

# Implementation of a Multi-Processor Garbage Collector in ProcessBase Abstract Machine

January 20, 2000

WILLIAM BRODIE-TYRRELL, DAVE MUNRO

{william, dave}@cs.adelaide.edu.au

Dept. of Computer Science, University of Adelaide

## Abstract

ProcessBase is a new persistent programming language targeted to support process modelling applications. It forms a central part of a compliant systems architecture, i.e. an architecture that is compliant to the needs of the running application.

The ProcessBase Abstract Machine (PBAM) is the execution engine for the language. It uses a heap-based storage architecture together with two contiguous stacks per thread to facilitate execution. Prototype implementations have already been built for Solaris, Arena and MIPS platforms, here we describe the early design strategy to incorporate a novel Multi-Processor Garbage Collector that has provable space and time bounds into the PBAM.

We describe in this paper the modification of a multi processor garbage collector so that it can operate on the object cache of the PBAM. The collector used exhibits the desirable properties of incrementality, multiple concurrent collectors, bounded execution time and therefore real-time operation, bounded space usage, simple synchronisation requirements and the support of multiple mutators without read barriers.

## 1 Introduction

ProcessBase is a language designed for modelling processes and their interactions. It is strongly typed, supports first class procedures, linguistic reflection and concurrency. The ProcessBase Abstract Machine (PBAM) [MBG+99] uses a two-layer heap based architecture with separate data and pointer stacks for each thread. The lower layer of the heap architecture is a persistent store, and the upper layer is a fast object cache holding short-lived objects that can be garbage collected independently of the persistent store. Each thread has separate pointer and data stacks which are stored together in a heap object. The heap architecture is based upon that of Napier's [MBC+89] Persistent Abstract Machine (PAM) [CBC+89].

Solaris, Arena [MB97] and MIPS prototypes of the PBAM exist and have a single-threaded mark and sweep garbage collector that suspends all other threads while it is running. The garbage collector described by [BC99] and implemented in this paper has the desirable properties of running concurrently on multiple processors, having primitive synchronisation requirements, incrementality, time-stealing and executing in bounded space and time. It is also described simply and elegantly.

The target architecture for this implementation is the Silicon Graphics PowerChallenge. It is a shared memory machine, and the instance used for testing has 20 MIPS R10000 processors.

## 2 Object Cache

The object cache uses a "Key to RAM Table" (KRT) [BR90] to translate between addresses used by the persistent store and local memory addresses. The KRT is an array of entries containing pointers to objects present in the cache, with entries hashed to based on the persistent address of the object that the entry represents. The KRT is used to resolve all references to objects that are not present in the cache and "pointer swizzle" the address that made the reference to the object.

Address resolution is optimised by assuming that all objects referred to by the threads' pointer stacks and the root object are present, bypassing the check for whether the address is a memory reference or persistent address.

The garbage collector described below was not designed for an object cache that interfaces to a persistent store, and so must be extended to keep consistency between the object space and the KRT used to hash from persistent addresses to the objects. Details of this design work are shown in *Section 4.2*.

### 3 Garbage Collector

The PBAM prototypes already have a single-threaded mark and sweep collector that runs before a stable store checkpoint and when the object cache becomes full. The collector in its present state is unsuitable for multi-threaded operation and the solution taken was to adapt an existing multi-processor algorithm to the requirements of a two-level persistent store rather than to re-engineer the existing mark and sweep algorithm to be thread-safe. The collector implemented here and described by Blleloch and Cheng [BC99] is based upon Baker's real-time collector [Bak77], Baker's tri-colour marking [Bak92] and the work done by Nettles & O'Toole [ON94].

#### 3.1 Machine and Memory Model

The collector assumes that there are P processors, all capable of modifying data and acting as collectors. Two methods are used to synchronise access to shared data: "test and set" and "fetch and add", each atomic.

Memory is organised as a linear addressable space which is divided into an object space and Key to RAM Table (KRT). The object space is further divided into discrete objects of arbitrary size, allocated linearly on demand starting at lower addresses. The KRT contains three-word entries, with enough entries to represent each object in the object space. The boundary between object space and KRT is fixed because of the hashing algorithm used to select entries in the KRT from objects' persistent keys. The format of all objects is:

- Collector word: used for locking of objects and to provide forwarding pointer once an object has been marked *Grey*
- Persistent Key: the address under which an object is stored in the persistent store, zero if not in store
- Flag word: used to store the number of pointer fields in the object (packed at front) and miscellaneous other flags
- Size word: number of words in object
- Pointer fields: pointers in object
- Non-pointer fields: other data in object
- Hash code: identifier used for fast sorting that is accessible from user programs

#### 3.2 Application Model

The application allocates memory from the object cache by calling `ALLOCATE(N)`, which returns a pointer to an object of size N, containing no pointers. The application must then call `INITLOC(V)` to initialise the next location in the object to V before accessing the object or allocating a new object. `INITLOC` must be called N times to initialise all locations within an object. Once an object is initialised, the program may modify the count of pointers in the object.

