# Dynamic Grouping in an Object Oriented Virtual Memory Hierarchy*

Ifor Williams*, Mario Wolczko†, Trevor Hopkins

Department of Computer Science, The University, Manchester M13 9PL, UK.

ifor@uk.ac.man.cs.ux, mcvax!uk!man.cs.ux!ifor
miw@uk.ac.man.cs.ux, mcvax!uk!man.cs.ux!miw
trevor@uk.ac.man.cs.ux, mcvax!uk!man.cs.ux!trevor

## Abstract

Object oriented programming environments frequently suffer serious performance degradation because of a high level of paging activity when implemented using a conventional virtual memory system. Although the fine-grained, persistent nature of objects in such environments is not conducive to efficient paging, the performance degradation can be limited by careful grouping of objects within pages. Such object placement schemes can be classified into four categories — the grouping mechanism may be either static or dynamic and may use information acquired from static or dynamic properties. This paper investigates the effectiveness of a simple dynamic grouping strategy based on dynamic behaviour and compares it with a static grouping scheme based on static properties. These schemes are also compared with near-optimal and random cases.

## 1 Introduction

Virtual memory enables programs much larger than primary memory to be implemented without the need for explicit management of primary memory by the user program. Virtual addresses are usually mapped onto a two-level hierarchy of memory devices consisting of primary random access memory and secondary disk memory. Contiguous portions of virtual memory (termed 'pages') are swapped between primary and secondary memory as required by the program. Pages are typically several thousand bytes long and page sizes are carefully chosen to minimise the amount of paging. When programs occupy contiguous sections of store large in comparison with page sizes, the virtual memory system performs well. However, in *Smalltalk-80*[1] [GR83] the unit of locality is a fine-grain object (50 bytes on average [Ung84]) small in comparison to page size. Consequently *Smalltalk-80* suffers serious performance degradation on conventional paging systems.

Applications are often developed by incrementally grafting new objects into a persistent environment, so that the static locality inherent in conventional programs is lost. The re-use of code is encouraged by the inheritance mechanism, which enables an application to use any portion of a large persistent environment. In contrast, conventional programs are limited to the code that represents the program text (with the exception of library routines and the operating system). In addition, conventional programs have statically bound procedure calls, whilst in *Smalltalk-80* binding is performed at run time.

The cumulative effect of such properties of object oriented systems is poor paging performance on conventional paging systems. To reduce the inefficiencies, various object grouping strategies may be employed. In general, several related objects are placed together on a page; the fundamental aim of this process is to maximise the useful content of a page when it is brought in from secondary storage. Ideally, when a page is brought in from secondary memory, it should be full of objects that will be required in the near future which are not already in primary memory. Similarly, a page to be swapped out to secondary memory should contain objects that will not be required for some time. In swapping out a page, any object in primary memory can be selected and included on the page to be transferred into secondary memory. However, when a page is brought in from secondary memory, it is much more efficient if objects

*A version of this paper was published in *Proc. ECOOP 87*, Springer-Verlag LNCS 276, pp.79-88.
†Supported by the Science and Engineering Research Council
[1] *Smalltalk-80* is a trademark of Xerox Corp.

contained within the page reside on adjacent disk blocks. Therefore it is impractical to satisfy the ideal grouping for both fetching and ejecting a group of objects from primary memory.

## 2 The Experiment

The purpose of this experiment was to investigate the effectiveness of two different grouping schemes. A *static* grouping scheme based on static information was compared with a *dynamic* grouping scheme based on dynamic behaviour. The comparison was performed by simulating the respective memory models and animating the simulations with long memory reference traces gathered on a *Smalltalk-80* system. These memory reference traces were obtained by modifying the *Smalltalk-80* virtual machine emulator to write all object memory accesses, object creation and garbage collection information onto a file. The trace file was then read by the memory model simulation which provided statistics to evaluate the effectiveness of the model. No compression of memory traces was necessary, as sufficient memory space and processor power was available to perform a realistic simulation.

The memory traces used to exercise the memory model simulation were derived from typical interactive sessions using *Smalltalk-80*. There were four traces, each representing approximately 4 000 000 bytecodes of activity. Trace 1 to trace 3 were 4M, 3.5M and 5.5M bytecodes in length respectively and consisted of compiling, browsing and editing activity and execution of some examples. Trace 4 was derived from a 19M bytecode session which involved the filing in and compilation of a 24Kbyte source file and the re-compilation of the hierarchy rooted at StandardSystemView.

