external interface with the store manager via GETMAIN and FREEMAIN SVCs. To obtain a granule the granule manager issues a special GETMAIN call which is recognised by the store manager; the granule size is passed as a parameter. If the store manager intends to satisfy the request it allocates a block of store in the dynamic area without, however, constructing the data structures normally used to describe a region. On receiving the new granule, the granule manager provides it with an SGE duly filled out and linked onto the SGE chain; the SGE acts as the descriptor for the allocated store. The store manager might refuse a request for a granule either because there is no store available or because it is waiting for granules to be cleared from a particular area of store which it needs in order to create a region. The store manager never refuses a FREEMAIN call to free a granule.

It is not uncommon for the store manager to refuse a request for a granule: at busy times over 70% of requests are turned down. To cushion the effect of this, the granule manager maintains a pool of unoccupied granules called the reserve. If the number of granules in the reserve falls below a fixed value called the reserve level, action must be taken to acquire new granules from the store manager or to release occupied granules by causing sessions to migrate. The reserve level is currently set at six.

There are three entries to the granule manager: to request a swap buffer, to release granules which are no longer occupied or which are wanted by the store manager, and to cause the granule manager to make good the reserve.

When a swapper routine calls for a swap buffer, the granule manager calculates how many granules are needed and tries to allocate that number from the reserve. If there are insufficient in the reserve the store manager is called for fresh granules. The requests may be satisfied, in which case the swap buffer is assembled from the reserve and the fresh granules, and handed back to the caller. If however the store manager fails to provide all the fresh granules required, any which have been obtained are added to the reserve, and the request for a swap buffer is turned down. In most cases there will be enough granules in the reserve.

Granules released by sessions are handed back to the granule manager. For each granule marked as migrating the predicted additional size of the reserve (see next paragraph) is decremented. A granule marked as wanted by the store manager is FREEMAINed; any remaining granules are added to the reserve. A reserve granule may be claimed by the store manager, in which case it must be handed over directly and the reserve made good in the usual way.

The third entry to the granule manager prompts it to check the health of the reserve. To this end the granule manager keeps a record of the predicted additional size, that is, the number of granules which are

shortly to be released by migrating sessions and which the granule manager might be able to add to the reserve. If the sum of the actual size of the reserve and the predicted additional size is less than the reserve level, the store manager is called for fresh granules to make good the deficit. If the fresh granules are forthcoming all is well; if not, the granule manager must look elsewhere. When the store manager turns down a request for a granule, the granule manager inspects the reason for the refusal. If the store manager is in the process of claiming a region no further steps are taken: there may in fact be plenty of store available. In the case of a refusal because of lack of store, the granule manager must select sessions for migration from swap buffers to disc.

First the oldest arrivals on the don't-keep chain are selected; if that chain is exhausted, the most recent additions to the keep chain are chosen, though in practice it is a rare event for the granule manager to migrate a session from the keep chain. The rationale for the latter choice is clear: the most recently swapped out ready session is, in the absence of other evidence, the ready session least likely to be swapped back soon. Deciding on an algorithm for the don't-keep chain is more difficult; at least one can argue that to select the oldest arrival is to ensure that a session whose user has gone for coffee does not sit in store indefinitely.

For each session chosen the granules are marked as migrating, the predicted additional size of the reserve incremented, and the session put on the transfer queue. This continues until the sum of the predicted additional size and the actual size reaches the reserve level. It can now be seen why the predicted additional size is used in the comparison rather than just the actual size. If the latter alone were used each entry to the granule manager would be liable to initiate migrations to make good the deficit in the reserve. The result of a rapid sequence of entries would be to make good the deficit several times over and so to cause too many sessions to migrate.

The reserve mechanism is one of the more crucial features of the system. The granule manager is able to detect a shortage of store several steps before requests for swap buffers are affected. Performance figures for a particular period of moderate-heavy load, discussed in detail in section 4, show that whereas 70% of requests for granules were refused, the failure rate for requests for swap buffers was less than 2%. Furthermore there were 10% more requests for swap buffers than for granules; the supply of granules from migrated sessions accounts for the difference. If there were no reserve each request for a swap buffer would generate at least one request for a granule, and one would expect more than 70% of the requests for swap buffers to fail. The proportion of swaps-out directly to disc, and of failed preswaps from disc, would be far greater than the 1.1% and 1.2% actually obtaining. The figures indicate that the use of reserve also induces a reduction in the number of GETMAINs and FREEMAINs for granules. In fact these figures were obtained under an earlier version of the granule manager. With the current version the saving is far greater: the number of granule requests may fall below 30% of the number of requests for swap buffers. This serves to reduce overhead rather than improve performance in any perceptible way.

3.5. The store manager

The store manager task handles the allocation, placement and freeing of regions and granules in the dynamic area; the former comprise regions for system components and offline jobs, and second regions for time-sharing sessions (a Cambridge innovation - unswapped regions in the dynamic area temporarily allocated to time-sharing sessions). The store manager is solely responsible for determining priorities for the provision of store, and for arbitrating in cases of conflict. In particular it is the store manager which ensures that granules have the lowest-priority claim on store.