Applications can read word I of object S by calling `READ(S, I)` and write value V to it by calling `WRITE(S, I, V)`. An object must not be read from or written to before it has been initialised completely.

#### 3.3 Collection Algorithm

The collector has two half-spaces: "fromspace" and "tospace". Objects are allocated in tospace. The collector is started atomically, typically when tospace is full or prior to a persistent checkpoint. The collector has a root node from which all objects that are reachable are copied into the new tospace and thereby compacted. All objects that have been modified and have a valid persistent key are also considered to be roots of reachability to ensure

consistency between the cache and the persistent store; this prevents a modified object from being garbage collected and not updated in the persistent store.

The collector uses a tri-colour [Bak92] representation of objects so that it knows when they are: uncopied, partially copied or completely copied. *White* objects are in the fromspace and have not yet been reached by the collection algorithm from the root. When a pointer in an object is copied, the object that it points to is made *Grey*: it has space allocated for it in the tospace, is placed on a copy stack, and the collector prefix word in the original is used as a pointer to refer to the copy. The prefix word in the copy is set to the number of words remaining to be copied. The copy stack grows downwards from the upper addresses of tospace while tospace is filled from the lower addresses. If the copy stack and the allocation in tospace intersect then the collector experiences a fatal out-of-memory error. At this point the abstract machine needs to purge objects from the cache, back into the persistent store. It can also attempt to expand the heap.

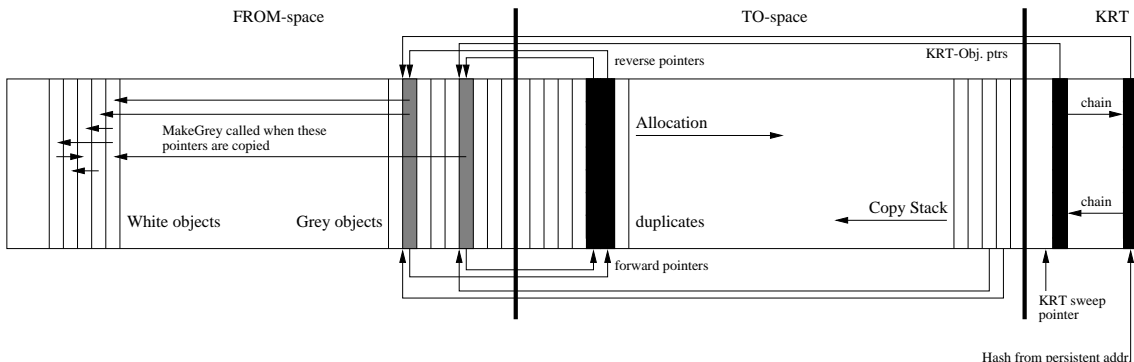


Fig. 1: Memory Map with collector on

Once an object has been completely copied, the prefix word in the copy will reach zero and the object is considered *Black*.

### 3.3.1 Work Stealing

When a thread requests that an object be initialised or written to, a fixed number of steps of collection are performed. The collect algorithm takes a set of objects from the copy stack and copies the words inside the object. If a pointer is found in the object, then the object that that pointer refers to is made *Grey* and will be eventually be collected itself. If the objects have been completely copied and the collector has not yet performed its allocated amount of copying, more objects will be taken from the stack. If the allocated amount of copying is completed without the objects being completely copied, then the objects are placed back on the shared copy stack. The prefix word in each object's copy is updated to reflect the number of locations remaining to be copied.

When the shared stack is empty and all threads have completed their allocated amount of copying, COLLECT will check the KRT sweep pointer; see Section 4.2 for an explanation of how this treats all modified objects with a valid persistent key as a root of reachability. If the KRT sweep pointer has reached the end of the KRT, the collector is atomically turned off; any calls to WRITE or INITLOC after this point and until the collector is restarted incur no overhead from the collector.

