Multi-level Garbage Collection in a High-Performance Persistent Object System*

Mario Wolczko†
Department of Computer Science, University of Manchester
Manchester, U.K.
mario@cs.man.ac.uk

Ifor Williams
Department of Computer Science, University of Manchester
Manchester, U.K.
ifor@cs.man.ac.uk

Abstract

Conventional garbage collectors exhibit poor virtual memory behaviour. This paper describes a novel garbage collection system that has been designed to cooperate with an object-based virtual memory system so that both their aims are satisfied. The garbage collection system has been separated into parts, each part tailored to the characteristics of one level in the memory hierarchy.

As the sizes of our persistent object systems increase, so do our garbage collection problems. Whereas for small systems it is acceptable to rely on fast memory access speeds in our garbage collectors, we cannot do this in large systems, where most of our objects are in secondary storage. The classic symptom of this problem is that *garbage collectors antagonize virtual memory systems*. They display the kind of behaviour that defeats virtual memory algorithms, and degrades system performance due to excessive paging.

We believe that many of the causes of these problems are removed if both the virtual memory and garbage collection systems are designed with persistent object storage in mind. In this paper we describe the garbage collection and virtual memory systems of the MUSHROOM object-based architecture, and show how they *cooperate* to their mutual benefit.

In the next section we outline the virtual memory problems of traditional garbage collectors. Next, we provide an overview of the MUSHROOM virtual memory system, and then describe the garbage collection system in detail. Finally, we compare our approach with previous work.

*A version of this paper is to appear in the Proceedings of the Fifth International Workshop on Persistent Object System, Pisa, Italy, Sept. 1–4, 1992.

†To whom all correspondence should be addressed

1 Garbage collection problems in virtual memory systems

Garbage collection is still an important problem. Although some years ago it appeared that garbage collectors had become fast enough to be insignificant consumers of CPU time [1], it has now become apparent that garbage collection performance has not kept pace with improving performance in object-oriented languages [2]. Hence, garbage collection can be a significant overhead in an advanced implementation of an object-oriented language [3].

Furthermore, garbage collection poses severe problems for virtual memory systems. Simple garbage collectors access all live objects, many of which are inactive and hence paged out, causing large numbers of page faults and increased disk traffic. The page faults may also cause active objects to be ejected from memory, increasing access times. These problems are in addition to those caused by the mutation of fine-grained data structures, resulting in a loss of locality within pages over time.

Several schemes have been devised in an attempt to ameliorate these problems. The Baker collector [4] compacted virtual memory as it traversed the network of objects, in an attempt to regain lost locality. However, it still accessed all live objects. In order to counter these problems, generational collectors were developed which concentrated their activity on those parts of the system in which most garbage was created [5, 6, 7, 1, 8]. These collectors access the majority of data infrequently, attempting to avoid unnecessary paging, and also compact virtual memory in the more active areas.

However, these schemes do not address the degradation of locality in older generations of objects. The problem here is that objects are allocated to pages in an essentially arbitrary way. When a page is fetched from disk, very few objects on the page are of immediate use; many are there "by accident". This can result in excessive paging at changes of working set.

To address these problems, we have devised an architecture and virtual memory system which tries to prevent locality problems from arising, rather than attempting to cure them later. It does this by using a virtual memory system in which relocating an object is cheap, and hence an object need not be condemned to share a page with the same neighbours for its entire life. (Schemes with similar aims, but different approaches, are compared in Section 8 [9, 10].)

2 The MUSHROOM virtual memory system[1]

The MUSHROOM Project at the University of Manchester has designed a high-performance architecture for object-oriented computing, and is currently constructing a prototype implementation of this architecture. Amongst other innovations, the MUSHROOM architecture incorporates a virtual memory system that caters for the unusual demands of fine-grained object systems:

- Addressing is object-based, using a two-part address (object identifier and offset).
- Each memory word and register is tagged; primitive types (integers, reals, instructions, etc.) are distinguished from object identifiers.

[1] This section may be safely skipped by readers familiar with the architecture from [11, 12, 13, 14].

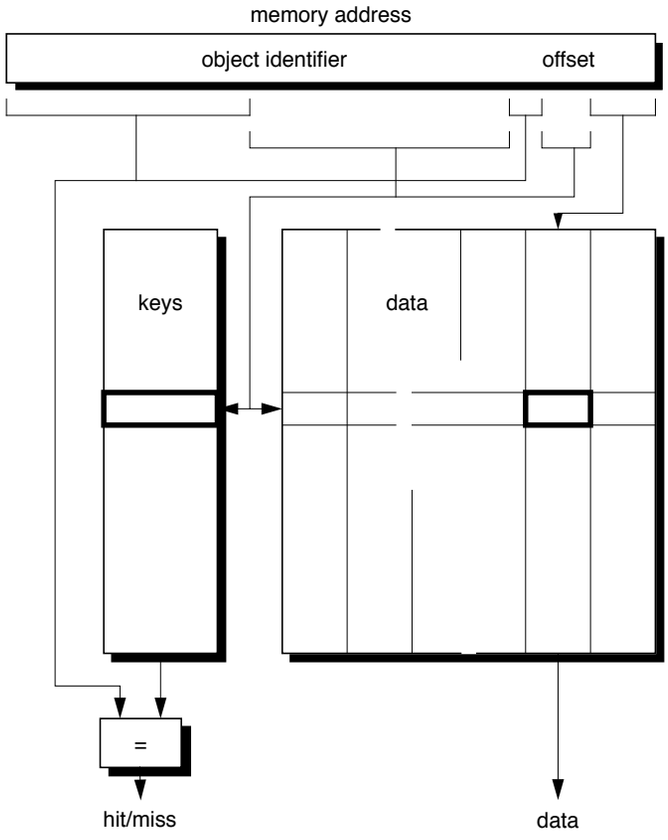

Figure 1: The structure of the MUSHROOM data cache

- A virtually-addressed object-level cache provides fast average access times, without constraining the arrangement of objects in memory or on disk.
- A dynamically-grouped virtual memory transfers collections of objects to and from secondary storage more effectively than conventional paging systems, by choosing the group of objects to place on a page based on recent system activity.

In the MUSHROOM architecture, all data accesses, due to LOAD and STORE operations, are first directed to a hardware data cache (see Fig. 1). A memory address, consisting of an object identifier and the offset within that object, is decomposed into three parts:

1. The row index selects a row from the cache (i.e., a cache line).
2. The column index selects a column from the row, (i.e., an individual data word).
3. The remainder is checked against a stored key to determine whether the cached value is truly associated with the address.

