Realisation of a Dynamically Grouped Object-Oriented Virtual Memory Hierarchy[*]

I. W. Williams[*], M. I. Wolczko[†] and T. P. Hopkins.

Dept. of Computer Science, The University, Manchester M13 9PL.

Abstract

Conventional paging systems do not perform well with large object-oriented environments (such as *Smalltalk-80*[1] [GR83]) due to the fine granularity of objects and the persistence of the object space. One approach taken to increase efficiency has been to group objects together in virtual space in an attempt to keep related objects on the same page. Conventionally, grouping has been performed whilst the system is inactive with the objects being ordered in virtual space according to a depth- or breadth-first traversal of the graph formed by the pointers between objects. Whilst such static grouping reduces the amount of paging compared with an arbitrarily grouped system [Sta84], it is not sufficiently effective to eliminate the paging problems of large persistent object-oriented applications.

As part of current research on architectures for high-performance object-oriented machines, a novel dynamically grouped virtual memory system was developed and extensively simulated. With dynamic grouping, objects reside in a virtual object memory and are accessed by pointer-offset addresses. Objects are transferred between primary memory and disk in groups which are determined at run time. Our simulations show that dynamic grouping strategies can achieve significantly better performance than static grouping. In [WWH87] static and dynamic grouping were simulated and it was discovered that for reasonable memory sizes dynamic grouping reduced page faults by up to three times compared with static grouping.

This paper will describe an ideal dynamically grouped virtual memory and will investigate different practical realisations of dynamic grouping schemes. The results of comprehensive simulations of practical implementations will be presented and cost-performance estimations will be compared with conventional memory systems.

1 Introduction

Virtual memory enables programs and data to be much larger than primary memory without the need for explicit management of transfers between primary and secondary memory by

[*] A version of this paper appeared in *Proc. Workshop on Persistent Object Systems: Their Design, Implementation and Use*, availiable as Persistent Programming Research Report PPRR-44-87, Universities of Glasgow and St. Andrews, Aug. 1987, pp.298–308.

[†] Supported by the Science and Engineering Research Council

[1] *Smalltalk-80* is a trademark of Xerox Corp.

the programmer. 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 and when required. Pages are typically several thousand bytes long and page sizes are chosen to help minimise the amount of paging.

Whilst this system works well with most applications written in conventional languages such as FORTRAN or C, modern symbolic processing languages such as LISP or *Smalltalk-80* exhibit properties which hinder efficient implementation on conventional paging systems. This is because the data structures used by symbolic processing languages are finer-grained and are more richly interconnected than those in conventional languages. The dynamic nature of the fine-grained, persistent information manipulated by symbolic processing languages contrasts strongly with the behaviour of conventional systems. However, little work has been done in the past into alternative virtual memory architectures for symbolic processing and the problem of locality is still prevalent.

2 Problems With Conventional Virtual Memory

2.1 Locality of Reference

A program in a conventional system contains one or more contiguous segments of code, operating on some data. A segment will typically occupy a few hundred Kbytes of memory and program execution will be constrained to fall within those segments (except for operating system calls). Thus programs occupy contiguous sections of store which are large in comparison with page sizes. Primary memory can contain a good working set of pages, and pages which are swapped in are likely to have a significant amount of useful content.

Figure 1 illustrates a schematic representing an object-oriented system implemented on a conventional virtual memory. In most object-oriented systems, the 'unit of locality' is a fine-grained object small in comparison to page size (50 bytes on average in a *Smalltalk-80* system [Ung84]). An application is developed by adding new objects to the system. The representation of a new application will consist of a relatively small number of objects scattered among the many objects that represent the whole system. Primary memory will contain a few objects that are required by the application, together with many more objects which happen to reside on the same pages as the objects required by the application. A working set of objects in primary memory is unlikely to be achieved unless most of the whole system is in primary memory. A page that is swapped in will not contain many useful objects other than the object that caused the page to be swapped in. Consequently, the granularity of conventional paging systems is far too large and causes serious performance degradation for object-oriented systems.

Grouping together in virtual memory the objects that represent a particular application would obviously alleviate these problems. However, most applications rely heavily on inherited behaviour from existing objects, thus enabling an application to reuse any portion of a large, persistent environment. In such circumstances, grouping objects by application is not feasible due to the sizeable body of code which is shared by many applications. In contrast, conventional systems do not share code to such a degree.

Additionally, conventional programs have statically-bound procedure calls, whilst in object-oriented languages binding is often performed at run time. This means that in general it is not feasible to determine the precise body of code that may be used by an application.

