Reconciling Buffer Management with Persistence Optimisations

Cutts, Q.I., Lennon, S.
{quintin,lennons}@dcs.gla.ac.uk
Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8RN, Scotland.

Hosking, A.
hosking@cs.purdue.edu
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA.

Abstract

The so-called 'read' and 'write' barriers present an obstacle to the efficient execution of persistent programs that use a volatile object buffer, non-volatile object store memory model. The barriers, implemented as checks added to the code, are required to ensure that objects are moved from store to buffer before being used and, if updated in the buffer, that they are written back on periodic checkpoints.

Static read and write barrier optimisations are identified that require run-time guarantees on certain objects' remaining resident in the buffer. These objects are said to be 'pinned'. Object pinning conflicts with the buffer manager's freedom to evict objects from the buffer when it is full . A contract between the buffer manager and the code optimiser guarantees a minimum level of pinning for each thread. Beyond this guarantee, heuristics added to the buffer manager allow adjustment of the level of pinning for each thread according to the prevailing collection and individual states of threads, to minimise pinning vs object management conflicts. Optimisation of the write barrier additionally requires the buffer manager to uphold certain guarantees across checkpoint operations.

This paper presents the design of buffer management mechanisms to uphold the pinning and update guarantees and identifies the heuristics used to adjust the level of pinning allocated to each thread. Details of the mechanisms' implementation in the PJama system are given. Measurements show that the additional functionality does not represent a significant overhead to the operation of the system, when running the OO7 benchmark.

1 Introduction

Existing implementations of persistence keep buffer management technology largely independent of the execution engine that runs on top of it. Pragmatically, this may have evolved to enable easier experimentation with system componentry and perhaps sometimes because of a "buffer manager knows best" policy with respect to object management. The interface between executing code and the buffer is typically

simple, allowing the code to check whether an object is resident in the buffer or whether it has been marked as updated or locked. Further buffer management operations such as object pinning, object faulting or buffer recycling are either side effects of the checks or are performed asynchronously with program execution. In [HNC+98], a number of persistence optimisations are considered in which the executing code requires guarantees on the states of objects in the object buffer. This paper examines the potential for conflict between buffer management and executing code in the presence of these optimisations and aims to show that collaboration rather than conflict is both feasible and sensible.

An object buffer holds the subset of the objects stored in the persistent store that are currently being, or were recently, used by the executing program. The minimum operational requirement of the buffer is that it always ensures there is space to fault in new objects required for program execution. In order to do this, it *must* evict objects from the buffer periodically. For good performance, a secondary requirement of the buffer manager is that it attempts to retain a good working set of objects in the buffer. This minimises needless object eviction and refaulting.

The optimisations around which the ideas in this paper have developed require that a restricted set of objects in the buffer must *not* be evicted. Further, the run-time mechanisms required to support the optimisations will be more efficient if a larger set of objects could be retained in the buffer.

The requirements of the buffer manager and the optimisations result in potential for both conflict and collaboration. The buffer manager's requirement for object eviction coupled with the optimisations' insistence on a minimum level of object retention could lead to conflict. A contract is required between the two which ensures that the sets of mandatory evictions and retentions do not overlap.

From the collaborative point of view, both buffer management and optimisations aim to retain further groups of objects in the buffer. Buffer managers already achieve this with object usage information gathered dynamically. The optimisations considered here operate over the code statically and can make an indication of those objects that could beneficially be retained, over and above the mandatory minimum requirement. However the static nature of the optimisations means that they cannot be more prescriptive, since they are unable to determine the dynamic demands on the buffer in a multi-threaded environment. Instead, the buffer manager may act collaboratively with the optimiser, taking into account its recommendations and the operational behaviour and buffer state experienced at run-time.

The paper develops these ideas by trading the various costs and benefits of optimisations and buffer management. Section 2 describes a canonical persistence architecture within which the mechanisms described here will run. Section 3 elaborates on the run-time requirements of the optimisations, whilst Sections 4 and 5 present the design of a run-time mechanism to support the persistence optimisations which enables collaboration with the buffer manager, along with an implementation in a persistent version of Java[GJS96], PJama[ADJ+96]. Measurements are given in Section 6 showing that the required contract between buffer manager and optimiser does not have a negative effect on buffer management and in fact that a good level of collaboration is achieved between the two mechanisms.

2 A Canonical Persistence Architecture

The aim of this section is to define a canonical persistence architecture against which the buffer management techniques described in the paper are designed to operate. Whilst the techniques have been specialised and implemented in the PJama system, they are intended to be applicable to any system broadly conforming to the canonical model.

In an orthogonally persistent system, there is conceptually a single storage system for all data, irrespective of its lifetime [AM95]. The data consists of objects connected to one another in an arbitrary graph. This conceptual model is implemented in the canonical architecture using a two level store: the lower level is a non-volatile store supporting the permanence requirements of a persistent system; the higher level is a volatile store, typically giving the fast access to and update of data required for efficient operation. The volatile store is smaller than the non-volatile store and should be viewed as a buffer holding copies from the permanent store of data involved in the ongoing computation. Operations over data only take place in this buffer. In the following, the non-volatile store will be referred to simply as the *store*, and the volatile store as the *buffer*. Although some architectures distinguish between transient and persistent data, using different memory areas and management techniques for each, the discussion here is orthogonal to such issues.

