The Memory Pool System

Thirty person-years of memory management development goes Open Source

Richard Brooksby
Ravenbrook Limited
PO Box 205
Cambridge, United Kingdom
rb@ravenbrook.com

Nicholas Barnes
Ravenbrook Limited
PO Box 205
Cambridge, United Kingdom
nb@ravenbrook.com

ABSTRACT

The Memory Pool System (MPS) is a very general, adaptable, flexible, reliable, and efficient memory management system. It permits the flexible combination of memory management techniques, supporting manual and automatic memory management, in-line allocation, finalization, weakness, and multiple concurrent co-operating incremental generational garbage collections. It also includes a library of memory pool classes implementing specialized memory management policies.

The MPS represents about thirty person-years of development effort. It contains many innovative techniques and abstractions which have hitherto been kept secret. We are happy to announce that Ravenbrook Limited is publishing the source code and documentation under an open source licence. This paper gives an overview of the system.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language constructs and features—*dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*memory management*; D.4.2 [**Operating Systems**]: Storage Management

General Terms

Algorithms, Design, Reliability

Keywords

Garbage collection, memory management, software engineering

1. INTRODUCTION

The Memory Pool System (MPS) is a flexible, extensible, adaptable, and robust memory management system, now available under an open source licence from Ravenbrook Limited.

Between 1994 and 2001, Harlequin Limited (now Global Graphics Software Limited) invested about thirty person-years of effort

developing the MPS. It contains many innovative techniques and abstractions which have hitherto been kept secret. In 1997, Richard Brooksby, the manager and chief architect of the project, and Nicholas Barnes, a senior developer, left Harlequin to form their own consultancy company, Ravenbrook Limited, and in 2001, Ravenbrook acquired the MPS technology from Global Graphics.

Our goals in going open source are that as many people as possible benefit from the hard work that the members of the (now defunct) Memory Management Group put in to the MPS design and implementation. We also hope to develop the MPS further, through commercial licensing and consultancy.

This paper gives an overview of the MPS, with particular emphasis on innovative things that it does.

2. BACKGROUND

2.1 History

The original Memory Management Group, set up in 1994, consisted of Richard Brooksby and P. Tucker Withington. Richard had previously worked in the ML Group and implemented the MLWorks[TM] memory manager and garbage collector. Tucker joined from the ailing Symbolics Inc, where he maintained the Lisp Machine's memory systems.

The initial brief of the group was to provide a memory manager for Harlequin's new Dylan system. Harlequin was also interested in a broader set of memory management products, and in absorbing the memory managers of other products, such as ScriptWorks[TM] (the high-end PostScript® language compatible raster image processor), LispWorks[TM], and MLWorks[TM]. Initial prototyping and design work concentrated on a flexible memory management framework which would meet Dylan's requirements but also be adaptable to other projects, and form a stand-alone product.

Richard's concerns about the subtlety of a generic memory management interface, the pain of debugging memory managers, and the complexity of the Dylan implementation, led him to push for a fairly formal requirements specification. This set the tone for the group's operations, and led to extensive use of formal software engineering techniques such as inspections. At its height, the group was operating a Capability Maturity Model level 3 process [12]. As a result, the MPS was very robust and had a very low defect rate. This enabled the group to concentrate on development.

The Memory Management Group collaborated with the Laboratory for the Foundations of Computer Science at Edinburgh University, with the goal of formally verifying some of the algorithms [16].

The MPS was incorporated into the run-time system of Harlequin's DylanWorks[TM] compiler and development environment (now avail-

able as *Functional Developer* from Functional Objects, Inc). Later, it replaced the highly optimized memory manager in Harlequin's ScriptWorks™, improving performance and reliability to this day as part of Global Graphics' Harlequin RIP®.

2.2 Requirements

The MPS had a fairly large and complex set of requirements from the beginning. The Harlequin Dylan project was formed from highly experienced Lisp system developers who knew what they wanted [10]. The requirements of ScriptWorks™ were even more complex [8]. On top of this, we were always striving to anticipate future requirements.

This section describes the overall architectural requirements that guided all aspects of the design [11]:

Adaptability The MPS has to be easy to modify to meet new requirements. This makes the MPS suitable for new applications and ensures it has long and useful life.