### 3.3.2 Write Locking

While the collector is running, the collector prefix word on the object in tospace (the copy) is used to lock access to that object so that it is not simultaneously copied and written to. If the prefix word is positive ( $N$ ), it indicates that the object is unlocked and has  $N$  remaining words to be copied, i.e. it is *Grey*. If the prefix is negative, it indicates that a copier has the object locked for copying at the location corresponding to the value in the prefix. No two collectors may have control of an object concurrently as the operations to add and remove objects from the copy stack are atomic.

If a COPYLOC has an object locked, any WRITE that is issued concurrently from another thread will be blocked from writing that location until the COPYLOC has completed.

### 3.3.3 Race Condition

The algorithm as described in [BC99] does not handle the case of concurrent writes to a single location from multiple threads while the collector is on, as the WRITE does not make any attempt to lock the location it is writing to. Each write requires that the original object and its copy are both updated; if two writes are issued concurrently then it is possible for the original and copy to become inconsistent. This will cause unpredictable behaviour because reads from the object while the collector is on will return the value from the original, and a read once the collector has turned off will return the value in the copy.

It is proposed that a negative value in the collector prefix word of the primary copy should be used to permit a WRITE to lock an object. Currently the values 0 and 1 have special meaning in fromspace: they indicate *White* and “currently allocating tospace copy” respectively. Positive values greater than one indicate a forwarding pointer to the tospace copy. A WRITE could lock an object by setting the top bit of the prefix word, making it negative and indicating to COPYLOC and WRITE in other threads that the object is locked. This requires a minor modification to TESTANDSET (described in [BC99]) so that it understands any non-zero value as being “set” and permits any value to be set.

A drawback of this locking mechanism is that it prevents concurrent writes to a single object from separate threads. The locking mechanism will serialize access to objects, causing a significant slowdown if all threads are operating on a single large object.

## 4 Design

It is believed that the collector described above and in [BC99] has a memory model sufficiently similar to that used in the ProcessBase abstract machine that it can be a near plug-in replacement. There are some issues, however:

- Suitability of application model, compiler modifications
- Interaction between collector and Key to RAM Table
- Independence of collection between object cache and persistent store
- Optimisation

### 4.1 Suitability of Machine & Application Models

Since objects are created by an atomic instruction which fills in all of the pointer and other data fields, the object will be completely initialised before it is visible to the user program and before the instruction has completed, i.e. before a collection can be invoked. The system must then trust the compiler to correctly manufacture the pointers used to initialise the object.

### 4.2 Interaction between Collector and KRT

An object can be discarded if: it has not been reached by the collector (is *White*), AND either it has not been modified OR it has no key in the persistent store. Modified objects that have been in the persistent store must be copied as they will need to be written back to the store at the next checkpoint to maintain consistency. This discard policy requires a sweep through the KRT to check for modified objects that have persistent keys and have not been reached.

There are two options for incorporating the KRT into the collector algorithm: duplication (two KRTs) with copying when an object is made *Grey*, or updating the KRT in a final phase after collection that begun at the root object has completed.

Duplication of the KRT is the simplest option and maintains the “replication invariant” of [BC99]. When an object is copied from fromspace to tospace, a duplicate entry is made in the tospace KRT which points to the new tospace object. This algorithm uses more space than a single KRT, but maintains referential integrity without leaks from the fromspace graph into the tospace graph.

A single KRT is feasible if the memory address in each entry is updated near the end of the collection to point to the tospace copy of each object. See 4.4 *Optimisation* for an explanation of why leaks out of fromspace are valid.

The sweep required to update KRT pointers at the end of a collection can be performed simultaneously with the sweep for modified objects.

Once all objects that are reachable from the root object have been completely copied (made *Black*), a sweep through the KRT occurs to find objects that have been modified but not reached by the collector: these objects must be considered roots by the collector to maintain consistency with the persistent store. This is done by having a shared pointer into the KRT which indicates how far through the sweep has progressed; when COLLECT is called and the shared stack of *Grey* objects is empty, the KRT sweep pointer is checked to see if the sweep has completed. If the sweep has not completed then the process which called COLLECT will lock, increment and unlock the KRT sweep pointer, otherwise it will wait to turn the collector off.

For all locations that a process incremented the KRT sweep pointer over, it must check for a modification bit and a valid persistent address. If both are present and no tospace copy exists then MAKEGREY is called. The KRT pointer to the object is then overwritten with the new tospace address. If no such address exists, then the object is to be discarded: the object next in the hash chain is copied over the current KRT entry and the location it was copied from cleared.