The store manager permits the swapper to use as much of the free store as is needed; this will frequently be all the free store. A condition, however, is that the store manager has the right to claim back at any time any part of store containing granules to construct a region for some other purpose. When it needs to do this, the store manager scans the SGE chain to determine which granules need to be moved. The SGE chain is two-way linked and ordered by granule address so that the store manager, which knows whether it is dealing with high store or low store, can scan from the nearest end. Doomed granules are marked as wanted, and the time-sharing control task is POSTed.

The swapper in turn scans the SGE chain noting the marked granules. Reserve granules are removed from the reserve and handed directly to the granule manager for immediate freeing. The remaining marked granules belong to sessions. The status of the sessions is inspected and those which are not in a preswapped state are queued for migration. Preswapped sessions are left untouched: the marked granules will be returned to the store manager as soon as their sessions are swapped in.

The system has been designed to allow the store manager to move granules to another part of store rather than migrating their sessions, although this mechanism is currently not in use. To this end interlocks are provided on individual granules in order to prevent interference between the store manager and the swapper.

3.6. The driver

An outline of the chief functions of the driver appears in section 2. It will be recalled that the driver is responsible for scheduling swaps and assigning time-slices to resident sessions. Many of the changes introduced into the driver for store-to-store swapping are detailed rather than sweeping; however, there are some substantial modifications. The control of preswapping is the one major new function. In addition certain significant changes in the scheduling algorithms have been made possible by the comparative cheapness of swapping under the new system.

The conventional term 'compute-bound' is here used of sessions heavily engaged in computation, disc I/O or both: it will be recalled that a session must occupy its region while it has disc I/O in progress. A session is considered to be interactive at the start of each new transaction; it

becomes compute-bound if the execution time of the transaction rises beyond a certain value.

At its simplest, the preswapping mechanism within the driver works as follows: whenever the driver is notified of the completion of a swap-in, it will seek to ensure that another session is preswapped for that region; the aim is always to have one session in hand for each region. The driver may if necessary preswap at times other than the completions of swaps. In choosing a session to preswap the driver imposes an order of priorities: only a session which is ready to run is eligible; then the choice will depend on whether a session is interactive or compute-bound, and whether or not it has just emerged from a wait state. The driver may find that the session of its choice is already preswapped or that for some reason it cannot be preswapped. In such a case the driver does not choose another session of lower priority but abandons the attempt to preswap. Too many sessions must not be preswapped at once, for a preswapped session may not be migrated, and so a large number in store at one time might prove a problem for the store manager. Clearly there is no advantage in allowing a larger number of preswaps and permitting them to migrate.

There are, as we have seen, occasions on which the driver may have more than one session preswapped for a region: a session may become ready and so eligible for preswapping after a session of lower priority has been preswapped. There is good reason for allowing more than one preswap in such cases. The driver pays no attention to preswaps when it chooses a session to swap in, though of course the priority rules for preswapping are modelled on those for swapping in. Hence a higher priority session may as well be preswapped since it will in any case be preferred over a lower priority session for the next swap-in.

The reduction in swap time in the new system made it practicable to change the algorithms for preempting. A swapped-out session which emerges from a wait state may preempt a resident session for the same region by in some way curtailing the latter's period of residence. The aim of the algorithm is to afford good response to the mass of trivial transactions. Formerly the cost of swapping meant that the action could not be too drastic; in practice the only effect was that a preempting interactive session would cut the residence time-slice of a resident compute-bound session to that of an interactive session. This might mean that the session was swapped out forthwith if it had already been resident for an interactive time-slice.

Store-to-store swapping is cheaper, and so a preempting session can be allowed to cause more drastic action. The current algorithm uses the execution time of the transaction in progress for a session, that is, the sum of the cpu time and disc I/O time since the session's last terminal I/O wait, as a measure of the degree to which the session is interactive. First the driver attempts to preswap a preempting session. When the driver is notified of the completion or failure of the attempt, it compares the states of the preempting and resident sessions to determine the next step. Broadly speaking, if the preempting session is newly out of a terminal I/O wait it will cause the resident session to be swapped, allowing a short period of grace if the resident session is itself highly interactive;

otherwise the residence time-slice of the resident session is cut to that of the preemptor, as in the former system.

A feature of the driver's strategy for assigning residence time-slices is also of interest. In TSO as provided by IBM the number of scheduling queues, their characteristic time-slices and the manner in which they are treated are all specifiable as parameters. Unfortunately the underlying mechanisms are rather inflexible; the only versions which have proved successful at Cambridge have been based on round robins. A modification contemporaneous with store-to-store swapping is a variable-slice round robin. Sessions which were in a ready state at their last swap-out are held on one of two circular chains; one chain is for interactive sessions, the other for compute-bound sessions. The pair of chains is treated as a single round robin. While there are sessions on the interactive chain all sessions are granted the shorter interactive time-slice; if the interactive chain is empty the remaining compute-bound sessions receive a longer time-slice. The relative cheapness of store-to-store swapping makes it reasonable to grant short slices to compute-bound sessions. There are additional mechanisms for granting priority to sessions emerging from a wait state over those already on the scheduling chains.

4. PERFORMANCE

4.1. Preliminary

The purpose of this section is to evaluate the performance of the time-sharing system in general and of store-to-store swapping in particular. First, however, the discussion concentrates on certain problems of performance evaluation and leads on to the solutions used at Cambridge.