Flexibility The MPS must fit into a number of different products and meet differing requirements in diverse environments. It must do this with as little modification as possible, so that it can be deployed at low cost. Flexibility gives the MPS broad application, and reduces the need to maintain special versions of the MPS for different clients. Code re-use also leads to robustness through use testing.

Reliability Memory management defects are very costly. In development they are difficult to find and fix, and once deployed they are virtually impossible to reproduce. The MPS may be shipped to third and fourth parties, further increasing the cost of a defect. Reliability is therefore very important to the viability of the MPS.

Efficiency Efficiency will always be required by clients; after all, memory management is about the efficient utilization of resources to meet requirements. However, the tradeoffs between those requirements will differ from application to application, hence the need for adaptability and flexibility. A generally efficient system will make it easier to meet these requirements.

3. ARCHITECTURE

The MPS consists of three main parts:

1. the Memory Pool Manager (MPM)

2. the pool classes, and

3. the arena classes.

See Figure 1.

Each pool class may be instantiated zero or more times, creating a *pool*. A pool contains memory allocated for the client program. The memory is managed according to the memory management policy implemented by its pool class. For example, a pool class may implement a type of garbage collection, or manage a particular kind of object efficiently. Each pool can be instantiated with different parameters, creating variations on the policy.

The arena classes implement large-scale memory layout. Pools allocate tracts of memory from the arena in which they manage client data. Some arena classes use virtual memory techniques to give control over the addresses of objects, in order to make mapping from objects to other information very efficient (critically, whether an object is not white). Other arena classes work in real memory machines, such as printer controllers.

The MPM co-ordinates the activities of the pools, interfaces with the client, and provides abstractions on which the memory management policies in the pools are implemented.

This architecture gives the MPS flexibility, its primary requirement, by allowing an application of the memory manager to combine specialized behaviour implemented by pool classes in flexible configurations. It also contributes to adaptability because pool classes are less effort to implement than a complete new memory manager for each new application. Reliability is enhanced by the fact that the MPM code can be mature code even in new applications. However, efficiency is reduced by the extra layer of the MPM between the client code and the memory management policy. This problem is alleviated by careful critical path analysis and optimization of the MPM, and by providing abstractions that allow the MPM to cache critical information.

4. IMPLEMENTATION

The MPS is about 62 Kloc of extremely portable ISO standard C [1]. Except for a few well-defined interface modules, it is freestanding (doesn't depend on external libraries[1]). We have been known to port to a new operating system in less than an hour.

The code is written to strict standards. It is heavily asserted, with checks on important and subtle invariants. Every data structure has a run-time type signature, and associated consistency checking routines which are called frequently when the MPS is compiled in "cool" mode[2]. Much of the code has been put through formal code inspection (at 10 lines/minute or less) by between four and six experienced memory management developers [15]. It was developed by a team working at approximately Capability Maturity Model level 3 [CMMI1.02]. As a result, it is extremely robust, and has a very low defect rate.

The MPS is designed to work efficiently with threads, but is not currently multi-threaded. Fast allocation is achieved by a non-locking in-line allocation mechanism (see section 5.2).

5. KEY FEATURES AND ATTRIBUTES

5.1 Flexible combination of memory management techniques

The most important feature of the MPS is the ability to combine memory management policies efficiently. In particular, multiple instances of differing garbage collection techniques can be combined. In the Harlequin Dylan system, for example, a mostly-copying main pool is combined with a mark-sweep pool for handling weak key hash-tables, a manually-managed pool containing guardians implementing user-level weakness and finalization, and a mark-sweep pool for leaf objects. The same codebase is used with a very different configuration of pools in the Harlequin® RIP[3].

An overview of the abstractions that allow flexible combination can be found in section 6.

---

[1]Ironically, a lot of clever design went into the interfaces (MPM and the plinth) to make robust and efficient binary interfaces for a closed-source MPS library.

[2]The MPS can be compiled with various flags to give different *varieties*. The "cool" varieties are intended for debugging and testing. The "hot" varieties are for delivery. Some level of internal consistency checking is present in all varieties.