The process of discarding a KRT entry (and therefore the object) must be atomic: another process must not be able to obtain the object's address by using a dereference of the persistent key while the object is deleted. This requires that the discard method lock the object using a flag bit in the KRT entry, all dereferences of objects must also use this bit for locking.

### 4.3 Independence of Collection

The object cache should be garbage collected before the persistent store; the objects in the object cache serve as roots of reachability for the collection of the persistent store. Once this collection of the cache has been performed and the list of roots made from the cache, a section of the persistent store can be collected.

However, for the object cache to be useful and maintain the property of "infant mortality", it must be able to be garbage collected independently from the persistent store. This property is guaranteed by not discarding objects that may have been referenced by the persistent store (have valid persistent keys) and that have been modified; this behaviour is provided by the sweep through the KRT after collection by reachability from the root.

Independence of operation from the persistent store requires also that there is a "faulting mechanism" for determining when an address refers to a local memory location or a persistent address, similar to the page fault mechanism used in virtual memory. The PBAM uses the top bit of addresses to do this: if it is set then the address is persistent, if it is clear, then the address is a memory reference.

### 4.4 Optimisation

Blelloch and Cheng's [BC99] paper describes a "replication invariant" which forces there to be no references between fromspace and tospace: each memory graph is entirely independent of the other. This invariant is a sufficient but not necessary condition to maintain the integrity of the data: leaks from fromspace into tospace are not important because they exist only for the life of fromspace, i.e. until tospace is completed at which point fromspace is discarded.

This means that it is not necessary to allocate two copies of new objects while the collector is on and that only a single KRT is required. When a single object is allocated in tospace, all pointers inside that object refer to other tospace objects; if a write is performed that uses a fromspace pointer, MAKEGREY is called and the address of the tospace copy is used. Where fromspace and tospace copies of an object exist, a reverse pointer from the tospace to the fromspace is required so that writes to a tospace object are reflected also in the fromspace object.

This optimisation is a space/time tradeoff - it saves space when objects are allocated while the collector is on, possibly a large amount if the collector needs to run for a long time due to large physical memory and a small collection factor  $\kappa$ . It also saves space because no duplicate KRT is required (three words per object). It saves this space at a slight time penalty: reads from an object in tospace, reached from a pointer in an object existing only in tospace or from a KRT entry that has been passed by the sweep (now has tospace address) may have to be forwarded to fromspace because the object being read may not have been completely copied.

## 5 Conclusions and Further Work

Design work to date suggests that this garbage collection algorithm with the modifications mooted seems appropriate for use in a multi-threaded object cache. Much work still needs to be done, in particular a set of benchmark programs to exercise the policy tradeoffs needs to be created.

## 6 References

- [MBG+99] "**A Compliant Persistent Architecture**", Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G. N. C., Mayes, K., Munro, D. S. & Warboys, B. C., *To Appear: Software-Practice and Experience (1999)*
- [BC99] "**On Bounding Time and Space for Multiprocessor Garbage Collection**", Blelloch, G. E. & Cheng, P., *Communications of the ACM, SIGPLAN PLDI (1999)*
- [Bak77] "**List Processing in Real-Time on a Serial Computer**", Baker, H. G., *AI Laboratory Working Paper 139 (1977)*, *Communications of the ACM (1978)*
- [Bak92] "**The Treadmill: Real-Time Garbage Collection Without Motion Sickness**", Baker, H. G., *ACM SIGPLAN Notices (1992)*
- [ON94] "**Concurrent Replicating Garbage Collection**", O'Toole, J. W. & Nettles, S. M., *ACM Symposium on Lisp and Functional Programming (1994)*
- [BR90] "**Persistent Object Stores: An Implementation Technique**", Brown, A. L. & Rosenberg, J., *Implementing Persistent Object Bases, Principles and Practice (1990)*
- [CBC+89] "**The Persistent Abstract Machine**", Connor, R., Brown, A. L., Carrick, R., Dearle, A. & Morrison, R., *Persistent Object Systems (1989)*
- [MBC+89] "**The Napier88 Reference Manual**", Morrison, R., Brown, A. L., Connor, R. & Dearle, A., *Universities of Glasgow and St Andrews Report (1989)*
- [MB97] "**Arena - A Run-Time Operating System for Parallel Applications**", Mayes, K. R. & Bridgland, J., *Proceedings of 5th EuroMicro Workshop on Parallel and Distributed Processing (1997)*, pp 253-258