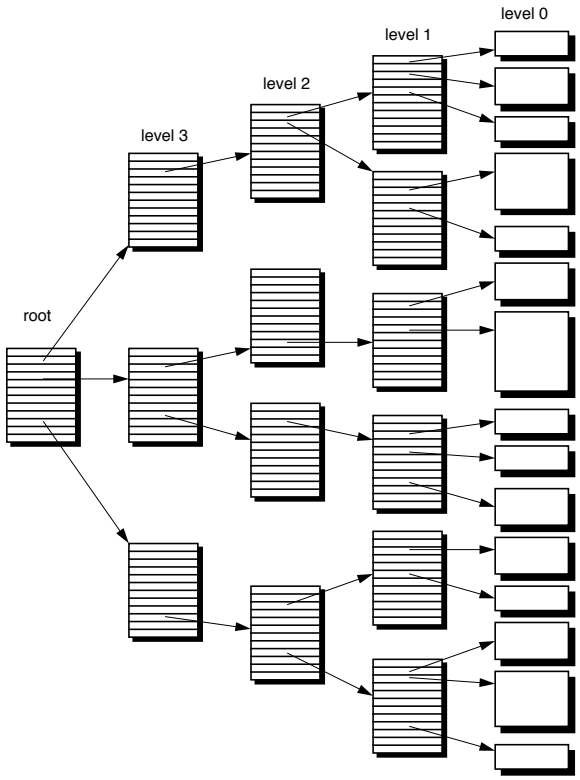

Figure 2: The structure of the MUSHROOM object table. Objects at level 0 are "user" objects; those at other levels are "object table objects". Each object table object can contain the addresses of up to 128 other objects.

Each row of the cache stores part of a single object (or all, if the object will fit). This structure of cache, with a suitable number of rows and columns, and a pipelined implementation, has a very high hit rate and low average access time for Smalltalk programs [14, 13].

Should the cache miss, the appropriate cache line has to be filled by obtaining the data from main memory. This may also require the current contents of the cache line to be written back to memory. To perform both these operations, we need to know the main memory address of an object. This information is held in a data structure known as an *object table*. Conceptually, the object table is a huge array, indexed by object identifier, that holds the real address of an object, and some housekeeping information. Actually, the object table is itself broken into objects, known as object table objects, each of these having its address stored in an object table table object, etc., until a single root object table (with fixed address) is reached (see Fig. 2).

The combination of this cache structure, and the object table, means that the real address of an object is only stored in one place: in its object table entry. This enables us to relocate an object cheaply, as we do not have to search for occurrences of its address. In conventional

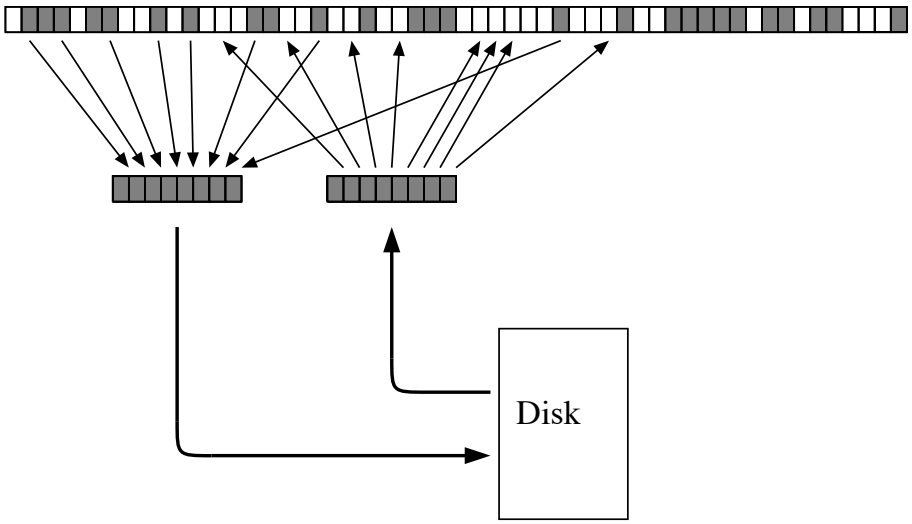

Figure 3: Dynamically grouping during paging. Objects being paged out are gathered from diverse parts of main memory. Objects being paged in are scattered into free areas.

systems without an object table, relocation is prohibitively expensive. Those that implement an object table entirely in software pay a high penalty for each object access. In the MUSHROOM architecture, object accesses are fast, but relocation is also cheap.

The ability to relocate objects cheaply is used by the virtual memory system. Rather than allocating an object to a particular page for its whole life, when the virtual memory system needs to eject objects to make room for others it chooses a group of objects sufficient to fill, or nearly fill, a page and ejects those (see Fig. 3). This choice is made dynamically, at the moment of ejection, in the hope that the group of objects is related. If they are related, when the page is later read back in (due to a page fault involving one of the objects on the page), many other useful objects will also be brought in at the same time. Earlier simulations of this technique, grouping objects by time of last access, suggest that it can substantially reduce the number of page faults in a system [11].

3 Design aims of the garbage collection system

The garbage collection system should reclaim the bulk of the garbage within the MUSHROOM virtual memory hierarchy. Allowing small amounts of garbage to remain on disk, uncollected, is acceptable. Small amounts of garbage in memory may be missed by any particular garbage collection cycle, but should be collected in the next cycle. To maintain the high performance of the system, the garbage collector must not negate the aims of the memory system:

- The garbage collector must have minimal impact on the performance of normal operations which change objects (so-called "mutator" operations). This excludes garbage collectors which substantially degrade the performance of elementary register or memory operations, or which increase the complexity of the hardware so much that the processor cycle time is significantly lengthened. Therefore, schemes which require a complex check or reference-count change for each mutator operation are unlikely to be acceptable.

- It must not adversely affect the performance of the storage hierarchy, e.g., by lowering cache hit rates or diluting main memory with inactive or garbage objects.

- It must not cause lengthy delays to users (i.e., it must be "non-disruptive"; real-time response is not required).

All of these criteria must be met in a high-performance system, i.e., one that is executing efficient codes emitted by an advanced compiler, with associated high rates of garbage creation.

4 Outline of the proposed solution

The MUSHROOM garbage collection system attempts to satisfy the above requirements by being integrated with the virtual memory system. Its parts are matched to the characteristics of the storage hierarchy, and it exchanges information with the virtual memory system.

The first level of garbage collection runs entirely within the data cache, in an attempt to reclaim the majority of garbage in the cheapest way. Confining the first phase to the cache has several benefits: the cache is relatively small (compared to main memory or virtual memory), fixed in size, and of constant, fast access time. This means that each phase of this collector has a small, fixed upper bound on the time it runs, and causes minimum disruption to the user.

The second level is an incremental collector operating in main memory. It too has a small, fixed upper bound on the time required for a garbage collection step, and uses information from the virtual memory system to avoid accessing disk, and to collect garbage before it migrates to disk.

The final level is used to reclaim garbage from disk, and is a reference counting system (with counts held separately from objects). Reference counting is a good choice for an area in which the death rate is low, and there are many live objects. Unlike marking collectors, it does not require a scan of live objects, but merely access to the data that change; these are present in main memory just before being written to disk.

5 The cache-based collector

In the MUSHROOM system, immediately after creation an object is resident only in the data cache. It is allocated an object identifier, but is not allocated a main memory address. Main memory space is allocated only when some part of the object leaves the cache. This speeds both object allocation, by saving unnecessary memory traffic loading data that will immediately be overwritten, and also reclamation, if the object never leaves the cache.

Table 1: Contents of an object's header

Size	8 bits
Local	1 bit
Marked	1 bit
Traced	1 bit
Tracing offset (see text)	8 bits
Free	1 bit

In-cache allocation is possible in the MUSHROOM architecture because the cache is *virtually-addressed*, *write-back* and *software-controlled*. By virtually-addressed, we mean that to access a word in an object the cache is probed using a function of only the object identifier and offset – no virtual-to-real address translation is performed. The cache is write-back because modifications to a cache line are propagated to memory only when the line is flushed. When a cache miss occurs, a *software* trap handler is invoked that writes back the current contents of the required cache line, if dirty, and loads the cache line. For new objects, allocation of a real address is delayed until part of it is flushed from the cache, i.e., until the last possible moment.