[3]The details of the Harlequin® RIP configuration and the the RIP-specific pool class implementations are confidential, and not available under an open source licence.
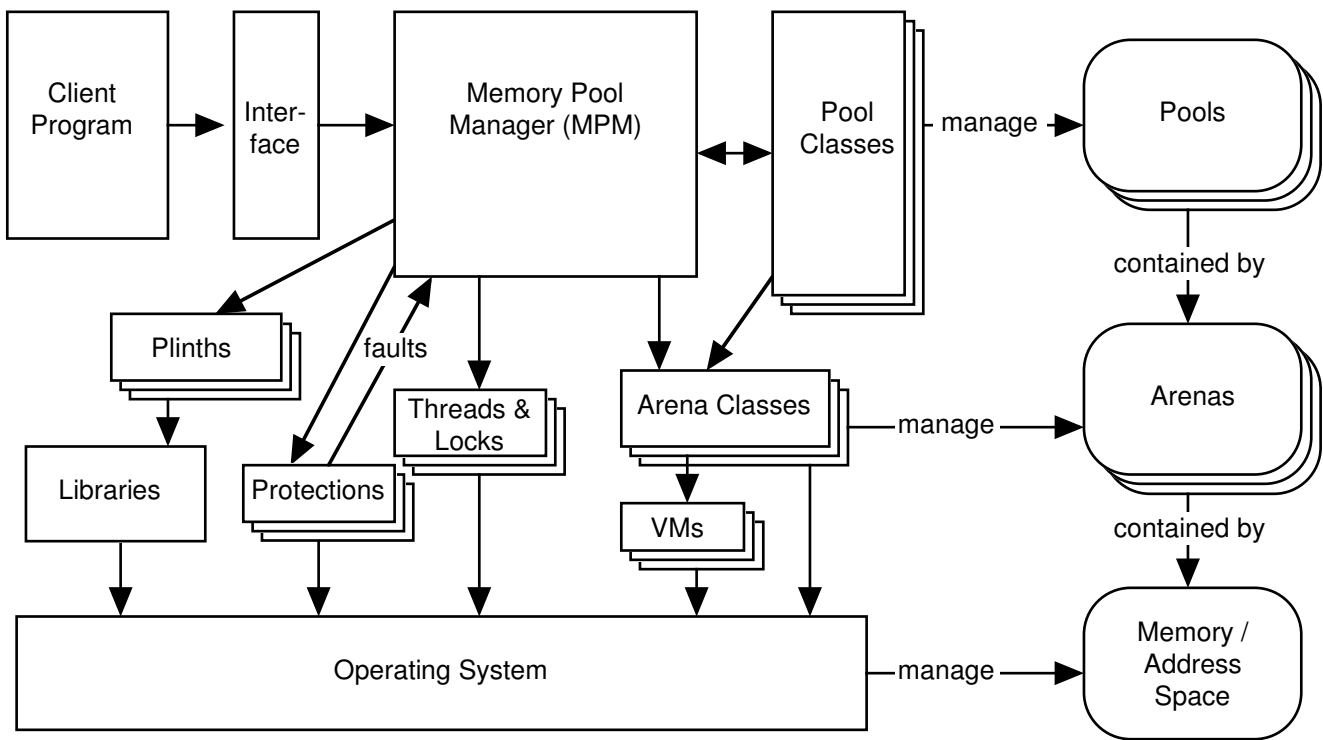
Figure 1: The MPS Architecture.

5.2 Efficient in-line allocation

The MPS achieves high-speed multi-threaded allocation using the abstraction of *allocation points* backed by allocation buffers [9]. The allocation point protocol also allows garbage collection to take place without explicit synchronization with the mutator threads.

An allocation point (AP) consists of three pointers: `init`, `alloc`, and `limit`. Before allocation, `init` is equal to `alloc`. The thread owning the AP *reserves* a block by increasing `alloc` by the size of the object. If the result exceeds `limit`, it calls the MPS to complete the reservation, but otherwise it can initialize the object at `init`. Once the object is initialized, the thread *commits* it, by setting `init` equal to `alloc`, checking to see if `limit` is zero, and calling the MPS if it is. At this point the MPS may return a flag indicating that the object has been invalidated, and must be allocated and initialized again. Both the reserve and commit operations take very few instructions, and can be inlined.

The exact implementation of the AP protocol depends on the pool from which the thread is allocating. Some pools may guarantee that an object is never invalidated, for example, and so the commit check can be omitted. Most pools implement APs by backing them with allocation buffers, fairly large contiguous blocks of memory from which they can allocate without ever calling the MPS.