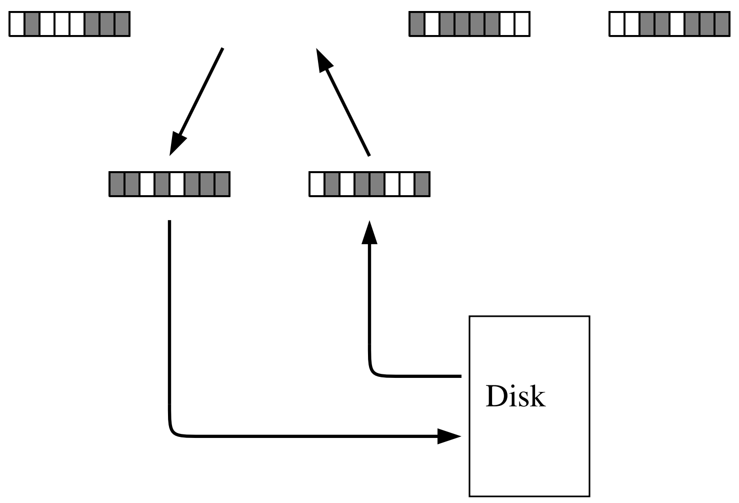

Figure 1: Schematic of a Conventional Paging System

2.2 Dilution of Virtual Memory

Having determined that locality is a severe problem because of the discrepancy between the size of objects and the size of pages, another attribute of object-oriented systems which causes performance problems can be identified. The objects accessed by any particular application constitute only a tiny fraction of the potentially accessible objects. Since the objects accessed by an application are distributed randomly amongst other objects, this leads to a *dilution* of the virtual memory, where dilution can be considered to be the ratio of accessed to accessible virtual memory locations. The natural persistence of many object-oriented systems and the generation of garbage exacerbate the dilution problem. In contrast, conventional systems have the representations of applications contained within large contiguous sections of virtual memory. Therefore it is straightforward to determine which pages constitute an application, and arrange for primary memory to be rich in those pages.

2.3 The Cumulative Effect

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, an attempt is made to place a number of related objects 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 practice, it is not possible to satisfy both of these aims simultaneously without unacceptable performance overheads.

There would be no performance problems if single objects could be swapped between disk and primary memory without much overhead (this was attempted in LOOM [Kae86]). Unfortunately, the time taken by a disk head to reach the disk block containing an object is significantly longer than the time taken to transfer the object into primary memory. Consequently, swapping single objects is not efficient. If a group of related objects were stored on adjacent disk blocks, the group could be swapped in whenever any of the objects within the group was required. With a sensible group size, the overhead of head seek times for virtual memory transfers could be reduced to an acceptable level.

3 Dynamically Grouped Virtual Memory

A number of *Smalltalk-80* implementations utilise grouping schemes on top of paged virtual memory systems by attempting to place related objects on the same page. However, such techniques have relied on the static structure of the system for grouping information, and perform grouping infrequently, usually off-line. In a previous study [WWH87], it was determined that grouping objects at run-time based on recent behaviour performed significantly better than other grouping strategies. The proposed Dynamically Grouped Virtual Memory (DGVM) dynamically groups objects based on their dynamic behaviour.

The principles of operation of the DGVM are as follows. Accessing an object not in primary memory causes the secondary memory page containing the object to be swapped in. 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, space is generated in primary memory, if required, by ejecting objects into secondary memory. Also, the garbage collector reclaims space occupied by a garbage object by adding it to a free list.

An efficient implementation of the DGVM requires pointer-offset object addressing. Contrary to this, some *Smalltalk-80* implementations use virtual addresses to directly access objects [UP83] thus saving an object table indirection. However, when objects are to be dynamically relocated in memory, the object table indirection is clearly invaluable. There are two possible implementation schemes for the object table indirections. First, the object table may translate pointers into virtual addresses for the underlying machine. Alternatively the pointers may be translated into real primary or secondary memory addresses. Because the DGVM is being developed for a new machine architecture, there is no requirement to consider an additional level of conventional virtual addressing. Consequently, pointer-offset virtual addresses are translated onto real addresses.

4 Consequences of Dynamic Grouping

The problems caused by the mismatch between object sizes and page sizes are eradicated by the dynamic grouping of objects onto a page. It has been shown that the group of objects which were last active together proves to be a good measure of which objects are related [WWH87]. Dynamically constructing a page by ejecting the least-recently-used objects proved to be an effective mechanism for ensuring that related objects were grouped together, thus significantly reducing the problems caused by the mismatch between page and object sizes.