In order to ease the capture of long traces, the virtual machine executing the sample sessions wrote event information onto a file. Another virtual machine implementation was then driven by these event traces and generated the memory reference traces. If the virtual machine used in executing the sample sessions traced all memory accesses, the reduction in performance would limit the length of obtainable traces. Since large and representative traces were required to provide an accurate simulation, this indirect method of generating memory reference traces was used.

## 3 Grouping Categories

In a static grouping scheme, objects are re-grouped whilst the system is in a 'frozen' state; this is done by creating an 'initial placement' that locates objects related in some way close to each other in memory. For example, *Smalltalk-80* has a facility whereby a 'snapshot' of the whole state of the system is taken and an 'image' created. The *Smalltalk-80* virtual machine can load the image into its memory and re-create a state identical to the state of the environment when the image was created. When the virtual machine loads the image it may arrange the objects in memory such that related objects are close together, thereby statically grouping objects.

Conversely, dynamic grouping is performed while the system is running and is inherently more complex because of the need to maintain memory integrity. In the incremental copying garbage collectors [Ste75, LH83] for example, objects are copied from one area in memory to another whilst the system is running. The principal aim is to identify unreferenced objects, but by incrementally copying related objects into a new area of memory dynamic grouping is performed.

Grouping strategies can be further categorised by the nature of the information used to group objects. Information from static relationships, such as from the graph formed by pointers from objects, may be used with either static or dynamic grouping. For example, these pointers may be followed in either a 'depth-first' or 'breadth-first' manner. Similarly, grouping based on dynamic behaviour can also be used — for example, a 'least recently accessed' mechanism. This gives a total of four categories for classifying grouping strategies.

## 4 Static Grouping

Static grouping involves the generation of an initial placement of objects within virtual memory. The initial placement may be determined by a wide variety of factors leading to many different initial placement schemes. However, a previous study of static grouping schemes [Sta84] showed that there is no appreciable difference between different object groupings. In the study, several static grouping schemes were compared and the results did not indicate a scheme which performed convincingly better than others. Thus it appears that the choice of static grouping strategy is not critical; hence a simple depth-first ordering is considered here.

The model used to simulate a conventional paged memory system is specified formally in Appendix A. Note that the assignment of objects to virtual addresses does not change after the initial placement. Except for objects created or deleted during the simulation run, a page will always contain the same set of objects. When an object not in primary

memory is accessed, the page on disk containing the object is brought into primary memory. If primary memory is full, a page must be purged to create space for the required page. In the model, page replacement is achieved by a true 'least recently used' (LRU) purging policy; this approach is considered reasonable as most conventional paging systems use an LRU approximation.

When an object is created, a best fit algorithm is employed to allocate a free space to the object. A list is maintained of the free spaces in memory and new objects are allocated into the smallest contiguous space that will contain them. If no free space is available then an existing page must be purged from primary memory. Additionally, for the purposes of simulation, the memory within a page is effectively compacted after an object is deleted from that page; fragmentation within a page is not modelled. Deleting objects by request from the reference counting garbage collector is immediate and the freed space is added to the free list.

# 5 Dynamic Grouping

As mentioned earlier, an effective grouping of objects to be purged is not necessarily the best grouping for objects to be fetched from secondary memory. Identifying the group of objects not in primary memory that are to be accessed in the immediate future is clearly impossible; in general, it is also difficult to find an adequate approximation. In contrast to this, assessing which objects are least likely to be used in the future is equally impossible but has a more readily realisable approximation in the form of a 'least recently used' heuristic. Consequently, the dynamic grouping scheme based on dynamic information modelled here will be in the form of a 'least recently used' (LRU) purging algorithm.

## 5.1 Least Recently Used Grouping

For an efficient implementation, a dynamic grouping scheme requires pointer-offset object addressing. In many *Smalltalk-80* implementations, virtual addresses are used to directly access objects [UP83] thus saving an object table indirection. However, when objects are dynamically relocated in memory, the object table indirection is clearly invaluable. There are two possible levels of implementation for the object table indirections. First, the object table may translate pointers into virtual addresses for the underlying machine or alternatively, the pointers may be translated into real primary or secondary memory addresses. This experiment assumes that the object table translates onto real addresses.