The AP protocol allows in-line allocation of formatted objects (see section 5.5) containing references that need tracing by a garbage collection. The MPS knows that objects up to `init` have been initialized and can therefore be scanned. It also knows that an object that is half-initialized (somewhere between reserve and commit) when a flip occurs (see section 6.7) can't be scanned, and may therefore contain bad references, that is, references which have not been fixed (see section 6.5). Hence the commit check, and the re-allocation protocol. The chances of a flip occurring between a reserve and commit are very low, and re-allocation rarely happens in practice.

The AP protocol relies on atomic ordered access to words in memory, and some care must be taken to prevent some processors from re-ordering memory accesses.

The design of allocation buffers was inspired by the Symbolics Lisp Machine allocator, which supports a similar protocol in microcode, but requires atomic initialization of objects.

5.3 A library of pool classes

5.3.1 A: Allocate only

A simple pool class which only supports in-line allocation. This is useful when objects need to be allocated rapidly then deleted together (by destroying the pool). Allocation is very fast. This pool is not currently in the open sources.

5.3.2 AMC : Automatic Mostly Copying

The most complex and well-developed pool class, this was originally designed as the main pool for Harlequin's implementation of Dylan, but is a general purpose moving pool. It implements a generational mostly-copying algorithm [5].

5.3.3 AMS : Automatic Mark Sweep

This is the general-purpose non-moving counterpart of AMC. Not generational.

5.3.4 AWL : Automatic Weak Linked

A specialized pool originally designed to support weak key hash tables in Dylan. In a weak key hash table, the value is nulled out when the key dies (see section 6.4). The pool implements mark sweep collection on its contents.

5.3.5 LO : Leaf Object

This pool stores leaf objects (objects not containing references). It was originally designed for use with the Dylan foreign function interface (FFI), guaranteeing that the objects will not be protected by the MPS under a hardware read or write barrier, because interactions with foreign code would be unpredictable.

5.3.6 MFS : Manual Fixed Small

A simple pool which allocates objects of fixed (regular) small (much less than a page) size, though each instance of the pool can hold a different size. Not garbage collected. Used internally to the MPS for the management of some of its own dynamic structures.

5.3.7 MRG : Manual Rank Guardian

This pool is used internally in the MPS to implement user-level finalization of objects in other pools. Some techniques from [14] were used in its design.

5.3.8 MV : Manual Variable

A manually managed pool for variable sized objects. This is the pool class used internally by the MPS as the control pool for many MPS data structures. It is designed to be very robust, and, like many other MPS pools, keeps all of its data structures away from the objects it manages. It is first-fit, with the usual eager coalescing.

5.3.9 MV2 : Manual Variable 2

An unfinished manually managed pool for variable sized objects, using bitmaps and crossing maps. Designed for high-performance freelists with subtle theory of block reuse. MVFF is bascially a light version of the same that just uses the high performance freelist and first fit.

5.3.10 MVFF : Manual Variable First Fit

A general-purpose manually-managed pool for variable sized objects, implementing address-ordered first-fit, but with in-line worst-fit allocation. It is optimized for high performance when there is frequent deallocation in various patterns.

5.4 Support for location dependency

Some data structures and algorithms use the address of an object. This can be a problem if the memory manager moves objects around. The MPS provides an abstraction called *location dependency* (LD) which allows client code to depend on the locations of moving objects.

The design of LDs was inspired by the Symbolics Lisp Machine which has hardware support for something similar, and an evolution of an algorithm developed by P. Tucker Withington for simulating the LispM hardware on stock hardware when writing the Lisp Machine emulator.

5.5 Client object formats

Garbage collectors always have some information about the format of the objects they manage. For example, a non-conservative garbage collector must be able to find and interpret references within the objects it manages. A garbage collector must be able to compute the size of an object given a reference to that object. If the object allocation is entirely handled by the client (as in the MPS; see section 5.2), the size information is encoded somehow in the object format.

This format information is usually entangled deeply in the source code of the collector, for instance in the innermost scan/fix loop. Adapting such a collector to a new object format may be difficult and may introduce some very complex defects.

The MPS includes no object format information; rather it allows each MPS client to specify one or more object formats, by providing a small set of methods which operate on pieces of memory.

When a client creates a pool of a formatted pool class, it specifies an object format for the objects allocated in that pool. The pool class is then able to invoke these methods as necessary to perform format-specific operations. In particular, the *scan* and *fix* methods of the pool class interact closely with the format methods (see section 6.5).