To illustrate how the DGVM can improve paging performance, consider the following example. When the user of a *Smalltalk-80* system chooses the “accept” menu item from a

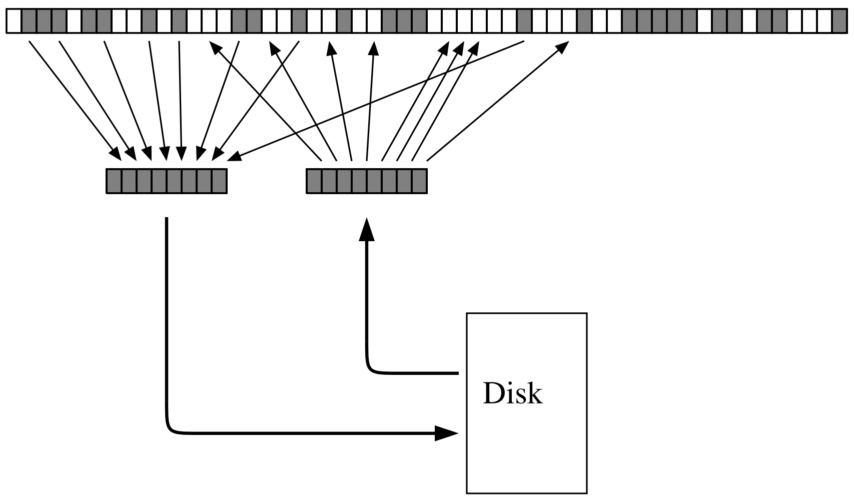

Figure 2: Schematic of a Dynamically Grouped Virtual Memory

browser, the compiler, consisting of several hundred objects, is invoked. During the compilation process, the objects required during compilation will be placed close to each other in an ordering of objects by time of access. If the compiler is not invoked again for some time, all the compiler related objects will paged out together. When the compiler is invoked again, access to any object in the group will cause all the related objects on the same page to be paged in, ready for immediate use.

Any erroneous or coincidental groupings will be removed over time. For example, if the user's clock icon ticks during the compilation, thus marking some of the clock objects as part of the group containing the compiler, those objects may be paged out as part of the compiler group. However, at the next use of either the compiler or the clock objects it is most unlikely that both sets of objects will be accessed at the same time, and the groups will be separated again. As the system evolves, any coincidental groupings will be filtered into their constituent parts by the regrouping mechanism, and the performance of the system will improve.

The sharing of large parts of an object-oriented system by an inheritance mechanism also proves to be a problem in a conventional system, since any grouping performed statically would only benefit applications that were grouped together with the shared code. With dynamic grouping, groupings can change constantly. An application will soon acquire a working set of objects that represent the required inherited behaviour.

One of the best results of dynamic grouping is the dramatic effect on the dilution problem. In conventional systems, the persistence of objects and the generation of garbage dilute pages and reduce their useful contents with serious effect. Dynamic grouping actively filters primary memory removing all objects which are not active, thereby maintaining a larger proportion of useful objects in primary memory.

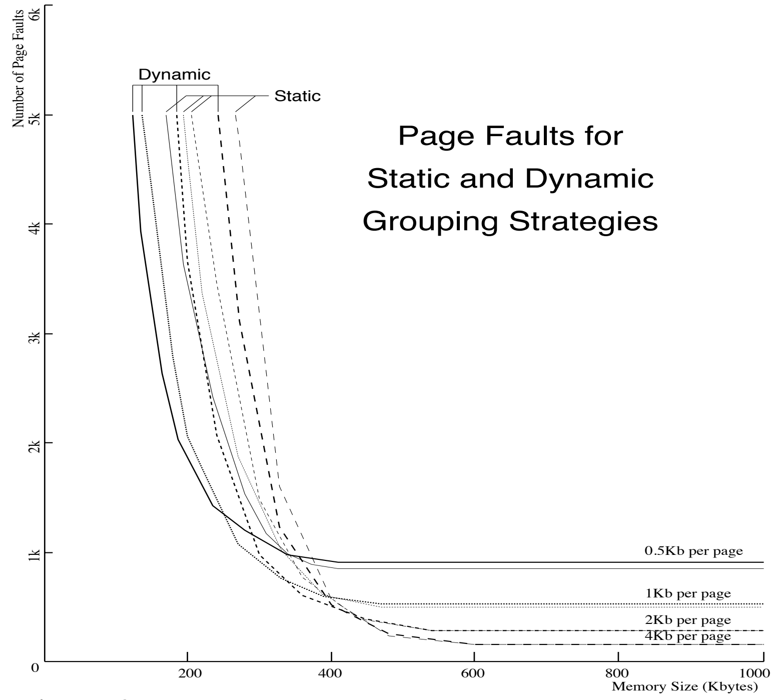

Figure 3: Graph of Page Reduction for Various Page Sizes

5 Effectiveness of Dynamic Grouping

Table 1 illustrates a typical page fault rate increase that would be experienced if the DGVM is replaced by a conventional paging system. These figures are derived from extensive simulations described in [WWH87]. When little paging is performed, no decrease in performance would be observed. However, when the primary memory size is 10% of the system size, using a conventional memory would decrease performance by 286%. When the total space occupied by all objects in the system is three times the primary memory size, dynamic grouping performs twice as well as static grouping (see Table 1). Furthermore, as the ratio of total object space to primary memory size is increased, the performance of dynamic grouping with respect to static grouping also increases.