The performance measures used for store-to-store swapping may be divided into two classes. Some measures are relevant only to the internal operation of the system; they may be used to compare the efficiency of different versions of the system and to check whether particular components are working as expected. Other measures, such as swap times, response times and transaction rates, quantify features common to all time-sharing systems; these may be called external measures.

External measures could potentially provide a direct comparison between the current and former versions of the time-sharing system, or in certain cases between the system at Cambridge and those elsewhere. Unfortunately many of the performance logging mechanisms now incorporated in the system were introduced at the same time as or after store-to-store swapping. Hence there are disappointingly few direct comparisons that can be made between the present and former Cambridge systems.

A problem that affects all the statistics-gathering exercises of the Cambridge time-sharing system arises from the great variation in load at different times. Store-to-store swapping is influenced not only by the number of active sessions and the work they do, but also by the demand for store in the dynamic area; the latter depends on the offline job load and the use of time-sharing second regions . As a result it is difficult to correlate performance figures so as to give a single composite representation of performance - or at least any representation so obtained would be of doubtful value. The method adopted here is rather to give specific and, as far as is possible, typical values, indicating in more general terms the variations that occur.

The measurement of certain characteristics, in particular of load and response, presents a second problem: what in fact constitutes an adequate measure? Is a single measure applicable in all circumstances? In practice one has to develop suitable operational definitions as reconstructions of the pre-formal notions. The next two subsections discuss this issue for load and response.

4.2. Measures of load

The number of active sessions is clearly a broad indication of the load on the time-sharing system; one readily observes a rough correspondence between this number and response. However it is not sufficient for all purposes. An effect noticeable at Cambridge is that the response in the first term of the academic year with, say, 80 active sessions is by and

large better than the response with the same number of sessions in the second term. The reason is not hard to determine: in the first term many users are novices, finding their way round the system and doubtless spending much of the time thinking rather than driving their sessions. By the second term the same users have greater experience and are driving their sessions harder, thus making greater demands on machine resources. This effect is in fact measurable: there are details later in this subsection.

Clearly the rate at which sessions work must be taken into account in characterising load. Now the rate of work of a session depends on the rate at which new transactions are initiated and on their execution times. As the rate and size of transactions increase, so the mean number of sessions which are running or ready to run also increases (that this is so depends on the near-random distribution in time of the starts of transactions).

Thus the mean number of ready sessions represents a good measure of load. It remains to decide how to calculate a mean; as the actual number of ready sessions varies wildly from moment to moment, smoothing must be applied. The procedure used in the system measures the mean over the preceding few minutes, weighting the measure in such a way that the most recent history is the most significant and the value of the measure converges exponentially with a half life of 1.75 minutes towards the actual number of ready sessions. The following values of the mean number of ready sessions can be taken as rough guidelines for the load perceived by users:

up to 5	light
5 to 10	medium
10 to 20	heavy
over 20	very heavy

There is a lacuna in the argument for using the mean number of ready sessions as measure of load; the mean has been described as a function of transaction rate and transaction size. It is, however, also dependent on the efficiency of the time-sharing system itself. An improvement in the system will produce a corresponding improvement in response precisely because transactions are completed more rapidly and, for a given transaction rate, the mean number of ready sessions falls. That is to say, the mean number of ready sessions is a measure of the apparent load. For a given time-sharing system, the apparent load varies with the real load, and may be used as an indicator of the latter in measuring performance at different times. On the other hand different time-sharing systems may exhibit different apparent loads for the same real load; the more efficient a system, the lower the apparent load will be.

The mean number of ready sessions, or apparent load, is used below as a measure of the variation in real load on the current system. It can also serve as a crude means of comparison between store-to-store swapping and the former system; crude because the comparison is dependent on rough indicators of real load such as the number of active sessions.

A measure independent of changes in the system is still needed; such might be provided by the distribution of execution times of transactions. Certainly this value seems dependent on the behaviour of the users rather

than on that of the time-sharing system. However, major changes in the time-sharing system will create a different environment as a result of which users' behaviour may alter; the claim should rather be that, for sufficiently similar systems, the distribution of transaction execution times is independent of the behaviour of the time-sharing system. Figure 4 shows that this distribution exhibits little variation: most observed distributions lie within the shaded area on the Figure. (The strange time co-ordinates used in the Figure and elsewhere have been chosen for the convenience of the logging software; the values are all powers of 2 multiplied by the 1/300 second timing unit common in 370/165 software.) It may be questioned, however, whether this is a truly independent measure of load, even under a single time-sharing system; might not the transaction rate fall as the transaction sizes increase purely as a result of users' behaviour? In fact higher transaction sizes have been associated with deteriorations in response times; the two abnormal distributions in Figure 4 were obtained at times of notably inferior response. The tentative conclusion is that the distribution of transaction execution times does indeed form an independent measure of load, provided that it is not used to compare radically different time-sharing systems.

The manner in which the five time-sharing user regions are treated separately has been mentioned previously: a session is restricted to the region in which it is initiated at logon. The analogy with five machines has been drawn. This arrangement has the disadvantage that the load at any one time may be unevenly distributed across the five regions. When this occurs some users will experience a response considerably worse than the momentary average, some better.