Different pool classes will use formats in different ways. For example, a copying garbage-collected pool may need methods to:

- calculate the size of an object;
- copy an object;
- scan an object;
- replace an object with a "broken heart" (containing a forwarding pointer).

A non-moving mark-and-sweep pool, on the other hand, may only need methods to calculate the size of an object and to scan an object.

The format methods include the following[4]:

skip Skips a pointer over an object.

copy Makes a copy of the object in another location. Objects are usually copied byte by byte, but some uncommon object formats might contain relative pointers that have to be fixed up when the object is moved.

pad Fills a block of memory with a dummy object. This should work just like a real object in all the other methods, but contain no data. This method is used by the MPS to fill in odd corners that need to be scannable.

fwd Replaces an object with a "broken heart" of the same size, containing a forwarding pointer. It is used when the MPS has moved an object to a new location in memory.

isfwd distinguishes between a broken heart and a real object, returning the forwarding pointer of a broken heart.

scan locates all references in a contiguous set of objects and tells the MPS where they are. The objects may include dummy objects and broken hearts.

align is an integer value defining the alignment of objects allocated with this format.

Note that a single client may use more than one object format, even within the same pool class. A client may choose to have many formatted pools, specializing format methods to the kind of object which is allocated within a given pool.

5.6 Multiple arenas

Memory managers, and garbage collectors especially, usually work well with exactly one client. They take over the whole memory infrastructure of a process and provide a single instance of an abstraction to a single client. For instance, they often assume that they have total control over memory protection and memory mappings.

In today's modular software world, such an approach has obvious drawbacks. How do you link together two components, possibly written in different languages, with different memory management infrastructures?

The design of the MPS avoids such assumptions about the environment in which it runs. It abstracts its entire interface with a client into an *arena* object. There is no "global state" of the MPS (apart from the set of arenas). Separate arenas are managed entirely independently. All MPS operations (e.g., allocation, pool creation, garbage collection) are per-arena.

There is more than one way to provide the underlying memory which the MPS manages: it may be a dynamic amount obtained from a virtual memory subsystem, or it may be a fixed amount of client memory (for instance, in an embedded controller or an application which needs a constant memory footprint). These are implemented as distinct *arena classes*. Note that the MPS may manage arenas from more than one arena class simultaneously.

For testing purposes, there is also an arena class which obtains memory from the C standard library `malloc` function.

6. THE TRACER

The Tracer co-ordinates the garbage collection of memory pools.

The Tracer is designed to drive multiple simultaneous garbage collection processes, known as *traces*, and therefore allows several garbage collections to be running simultaneously on the same heap. Each trace is concerned with refining a *reference partition* (section 6.1) using a five-phase garbage collection algorithm that allows for incremental generational non-moving write-barrier type collection (possibly with ambiguous references) combined with incremental generational moving read-barrier type collection, while simultaneously maintaining generational and inter-pool remembered sets. Furthermore, the Tracer co-ordinates garbage collection across pools. A trace can include any set of pools. The Tracer knows nothing of the details of the objects allocated in the pools.

This section describes the abstractions used to design such a general system. The definitions are rather abstract and mathematical, but lead to some very practical bit twiddling. The current MPS implementation doesn't make full use of these abstractions. Nonetheless, they were critical in ensuring the that the MPS algorithms were correct. It is our hope that they will be of great use to future designers of garbage collection algorithms.

6.1 Reference Partitions

The MPS was originally based on a theory of *reference partitions*, which we developed to generalize the familiar idea of "tri-colour marking" [13]. Subsequently the MPS was refined to include more varied barrier techniques[19], but we present the basic theory here to give a flavour of the MPS.

A *reference partition* is a colouring of the nodes in a directed graph of objects. Every object is either "black", "grey", or "white" in any reference partition. A partition (B, G, W) of a directed graph is a reference partition if and only if there are no nodes in B which have a reference to any node in W, that is, nothing black can refer to anything white.

An *initial* partition is one with no black nodes: $(\emptyset, G, W)$. All initial partitions are trivially reference partitions.

A *final* partition is one with no grey nodes: $(B, \emptyset, W)$.

For a predicate P, we say some reference partition (B, G, W) is a reference partition *with respect to P* if and only if everything with P is in W, that is, $P(x) \Rightarrow x \in W$.