The cache-based collector is generational in nature, reclaiming objects that die in the cache, so long as they have not been referenced from objects outside the cache. Previous studies have shown that most objects in Smalltalk or Lisp systems die relatively soon after their birth: "most objects die young." [1, 15]. Our own simulations, driven by lengthy memory traces from a Smalltalk-80 system, indicate that the majority of garbage, as much as 90%, can be reclaimed in this way [14]. Hence, most objects will be reclaimed by a fast, efficient collector. They will also benefit from having had no main memory image, saving on unnecessary fetches at allocation time, and unnecessary stores after they have been reclaimed; both problems can occur on conventional systems [16].

The collector can be triggered at any convenient time, performing a mark and sweep of the cache. The marking phase uses the contents of the registers as roots, as well as those objects that have been, and probably still are, referenced from main memory or disk. Clearly, determining the former of these is straightforward; the difficulty is knowing which objects in the cache may be referenced from outside. In order to achieve this, the objects are divided into those which are *local* to the cache, and those which are *non-local*. A bit in each object's header (the first word of the object, see Table 1) records whether or not it is local. When created, an object is marked as local. If its identifier is ever written into main memory (due to part of an object containing a reference to it being flushed from cache), or its header is flushed from the cache, then it is marked as non-local, and can never become local again. Hence, the test for locality is:

1. Probe the cache to see if the object's header is in cache; if not, it is not local.

2. If the header is in cache, examine its "local" bit.

Because the MUSHROOM cache is software-controlled, it is possible to probe the cache for the

presence of an object without causing a cache-miss trap.

The in-cache collector uses the non-local objects in the cache as the roots for the marking phase; only local objects that are garbage are reclaimed by this collector.

Fig. 4 illustrates an example. At the top we see the state of the cache when the collector is triggered. For simplicity, we show only the objects in cache, paying no attention to the detailed structure of the cache, or to parts of the cache that are empty. Local objects are marked with an "L", non-local objects have no designation. Parts of some objects (3, 6 and 9) are not in cache; these are indicated by shaded areas. Objects 4, 6 and 8 are not local, because:

- Object 8 has had a reference to it stored in memory.
- Object 4 has also had a reference to it stored in memory, but this has been subsequently overwritten. Despite the fact that object 4 is garbage, this collector will not reclaim it.
- Object 6 has had its header flushed from cache.

The marking phase starts by marking as live all local objects referenced from registers; the mark bit is also stored in the object header. Then it runs through all cache lines, and any cache line containing a portion of a non-local object is scanned; local objects referenced from such cache lines are marked as live. In the second part of Fig. 4, these objects (3, 5 and 7) have been marked with an "M". Finally, another pass through the cache uses the live, local objects marked in the previous phase as roots for a recursive trace of all live, local objects accessible from those roots. This picks up an additional object, 9, in the third part of the diagram.

The sweep phase runs through all cache lines, and reclaims those lines belonging to local objects that were not marked as live, i.e., objects 1 and 2, and resets the marks of other objects in preparation for the next scavenge (last part of Fig. 4). Thus, the primary garbage collector can reclaim all local garbage without a single access to main memory.

The local garbage objects are placed on a doubly-linked free list (actually, there is one free list for objects that will fit in a single cache line, one for objects that need two cache lines, etc.). When allocating an object, one is taken from the appropriate list if possible. If a cache line occupied by an object on a free list is required, it is immediately surrendered, the object is removed from the list, and its identifier added to a list of unused identifiers.

5.1 Handling the mark stack

One complication is due to the recursive nature of the tracing phase: storage is required for a stack. We could reserve some part of the cache for this, but in the worst case (in which every cache line contains an object, and all the objects form a single chain) this would be very expensive. Instead we use two properties of the system to make this phase more space-efficient:

1. This collector is uninterruptible, and therefore can manipulate the cache contents in any way it desires, so long as all live objects are restored to their initial states when it exits.
2. In the MUSHROOM architecture, objects are limited to being quite small (256 words in the prototype implementation) [12].[2] The size of an object is stored in its header.

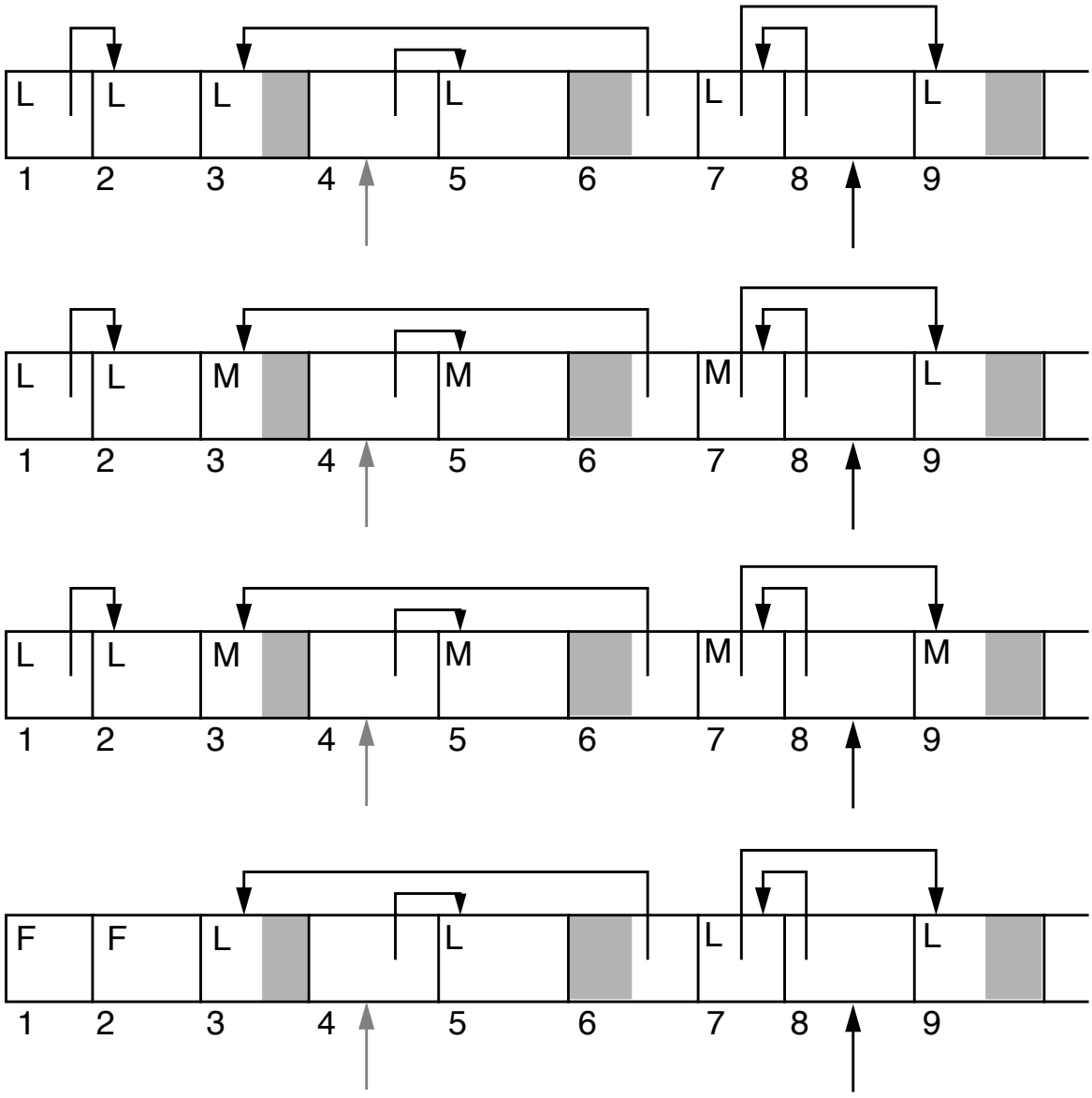

Figure 4: Four steps in the operation of the in-cache garbage collector (see text for explanation).