When no objects are swapped out during the course of the application, static and dynamic grouping have no useful effect, as every object is paged in and out precisely once. This occurs when there is sufficient primary memory for all the objects required by the application. As the memory size decreases, and swapping starts to occur, the page fault rate for each grouping scheme increases. However, the page faults incurred by dynamic grouping are always fewer than those incurred by static grouping.

In Fig. 4, the number of page faults per 10^5 memory references is plotted against time (in units of memory references) for an application running in simulated statically- and dynamically-grouped virtual memories. As can be seen, the peaks of activity within each system correspond closely in time, but the activity is much lighter in a dynamically-grouped system. Careful examination of these traces has shown that the effect described in the earlier 'compilation' example is the main cause of the decreased paging in a dynamically-grouped system.

Some attempt has been made to measure the dilution of primary memory in the two systems. Whenever a page is fetched from secondary to primary memory, it is possible that a number of objects on that page will be ejected back to secondary memory without ever being accessed

Size primary memory : application size (%)	Page Faults rate increase (%)
50	0
40	133
30	192
20	263
10	286

Table 1: Summary of Page Fault Rate Increase Using Conventional Memory and 1Kb pages

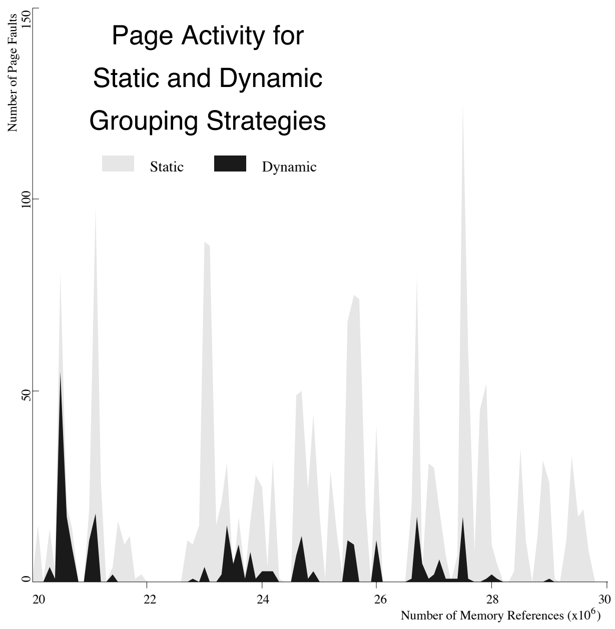

Figure 4: Trace of Paging Activity

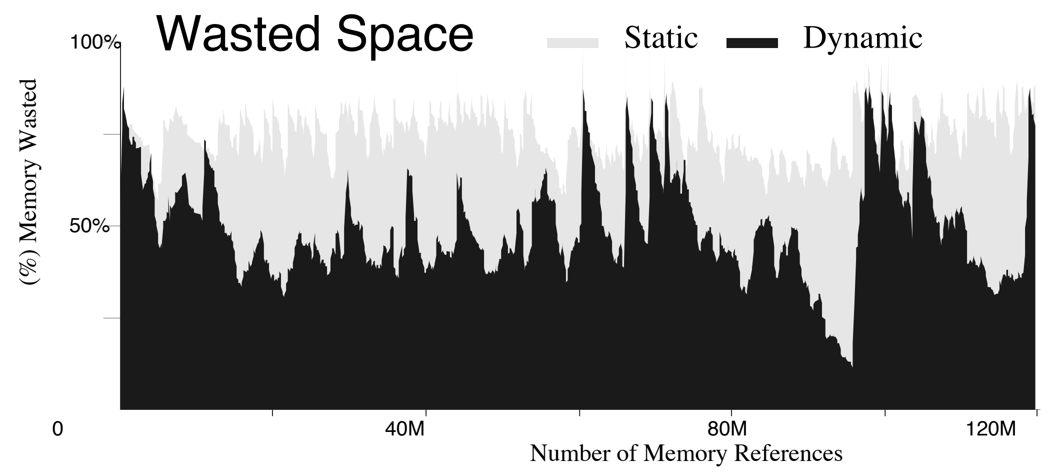

Figure 5: Trace of Wasted Space

while in primary memory. The space these objects occupy in primary memory is essentially wasted. Plotting the total amount of wasted space (as a percentage of primary memory size) in a particular system against time yields some insight into the effectiveness of the page- or object-purging policies in use. In Fig. 5, the percentage of wasted space is plotted against time for an application running for a duration of 120 million memory references. The amount of space wasted by a statically-grouped conventional virtual memory remains constant at about 75%, whilst the DGVM system shows changing activity, averaging about 40% wasted space. Relating this trace to activity in the application, it has been observed that sudden increases in paging activity correspond to changes in activity in the application. Thus while the activity remains in one part of the application, the page fault rate in a dynamically-grouped system decreases as the working set is rearranged and improved.

6 Implementation of Dynamic Grouping

The effectiveness of the dynamically grouped virtual memory concept has been demonstrated in previous sections [WWH87]. However, this is of limited use unless an efficient realisation of a DGVM can be developed. The remainder of this paper considers the requirements of such an implementation, and evaluates the approximate costs and effectiveness of a practical implementation. Clearly, the speed of address translation and the size of the object table are issues crucial to the efficient implementation of a DGVM system. Another issue is the least-recently-used ordering that is required on all objects. Some mechanism is required to maintain LRU ordering, or a good approximation to LRU ordering.