2.1 Read and Write Barriers

When the fields of a persistent object are to be accessed or updated, the object must be present in the buffer. If not already present, a copy of the object must be transferred from the store, or *faulted in*.

The two-level store potentially adds to the complexity of the object addressing mechanism. When objects are in the store, the addresses will be relative to some characteristic of that store. For efficiency, addressing within the buffer uses the latter's addressing mechanism. Address translation mechanisms are required to convert embedded object references between these two addressing mechanisms. Access from one object in the buffer to another that it references cannot take place until the referencing address has been converted, or *swizzled* [Mos90]. Many techniques exist for swizzling e.g. [CBC+90, Wil90, Hos91, KK95], trading flexibility, efficiency and simplicity. From the point of view of the canonical architecture described here, it is enough to note that the store and the buffer use different addressing mechanisms and that addresses within objects resident in the buffer must be swizzled before they can be used.

The two addressing mechanisms introduce the requirement for a check to be made, at some point before a reference is used, to ensure that the object addressed is *resident* in the buffer, and that the reference is in the correct format. The check, and any resulting operations, form a barrier to the ongoing computation known as the *read barrier*. In the canonical architecture, the check, known as a *residency check*, must be inserted into the code stream at any point before the corresponding object is accessed. Operating-system-supported object faulting and swizzling, which remove the

requirement for checks, are not considered here because of efficiency findings presented in [HM93]; these findings are incorporated into the optimisation mechanisms referred to in this paper.

The permanence of data in a persistent system requires that any changes made to objects whilst resident in the buffer are copied back to the original version of the object in the store. For efficiency, these copy operations are batched up and performed at periodic *stabilise* points. To avoid writing all objects in the buffer back at this time, updated objects are identified using a mark placed on the objects at the time of their first update. The need for this marking introduces a second barrier to the ongoing computation known as the *write barrier*: a check must be made to ensure that the relevant object is marked before proceeding with the update.

2.2 Buffer Management

As already stated, the buffer is smaller than the store. A buffer manager is therefore required to ensure that a set of objects appropriate to the current computation is always resident in the buffer – analogous to the operation of a pager in a virtual memory system which aims to keep a good working set of pages in main memory. The buffer management operations of interest here are as follows.

- Co-ordination of the faulting of objects into the buffer as required by the current computation.
- Recycling of the buffer space when it is full. Using appropriate selection algorithms, the buffer manager chooses objects to evict from the buffer that are no longer required by the ongoing computation. Effective selection algorithms are key to good buffer management and hence good system performance.
- Stabilisation of the changes to updated objects back to the store. Once the objects are written back to the store, the marks identifying the objects as updated are removed. An issue here is whether updated objects can also be written out to the store during buffer recycling, independently of a stabilise operation. In such cases the store must be able to record both the original and updated versions.

2.3 Execution Model

The model for execution over the persistent store consists of a number of threads operating over a single shared buffer of objects. Of interest here is the fact that the demand on the buffer cannot be determined statically in general because it depends on the particular group of threads executing at any time. The execution model assumes that each thread models a stack holding for example function or method arguments, locally-declared variables and the results of intermediate calculations.

3 Run-time Guarantees for Persistence Optimisation

As described in the previous section, an unoptimised persistent system performs a residency or update check before every read from or write to a persistent object. Optimising the persistence overhead entails reducing the number of these checks

executed. The checks can be avoided if the system can guarantee that the relevant object is already resident or marked, respectively. Many existing persistence optimisations depend on such guarantees built into the execution model of the system. For example, in an object-oriented paradigm, if on a method call the receiver object is guaranteed to be made resident, residency checks on it are not required in the method body. Alternatively, optimisations may be driven by the shape of an application's data [MH95]: annotations on data structures may instruct the system on the object management approach to be used with data of these types and so permit the compiler to avoid planting some checks.

This paper depends on a more general optimisation approach in which the pattern of application of persistence operations to objects, and hence the objects' state at run-time, e.g. with respect to residency or update, is determined from the code statically. Assuming that the operations in question are idempotent, after an initial application is detected in the code, subsequent invocations of the same operation shown to act on the same object can be eliminated. Full details of the code optimisation technique are beyond the scope of this paper, but are available in [HNC+98].

Crucially, the operations are typically only idempotent when certain guarantees can be assumed on the state of the operations' operands in the object buffer at run-time. The code is statically rewritten by the optimiser, eliminating some operations and hence rendering the code unsafe unless the guarantees are upheld. The guarantees required for read/write barrier optimisations are as follows.