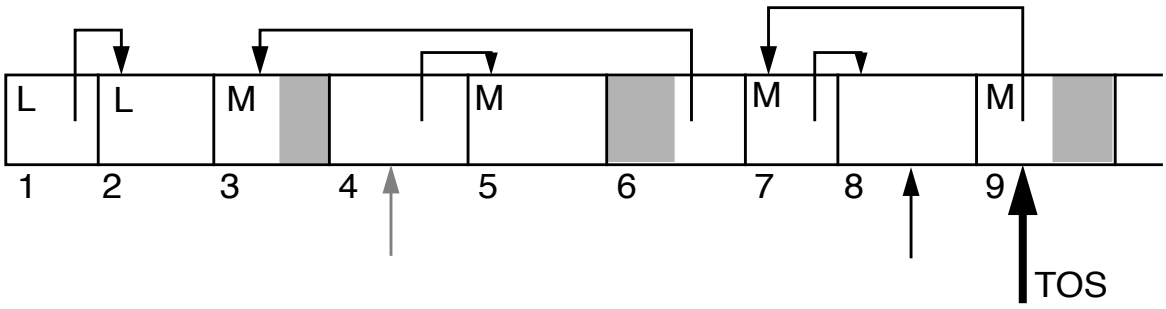

Figure 5: Reversing pointers while tracing

The recursive tracing process, when tracing a reference from object *A* to *B* to *C*, overwrites the reference to *C* that was previously in *B* with a reference to *A* so that it can "find its way back." That is, it reverses the chain of references it has traced, holding the identifier of the current and previous objects in registers. Fig. 5 (which logically goes between the second and third parts of Fig. 4) shows this at the time objects 8, 7 and 9 are on the stack.

Reversing pointers is sufficient to record *which* object was last traced, but we also need to know *how far* through the object we were, i.e., the offset of last word traversed in the last object. A solution to this problem is to reserve eight bits in each object's header to record the offset of traversal in the previous object (recall that every local object will have its header in cache).[3]

The storage overheads of the scavenger are:

- In each object's header, at least two bits are needed to represent the states: non-local; local and marked; local and traced; local garbage. To speed the inner loop of the collector it may be worthwhile encoding these alternatives using three bits (local, marked, traced).
- Eight bits in each object's header to record the traversal offset.
- A few registers during the garbage collection phase. These need not be dedicated to the collector, but can be saved when the collector starts, and restored when it exits.

The garbage collector operates in time proportional to the cache size.

[2]To recap the reason for this decision from [12], very few objects in a Smalltalk system are larger than 1 Kbyte. Rather than accommodating these objects in the architecture, and paying the price (wider buses, wasted memory, wider cache keys, etc.), we use the encapsulation mechanisms of object-oriented languages to build large objects from collections of smaller ones, rather like a file system provides the illusion of large contiguous files from assemblages of smaller blocks. The illusion of large objects can still be presented to the programmer by the use of suitable library classes, and a smart compiler.

[3]Actually, we use less than eight bits. Because each item in the chain is an object reference, and not some primitive type, there are some tag bits in each word which can be overwritten while constructing the stack, and replaced when unwinding the stack.

6 The main memory collector

The second level of the garbage collection system reclaims garbage in main memory. Because main memory is large, this is done by an incremental mark/sweep collector, similar to that described in [17]. As with the cache-based system, this system relies on intimate knowledge of the memory structure to achieve good performance.

To simplify the explanation, let us first consider the collector in abstract terms, devoid of any architecture-specific details. Using the terminology of [17], during the marking phase of the garbage collector each object is in one of three states:

White objects have not been examined by the garbage collector since its last sweep phase.

Grey objects have been examined by the garbage collector, and are therefore considered live, but their contents have not yet been scanned to see if they contain any references to white objects.

Black objects have also been examined by the garbage collector, and are also considered live, like grey objects. Additionally, they have been scanned and any white objects they referenced have been marked as grey.

At the start of the marking phase, the system roots are marked as grey; all other objects are white. The marking phase of the collector repeatedly performs the following actions: locate a grey object, mark it black, and mark any white objects it references as grey. When there are no more grey objects, the remaining white objects are known to be garbage, and the sweep phase can then begin; this will reclaim the white objects.

In order to assure the correct operation of this algorithm, the mutator must not be allowed to store a reference to a white object into a black object. If this is attempted, the white object is coloured grey before the store completes. As an optimisation, if the mutator accesses any white object, we can colour it grey immediately as it cannot be garbage (by virtue of being accessed).

In some ways this collector is also similar to a simple copying collector [4]. In a copying collector, an object's colour is encoded into its virtual address, and to change colour the object must either be copied from one virtual address space to another (when becoming grey), or the boundary between spaces shifted (when an object becomes black). The on-the-fly mark/sweep collector operates in a similar fashion, except that it does not need to copy objects. However, it cannot sweep up garbage by simply discarding a portion of virtual address space, but must search the object table for white objects.

6.1 Using a queue of grey objects

To locate grey objects, the garbage collector operates on a queue of object references, each object in the queue being grey. The queue is initialised with the system roots (e.g., contents of registers, and other root objects). In the marking phase, the collector proceeds in steps, each step taking one object from the queue, and tracing it. In tracing an object, its contents are scanned for references to white objects, and all such objects are added to the queue. Finally, the