As an example consider the following measurements of load (mean number of ready sessions) taken by region at the start and end of an eight-minute period in November 1979:

82 active sessions:	1.66	0.91	0.50	0.52	1.09	total 4.68
98 active sessions:	1.56	4.12	1.21	1.39	2.73	total 11.03

The overall load changes from light-medium to medium-heavy. However the users in region 2 perceive a change from an apparent system-wide value of 4.55 (5 * 0.91) to 20.6 - from light-medium to very heavy. For users in region 1 there would be no perceptible change. Such peaks in the load in single regions seem generally to be short-lived - about 2 or 3 minutes is common. Indeed the observed times depend on the degree of weighting used in the calculation of the mean number of ready sessions; the actual peaks are likely to be shorter and sharper. One could ensure that a new session was always started in the region with the lightest momentary load at the time of logon, but this would do little to mitigate the effects of short-term peaks, and might rather lead to greater unevenness. In practice the region with the fewest number of active sessions is chosen, as this quantity is more stable than momentary load. The only way to solve the problem of uneven load would be to use a single time-sharing region and ensure that everyone got the same appalling response.

4.3. Measures of response

A value often used to characterise the performance of a time-sharing system is mean response time; this is clearly unsatisfactory in most cases. It would perhaps be a useful index of improvement after a modification in a time-sharing system as long as the load on the system showed little variation. In comparisons between different periods under one system with variable load, or between systems at different installations, the mean can be quite misleading.

First, a low mean response time with a high variance might be less satisfactory than a higher mean with low variance. Indeed one might deliberately choose the latter as a feature of the design of a time-sharing system. Thus to sacrifice rapid response for trivial transactions to overall steadiness might be desirable if it removed irregularities in response time for transactions of identical sizes; it is a less convincing proposition if the smoothing conceals the distinction between trivial and non-trivial transactions.

Second, in order both to understand the performance of one system and to compare different systems it is necessary to take account of the relationship between response time and transaction size. The mean response time does not reflect this relationship.

The procedure used in the current system at Cambridge takes account of both these issues. The response time of a transaction is defined as simply its elapsed time; this excludes delays in the communications system. For each transaction the response factor is calculated:

$$\text{response factor} = \frac{\text{response time}}{\text{execution time}}$$

Frequencies are counted for the distributions of both response and (the logarithm of) execution time over all transactions. The result is a matrix of the kind illustrated in Figure 2. The timing unit is 1/300 second. Transactions with execution times less than this value (about 10% of all transactions) cannot be counted in the same way. For these the system simply records the proportion with response times less than 0.853 second.

If one wanted a single figure to characterise response, one might give the percentage of all transactions with response time less than a certain value; but this would not make allowance for transactions with execution times comparable to or longer than the chosen value. Yet to quote the proportion of transactions with response factors less than a certain value would distort the picture for short transactions: for a transaction of 10 msec a response factor of 20 is as good as one of 4. The solution is to use a combination of the two. The figure used at Cambridge is the percentage of transactions with response time less than 0.853 second or (4 * execution time), whichever is the greater. This figure generally lies between 75% and 95%.

4.4. Observed performance

This subsection presents and discusses performance figures for the store-to-store swapping system. Subsection 4.1 mentioned the difficulty of assembling a universal set of statistics. The method adopted here is to present measurements made during a particular period and then to indicate the sorts of variations encountered. The eight-minute period chosen was characterised by a large number of active sessions and a moderate-heavy load. Figure 2 gives details of response times and certain other measures. Figure 3 represents the movement of sessions between region, store and disc and indicates the frequencies with which particular routes were traversed. The distribution of transaction execution times for the period under study is marked on Figure 4.

Figure 2 will be discussed first. Item (a) in Figure 2 shows that the load in the period was medium-heavy. There was little variation across regions, or for each region between the start of the period and the end; compare the figures for another eight-minute period in November 1979 quoted in subsection 4.2.

A transaction has been defined as the period of activity of a session between terminal input or output waits. However, performance evaluations sometimes use a slightly different definition of the term, whereby a transaction is a period between input waits only, so that terminal output wait states are counted as part of the transaction. The mean transaction rates for both definitions are given in Figure 2(b); generally about 88% of transactions (in the sense normally used here) end in terminal input waits, that is, are transactions according to the second definition. The mean rate of transactions is commonly in the range 4.5 to 5.5 per second for heavy loads.

Figure 2(b) includes a table showing the frequencies of different values of the response factor and transaction execution time. The value to the left of each row in the table indicates the upper limit of the range of execution times for that row; the lower limit is equal to the value of the upper limit for the row above. For example, the third row includes transactions with execution times between 0.053 sec and 0.107 sec. The first row is for execution times between 0.0033 sec (1/300) and 0.027 sec; the last row covers execution times greater than 6.827 sec. Each position in the table shows, as a percentage of all transactions, the proportion of transactions whose execution times and response factors lie within the appropriate ranges. Only transactions whose execution times are greater than 0.0033 sec are included in the table; they comprise 91.27% of the total. Commonly this figure is in the range 88% to 90%.