Guaranteed Object Residency

Optimisations involving the removal of residency checks, update checks and lock acquisitions are all based on a run-time guarantee that the associated object will be resident during the execution of the code from which checks have been eliminated. The object must be *pinned* at this time, requiring that it remain in the object buffer. The pinning guarantee is upheld by the buffer manager, which ensures that any object so identified by the optimisations will not be evicted during the buffer recycling operation.

Guaranteed Object Marking

Update checks require a mechanism to identify whether objects have been updated since being made resident. Identification may be made using a mark of some kind on the object. The removal of update checks requires a guarantee that once an object has been marked as updated, it will remain so marked. This guarantee is again upheld by the buffer manager, which ensures that updated objects involved in update check optimisations retain their update markers across stabilisation operations. Failure to retain these markers would result in incorrect operation on a subsequent update from which the update check had been eliminated.

3.1 Overheads of per-thread run-time guarantees

The run-time guarantees described above may have a large impact on the performance of the run-time system. Careful design is required to ensure that the optimisation gained is large by comparison with any performance penalty. The sets depicted in Figure 1 help to discuss the trade-offs in making the optimisations described in this

paper. Taking residency checking as an example, set U contains all the residency checks performed during the execution of a program P. A particular optimisation will be able to consider a subset C of these checks as candidates for removal, and the result of the optimisation is that the further subset O of checks are statically removed. The residency of objects checked by operations in the set G is guaranteed at run-time. Strictly, it is only the objects checked by operations in O that must be guaranteed to remain resident, but the extent of set G demonstrates that it is very hard to achieve this situation in practice. The choice of an optimisation strategy aims to maximise the size of C and the size of O within C, whilst setting the size of G close to O and minimising the costs associated in upholding the associated guarantees.

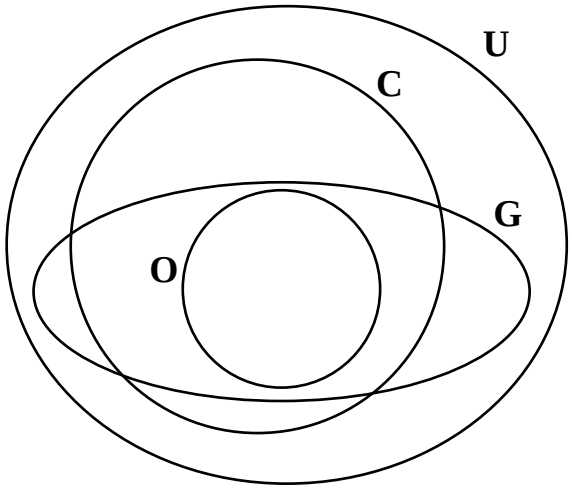

Figure 1: Partitioning operations involved in an optimisation

In the context of this paper, it is the run-time costs that are of interest, in particular the following:

Identification of objects to be pinned or marked

The cost of the mechanism that enables identification of the objects involved in the operations in set G must be smaller than the savings gained from the optimisation. Whilst some assistance may be provided statically by the optimiser (e.g. for pinning, in the form of tables indicating objects to be pinned for the ongoing computation), a run-time cost will always be incurred since the timing of operations such as buffer recycling and stabilisation is asynchronous with program execution.

Maintaining a good working set of objects in the buffer

The operation of the mechanism to uphold run-time guarantees must not conflict significantly with other existing run-time mechanisms. If optimisations require a large number of objects to be pinned, the flexibility for the buffer manager to evict objects on a cache recycle may be seriously affected. The buffer manager aims to keep a good working set of objects in the cache. If this set is a poor match with the

set of pinned objects identified by the optimisation then buffer management will be adversely affected. On the other hand, if the two sets largely overlap, the effect of the optimisation on buffer management will be slight, and may even improve it by more accurately defining a good working set.

Multiple independent threads

Whilst the code-based static optimisation considers single threads in isolation, the demands of an arbitrary collection of threads upon the buffer manager may significantly affect performance. Different threads may require different numbers of objects to be pinned, may work over matching or clashing sets of objects, and may be of differing priorities. Since these conditions are unknown at the time optimisations are made to the code, a contract is required between the optimiser and the run-time system to guarantee a minimum level of pinning that can assumed by the optimiser, whilst allowing the buffer manager free reign outside this level. The buffer manager can then use dynamic information unavailable to the optimiser in order to determine an appropriate pinning level.

4 Stack-directed Pinning and Marking

This section introduces a particular mechanism for pinning and retaining marks on objects in a restricted but effective subset of all objects affected by the read and write barriers. Crucially, the run-time guarantees required over this subset are cheap to uphold.

The optimisation is based on objects directly reachable from a thread's stack. A stack holds local variables, method or function arguments and intermediate calculations, all of which form a focus for the computation of the thread. In particular, the values on or directly accessible from the top section of the stack are likely to represent a significant fraction of the current working set of the thread.

4.1 Pinning

The guarantee made for read barrier optimisations is that any object both resident and reachable from a "hot" area at the top of the stack will remain resident across buffer recycle operations. Limiting the number of pinned objects to those in an area at the top of the stack reduces the effect of pinning on other object management activities. This area is known as the *pinning area*. The set of operations on objects referenced from pointers in the pinning area corresponds to set C in Figure 1. Ideally this area contains all stack references that can be reached from active code, in which case the optimiser can consider all stack references to be in the candidate set C. If the size of the area causes excessive pinning, it may have to be reduced, limiting the size of C.