If we can determine a final reference partition such that the client process roots are contained in B then W is unreachable by the mutator, cannot affect future computation, and can be reclaimed.

Tracing is a way of finding final reference partitions by refinement. We start out by defining an initial reference partition with respect to a "condemned" property, such as being a member of a generation. We then move reachable objects from G to B, preserv-

ing the reference partition invariant by moving objects from W to G where necessary, until we end up with a final reference partition.

The key observation here is that any number of partitions can exist for a graph, and so there's no theoretical reason that multiple garbage collections can't happen simultaneously.

A second important observation is that because reference partitions can be defined for any property, one could have, for example, a reference partition with respect to a certain size of objects. The MPS uses the reference partition abstraction to implement something equivalent to "remembered sets" [20] by maintaining reference partitions with respect to areas of address space called *zones*. This is described further in section 6.2.

Reference partitions can be usefully combined. If P and Q are reference partitions, then we can define reference partitions $P \cup Q$ as $(B_P \cap B_Q, (G_P \cup G_Q) - (W_P \cup W_Q), W_P \cup W_Q)$ and $P \cap Q$ as $(B_P \cup B_Q, (G_P \cup G_Q) - (B_P \cup B_Q), W_P \cap W_Q)$.

6.2 Induced graphs

Given a directed graph and an equivalence relation on the nodes we can define an *equivalence-class induced graph* whose nodes are the equivalence classes. If there's an edge between two nodes in the graph, then there's an edge between the equivalence classes in the induced graph. We can define reference partitions on the induced graph, and do refinement on those partitions in just the same way as for the original graph. We can garbage collect the induced graph.

In fact, you can think of conventional garbage collectors as doing this all the time. Consider a language in which an object may have sub-objects (inlined within its representation in memory), and an object may refer directly to sub-objects of other objects. You can represent the sub-object relationship with implicit references between sub-objects. Then there is a graph on the sub-objects, including both the usual references and these implicit references as edges. The graph of objects we normally discuss is induced from this graph, the equivalence classes being the sets of sub-objects of separate objects. Theoretically we could garbage collect individual sub-objects by tracing this lower level graph, and reclaim the memory occupied by parts of objects! The MPS is general enough to support this, though we have not implemented a pool class which does it.

Given *two* equivalence relations R and S on a directed graph G, we can define an *equivalence-class relation induced by R and S*, which is a binary relation between the equivalence classes of R and the equivalence classes of S. $(R(x), S(y))$ is in the relation if the graph includes an edge from x to y in the graph.

The MPS divides address space into large areas called *zones*. The set of zones is called Z. The number of zones is equal to the number of bits in the target machine word, so any set of zones (subset of Z) can be represented by a word. Given an equivalence relation R, the equivalence-class relation induced by R and Z is called the *summary* of each equivalence class of R. Roughly speaking, it summarizes the set of zones to which that class refers. This is a BIBOP-like technique adapted from the Symbolics Lisp Machine's hardware assisted garbage collection [18].

The Tracer groups objects into *segments*, and maintains a conservative approximation of the summary of each segment. By doing this, it is maintaining a reference partition with respect to each zone. Segments which don't have a zone in their summary are "black" for that zone, segments which do are "grey" if they aren't in the zone, and "white" if they are. The Tracer uses this information to refine traces during phase 1 of collection; see section 6.7.

6.3 Segments

The Tracer doesn't deal with individual client program objects.

All details of object allocation and format is delegated to the pool. The Tracer deals with areas of memory defined by pool classes called *segments*. The segment descriptor contains these fields important to tracing:

white The set of traces for which the segment is white. A superset of the union of the trace whiteness of all the objects in the segment. More precisely, if a trace is not in the set, then the segment doesn't contain white objects for that trace.

grey The set of traces for which the segment is grey. (See "white" above.)

summary A summary (see section 6.2) of all the references in the segment.

ranks A superset of the ranks of all the references in the segment (see section 6.4).

In addition, the Tracer maintains a set of traces for which the mutator is grey (and assumes it's black for all other traces), and a summary for the mutator. The mutator is a graph node which consists of the processor registers, and any references in memory that can't be protected against transfers to and from the registers. This is *not* usually the same as the root set.