The column totals, giving the overall distribution of response factors, show a considerable variance. In 25.86% of all cases the response factor was greater than 14. However, the largest contribution to this total comes from transactions in the first row of the table. In fact many of the transactions included in the figure of 20.67% in that row will have had execution times nearer to 0.0033 sec than to 0.027 sec; in such cases large response factors may yet be consistent with good response. Only 2.9% of transactions had execution times in this range and response times greater

than 0.853 sec. If we exclude the first row from the calculation, the column totals become

<2	2-4	4-6	6-8	8-10	10-12	12-14	>14
9.56	14.05	5.69	3.51	2.11	1.86	0.93	5.19

The variance is now lower, but it is still significant. It is unlikely that it could be reduced to insignificance with the current hardware.

8.73% of transactions had execution times less than 0.0033 sec, but only 0.04% had both these short execution times and response times greater than 0.853 sec.

The overall figure for response is 79.55%; typically the value lies within the range 75% to 95%, depending on load.

The mean swap rate of 21 per second given in Figure 2(c) is typical of a busy system; under quieter conditions the rate falls to well below 10 per second. The mean swap time of 5.5 msec compares with a value of about 30 msec before the introduction of store-to-store swapping. The mean swap time generally lies in the range 5 to 6 msec. On occasions when the fixed-head disc has been out of order and the swap file has been on a moving-head disc, the mean swap time has increased only a little, about 7 msec apparently being the maximum. On the other hand, the value may rise above 7 msec if there is a shortage of store for granules, even though the fixed-head disc is in use. These observations are consistent with the dictum that in a virtual-memory time-sharing system the quantity of main storage is often more important than the speed of the swapping devices; store-to-store swapping does to some extent imitate a virtual-memory system.

Figure 2(d) concerns granule management. The number of granules in existence varies considerably over time. It is unlikely often to be near 83, the maximum in the period under study; a normal figure would be 50, or 1300 Kbytes. This is still a large amount of store; it is worth bearing in mind that without store-to-store swapping this store would be unused, and that the store manager has granules removed whenever their store is needed for other purposes. The 54% failure rate of granule requests due to lack of store is typical of heavy loads coupled with scarcity of store in the dynamic area. With a light load the figure is far lower, sometimes as low as 2%. On the other hand, there is little variation in the rate of refusals which occur because the store manager is busy; a figure between 10% and 15% is common.

It is noteworthy that the failure rate for requests for swap buffers - 1.6% in Figure 2(e) - is much lower than the 70.2% failure rate for granule requests. The former figure is invariably low, sometimes only a fraction of a percent, bearing witness to the success of the algorithms in the granule manager.

Figure 2(f) states that 85% of migrations were initiated by the granule manager, 15% by the store manager. These proportions can vary considerably with load. Here are the figures compared with two other sets of readings from the end of 1979:

migrated by granule manager	85%	30%	2.5%
migrated by store manager	15%	70%	97.5%

In general a heavy load and scarcity of store produce a high proportion of

Figure 2: observed performance during an eight-minute period.

Period: 12:06 to 12:14 on 29.11.79, duration 8 minutes
107 - 115 active sessions

(a) mean number of ready sessions by region and overall,
at beginning and end of period

1.71 2.13 2.28 1.60 2.17 total 9.89
1.76 2.08 2.40 1.62 2.19 total 10.05

(b) transactions and response:
frequencies are given as percentages of all transactions

5.5 transactions per second, 4.8 terminal input waits per second

response factor against transaction execution time,
for transactions with execution times > 0.0033 sec:

exec times	range of response factor <2	2-4	4-6	6-8	8-10	10-12	12-14	>14	totals
0.027	0.19	4.03	6.22	4.82	4.70	4.10	3.64	20.67	48.37
0.053	0.46	1.60	0.46	0.30	0.04	0.16	0.04	0.56	3.62
0.107	2.24	2.88	0.83	0.23	0.16	0.26	0.04	0.91	7.55
0.213	2.35	2.50	0.42	0.26	0.23	0.19	0.07	1.57	7.59
0.427	2.43	2.85	1.25	0.68	0.65	0.38	0.19	0.91	9.34
0.853	1.44	2.13	0.95	0.76	0.38	0.23	0.26	0.72	6.87
1.707	0.53	1.18	1.06	0.53	0.42	0.42	0.26	0.34	4.74
3.413	0.07	0.57	0.42	0.57	0.19	0.11	0.07	0.18	2.18
6.827	0	0.26	0.19	0	0.04	0.11	0	0	0.60
	0.04	0.08	0.11	0.18	0.04	0.11	0	0	0.41
totals	9.75	18.08	11.91	8.33	6.81	5.96	4.57	25.86	91.27

2.9% had execution time between 0.0033 sec and 0.027 sec,
and response time > 0.853 sec
8.73% had execution time < 0.0033 sec
0.04% had execution time < 0.0033 sec and response time > 0.853 sec
about 73% had response time < 0.853 sec
79.55% had response time < maximum (0.853 sec, 4 * execution time)

(c) Swaps:
10.5 swaps per second
mean one-way swap time 5.5 msec
mean residence time 323 msec.