Accessing an object not in primary memory causes the secondary memory page containing the object to be fetched. Purging primary memory is achieved by swapping out pages of objects into secondary memory. Grouping objects onto the page to be ejected is achieved by constructing a collection of least recently used objects whose cumulative size is not greater than the page size. When objects are created during the simulation run, space is reclaimed in primary memory by ejecting objects into secondary memory. Also, the garbage collector reclaims space occupied by an object by marking the object as garbage and adding it to the free list.

## 5.2 Random Grouping

To give some indication of the effectiveness of the LRU dynamic grouping scheme, a random dynamic grouping scheme is also simulated. Purging involves identifying a random group of objects to be swapped into secondary memory. This is achieved by pseudo-random probing into the object table until sufficient objects have been collected to fill a page.

## 5.3 Near-Optimal Grouping

A near-optimal dynamic grouping strategy is also simulated to measure the effectiveness of the LRU dynamic grouping scheme. An optimal scheme is not realisable in practice but, given a complete memory trace where 'future' memory references may be analysed, it may be approximated with a reasonable amount of accuracy. In the ideal case, if a group of $X$ objects are to be swapped into primary memory, the group would contain the next $X$ objects to be accessed that are currently not in primary memory. However, for the purpose of simulation, $X = 1$ since it is assumed that the secondary memory has no latency. If all objects are the same size and primary memory is full, fetching an object from secondary memory necessitates ejecting an object from primary memory. The object to eject is obviously the one that is next needed furthest in the future. This object can be identified by scanning the memory traces forward in time from the faulted reference until references to $N-1$ different objects in primary memory have