Moving the pinning area

The pinning area must be moved up the stack as the latter expands to allow the "cool" objects reachable from lower areas of the stack to be evicted by the buffer manager if necessary. When the stack shrinks, the pinning area must be moved back down the stack. The new area may now contain objects that were previously pinned

but have since been evicted. A mechanism is required to ensure that all such objects are *repinned*, reinstating the guarantees required by the optimisations.

Deciding on the size of the pinning area

The size of the pinning area is crucial to good performance of the system as a whole. If it is too small, the set C of candidate objects available for the read barrier optimisation itself may become too small to be effective. Additionally, a small pin area increases the cost of the run-time guarantee mechanism as the repin operation is called frequently as the pin area moves down the stack in small steps. However, a small area minimises the effect of the pinning guarantee on the buffer manager's ability to recycle space.

These considerations can be summarised in the following two guides to choosing the size of the pinning area:

- In order for the optimiser to adjust code statically prior to execution, a guaranteed minimum size for the pinning area per thread must be part of the contract between optimiser and run-time system. Irrespective of its management of objects outside the pinning area, the run-time will always ensure that objects inside that area are pinned, or behave as if they are pinned.
- The run-time is free to adjust the pinning area dynamically to any size beyond the minimum guaranteed size. The optimal size for each thread cannot be determined statically and depends, for example, on the blend of threads at any one time, the speed with which their stacks grow and shrink, and the size of their object working set reachable from the stack. Dynamic profiling can be used by the buffer manager on each recycle operation to educate its choice of an appropriate pin area size for each thread.

4.2 Update Marking

Similarly to the pinning guarantee, any object reachable from the stack and marked as updated will be guaranteed to retain its mark across invocations of the stabilise operation. This is in addition to the pinning guarantee required for updated objects – they should not be evicted as a result of a stabilise operation. Care is required here to ensure that large numbers of written objects do not retain their marks on stabilisation even though they may not be updated again. If they were to retain marks in this way, a severe cost would be incurred in terms of unnecessary writes back to the store on subsequent stabilisations.

5 Implementation in PJama

The mechanisms for pinning and retaining update markers have been implemented in a persistent extension to SUN's Java Development Kit (JDK) implementation of Java, known as PJama version 0.4.5, corresponding to Java version 1.1.5. In parallel, an optimisation tool, BLOAT [NHC+98], has been constructed which operates over programs expressed at the Java bytecode level. Optimisation at this level is sensible for two reasons: sufficient program semantics can be retrieved from the bytecodes for effective optimisation; and the optimiser is free from any particular

compiler or Java Virtual Machine (JVM) [LY96] implementation. A general framework for optimisation in this context is proposed in [CH97].

5.1 PJama Persistence Mechanisms

In the following, the descriptions of PJama mechanisms are simplified down to the requirements of this paper. Full details of the implementation are available in [HNC+98].

Swizzling and Residency Checks

PJama uses an object buffer, known as the Object Cache [DA97], in line with the architecture described in Section 2. To maintain consistency with the original JDK, PJama uses indirection handles for references between objects. Every object in the buffer has an associated handle containing a pointer to the object. All references to an object point to the handle for that object. When faulting objects into the buffer from the store, a mix of eager and lazy swizzling is used. References in objects are eagerly swizzled to point to handles. The latter can be in one of two forms – if the object associated with the handle is resident in the buffer, the handle is in the standard indirection format. If the object is *not* resident, a new handle, known as a fault block, is created and the location in it for the pointer to the object is left blank. For this reason, all inter-object references require a residency check testing whether the reference is to a fault block or an indirection handle.

Cache Management and Cache Recycling

The PJama object buffer [DA97] is divided into a number of regions. Regions may be designated to hold for example handles, objects, method code or bootstrapping information. In the context of this paper, regions holding only objects are of primary concern; note that arrays are held in regions not currently considered by these optimisations and are the subject of further research [Bra98]. At any time, one region is designated as the *allocating* region, and newly faulted objects are placed into this region.

An asynchronous thread periodically checks the loading of the buffer and if the quantity of objects faulted from the store exceeds a threshold level, then it initiates recycling of the buffer space. Recycling involves a number of phases, as follows.

- Hiding Phase. All objects in the buffer, except for updated objects, are marked to indicate that they are potential candidates for eviction. Updated objects cannot be evicted from the buffer until stabilisation due to the no-steal policy of PJama's underlying storage system, RVM [HNC+98].
- Resurrection Phase. The recycling thread sleeps for a fixed period allowing execution to continue. During this time, the marks placed on objects during the hiding phase are removed from objects if they are accessed by running threads. Such objects are said to be *resurrected*.
- Recycling Phase. The unmarked objects in the buffer represent those currently in use by the executing program. The regions of the buffer are divided into groups: one is for regions containing only marked objects, the *Empty* regions; one is for

regions containing updated or marked objects, the *Non-empty* regions. The objects in Empty regions can be immediately evicted. If this does not reduce the buffer loading below a second threshold level, then objects in the remaining Non-empty regions are copied and compacted into a reduced number of regions, retaining hot objects and updated objects. If compaction still does not free up enough space, then hot objects that are not updated will be selectively evicted: the requirement for this has not been observed in practice by the authors.