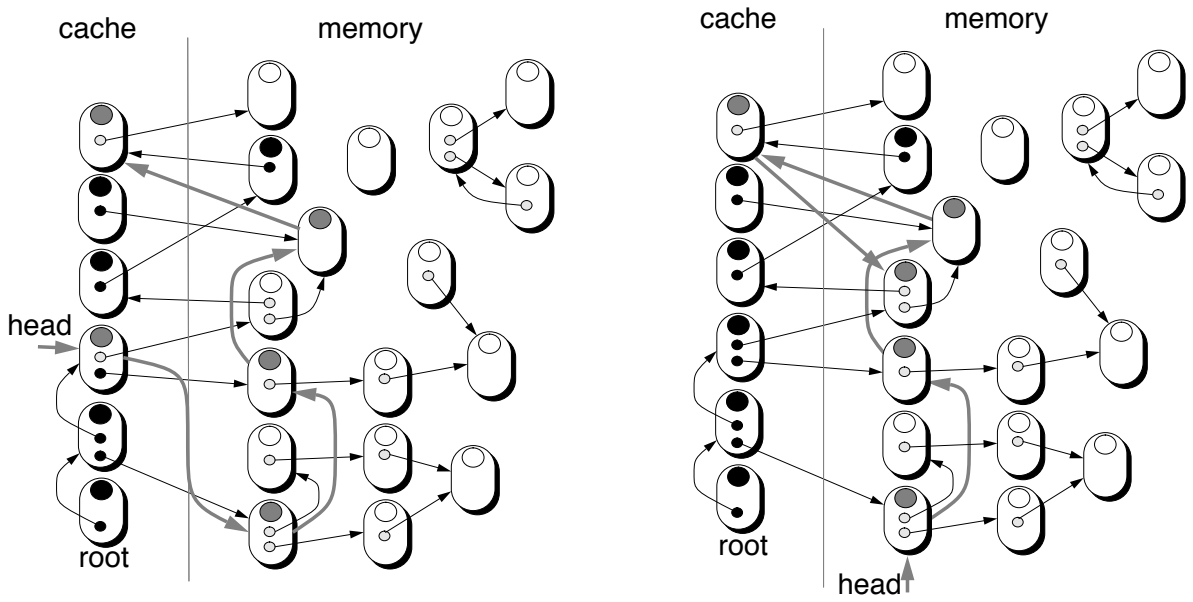

Figure 6: Colouring objects while marking, by traversing the "grey" queue

object is marked black. If, during the marking phase, the mutator attempts to store a reference to a white object into a black object, then the white object must be marked grey, and added to the queue.

When the queue is exhausted, the marking phase is complete, and all objects are either black or white. The white objects can be swept up and reclaimed. This can also be performed incrementally, as no white object can subsequently change state. Finally, all the black objects are marked as white, the queue is re-initialised, and the whole process repeats.

6.2 The MUSHROOM implementation

The MUSHROOM implementation of the collector described above is designed to cooperate with the virtual memory system as much as possible. It uses the movements of objects in the memory hierarchy to assist it in determining which objects are live. Occasionally it will mark an object as live when in fact it has recently become garbage, but all garbage present at the start of a collection cycle, resident in main memory and not referenced from in-cache garbage, will be reclaimed at the end of that cycle.

For the moment, let us ignore secondary (i.e., disk) storage, and consider only cache and main memory. The first assumption made by the collector is that all objects partially or wholly in cache are live. This is a reasonable assumption: an object must have been live when loaded into cache, and the normal cache turnover will tend to eject garbage from cache, to be reclaimed during the next cycle of garbage collection.[4] Thus, an object is marked as either black or grey

[4]Should this be a problem, due to large amounts of main memory garbage being retained by cached garbage, a background process could be run that cycled through the cache using a "clock" algorithm, ejecting objects from cache. If this were

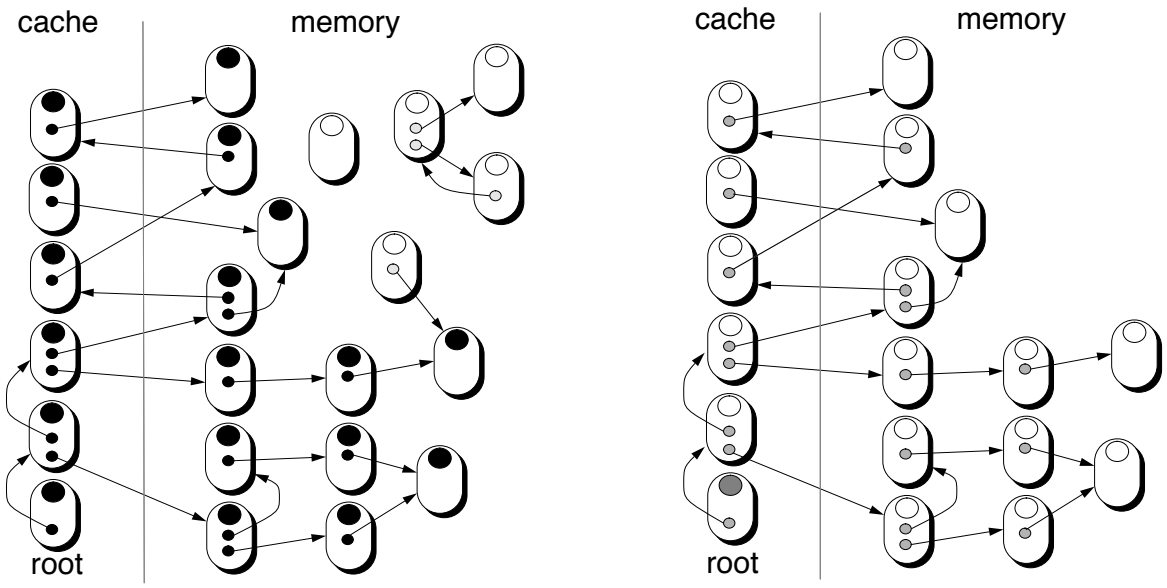

Figure 7: Final stage of marking (left) and after a "flip" (right)

when loaded into cache for the first time (depending on whether the whole object is loaded or not, respectively). If the newly-filled cache line contains any references to white objects, then these are added to the "grey" queue. The marking process consists of removing an item from the front of the queue, loading all parts of the object into the cache, scanning the parts for references to white objects, and then marking the object as black (see Fig. 6). By this process, all live objects will eventually pass through the cache and be marked black (Fig. 7, left). The sweep phase can then work through memory, scanning object-table objects for white objects, and reclaiming them (Fig. 7, right).

6.3 Avoiding cache misses

Determining the colour of an object referenced from a newly-loaded cache line could be expensive. Each cache fill could result in a further series of cache fills, simply to examine the colour of each object referenced in the first fill. Clearly, we cannot store the colour with the object, as this could result in a potentially unbounded series of cache fills. To avoid this, the colour field is stored in the object table entry (see Table 6.3), along with the real address of the object. This entry has to be accessed whenever the object crosses a boundary in the memory hierarchy (e.g., is loaded into or flushed from cache, or is paged in or out), so is conveniently to hand at these times. Also, each object reference has a tag bit dedicated to the main memory garbage collector, and which is used as a hint to the colour of the object. The tag bit dis-

done slowly enough it would not degrade cache performance significantly, but would force all garbage out of cache. Early simulation results, however, suggest that such a process is unlikely to be necessary and cache turnover will be sufficient to achieve this.

Table 2: Contents of an object table entry

Field	State(s) when required	Size
State[a]	L, M, D[b]	2 bits
Colour	L, M, D	2 bits
Address	L, M, D, F	24 bits[c]
Link (for free or local lists)	L, F	32 bits
LRU Link for grouping	M	32 bits
Disk reference count	M, D	8 bits
Size	L, M, D	8 bits

[a]Possible states are: local L; non-local, in main memory M; non-local, on disk D; free F.

[b]Free objects can be identified by their presence on a free list.

[c]This field is dependent of the size of memory and disk. Memory addresses are in units of words, disk addresses are in units of blocks.

tinguishes between *possibly white* and *definitely non-white* objects (shown as grey and black references, resp., in the diagrams). If the tag bit is set to "non-white", then the referenced object has already been examined by the collector, and is considered live; if set to "possibly-white", then it may or may not have been examined – the object's colour field must be examined to ascertain the precise colour.

When a cache line is filled, only the object identifiers that are tagged "possibly-white" need be considered. If the object table entry for a "possibly-white" identifier happens to be in cache, then the true colour of the object can be directly ascertained. If not, the object is added to the "grey" queue (even though it may already be in the queue, or even black), so as to avoid further cache fills. The tags of the newly-loaded identifiers are set to "non-white" before the mutator is allowed to proceed.