Memory barriers[4] are used to preserve the reference partitions represented by the traces in the face of mutation by the client process. These invariants are maintained by the MPS at all times:

- Any segment whose grey trace set is not a subset of the mutator's grey trace set is protected from reads by the mutator. This prevents the mutator from reading a reference to a white object when the mutator is black. If the mutator reads the segment, the MPS catches the memory exception and scans the segment to turn it black for all traces in the difference between the sets. (An theoretical alternative would be to "un-flip", making the mutator grey for the union of the sets, but this would seriously set back the progress of a trace.)

- Any segment whose grey trace set is not a superset of the mutator's grey trace set is protected from writes by the mutator. This prevents the mutator from writing a reference to a white object into a black object. If the mutator writes to the segment, the MPS catches the memory exception and makes the segment grey for the union of the sets. (An alternative would be to "flip", making the mutator black for the difference between the sets, but this would generally be premature, pushing the colletion's progress along too fast.)

- Any segment which has a summary which is not a superset of the mutator's summary is protected from writes by the mutator[5]. If the mutator writes to the segment, the MPS catches the memory exception and unions the summary with the mutator's summary, removing the protection. Abstractly, this is the same invariant as for the grey trace set (see above), because the summaries represent reference partitions with respect to zones. The barrier prevents the mutator writing a pointer to a white object (the zone) into a black object (which doesn't refer to the zone).

---

[4]The MPS uses memory protection (hardware memory barrier) for this, but could easily be adapted to a software barrier (if we have control over the compiler). The MPS abstraction of memory barriers distinguishes between read and write barriers, even if the specific environment cannot.

[5]The current implementation of the MPS assumes that the mutator has a universal summary. In other words, it assumes that the mutator could refer to any zone. This could be improved.

6.4 Reference Ranks for Ambiguity, Exactness, Weakness, and Finalization

The *rank* of a reference controls the order in which references are traced while refining the reference partition. All references of lower numbered rank are scanned before any references of higher rank. The current MPS implementation supports four ranks:

1. **ambiguous** An *ambiguous reference* is a machine word which may or may not be a reference. It must be treated as a reference by the MPS in that it preserves its referent if it's reachable from the client process roots, but can't be updated in case it isn't a reference, and so its referent can't be moved. Ambiguous references are used to implement conservative garbage collection [7].

2. **exact** An *exact reference* is definitely a reference to an object if it points into a pool. Depending on the pool, the referent may be moved, and the reference may be updated.

3. **final** A *final reference* is just like an exact reference, except that a message (sometimes called a "near death notice") is sent to the client process if the MPS finds no ambiguous or exact references to the referent. This mechanism is used to implement finalization [17, 14].

4. **weak** A *weak reference* is just like an exact reference, except that it doesn't preserve its referent even if it is reachable from the client process roots. So, if no reachable ambiguous, exact, or final references are found, the weak reference is simply nulled out. This mechanism is used to implement weakness.

Note that the pool which owns the reference may implement additional semantics. For example, when a weak reference is nulled out, the AWL pool nulls out an associated strong reference in order to support weak key hash tables.

Ranks are by no means a perfect abstraction. Parts of the Tracer have to know quite a bit about the special semantics of ambiguous references. The Tracer doesn't take any special action for final and weak references other than to scan them in the right order. It's the pools that implement the final and weak semantics. For example, the MRG pool class is one which implements the sending of near death notices to the client process.

The current implementation of the Tracer does not support segments with more than one rank, but is designed to be extended to do so.

The current MPS ordering puts weak after final, and is equivalent to Java's "phantom references". It would be easy to extend the MPS with additional ranks, such as a weak-before-final (like Java's "weak references").

6.5 Scanning and Fixing

In order to allow pools to co-operate during a trace the MPS needs a protocol for discovering references. This protocol is the most time critical part of the MPS, as it may involve every object that it is managing. The MPS protocol is both abstract and highly optimized.

Each pool class may implement a *scan* and *fix* method. These are used to implement a generic scan and generic fix method which dispatch to the pool's method as necessary. The scan method maps the generic fix method over the references in a segment. The fix method preserves the referent of a reference (except when it is applied to weak references), moving it out of the white set for one or more traces.

The most important optimization is part of the generic fix method which is inlined into the scan methods. The generic fix method first looks up the reference in the interesting set (see section 6.7), which takes about three instructions. This eliminates almost all irrelevant references, such as references to generations which aren't being collected, or references to objects not being managed by the MPS.