The remainder of this section describes the adjustments made to the PJama system to support the requirements of the optimisations.

5.2 Bytecode Optimisation

In the existing PJama virtual machine, residency and update checks are performed implicitly as part of the bytecode instructions that access and update objects. Using new instructions added to the virtual machine, the optimiser first makes all the residency and update checks explicit in the code. Analysis of the code with respect to the run-time pinning and marking guarantees allows the optimiser to eliminate these explicit checks from the code. The analysis is a type-based alias analysis with partial redundancy elimination. The implicit checks are removed from the existing access/update instruction definitions inside the virtual machine.

5.3 Pinning

In Java, only the topmost stack frame can be accessed by the currently executing method. The guarantee between the optimiser and the buffer manager is therefore that only references from the top stack frame guarantee the residency of their targets. This is the ideal situation referred to in Section 4.1 where all stack references accessible to active code are contained in the pinning area. Beyond this, the buffer manager is free to set the *pinning level*, determining the number of frames at the top of the stack from which objects should be pinned, to any value depending on the prevailing conditions.

Setting the pinning level

In the current implementation, the pinning level is set and fixed when the machine is initialised. The larger the pinning level, the cheaper is the optimisation support mechanism. This limited mechanism can take no account of varying thread loads, but does give the opportunity to assess the pinning requirements of a variety of executing programs.

Retaining pinned objects at recycle time

A minimal adjustment to the hiding phase of the recycling algorithm ensures that references to resident objects in the stacks' pinning areas are not evicted. Thread stacks are scanned from the top down to the specified pinning level for each thread (see below for how this is set). Any resident object reachable from a reference found during the scan is marked as pinned. The subsequent hiding scan, marking all candidates for eviction, removes the pin-marks from the pinned objects it encounters and does not re-mark them as candidates for eviction. Pinned objects therefore appear

resurrected to the cache manager later on during the recycle phase and so are not chosen for eviction.

In the very rare cases where compaction does not free up sufficient space and resurrected objects must be evicted, some of those evicted objects may require repinning if they are in a stack's pinning area. This is achieved with a further scan of the stack once recycling is complete, or at any time before the associated thread resumes execution.

Repinning an earlier pinning area

The buffer manager is able to set the base of the pinning area to be any arbitrary frame boundary. When the stack retracts across this point, a new pinning area must be set up below. Any previously pinned objects in the new area, evicted in the meanwhile, must be faulted in, re-swizzled and repinned.

When the buffer manager sets a pinning area, it records the return address held in the lowest frame of the area in a thread variable. The return address slot is then overwritten with the address of a fragment of code which, when executed, will repin objects in the new pinning area as necessary, before retrieving the original return address from the thread variable, performing the correct return and then continuing execution.

Repinning a stack area consists of scanning the new area searching for references to objects that have been evicted but were previously resident. This is achieved by examining the object handle. Handles in standard PJama can be in three states, corresponding to whether the associated object is resident, stolen or not resident. For this optimisation, a further state has been added to represent an object made resident but subsequently evicted. Whilst the new state ensures that only previously resident objects are re-faulted, note that there is no guarantee that a thread only repins objects that it previously made resident. The objects may in fact have been made resident by some other thread no longer requiring them in the buffer.

Dynamically setting the pinning level

Once measurement has given a feel for gross system behaviour, in the form of general rules about acceptable numbers of pinned objects, the following mechanism is being considered for dynamic adjustment of the pinning levels of threads. Each thread is allocated a pinning budget, perhaps in relation to its priority. On the hiding phase of the first recycle in which the thread is involved, the buffer manager runs down from the stack top, pinning resident objects referenced from the stack, until the number of objects pinned is equal to thread's budget. The base of the pinning area is set to be the frame boundary at the base of the frame in which the budget ran out, and so the remaining objects in that frame are pinned. If a repin is required subsequently, the same mechanism is used again to set up the new pinning area. The following guidelines govern the setting of pinning budgets:

- The sum of all pinning budgets should be set at a level deemed not to seriously affect buffer management performance, this level to be determined both from previous system executions and from the manner in which the current set of pinned objects affects buffer operations.

- Threads that do not return out of the frame at the base of their pinning boundary before the next recycle need not have such a large budget and so it can be reduced
- Threads that do recross the boundary of their pinning area before the next recycle may have their budget allocation increased in relation to the number of repin operations occurring between recycles.
- When all threads require budget increases exceeding the overall limit permissible, the buffer manager sets allocations according to considerations such as thread priority and thread activity.