Presence of an object's identifier in the queue implies that the object is non-white, and guarantees that eventually it will be marked black. It may or may not be marked grey when entered in the queue; this decision is based on local circumstances (such as whether marking it grey would require a cache-memory transfer or a disk access). Marking it grey immediately may save it from later being added to the queue again (which is harmless, but inefficient). However, even this is not guaranteed, as determining the colour at that stage may also be inconvenient or expensive.[5]

6.4 Resuming marking after a sweep

One complication remains: when the collector has "flipped" state (i.e., black objects are now considered white), some objects, previously black and now white, and their references to other white objects, will still be in the cache (Fig. 7, second part). We must ensure that these references are accounted for in the next cycle of marking. A simple solution would be to add

[5]One might think that this would lead to much inefficiency, due to objects being entered into the queue many times. However, simple examination of a Smalltalk image shows that fewer than 30% of objects are referenced more than once, so at least 70% of objects cannot suffer this fate. Additionally, 90% of objects are referenced no more than four times, suggesting the probability of repeated enqueuing is low. Even fewer objects are multiply-referenced in Lisp systems [18].

them to the "grey" queue immediately, but this could potentially result in an extremely long queue, and take a substantial period of time. Instead, we distribute this activity over the next mark/sweep cycle, by having the cache flush routine detect any attempts to store possibly-white references into main memory, convert them to non-white references, and add the referenced object to the "grey" queue. However, this does not guarantee that all such references are dealt with, so that when the queue has been exhausted, a single scan of the cache for any remaining possibly-white references is performed, and these are added to the queue. Further checks for possibly-white references in the cache flush routine are no longer required, as we can be certain that the cache has no more such references, and the cache fill routine will prevent them from entering cache. Therefore, the next time that the queue is empty, marking has finished and sweeping can begin.

6.5 Benefits

Chambers has identified store checks (e.g., when a reference to a white object is stored into a black object) as a significant obstacle to high performance in compiled systems [3]. In our scheme, the store checks are not performed on every store instruction (as in the SOAR architecture, Lisp Machine ephemeral collector, TI Explorer, or SELF system [19, 6, 9, 3]) but only at cache misses – a much less frequent operation. Similarly, load checks (to ensure that the cache only contains references to non-white objects) are performed only at cache fills, and not at every load (as in the Baker collector or the i432 architecture [4, 20]). This means that normal memory operations can proceed at full speed. Stores into locations which never leave cache are not slowed at all.

Note also that the breadth-first approach used by the mark/sweep system does not have the same disastrous consequences on locality as observed in copying collectors [6], as no objects need to be relocated.

7 Disk-based garbage collection

As described, the garbage collection system would access all live objects on disk in every cycle. This can be avoided by having the main-memory collector cooperate with the virtual memory system, reducing the amount of garbage on disk to such an extent that it is not worth reclaiming by such brute-force techniques.

The first noteworthy gain can be made by observing that, by definition, garbage is never accessed. Hence, in MUSHROOM's dynamically-grouping virtual memory system (where grouping uses an LRU criterion, with ejection of an object header from cache being used as the time of last access), garbage will drift rapidly down the LRU chain. The paging system avoids paging out objects at the end of this LRU chain if they have not been marked as live by the garbage collector. When it can do this, then there is a good chance that such objects will be reclaimed in memory, where such reclamation is cheaper. Hayes has shown that for older garbage, there is strong tendency for garbage to form in clumps [21], explaining the "pig in a python" behaviour observed by Ungar and Jackson [22]. One possibility that we are exploring is to insert timestamps into the LRU chain so that we can detect such clumps, delay paging,

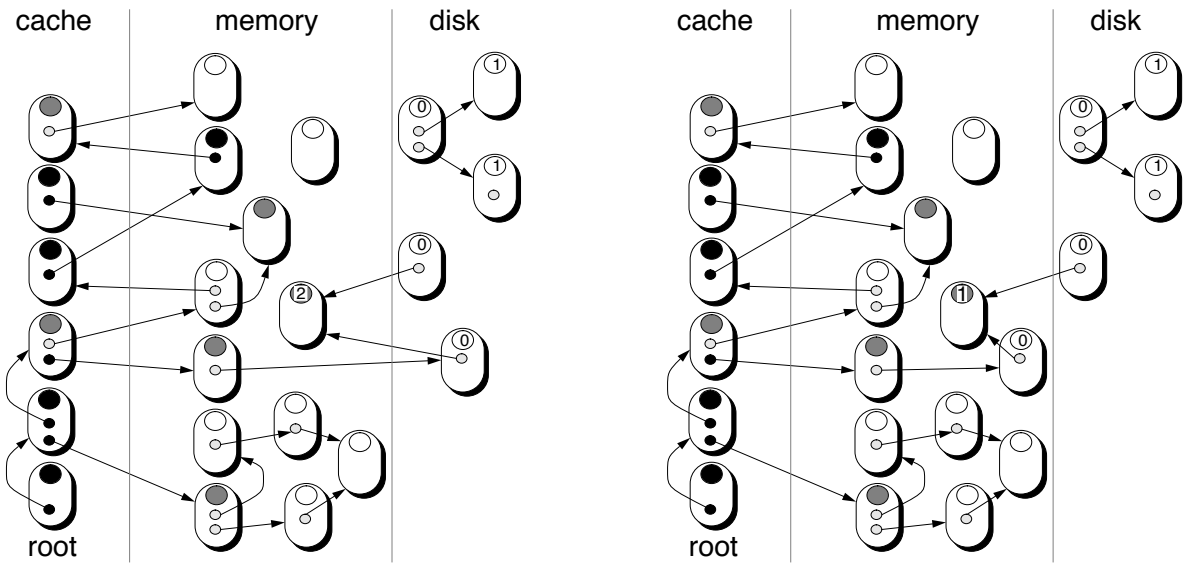

Figure 8: Using reference counts. Each object has a count of how many references to it are stored on disk (those not showing counts are zero). When an object is moved into memory (right) or out, any objects it refers to have their counts altered.

and increase garbage collection activity (particularly if the collector is in the sweep phase, or thought to be near the end of the mark phase).

To avoid unnecessary disk accesses, we can go further. We confine the main memory garbage collector so that to reclaim in-memory garbage it does not access disk at all. Object references are partitioned between main memory and disk in a similar fashion to the partition between cache and main memory. To do this, every object referenced from disk is marked as such, and will not be reclaimed by the main memory collector (such objects are added to the "grey" queue when they are first referenced from disk). Objects in memory which are referenced from disk at the start of the marking phase can be located, scanned, and marked black by walking the in-memory object table objects.

Rather than use a single-bit mark to record when an object is referenced from disk, we use a reference count (of eight bits in the current implementation). The number of references to an object from disk is recorded in the reference count, and the counts are modified whenever a group of objects is read from, or written to, disk, by the paging routines (Fig. 8). The reference count is held in the object table entry.

An object on disk is only eligible for reclamation when its reference count has fallen to zero, and the main memory collector has not found a reference to the object during its mark phase. Any complete cycle of garbage on the disk, unreferenced from garbage in main memory, will not be reclaimed, but we expect this phenomenon to be rare. Similarly, objects referenced from disk 255 times or more will not be reclaimed, as their reference counts will stick at 255.