If an implementation of a DGVM system is to be economically viable, it must achieve its benefits with little cost additional to that associated with a conventional memory system. In the following sections it is outlined how such an implementation may be achieved, and the

No. Bits	Frequency	%	Cum. %	Wasted %
1	26258415	20.44	20.44	0.00
2	24426589	19.01	39.45	1.52
3	27373669	21.31	60.76	2.82
4	32920993	25.62	86.38	3.88
5	10427741	8.12	94.50	5.92
6	5819090	4.53	99.03	8.01
7	927129	0.72	99.75	10.30
8	167225	0.13	99.88	12.52
9	29954	0.02	99.90	14.65
10	8288	0.01	99.91	16.68
11	11564	0.01	99.92	18.62
12	22676	0.02	99.94	20.47
13	26429	0.02	99.96	22.23
14	9728	0.01	99.97	23.92
15	16384	0.01	99.98	25.53
16	29532	0.02	100.00	27.08
TOTAL	128475406	-	-	-

Table 2: Efficiency of Pointer-Offset Addressing

principles upon which it is based. By dynamic analysis of certain properties of a typical object-oriented system (*Smalltalk-80*) we can find techniques to improve the economy of design of a dynamically-grouped virtual memory to a level competitive with conventional paging systems whilst maintaining its significant performance advantage. The aim is to show that:

1. Pointer-offset addressing is efficient in its use of address bits.

2. The size of a hardware object table is comparable to hardware page table sizes.

3. Object level LRU purging approximations are sufficiently effective.

6.1 Efficiency of Pointer-Offset Addressing

Objects are accessed by pointer-offset addressing, the pointer being the unique name for the object and the offset being the index into that object. The pointers are virtual addresses that require translation into real addresses by looking up an entry in the object table. The object table contains information about the location of all objects in the system and is consequently large. Each access to an object must interrogate the object table to determine the real address of the object to be accessed. Obviously, objects vary greatly in size but the number of bits required to represent the offset remains the same. A 16 word object only requires 4 bits to index all of the object. If offsets were 16 bits in length for example, 12 bits would be wasted. Such inefficiencies are clearly undesirable. However, choosing smaller offsets will limit the size of objects directly representable.

The size of the offsets is a tradeoff between wasted address bits and size of object representable in contiguous memory locations. Table 2 contains statistics for the dynamic usage of offset bits for a typical *Smalltalk-80* system during 19M bytecodes of activity. For each memory access the minimum number of bits required to represent the offset was determined and the number of occurrences of the computed minimum noted. This gives a measure of the nature of accesses to objects. Approximately 20.4% of accesses are to the first two words of an object whilst 25.6% of accesses are to words 8 through to 15 of an object. A cumulative figure indicating the fraction of all accesses representable for each offset size is also given. As can be seen, the vast majority of accesses (99.88%) require 8 offset bits or less. Furthermore, static analysis of a number of *Smalltalk-80* images showed that 99.5% of objects contained less than 256 words of data.