5.4 Update Marking

The guarantees for eliminating update checks require that the object is pinned and that the update marker is retained across stabilisation. Due to the no-steal policy imposed by RVM, an updated object cannot be evicted from the buffer during recycling, so the pinning guarantee is already upheld.

The marking guarantee is implemented using a scan of the entire stack at stabilisation time. All updated objects accessible directly from the stacks are identified during stabilisation and their update marks are retained, not removed.

It is accepted that this mechanism will potentially cause re-marked objects to be written back to the store on the subsequent stabilisation even though they have not been updated. The number of phantom writes caused this way is application dependent and is currently being measured. If the problem is large, updated-object maps of the stack can be constructed at optimisation time, keyed by program counter, to ensure that only those objects updateable in the future retain their marks over stabilisation. The construction of such maps is feasible and already used by some systems for accurate garbage collection [DMH92, ADM98].

6 Measurements

All of the mechanisms described in Section 5 have been implemented in the PJama system, except for the dynamic adjustment to the frame pinning level. The measurements presented here and those currently being taken will be used to calibrate the operation of that aspect of the system. Those given here are intended to answer the following questions.

- Do the pinning/marking mechanisms have a positive or negative effect on buffer management?
- How does buffer management performance change as parameters controlling the buffer management and pinning/marking mechanisms are adjusted?
- Can base parameters of buffer management be derived around which the dynamic adjustment of the frame pinning level can be calibrated?

The direct benefits of the optimisations are not considered in this paper. These are reported in [HNC+98], which discusses the optimisations themselves in detail.

6.1 Measurable system characteristics

In order to answer the questions posed, the following characteristics of the PJama system will be measured. Measurement of each will be taken against the unoptimised system, if appropriate, and against the optimised system with varying pinning depths and object buffer sizes.

- Number of object faults in a program's execution. This shows how effectively the buffer manager is able to retain a good working set of objects as pinning depth and buffer size are adjusted.
- Number of recycle operations in a program's execution. This shows whether an excessive number of objects is being redundantly pinned, by comparison with the non-optimised system.
- Number of Non-empty regions encountered during all recycling operations in a program's execution. This will be represented as a percentage of *all* regions considered during recycling operations. This measurement gives an indication of the effect of the stack pinning on buffer management. The number of compacting recycles is also measured.
- Number of repin calls. Across a range of applications, this will give an indication of a suitable mean value at which to set the pinning depth. Dynamic adjustment of the level can work around this mean value.
- Number of repinned objects. This gives a secondary indication of whether an effective working set is being retained and also a general indication of the cost of repinning on the buffer manager.

6.2 Measured Programs

Two program sets were to be used to gain measurements of the adjusted system's operation, to measure both single-threaded and multi-threaded performance. Time pressure has only allowed measurement of the first, which is OO7 [CDN93], typically used to test object-oriented database performance, and in this case used in single-thread mode. The *medium* OO7 database was used, with three connections per atomic part. The benchmark programs measured were: T1 traversal, giving a heavy read-only load to the buffer manager; and T2a, the same traversal as T1 with an update load. The OO7 benchmarks test the effects of the pinning mechanism, but not the update marking mechanism because they only stabilise at the end of each program run.

6.3 Results

The results of the experiments are presented in Figures 2 - 11, in two groups, for the T1 and the T2a traversals respectively. All the graphs plot one of the characteristics identified in Section 6.1 on the Y-axis against increasing buffer sizes on the X-axis, for a number of different pinning area sizes, and where appropriate, the non-optimised PJama system. Note that although the full size of the PJama cache is given, up to about half of this space is used for handles, code and house-keeping, the rest is for object regions.

For the read-only traversal, the effect on the number of recycles and on object faulting is slight. The optimisations increase the number of Non-Empty regions encountered at recycle time, by up to 10%, but not enough to cause more than a few percent extra recycling. No T1 recycles needed region compaction. The number of repinning operations decreases significantly with pinning depth and cache size. As discovered OO7 traversal are highly recursive, resulting in stack depths of more than 100 frames, indicating that more experiments are required here with larger pinning depths.