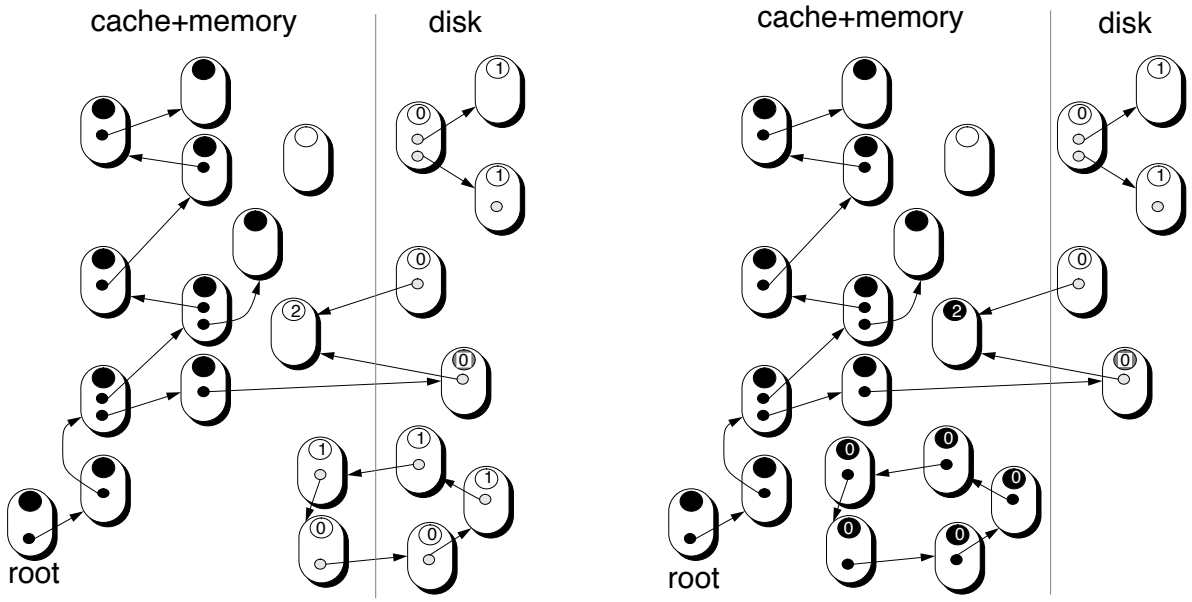

Figure 9: Reclaiming cycles partially on disk by bringing them into memory.

In an attempt to reclaim cycles that are partially on disk, the marking part of the mark/sweep collector is split into two phases. In the first phase, marking starts from objects referenced from registers, and root objects that are in memory (Fig. 9, left). This phase is confined to main memory. In the second phase, objects in main memory with non-zero reference counts, that were not marked in the first phase, are used as the roots. These, and any objects reached from them that were not marked in the first phase, could be parts of garbage cycles. In this phase, if a reference is scanned to an object on disk, then that object may be fetched into memory for further scanning (Fig. 9, right). If the whole cycle is brought into memory, then it can be reclaimed by the next pass of the mark/sweep collector. Heuristics are used to limit the number of objects fetched into memory by this phase.

8 Comparison with other work

As mentioned in the introduction, all generation-based garbage collectors partition the object space so as to concentrate garbage collection activity in the most profitable way. However, few schemes have paid close attention to the interaction between garbage collection and the storage hierarchy. Those which have provided hardware support for cross-generational checks have usually made these checks on every fetch or store operation (depending on the particular scheme in use); ours checks only on cache fills and flushes. Unlike previous schemes, our scavenger does not attempt to remember the locations of cross-generational references using a remembered set, page marking or card marking, or use indirection tables [1, 6, 8, 23, 24, 5, 9].

In conventional systems, garbage collection of older objects can disrupt the cache by caus-

ing newer, more active objects to be ejected [16]. Our incremental mark/sweep collector can make use of MUSHROOM's exposed cache structure to avoid ejecting active objects from cache, by ensuring that it always loads objects into a reserved cache area, unused by other objects.

An early description of the iMAX operating system for the iAPX-432 architecture suggested that garbage could be managed by being paged out rather than reclaimed [20], the complete opposite of our approach. Whether this was suggested because the garbage creation rate was expected to be much lower (the main programming language of the 432 was Ada), we do not know. Our view is that it is best to reclaim garbage sooner than later, and that reclaiming it in memory is better than letting it drift to disk. For systems with Lisp- or Smalltalk-like behaviour, it is essential that the reclamation rate match the allocation rate fairly closely over short time intervals (tens of seconds) if the system is not to run out of storage. We have been unable to locate any data as to the performance of the iMAX scheme.

In [25], Chiueh describes an in-cache garbage collector. This garbage collector runs during idle processor cycles at cache misses, and so requires a processor capable of switching to an alternate thread during these periods. The scheme is based on associating a three-bit reference count with each cached object, and performing reference count operations on objects that have been allocated but never left cache (similar to the local objects in the MUSHROOM scheme). To support this scheme, every store operation which may involve a reference to such an object (either as the value being stored, or the value being overwritten) must fetch the overwritten value, decrement its reference count, and increment the count of the stored value. Although the increments and decrements are deferred to the next cache miss, the extra fetch will undoubtedly slow store operations, even more than cross-generation checks. The MUSHROOM architecture has no provision for task switching during cache-memory transfers; this would complicate the architecture substantially.

Wilson describes using *opportunistic* garbage collection to hide garbage collection pauses from users [26]. Because our in-cache scavenger can be triggered at any time, and the mark/sweep collector is incremental, we can also exploit opportunism.

Our garbage collection scheme should also be well-suited to a shared-memory multiprocessor (the MUSHROOM architecture has provisions for multiple processors, but the prototype is a uniprocessor). The in-cache scavenger places no demands on main memory bandwidth. On a multiprocessor there would be no need to synchronise one processor's in-cache collection with any other processor's, unlike the scheme in [27]. Cache lines can even be invalidated during in-cache scavenging, to accommodate a cache coherency scheme, as other processors cannot possess references to local objects and can therefore only invalidate lines containing non-local objects.

Wilson *et al*. have pointed out that unless the data cache can contain the whole of the youngest generation, poor cache hit rates may result [16]. Our prototype implementation has a 512 Kword (2.5 Mbyte) cache. We realise this is large by current standards, but we wanted to have too much, rather than too little, for experimentation. The amount of cache actually in use can be controlled at system configuration time. It may also be that such cache sizes will be commonplace in a few years' time, especially using two-level structures. Our in-cache garbage collector needs no modification for a two-level cache in which the first level is not exposed but the second is.

8.1 Other systems that attempt to group objects

8.1.1 Courts' system

Courts has described a variety of techniques implemented on a TI Explorer system [9]:

- His temporal garbage collector divides objects into age-based address spaces, using indirections to locate cross-generational references. The Explorer has hardware support for the detection of cross-generation stores. A copying collector is used to move objects between spaces.
- The use of *training* to segregate active objects from inactive ones.
- An *adaptive training* system that attempts to segregate active objects from live but inactive ones (by delaying the action of the garbage collector after a flip).

The adaptive facility is of particular relevance, as it will perform a dynamic grouping function. However, the groupings so formed will be based not only on object activity (i.e., groups of objects accessed at approximately the same time will be placed together), as in the MUSHROOM system, but also on the ages of objects (i.e., a group will consist of objects of approximately the same age). Objects in the oldest age band are regrouped by being activated. Garbage in oldest age band is reclaimed by a scavenger that must copy all live objects – an expensive operation, as the ratio of live to garbage objects is likely to be high.