However, using pointer-offset addressing is potentially inefficient since many objects do not require the full complement of index bits to access all their parts. The number of bits wasted is evaluated by subtracting the minimum number of bits required to represent the offset from the number of bits used to represent the object index. Thus 3 bits would be wasted if location 27 of an object was accessed using an address format with 8-bit offsets. By computing this value for each memory access and noting the frequency of occurrence, a good measure of wasted address bits can be derived. The rightmost column of Table 2 illustrates wasted address bits as a fraction of the address width (pointer-offset width); with the above example, the 3 wasted bits are taken as a fraction of 40 bits (32-bit pointers and 8-bit offsets). As can be seen, using 16 bits for the offsets wastes 27% of the address bits, although a system with 4-bit offsets only wastes 3.9% of address bits.

A ratio of merit could be derived by comparing wasted address bits with number of objects directly representable for each offset size. However, this would be unfair as both measures are not equally important. It is important to note that the figures derived for the wasted address bits are only a measure of redundancy for various numbers of index bits. Pointer-offset addressing is fundamental to the implementation of object-oriented systems and cannot be compared sensibly with linear virtual addressing. For the design of the DGVM, it was considered that 8-bit offsets were the best choice since they gave a good number of directly representable objects (0.12% less than the optimal) and incurred only a 12.4% redundancy in the address bits (a 54% improvement on 16-bit offsets). For the 0.12% of accesses not directly representable, a tree of indirection pointers would be used.

6.2 Size of Hardware Object Table

An important issue in the design of a DGVM is the object table. The object table is used on every memory access to translate from a pointer-offset virtual address to a real address. Consequently, it is of vital importance that object table translations can be completed with low latency and high throughput. It is expected that a typical object-oriented system running on such a machine will have many thousands, possibly millions of objects. Clearly, it is not economic to provide a hardware object table with one entry for every object in the system. The hardware object table must therefore be a cache of a much larger structure, partially implemented by tables in main memory. For performance reasons, the cache hit-rate must be high, although this may imply an impractically large object table cache.

To try to determine how large an object table cache must be, a long memory trace (120M memory references) for a typical *Smalltalk-80* session was analysed. For the trace, a list was maintained for all the objects that had been referenced, ordered by time of access. Thus, the most recently accessed object was at the head of the list, and the least recently accessed object at the tail. For each access, the object accessed was moved to the head of the list, and the position

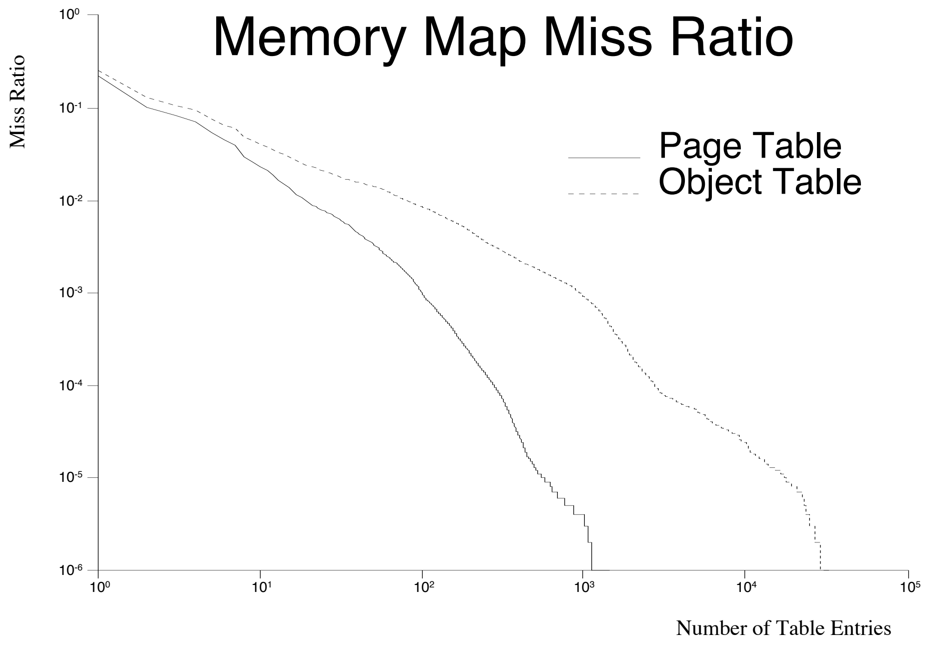

Figure 6: Memory Map Miss Ratio

Hit Rate (%)	OT Entries	PT Entries	Object Table Size (×Page Table Size)
90.0000	4	3	1.33
99.0000	82	20	4.10
99.9000	959	101	9.50
99.9900	2790	282	9.89
99.9990	17644	548	32.20

Table 3: Object Table Size Summary.

it had come from was noted. The frequency of each position gives a measure of locality from which the effectiveness of assorted object and page table cache sizes can be determined. The maintenance of this list essentially models a fully associative object table cache with true LRU purging of entries. Since the frequency of access to each position in the list is available, the hit ratio for various sizes of page and object table caches can be derived (Table 3).