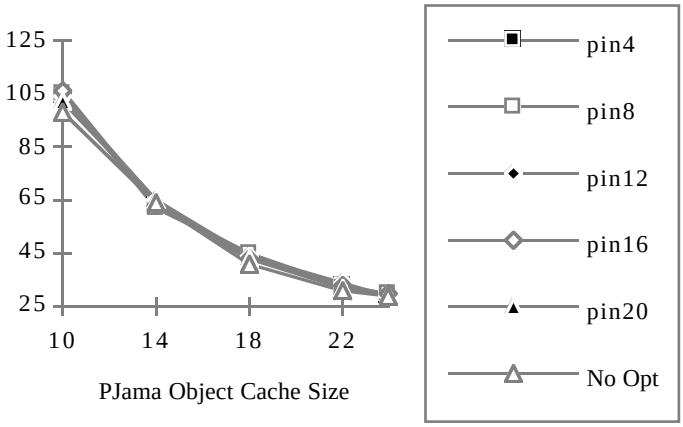

Fig 2: OO7 T1 Recycle Counts

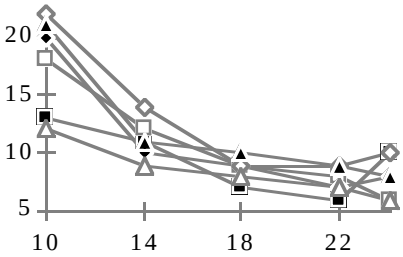

Fig 3: OO7 T1 % Non-Empty Region

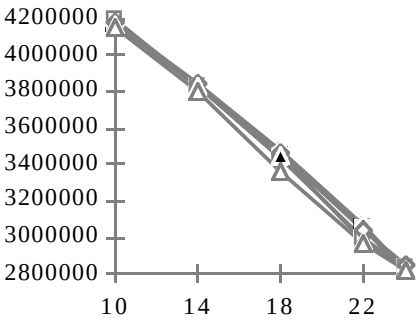

Fig 4: OO7 T1 Object Fault Counts

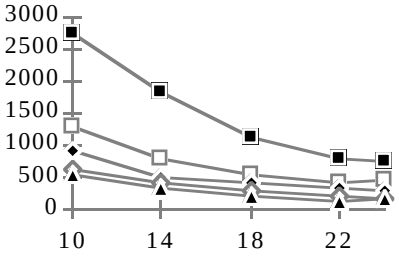

Fig 5: OO7 T1 Repin Operation Calls

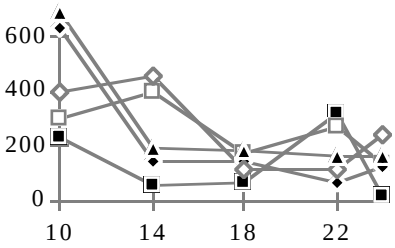

Fig 6: OO7 T1 Repinned Obj. Counts

The effect of pinning depth on recycle counts appears reasonably small from the measurements of updating traversals shown in Figures 7 and 8. However, they are somewhat erratic, and further measurement is required. Note that larger cache sizes were required for the updating traversals because of the No-Steal policy currently adopted by PJama. Faulting caused by the repin operation bears little relation to pinning depth, unsurprisingly since fewer than 1 fault per repin operation is required.