(d) Granules:
maximum number 83, totalling 2158 Kbytes
5790 requests to store manager, or 12 per second
54% failed through lack of store
16.2% failed because store manager was busy

(e) Swap buffers:
6420 requests to granule manager, or 13.4 per second
1.6% failed

(f) Migrations:
37% of swaps to store subsequently migrated
85% initiated by granule manager (squeeze on store)
15% initiated by store manager (to reclaim specific granules)

migrations initiated by the granule manager.

Figure 3 illustrates the internal dynamics of store-to-store swapping. The percentages on upward routes in the Figure are of all swaps-out during the period, those on downward routes of all swaps-in. There is a discrepancy in these figures: 37.7% of swaps-out arrive on disc, whereas transfers from disc constitute 38.1% of swaps-in. The explanation is that each session begins at logon with a swap of a standard session framework from disc, and ends at logoff also with a swap-in. Hence every complete session contributes an unpaired swap-in from disc.

The figures for swaps directly between regions and disc, and for swaps out to store, generally exhibit little variation with load; the values here may be taken as typical, though slightly high for the disc swaps. The figures for preswapping and migration, however, are noticeably high. An overnight period in February 1979 registered 14% migrations and 28% preswaps, with a corresponding 61% swaps-in from the keep and don't-keep chains; compare 36.3%, 74.7% and 13.8% respectively for the period under study. In fact these three values tend to vary in unison.

Some observations are not included in the Figures. In the period under study 1.2% of all preswap attempts failed (because no swap buffer was available); this value is generally below 1%, though it has been seen to exceed 10%. A small number of sessions were preswapped or swapped in while in the process of migrating (see subsection 3.3). Such cases constituted about 0.1% of all swaps-in; this figure is typical.

The Cambridge system may be further characterised by extrapolating and interpreting the sorts of statistics described so far; thus one can obtain figures which more readily permit direct comparison with other systems. Naturally such extrapolations can serve only as rough guidelines. The estimates quoted here for moderate-heavy loads are examples.

The first table indicates the proportions P of transactions with execution times less than E secs which have response times less than R secs.

P	E	R
80%	0.85 sec	0.85 sec
88%	0.85 sec	1.70 sec
85%	1.70 sec	1.70 sec

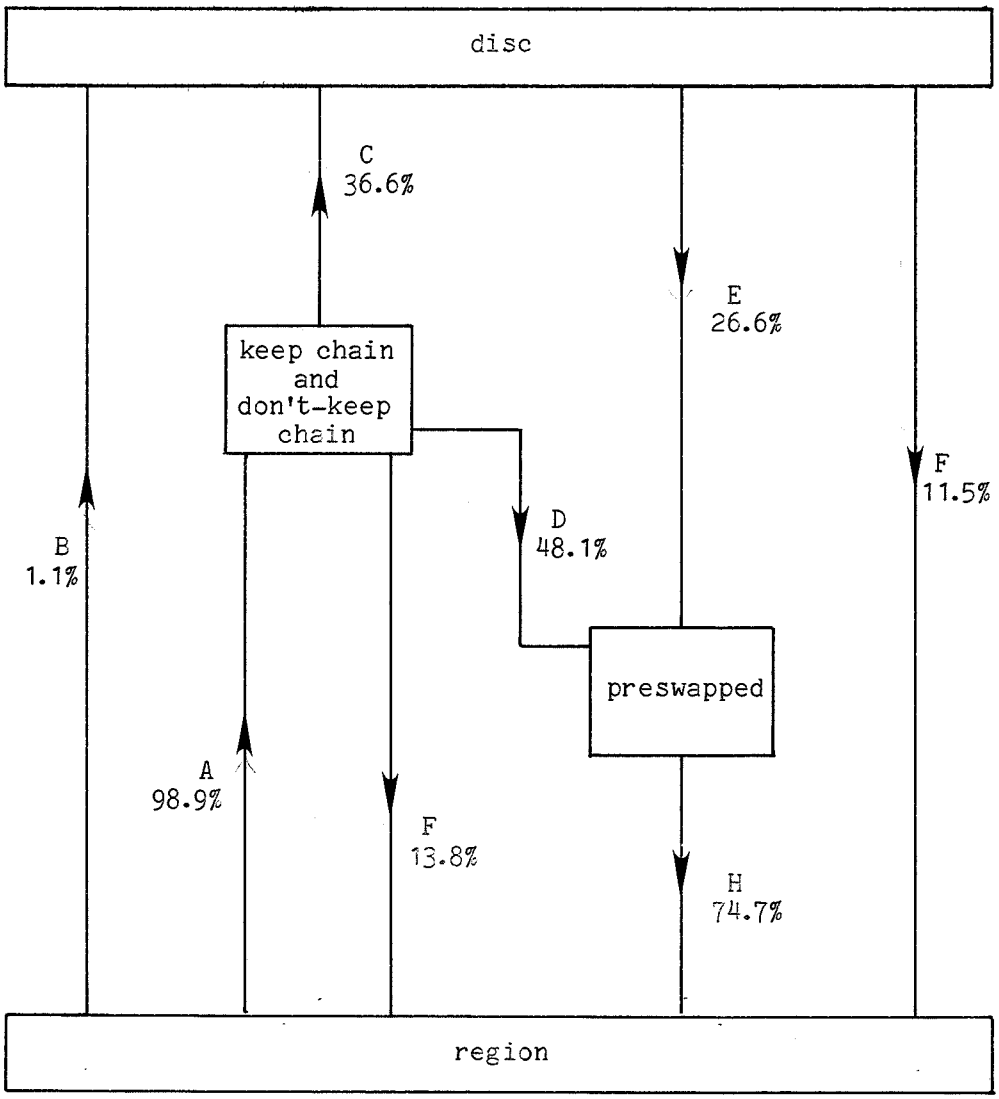
The proportions P of <u>all</u> transactions with response times less than R secs are as follows:

P	R
75%	0.85 sec
88%	3.50 sec
93%	7.00 sec

4.5. Costs of store-to-store swapping

An adverse effect of store-to-store swapping is to introduce delays into the process of region allocation. Frequently the store manager must wait for granules to be freed before it can hand over a new region. Tests show that the mean delay per allocation is 50 msec, rising to 90 msec at busy

Figure 3: path flow for change of state of sessions.



A: swap out to store
B: swapout to disc
C: migrate
D: preswap in store
E: preswap from disc
F: swap in from store
G: swap in from store, from preswapped state
H: swap in from disc

The percentages indicate the traffic along each route; those on upward paths are of all swaps-out, those on downward paths of all swaps-in.

times.

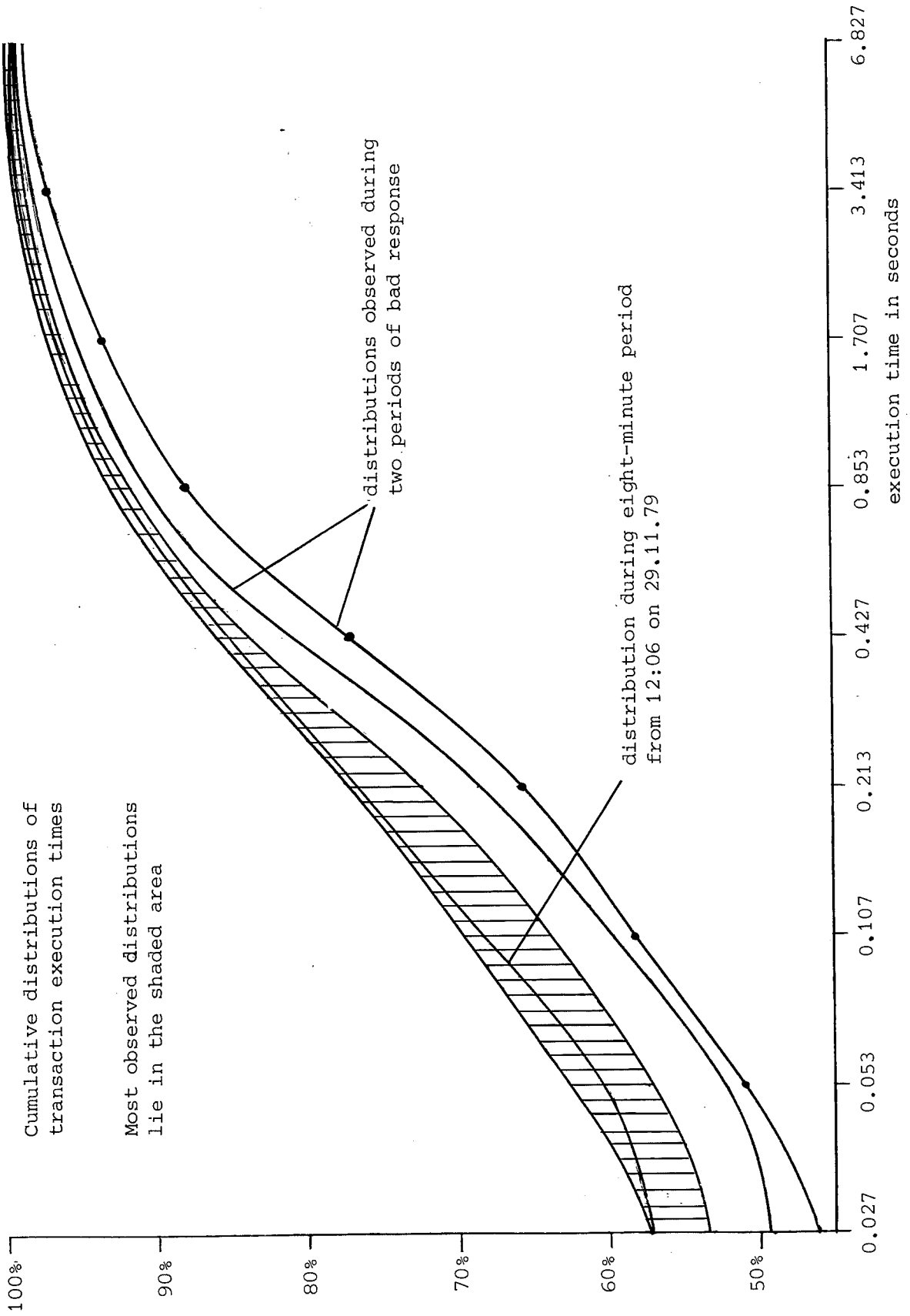
The major cost, however, is in cpu time. To move the mean swap size of 22 Kbytes within store takes about 2.53 msec. During the period studied in subsection 4.4 there were 21 swaps per second, 93.7% of which were within store. Hence the time spent in store-to-store swaps was 49.8 msec per second, or about 5% of the total cpu capacity. In addition there is some overhead for granule management and the new functions in TSO and the store manager.