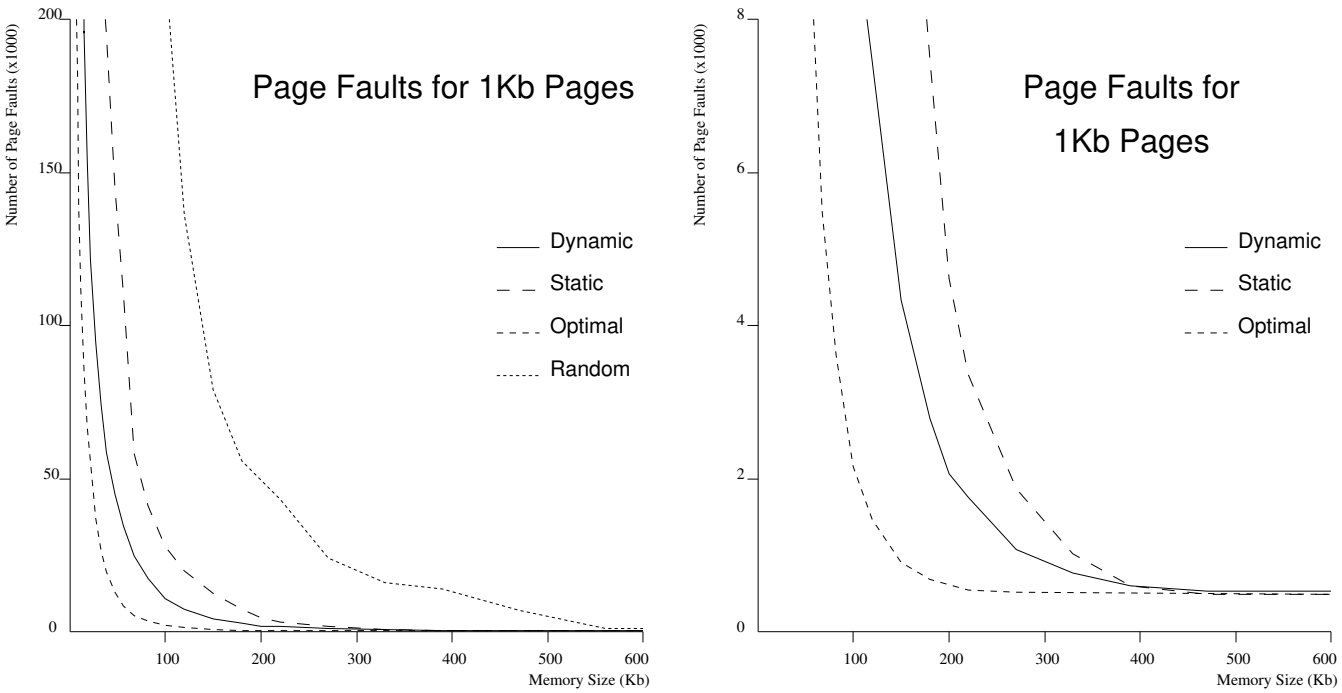


Figure 1: Page Faults (a) With Random Curve (b) Without Random Curve

been observed (where *N* is the number of objects in primary memory). However, due to the varying sizes of objects, the *N*th object is not necessarily the optimal choice of object to purge.

# 6 Discussion

The four models are implemented in C and driven with the memory reference traces gathered from a *Smalltalk-80* system. Three models provide near-optimal, random and LRU dynamic grouping whilst a fourth static grouping scheme is included for comparison. Near-optimal static grouping and random static grouping are not included since they have no implication on the dynamic grouping scheme and have been covered in a previous study [Sta84]. An attempt has been made to keep the models simple without introducing inaccuracies that may bias the results. However, there are several points which deserve mention:

- In the static grouping scheme on a conventional paging system, the model is generous in that it assumes a true LRU page purging policy. However, studies have shown that LRU approximations are sufficiently effective for this to be realistic [EF79].
- For the same amount of primary memory, it is more difficult to approximate LRU purging for objects than for page purging. This is due to the smaller granularity of object-based LRU purging.
- In the dynamic grouping scheme, when a group of objects are brought in from secondary memory, they are considered to be a group of 'most recently used' objects. Altering the position of the newly fetched group in the LRU selection may improve the effectiveness of the dynamic grouping scheme. Clearly, considering a newly fetched group of objects as least recently used may lead to their premature ejection.
- All objects are smaller than the page size — large *Smalltalk-80* objects which do not fit onto a page are dealt with as special cases by the simulator. Objects do not cross page boundaries.
- For simplicity, only one static grouping scheme is modelled. It is considered that previous studies of static grouping schemes [Sta84] are conclusive in that there is no single scheme which is identifiably more effective than any other.
- No 'dirty bit' is included in the conventional paging model. Thus, unchanged pages are purged in the same way as modified pages. This is in accordance with the dynamic grouping scheme which does not create space by deleting unmodified objects.