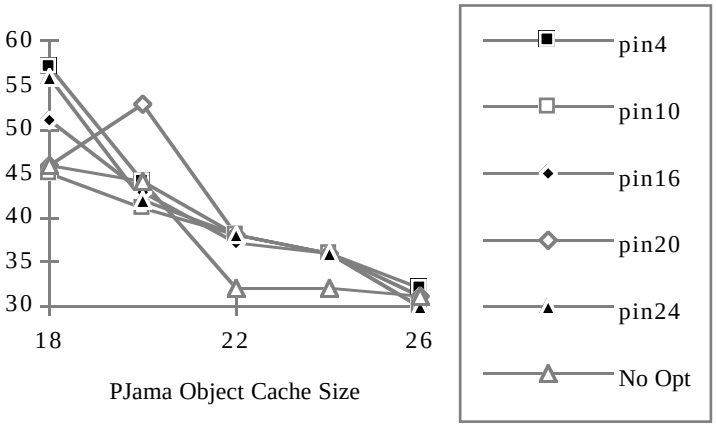

Figure 7: OO7 T2a Recycle Counts

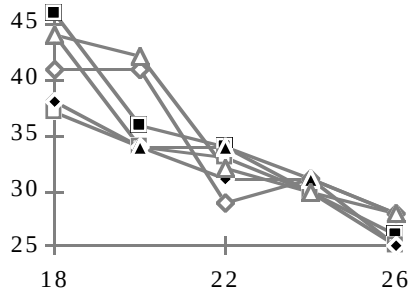

Figure 8: OO7 T2a No. of Compactions

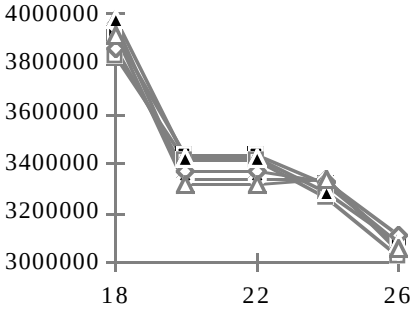

Figure 9: OO7 T2a Object Fault Counts

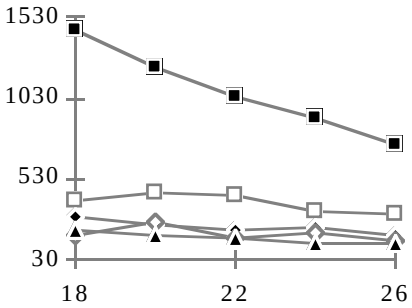

Figure 10: OO7 T2a Repin Operation Calls

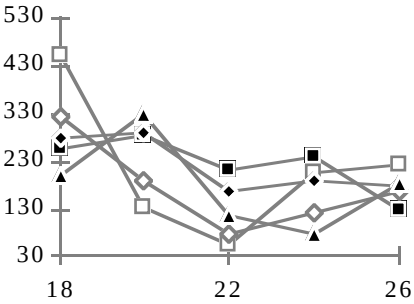

Figure 11: OO7 T2a Repin'd Obj. Counts

7 Conclusions

In answer to the three questions presented at the start of Section 6, the measurements presented here give an indication that the proposed run-time guarantee mechanism does not have a significant effect on the existing buffer manager. The pinning contract between optimiser and buffer manager required for their cohabitation is therefore acceptable. There is little evidence either way to suggest that the collaborative aspect of the two mechanisms with respect to object retention is beneficial overall to the object management costs. However, the performance of the optimisation reported in [HNC+98], in which around 75% of residency checks are eliminated, indicates that the run-time cost, primarily in the repinning operation, will be worth bearing. The measurements give an indication of appropriate pinning depths to be used to calibrate the automatic mechanism to adjust these depths at run-time, although it is noted that the initial set of depths chosen for measurements is not large enough.

8 Acknowledgements

This research is supported in part by gifts from Sun Microsystems, Inc., and by the NSF under Grant No. CCR-9711673. Sincere thanks are due to Frances Flood and Stuart Blair for constructing and using measurement technology, to Nate Nystrom for his work on the BLOAT tool, and also to the PJama team for assisting in our understanding of their system, providing its source code and that of the OO7 benchmark.

9 References

[ADJ+96] Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T. & Spence, S. “An Orthogonally Persistent Java”. ACM SIGMOD Record 25, 4 (Dec) (1996) pp 68-75.

[ADM98] Agesen, O., Detlefs, D. & Moss, J.E.B. “Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines”. In Proc. PLDI to appear (1998).

[AM95] Atkinson, M.P. & Morrison, R. “Orthogonally Persistent Object Systems”. Int. J. Very Large Data Bases 4, 3 (1995) pp 319-401.

[Bra98] Brahnmath, K. “Optimising Orthogonal Persistence for Java”. M.S. Thesis, University of Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, USA (1998).

[CBC+90]* Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (ed), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia (1990) pp 353-366.

[CDN93] Carey, M.J., DeWitt, D.J. & Naughton, L.T. “The OO7 Benchmark”. In Proc. ACM International Conference on Management of Data, Washington, DC (1997) pp 12-21.

[CH97] Cutts, Q. & Hosking, A., L “Analysing, Profiling and Optimising Orthogonal Persistence for Java”. In Proc. Second International Workshop on Persistence and Java (Half Moon Bay, California) (1997).

[DA97] Daynes, L. & Atkinson, M. “Main-Memory Management to Support Orthogonal Persistence for Java”. In Proc. Second International Workshop on Persistence and Java, Half Moon Bay, California (1997).

[DMH92] Diwan, A., Moss, J.E.B. & Hudson, R.L. “Compiler Support for Garbage Colection in a Statically Typed Language”. In Proc. ACM Conference on Programming Language Design and Implementation, San Francisco, California (1992) pp 273-282.

[GJS96] Gosling, J., Joy, B. & Steele, G. **The Java Language Specification**. Addison-Wesley (1996).

[HM93] Hosking, A. & Moss, J.E.B. “Protection Traps and Alternatives for Memory Management of an Object-Oriented Language”. Proceedings of the ACM Symposium on Operating Systems Principles - ACM Operating Systems Review 27, 5 (Dec) (1993) pp 106-119.

[HNC+98] Hosking, A., Nystrom, N., Cutts, Q. & Brahnmath, K. “Optimising the Read and Write Barrier for Orthogonal Persistence”. submitted to the Eighth International Workshop on Persistent Object Systems, Tiburon, California .

[Hos91] Hosking, A.L. “Main Memory Management for Persistence”. OO Systems Laboratory Technical Report, Dept of Computer & Information Science, University of Amherst (1991).

[KK95] Kemper, A. & Kossman, D. “Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization and Quantitative Analysis”. Int. J. Very Large Data Bases 4, 3 (Aug) (1995) pp 519-566.

[MH85] Moss, J. E. B, Hosking, A. L. "Expressing Object Residency Optimisations using Pointer Type Annotations". In **Persistent Object Systems**, Atkinson, M., Maier, D. & Benzaken, V. (ed), Springer-Verlag, Proc. 6th International Workshop on Persistent Object Systems, Tarascon, France (1995) pp 3-15.

[Mos90] Moss, J.E.B. “Working with Persistent Objects: To Swizzle or Not to Swizzle”. COINS, University of Massachusetts Technical Report 90-38 (1990).

[NHC+98] Nystrom, N., Hosking, A.L., Cutts, Q. & Diwan, A. “Partial Redundancy Elimination for Access Path Expressions”. In Proc. Submitted to OOPSLA'98 (1998).

[Wil90] Wilson, P.R. “Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard hardware”. University of Illinois at Chicago Technical Report UIC-EECS-90-6 (1990).