Against this we may set the savings in cpu and channel time gained by the reduction in disc traffic. In the period studied there occurred only about 38% of the disc transfers that would have been required to maintain the same swap rate without store-to-store swapping. Thus the number of transfers saved was 62% of 21, or 13 per second. The actual time needed to transfer 22 Kbytes to or from the fixed-head disc is 15 msec. Hence the channel capacity saved was 195 msec per second, or about 20% of the total capacity. The cpu cost of 13 disc transfers is about 10 msec, which more than balances the extra housekeeping costs in store-to-store swapping.

We see that in the period under study store-to-store swapping cost about 5% of the cpu capacity and saved 20% of the capacity of the channel serving the fixed head disc. In general the cpu of the Cambridge 370/165 is not utilised 100%. Store-to-store swaps do in part soak up what would otherwise have been cpu idle time; the real cpu cost will have been below 5%. The saving in channel utilisation reduces contention and so improves the retrieval time for OS and HASP overlays.

The figures quoted here refer to busy periods; with a lower swap rate the cost is considerably lower.

Figure 4: distribution of transaction execution times.



5. CONCLUSION

The store-to-store swapping system has now been operating successfully for over a year. The original aim, to increase the maximum number of simultaneously active terminals, has been achieved: the maximum is now set at 120. It is difficult to make detailed comparisons of performance before and after the change; no figures on response are available for the previous system, and patterns of use change. However comparison of the mean number of ready sessions before and after the change suggest that the apparent load with 120 sessions is now lower than it was formerly with 90. The most dramatic change has been in swap time: the mean, formerly 30 msec, is now 5.5 msec. The cost is less than 5% of the total cpu capacity of the machine, while there is a saving of up to 20% in use of the channel serving the fixed-head disc.

Store-to-store swapping was designed and implemented by the author. Barry Landy provided the new features in the store manager. Important changes to the driver had previously been made by Tony Stoneley, who also devised the mean number of ready sessions as a measure of load. Chris Thompson improved the handling of disc I/O in the swapper.

I would like to thank those who, through their advice and criticism, helped in the production of this report.

APPENDIX A: GLOSSARY

Terms defined and used in just one subsection are not included in the glossary.

active session	a session serving a user logged onto the system.
don't-keep chain	the chain of sessions which have been swapped out to store and which were not ready to run when the swap-out occurred.
driver	the TSO routine responsible for scheduling swaps.
dynamic area	an area of store not occupied by OS/MVT; part of the dynamic area is allocated to system components like TSO which are regarded as ancillary to OS/MVT, the rest is available for users' jobs, granules and some other applications.
FREEMAIN	the SVC used to free a block of store.
keep chain	the chain of sessions which have been swapped out and which were ready to run when the swap-out occurred.
GETMAIN	the SVC used to obtain a block of store.
granule	a block of store in the dynamic area used for store-to-store swapping.
granule manager	a set of routines in the swapper which manage granules and construct swap buffers.
migration	the transfer of a session from a swap buffer to disc in order to release the granules for some other use.
OS dispatcher	a component of OS/MVT responsible chiefly for allocating the cpu to tasks.
POST	an SVC by means of which one task can signal another task, possibly waking the signalled task out of a wait state.
preswap	the process of ensuring that a swapped-out session is in a swap buffer in readiness for a swap-in.
ready session	a session which is ready to run, i.e. not in a wait state.
region	a block of store in the dynamic area allocated to a job, session or system component.
residence time	the time a session spends in its region between end of swap-in and start of swap-out; this includes the time taken to reconnect and disconnect the session.
response factor	the response time of a transaction divided by its execution time.
response time	the total time taken by a transaction from start to finish.
SGE	a swap granule element: a data structure describing a granule.
SGE chain	the two-way chain of the SGEs describing all the allocated granules; the chain is ordered by granule address.
store manager	a system component responsible for managing store in the dynamic area for regions and granules.

swap	a transfer of a session into or out of a time-sharing user region.
swap buffer	a set of one or more granules into which a session is swapped or preswapped.
swapper	a TSO routine which manages swaps, preswaps and migrations; the swapper also contains the granule manager routines.
swap time	the time between the queueing of a swap by the swapper, or the start of the swap if no queueing is needed, and the end of the swap; this does not include the time to disconnect or reconnect a session.
SVC	supervisor call: a call on the operating system to provide some special service.
time-sharing user region	a region in which time-sharing sessions execute; there are five such regions in the Cambridge system.
transaction	the period between a session's emergence from a terminal I/O wait state and its return to a terminal I/O wait state.
transaction execution time	the sum of the cpu time and disc I/O time used by a transaction.
TJB	time-sharing job block: a data structure describing a session; each session has one.
TSEVENT	the SVC used to call the driver.
WAIT	an SVC by means of which a task can put itself into a wait state (though a task can enter a wait state without calling WAIT - an example is the terminal I/O wait state of a session).