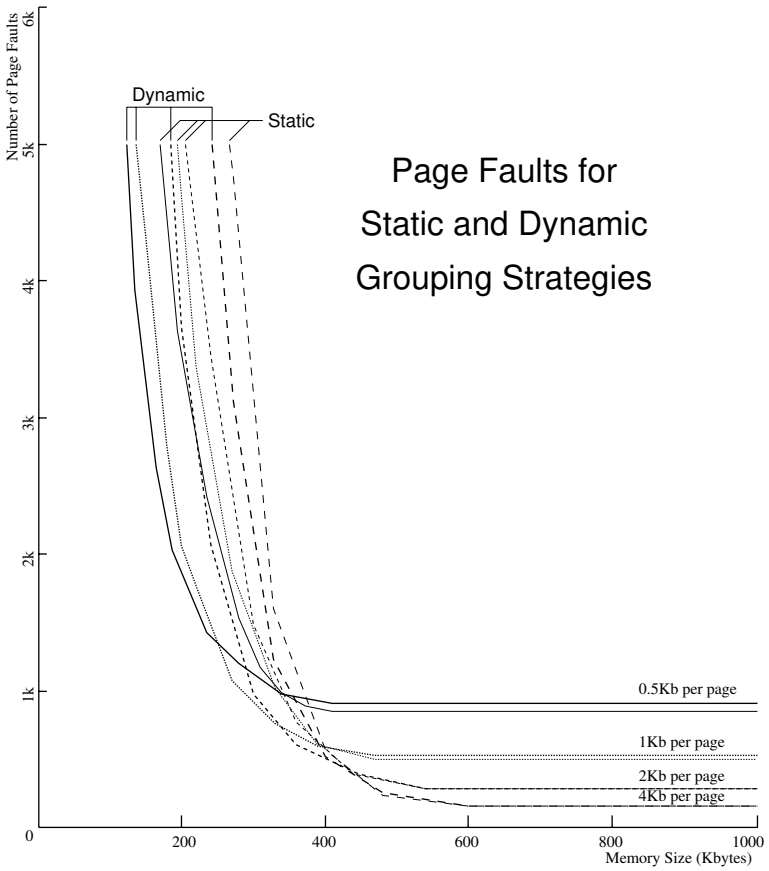


Figure 2: Page Faults for Different Page Sizes

- The memory reference traces do not include accesses to bitmaps and accesses to class fields by the someInstance or nextInstance primitives.
- The garbage collection scheme used conforms to the definition of the virtual machine [GR83] and is the same for all memory traces.

# 7 Results

For the three main traces, each combination of grouping scheme and page size had 15 primary memory sizes. Given that four page sizes were simulated for four grouping schemes, this gives a total of 720 points for all three main traces. The traces represented approximately four million bytecodes and 40 million memory references of *Smalltalk-80* activity each, with a fourth, 19 million bytecode trace being run with much fewer points. No statistical manipulation is attempted, relative comparisons are not made and the results are presented in their absolute form. Results are only presented from trace 1. The graphs obtained from traces 2-4 were very similar to those from trace 1. The simulations predict how real implementations with similar memory and image sizes behave with a reasonable amount of accuracy. However, it is unknown whether the results scale to larger memory and image sizes.

## 7.1 Page Faults

The graph in Figure 1(a) shows the page faults obtained for trace 1 for 1Kbyte pages. Note that each curve falls towards a constant level which represents the page activity caused by starting and terminating the simulations with all objects on the disk. The random figures are an order of magnitude worse than figures for static, dynamic and near-optimal grouping. Figure 1(b) shows the same graphs on a different scale without the random curve. With 200 Kbytes of main memory the dynamic grouping scheme pages 45% as much as the static grouping scheme. At 56 Kbytes of main memory, dynamic grouping activity is 31.5% of static grouping. When paging occurs during *Smalltalk-80* activity (not including that caused by startup or shutdown), the graph indicates that dynamic grouping is always better than static grouping — the curves only meet when no paging is performed.

The graphs in Figure 1 show only 1Kbyte pages. Other graphs were plotted for different page sizes, and Figure 2 shows curves for all the page sizes for static and dynamic grouping with trace 1. For the range of page sizes simulated, all the dynamic grouping simulations performed better than the corresponding static grouping simulations.

### 7.2 Page Activity

Whilst graphs of the page fault totals are useful for comparing the effectiveness of various paging schemes, it is interesting to observe the page fault activity in order to gain an insight into the effect of paging schemes on interactive response. Figure 3(a) illustrates one of the activity traces taken for a 200 Kbyte main memory with 1 Kbyte pages for trace 1. Many of the paging peaks for dynamic grouping are of similar magnitude, but are generally shorter in duration compared with the static grouping trace. Relating the peaks of page activity with new applications being run in the *Smalltalk-80* image shows that the dynamic grouping scheme changes the working set much more rapidly than static grouping. Figure 3(b) shows a section of the same activity graph expanded.

## 8 Conclusion

Dynamic grouping works well for the given combinations of image and memory sizes. It is unknown whether the results will scale to larger images and memory sizes, and it is not envisaged that any accurate predictions may be made until large applications are available. Future research will concentrate on investigating practical implementations of pointer-addressed dynamic grouping virtual memory and simulating such memories in order to estimate and compare cost and performance with conventional paging hardware.

## 9 Acknowledgements

The authors would like to thank Cliff Jones for commenting on the formal specification in Appendix A. Thanks also go to the Department for the provision of computing facilities.

## References

[EF79] Malcom C. Easton and Peter A. Franaszek. Use bit scanning in replacement decisions. *IEEE Transactions on Computers*, 28(2):133–141, 1979.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[LH83] H. Lieberman and C. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Sta84] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[Ste75] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[Ung84] David Ungar. Generation scavenging: A non-disruptive, high performance storage reclamation algorithm. In *Proc. Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh PA, May 1984. ACM SIGSOFT/SIGPLAN.

[UP83] D. M. Ungar and D. A. Patterson. Berkeley Smalltalk: Who knows where the time goes? In Glenn Krasner, editor, *Smalltalk-80: bits of history, words of advice*, pages 189–206. Addison-Wesley, 1983.