8.1.2 Static graph reorganisation

Initial attempts at static grouping showed disappointing improvements in paging activity [28], which led us to invent our dynamically grouped system. More recently Wilson *et al*. have suggested that alternative static grouping techniques could yield much better improvements [10].

As Courts has pointed out [9], any technique based on traversing graphs will group objects as to how they *can* be accessed, not how they *are* accessed. Whilst the suggestion that hierarchical groupings (rather than depth- or breadth-first groupings) may improve locality for Smalltalk is probably true, we are unsure as to how one might order the traversals of hash tables suggested in [10], given the lack of a "program" text. However, the techniques suggested in [10] are worthy of further investigation, particularly as they do not require special-purpose hardware.

8.1.3 The interaction between garbage collection and locality

Perhaps the best way of summarising our approach is that we do not attempt to make the garbage collector perform double-duty by both collecting garbage and improving locality. Responsibility for locality lies with the cache structure and the virtual memory system (in the way it manages memory and disk). The garbage collector merely has to collect garbage, while interfering with the activity of the virtual memory system as little as it can.

The only age-related collection that takes place is that done by the in-cache scavenger. The other two collection systems distinguish between garbage that was recently active (and hence in memory), or has been inactive (and hence is on disk). Neither is concerned with the age of an object. The division of labour between these two collectors will be based on the ratio of objects that become garbage having recently become active to objects that become garbage while inactive. To our knowledge, there have been no data published on this ratio for any system. We have attempted to estimate this ratio using traces from a Smalltalk system. A number of traces of object activity were taken, which included the times of every object creation, reclamation, and access (measuring time by ticks of a clock that advances with every memory reference). Plotting lifetimes, we obtained the familiar curve showing that 90% of objects lived less than 10^4 ticks; 50% live only for 100 ticks. If we assume that those with lifetimes less than 10^4 ticks are mostly reclaimed by the in-cache scavenger, then the remainder are eligible for collection by the other two techniques. For these objects, we measured the period of inactivity between the last access to each object, and it becoming garbage. Approximately 40% of objects become garbage within 10^4 ticks of their last access; this rises to 85%–90% within 10^5 ticks. Hence, most objects will become garbage within the period of 10 in-cache scavenge cycles since they were last active, and are therefore unlikely to be on disk. Increasing the scavenge cycle time by factors up to 100 had little effect on this ratio.

This would suggest the bulk of the garbage should be reclaimed by the in-memory collector, as hoped. The figures should be treated with caution, as the traces are not particularly long (between 10^7 and 10^8 ticks). Also, the traces came from a vintage Smalltalk system that dates from the era when reference-counting was the primary reclamation technique. Therefore, some of the code may explicitly break cycles to assist in reclamation, and this would skew the figures in favour of the in-memory collector. Nevertheless, the results are promising.

9 Summary and conclusions

We have described a garbage collection system which works hand-in-hand with a virtual memory system. The collector is split into three levels, to match the memory hierarchy: cache, main memory and disk. The in-cache collector is expected to reclaim most of the garbage (that due to young deaths), using no main memory bandwidth. An estimated time for the operation of the in-cache collector, on a 20 MHz machine, with a 128 Kword cache, is in the order of 20 ms; the worst case is around 100 ms (these figures are based on estimated timings using hand-coded inner loops). Of the remaining garbage, most will be reclaimed by the incremental in-memory collector. This assists the virtual memory system in determining which objects should be paged out, and tries to prevent garbage migrating to disk. Live, but inactive objects, should form the bulk of the traffic to disk. The main memory collector will only access disk in an attempt to reclaim cycles which are partially on disk.

10 Future work

This system is being implemented on the MUSHROOM prototype. Future work consists of measuring the performance of the various parts of the system, and examining the various time-space

trade-offs we have described. Attempting to compare the performance of this scheme with a static-graph scheme looks particularly interesting.

We are also interesting in the use of compression techniques and "swizzling" to decrease disk bandwidth and increase the address space [29].

11 Acknowledgements

The authors would like to thank Trevor Hopkins and the referees for their comments on a draft of this paper. This work was supported by the Science and Engineering Research Council, under grants GR/E/65050 and GR/G/47568, and a SERC Research Fellowship.

References

[1] 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.

[2] Craig Chambers. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, March 1992.

[3] Craig Chambers. Cost of garbage collection in the SELF system. OOPSLA '91 Garbage Collection Workshop Position Paper, 1991.

[4] Henry G. Baker. List processing in real time on a serial computer. *Comm. ACM*, 21(4):280–294, April 1978.

[5] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Comm. ACM*, 26(6):419–429, June 1981.

[6] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, 1984. Association for Computing Machinery.

[7] S. Ballard and S. Shirron. The design and implementation of VAX/Smalltalk-80. In Glenn Krasner, editor, *Smalltalk-80: Bits of history, words of advice*, pages 127–150. Addison-Wesley, 1983.

[8] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, March 1987.

[9] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Comm. ACM*, 31(9):1128–1138, September 1988.

[10] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[11] 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.

[12] Ifor Wyn Williams, Mario I. Wolczko, and Trevor P. Hopkins. Realisation of a dynamically grouped object-oriented virtual memory hierarchy. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 298–308, August 1987. Persistent Programming Research Report, Universities of Glasgow and St. Andrews (PPRR-44-87).

[13] Ifor Williams and Mario Wolczko. An object-based memory architecture. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130. Morgan Kaufmann Publishers, Inc., 1991.

[14] Ifor W. Williams. *Object-Based Memory Architecture*. PhD thesis, Department of Computer Science, University of Manchester, May 1989.

[15] Robert A. Shaw. Empirical analysis of a Lisp system. Technical Report CSL-TR-88-351, Computer Systems Laboratory, Stanford University, February 1988.

[16] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection: a case for large and set-associative caches. Technical Report UIC-EECS-90-5, University of Illinois at Chicago, December 1990.

[17] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, 21(11):966–975, November 1978.

[18] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *Comm. ACM*, 20(2):78–87, February 1977.

[19] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1987.

[20] Fred J. Pollack, George W. Cox, Dan W. Hammerstrom, Kevin C. Kahn, Konrad K. Lai, and Justin R. Rattner. Supporting Ada memory management in the iAPX-432. In *Proceedings of the First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, March 1982.

[21] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 33–46, Phoenix, Arizona, October 1991. Association for Computing Machinery, ACM Press.

[22] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–17, San Diego, California, September 1988. Association for Computing Machinery, ACM Press.

[23] P. G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, 1988.

[24] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 23–35, New Orleans, Louisiana, October 1989. Association for Computing Machinery, ACM Press.

[25] Tzi-cker Chiueh. An architectural technique for cache-level garbage collection. In R. J. M. Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 520–537, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[26] Paul R. Wilson. Opportunistic garbage collection. *ACM SIGPLAN Notices*, 23(12):98–102, December 1988.

[27] Kazuhiro Ogata, Satoshi Kurihara, Mikio Inari, and Norihisa Doi. The design and implementation of HoME. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 44–54, San Francisco, California, June 1992.

[28] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2), May 1984.

[29] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM Comp. Arch. News*, 19(4), June 1991.