It might be expected that an object table cache would be 20 times the size of an equivalent page table cache because there are on average 20 objects per page (for 1Kb pages). However, the dynamic locality inherent in object-oriented systems ensures this is not true and a high hit rate can be achieved with a small object table cache. The measure of locality described above was performed on both conventional and dynamically grouped virtual memory simulations. The results of both simulations are presented in Fig. 6 and represent the logarithm of the miss ratio (y-axis) against the logarithm of the number of entries in a fully associative mapping table (x-axis).

As can be seen from the summary in Table 3, when very low miss ratios in the order of 1 per 10^5 accesses are required, the size of object table cache required to sustain such performance is significantly larger than the equivalent page table cache size. As miss ratio performance drops to 1 in every 10 access, the required object and page table cache sizes are approximately the same. For the design of the DGVM, it is considered that a hit rate of 99% is desirable, thus requiring a fully associative object table cache with only 82 entries. Whilst this is approximately 4 times the size of an equivalent page table, the ease by which such a small number of entries can be implemented reduces the significance of this figure.

6.3 Effectiveness of LRU Purging Approximations

In the previous section it was assumed that the hardware object table cache would be purged in a true LRU fashion. Also, the DGVM paging simulations assumed that objects were purged from primary memory in a true LRU fashion. Whilst it is feasible to implement such a purging policy, it is an apparently expensive operation requiring a total ordering on objects by time of last access. Approximations to true LRU purging have been investigated thoroughly in conventional operating systems research [EF79] and it has been shown that simple algorithms provide a good approximation to true LRU behaviour. If such simple approximations worked equally well with object level purging, then the effectiveness of dynamic grouping could be maintained. Of the myriad of LRU approximations, the *clock* algorithm is one of the simplest [EF79]. The clock algorithm maintains two fields per entry in the memory map table. The first field represents the *use bit* and is set to a 1 every time the entry is accessed. The second field contains the *unreferenced scan count* which contains the number of scans the entry has survived whilst being inactive (use bit = 0). When an entry is to be purged a scan of the memory map table is performed. If the used bit of an entry is set to a 1, the used bit *and* the unreferenced scan count are cleared. If an used bit is 0, the unreferenced scan count is incremented. When an unreferenced scan count is incremented past a certain value termed the *ejection threshold*, the entry is purged and the scan halted. If none of the unreferenced scan counts exceed the ejection threshold, the ejection threshold is decremented and the whole scan repeated.

There are two measures of effectiveness for such a scheme. The most important result is the number of page faults experienced in comparison with true LRU purging. A second useful measure is the number of increments to the scan pointer — this gives an indication of the amount of work done to purge table entries. To determine whether object level LRU approximation is less effective than page level LRU approximation several simulations were performed with a typical session on a *Smalltalk-80* system lasting 5M bytecodes. True LRU dynamic and static grouping schemes were compared with their equivalents using the clock approximation algorithm. Table 4 illustrates the results.

The number of scans an inactive entry is allowed to survive can have a bearing on the effectiveness of the approximation. Several versions of the clock algorithm were simulated with different ejection thresholds. The values of the ejection thresholds were chosen as powers of two with an 8-bit scheme starting with an ejection threshold of 255. Table 4 contains absolute values of page faults and number of scan increments for the trace. Comparisons are made by giving a percentage difference between approximations and the best case for LRU static and dynamic grouping.

LRU approximations for conventional systems are successful providing performance which is 1.4% down on optimal with only a 1-bit ejection threshold. As the ejection threshold value is increased the performance increase is negligible. With object level purging on the other hand, LRU approximations are significantly less effective. The best performance was achieved with