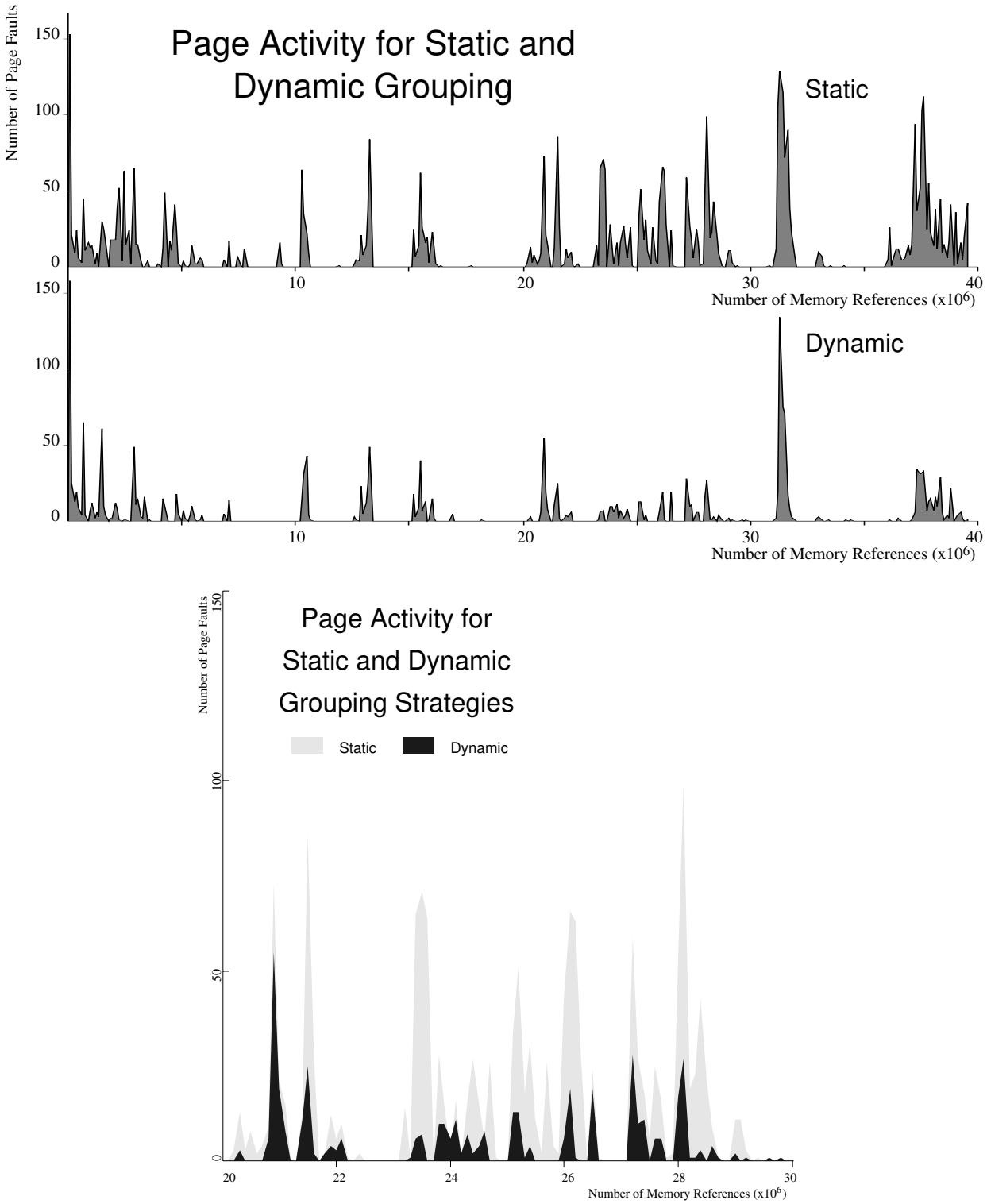


Figure 3: (a) Page Activity for Trace 1 (upper) (b) Expanded Section (lower)

# Appendix A

This appendix contains formal specifications written in VDM [Jon86] of the models used to simulate both static and dynamic grouping schemes. Two specifications are given of both models, the second specification being a more detailed refinement of the first (the refinements have been proved to model the abstract specifications, but the proofs are not included for brevity).