Scheme	Page Faults	% Diff. Stc$_{LRU}$	% Diff. Dnmc$_{LRU}$	Scans	% Diff. Stc$_{LRU}$	% Diff. Dnmc$_{LRU}$
Stc$_{LRU}$	6720	0.00	-271.27	—	—	—
1-bit	6811	-1.35	-276.30	23872	0.00	82.86
2-bit	6741	-0.31	-272.43	31272	-31.00	77.54
3-bit	6745	-0.37	-272.43	31343	-31.30	77.49
4-bit	6732	-0.18	-271.93	31540	-32.12	77.35
5-bit	6718	-0.03	-271.16	31564	-32.22	77.33
6-bit	6729	-0.13	-271.77	31166	-30.55	77.62
7-bit	6745	-0.37	-272.65	31343	-31.30	77.49
8-bit	6745	-0.37	-272.65	31343	-31.30	77.49
Dnmc$_{LRU}$	1810	73.07	0.00	—	—	—
1-bit	3437	48.85	-89.89	139258	-483.35	0.00
2-bit	2869	57.31	-58.51	129052	-440.60	7.33
3-bit	2798	58.36	-54.59	141303	-491.92	-1.47
4-bit	2751	59.06	-51.99	136358	-471.20	2.08
5-bit	2781	58.61	-53.64	148175	-520.70	-6.40
6-bit	2760	58.92	-52.48	143251	-500.08	-2.86
7-bit	2744	59.17	-51.60	139485	-484.30	-0.16
8-bit	2799	54.64	-58.34	150525	-530.55	-8.09

Table 4: Effectiveness of LRU Approximations. Page faults for true LRU statically (Stc$_{LRU}$) and dynamically grouped (Dnmc$_{LRU}$) virtual memories are given. The differences between true LRU performance and LRU approximations are given for each scheme. Also, the number of scan increments are compared.

a 7-bit ejection threshold which increased paging by 51.6% compared with true LRU object purging. A single bit LRU approximation increased paging by almost 90% compared with true LRU DGVM. Clearly, the clock algorithm of LRU approximation is unsuitable for object level memory management. Although it performs well with conventional virtual memory, the smaller granularity of objects have a drastic effect on its efficiency.

7 Summary of Implementation Results

It is difficult to derive an accurate cost of a DGVM implementation as there are many tradeoffs to be made. A high performance implementation could be constructed, or a slow inexpensive solution may be chosen. Under heavy paging loads for example, a single bit LRU approximation, 4 entry object table cache with 4-bit offsets would probably provide a system with better performance than a conventional statically grouped system. However, the choice of parameters involves an unavoidable subjective compromise between performance and cost.

The number of offset bits was chosen to be 8 since it substantially reduces the redundancy of 16-bit offsets without compromising the accessibility of objects. The next obvious choice of 4-bit indices would further reduce redundancy to 3.9% but the amount of non-primitive accesses is too large at 13.6%. Anything greater than 8 bits does not give sufficient increase in primitive accesses to warrant the extra redundancy. A hit rate of 99% on the cached object table is acceptable and requires a 128 entry fully associative memory map. However, the full implication of object table cache size cannot be estimated at present as it is critically dependent on the cost of a miss — a value which cannot be accurately determined until more detailed simulations have been performed.

It has been shown that a traditional LRU approximation algorithm does not perform well on an object purging basis. Whilst the paging performance of a DGVM utilising such poor LRU approximation will still perform significantly better than a conventional virtual memory, great improvements are to be gained if an efficient and effective method of LRU approximation could be found. According to the results in Table 4, using 1-bit LRU approximations, DGVM would reduce the paging of a conventional system by 49.54% (when main memory is 20% of system size).

8 Conclusions and Future Research

Dynamically grouped virtual memory has significantly better paging performance than conventional statically grouped virtual memory systems. When only 20% of a *Smalltalk-80* system can occupy main memory, a conventional paging system with 1Kb pages increases paging by 260% compared with a DGVM with the same page size. With light paging loads, there is little difference between conventional virtual memory and DGVM. As paging loads increase, the performance of DGVM relative to conventional paging also increases.

DGVM can be implemented efficiently at little cost in excess of the cost of conventional paging implementations. Pointer-offset addressing can be made efficient by careful choice of offset size to reduce redundancy. The object table can be effectively cached with approximately 100 entries of a fully associative memory providing 99% hit rate during address translation. Finally, traditional LRU approximations do not work sufficiently well for object level purging. Whilst they provide a system that performs substantially better the conventional paging systems, large performance improvements could be gained by developing better object purging LRU approximations. This is a target for future research.

9 Acknowledgments

The authors would like to thank Michael Fisher for suggesting the method of determining the miss ratios for various sizes of object table and page table. 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 Publishing Company, Inc, 1983.

[Kae86] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. *ACM SIGPLAN Notices*, 21(11):87–106, November 1986. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[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.

[Ung84] David Ungar. Generation scavenging: A non-disruptive, high performance storage reclamation algorithm. In *Proceedings of the 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 Publishing Company, Inc, 1983.

[WWH87] Ifor Wyn Williams, Mario I. Wolczko, and Trevor P. Hopkins. Dynamic grouping in an object oriented virtual memory hierarchy. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, volume 276, pages 79–88. Springer-Verlag, Paris, June 1987.