## Conventional Paging

The simplest specification is that of the conventional paging system where the primary memory is modelled as a set of pages. Pages are classed as being either in or out of primary memory. A reference to a page in memory does not affect the state of the system, whilst a reference to a page not in memory causes the required page to be included into primary memory. Clearly, if the primary memory is not full then no purging is required.

$Mem :: PS : Pg\text{-set}$

where

$inv\text{-}Mem(mk\text{-}Mem(ps)) \triangleq \mathsf{card}\, ps \leq store_size_in_pages$

$REFCE\ (p: Pg)$

ext wr $m : Mem$

post if $p \in PS(\overleftarrow{m})$ then $m = \overleftarrow{m}$ else if $\mathsf{card}\, PS(\overleftarrow{m}) = store_size_in_pages$
then $PS(m) - PS(\overleftarrow{m}) = \{p\}$
$\wedge\ \mathsf{card}\,(PS(m) \cap PS(\overleftarrow{m})) = store_size_in_pages - 1$
else $PS(m) = PS(\overleftarrow{m}) \cup \{p\}$

## Conventional Paging—LRU

By adding the concept of sequence to pages in real memory, least recently used purging can be specified. The sequence of pages reflects the sequence of references — the head of the sequence contains the least recently referenced page. If the referenced page is not in primary memory, the LRU page is ejected and the required page fetched.

$Mem_{LRU} :: PL : Pg^*$

where

$inv\text{-}Mem_{LRU}(mk\text{-}Mem_{LRU}(pl)) \triangleq \mathsf{len}\, pl \leq store_size_in_pages \wedge \mathsf{card}\, \mathsf{rng}\, pl = \mathsf{len}\, pl$

$LRU_REFCE\ (p: Pg)$

ext wr $m : Mem_{LRU}$

post if $p \in \mathsf{rng}\, PL(\overleftarrow{m})$ then $PL(m) = remove_pg(p, PL(\overleftarrow{m}))^\frown [p]$
else if $\mathsf{len}\, PL(\overleftarrow{m}) = store_size_in_pages$
then $PL(m) = \mathsf{tl}\, PL(\overleftarrow{m})^\frown [p]$
else $PL(m) = PL(\overleftarrow{m})^\frown [p]$

$remove_pg : Pg \times Pg^* \to Pg^*$

$remove_pg(p, pl) \triangleq$ if $pl = [\,]$ then $[\,]$ else if $p = \mathsf{hd}\, pl$
then $\mathsf{tl}\, pl$
else $\mathsf{hd}\, pl^\frown remove_pg(p, \mathsf{tl}\, pl)$

## Object Swapping

Object swapping groups objects to be purged onto a page to be ejected into secondary memory. In statically grouped systems, the content of a page is the same in primary memory as in secondary memory. With the dynamic grouping scheme, secondary memory is divided into pages whose contents are static. When pages are brought from secondary memory, the objects they contain are scattered into free spaces in primary memory. Similarly, pages being written to secondary memory are composed dynamically by selecting the appropriate objects to purge from primary memory.

The existence of an auxiliary function, $size: Obj \to \{1, \ldots, pg_size\}$, is assumed. This gives the size (in words) of a particular object.

$Obj_mem$ :: $Disk$ : $Pg$-set
$Mem$ : $Obj$-set

where

$inv\text{-}Obj_mem(mk\text{-}Obj_mem(d, m)) \triangleq$
$is_disjoint(m, \bigcup\{OS(p) \mid p \in d\})$
$\wedge \sum_{obj \in m} size(obj) \leq store_size$
$\wedge\ is_prdisj(d)$

$Pg$ :: $OS$ : $Obj$-set

where

$inv\text{-}Pg(mk\text{-}Pg(os)) \triangleq os \neq \{\} \wedge \sum_{obj \in os} size(obj) \leq pg_size$

The following function states that an object may not be on more than one page in a given set of pages, i.e. the sets which constitute each page are pairwise disjoint.

$is_prdisj : Pg\text{-set} \to \mathbb{B}$

$is_prdisj(d) \triangleq \forall d_1, d_2 \in d \cdot OS(d_1) = OS(d_2) \vee is\text{-}disjoint(OS(d_1), OS(d_2))$

$REFCE\ (p: Obj)$
ext wr $disk$ : $Pg$-set
wr $mem$ : $Obj$-set

pre $p \in mem \cup \bigcup\{OS(p) \mid p \in disk\}$
post if $p \in \overleftarrow{mem}$
then $mem = \overleftarrow{mem} \wedge disk = \overleftarrow{disk}$
else $\exists pg_in \in \overleftarrow{disk}, objs_out \subseteq \overleftarrow{mem}, pgs_out \in Pg\text{-set} \cdot$
$p \in OS(pg_in)$
$\wedge\ mem = \overleftarrow{mem} \cup OS(pg_in) - objs_out$
$\wedge\ objs_out = \bigcup\{OS(p) \mid p \in pgs_out\}$
$\wedge\ disk = \overleftarrow{disk} - \{pg_in\} \cup pgs_out$
$\wedge\ is\text{-}prdisj(pgs_out)$
$\wedge \sum_{obj \in mem} size(obj) \leq store_size$

## Object Swapping–LRU

The following specification is the same as the previous object swapping model, except for the inclusion of 'least recently used' (LRU) purging. LRU purging is specified by modelling primary memory as a sequence of objects which reflects the sequence of references to the objects. The head of the sequence contains the least recently referenced object and objects are added to the tail whenever they are accessed. When objects are brought in from the disk, they are also added to the tail of the sequence.

$Obj_mem_{LRU}$ :: $Disk$ : $Pg$-set
$Mem$ : $Obj^*$

where

$$inv\text{-}Obj_mem_{LRU}(mk\text{-}Obj_mem_{LRU}(d, m)) \triangleq inv\text{-}Obj_mem(mk\text{-}Obj_mem(d, \mathsf{rng}\, m)) \wedge \mathsf{len}\, m = \mathsf{card}\, \mathsf{rng}\, m$$

$LRU_REFCE\ (p: Obj)$
ext wr $disk$ : $Pg$-set
wr $mem$ : $Obj^*$

pre $p \in \mathsf{rng}\, mem \cup \bigcup\{OS(p) \mid p \in disk\}$
post if $p \in \mathsf{rng}\, \overleftarrow{mem}$
then $disk = \overleftarrow{disk} \wedge mem = remove_obj(p, \overleftarrow{mem})^\frown [p]$
else $\exists pg_in \in \overleftarrow{disk}, obj_seq_out, tmp_mem \in Obj^*,$
$pg_seq_out \in Obj^{**}, pgs_out \in Pg\text{-set} \cdot$
$p \in OS(pg_in)$
$\wedge\ \overleftarrow{mem} = obj_seq_out^\frown tmp_mem$
$\wedge\ tmp_mem \neq [\,]$
$\wedge\ obj_seq_out = \mathsf{dconc}\, pg_seq_out$
$\wedge\ pgs_out = \{mk\text{-}Page(\mathsf{rng}\, p) \mid p \in \mathsf{rng}\, pg_seq_out\}$
$\wedge\ post\text{-}add_to_mem(OS(pg_in), tmp_mem, mem)$
$\wedge\ disk = \overleftarrow{disk} - \{pg_in\} \cup pgs_out$
$\wedge \sum_{obj \in mem} size(obj) \leq store_size$
$\wedge\ \forall i \in \mathsf{dom}\, pg_seq_out \cdot$
$$\left(\left(\sum_{obj \in \mathsf{rng}\, pg_seq_out(i)} size(obj)\right) + \begin{array}{l}\mathsf{if}\ i = \mathsf{len}\, pg_seq_out \\ \quad \mathsf{then}\ size(tmp_mem(1)) \\ \quad \mathsf{else}\ size(pg_seq_out(i+1)(1))\end{array} > pg_size\right)$$

The *remove_obj* function (signature $Obj \times Obj^* \to Obj^*$) removes an *Obj* from a sequence of *Obj*s; it can be defined in a similar way to *remove_pg*.

$add_to_mem\ (objs: Obj\text{-set})$
ext wr $mem$ : $Obj^*$
post $mem = \overleftarrow{mem}^\frown objlist \wedge \mathsf{rng}\, objlist = objs \wedge \mathsf{len}\, objlist = \mathsf{card}\, objs$