A second level optimization in the generic fix method checks to see if the reference is to a segment managed by the MPS, and then whether the segment is white for any of the traces for which the reference is being fixed. This eliminates many more references.

Only if a reference passes these tests is the pool's fix method called.

A pool need not implement both scan and fix methods. A pool which doesn't contain references, but does contain garbage collected objects, will have a "fix" method but no "scan" method. Note that such objects are either black or white. A pool which contains references involved in tracing, but not garbage collected objects, will have a "scan" method but no "fix" method. Such a pool would be a pool of roots, its objects either grey or black. A pool with neither method is not involved in tracing, for example, a manually managed pool storing strings.

6.6 Roots

Roots are objects declared to the MPS by the client process as being *a priori* alive. The purpose of tracing is to discover objects which aren't referenced by transitive closure from the roots and recycle the memory they occupy.

The MPS supports various kinds of roots. In particular, a thread can be declared as a root, in which case its stack and registers are scanned during a collection.

Roots have "grey" and "summary" fields just like segments, and may be protected from reads and writes by the mutator using the same rules. However, roots are never white for any trace, since their purpose is to be alive.

6.7 Five phase collection

The Tracer runs each trace through five phases designed to allow pools to co-operate in the same trace even though they may implement very different kinds of garbage collection.

6.7.1 Phase 1: Condemn

The set of objects we want to try to recycle, the *condemned set* is identified, and a newly allocated trace is added to the white trace set for the segments containing them.

At this stage, all segments containing any references (even the white ones) are assumed to be grey for the trace, because we don't know whether they contain references to objects in the white set. The roots are made grey for the trace, because they are *a priori* alive. The mutator is also assumed to be grey for the trace, because it has had access to all the grey data. Thus we start out with a valid initial reference partition (see section 6.1).

We then use any existing reference partitions to reduce the number of grey segments for the trace as much as possible, using this rule: Let P be the set of reference partitions whose white sets are supersets of the new white set. Any node which is in the union of the black sets of P cannot refer to any member of the new white set, and so is also black with respect to it.

In practical terms, we work out the set of zones occupied by the white set. We call this the *interesting set*. We can then make any segment or root whose summary doesn't intersect with the interesting set black for the new trace. This is just a bitwise AND between two machine words. A pool will usually arrange for a generation to occupy a single zone, so this refinement step can eliminate a large number of grey segments. This is how the MPS implements remembered sets.

In a similar way, the MPS could also use the current status of other traces to refine the new trace. Imagine a large slow trace which is performing a copying collection of three generations. A fast small trace could condemn the old space of just one of the generations. Any object which is black for the large trace is also black for the small trace. Such refinement is not implemented in the MPS at present.

Note that these refinement steps could be applied at any time: they are just refinements that preserve reference partitions. The MPS currently only applies them during the condemn step.

6.7.2 Phase 2: Grey Mutator Tracing

This phase most resembles a write-barrier non-moving garbage collector [2]. Any segment "blacker" than the mutator is write protected (see section 6.3).

At this point the mutator is grey for the trace. Note that, at any stage, the mutator may be grey or black for different traces independently. In addition, newly allocated objects are grey, because they are being initialized by the mutator.

An object can be moved provided that it is white for any trace for which the mutator is black, because the mutator can't see references to that object. [What about ambiguous references?]

During phases 2 and 4 the Tracer makes progress by scanning segments which are grey for one or more traces (see section 6.5) in order to make them black. Thus we make progress towards a final reference partition (see section 6.1).

6.7.3 Phase 3: Flip

Flipping for a set of traces means turning the mutator black for those traces. This may entail scanning the client process thread registers and any unprotectable data. The mutator can't be running while this is happening, so the MPS stops all mutator threads.

This is also the point at which the MPS sets the `limit` fields of any formatted allocation points to zero, so that unscannable half-allocated objects are invalidated (see section 5.2).

6.7.4 Phase 4: Black Mutator Tracing

This phase most resembles a read-barrier possibly-moving garbage collector [6]. Any segment "greyer" than the mutator is read protected (see section 6.3).

At this point the mutator is black for the trace. In addition, newly allocated objects are black, and don't need to be scanned.

6.7.5 Phase 5: Reclaim

When the grey set for a trace is empty after flip then it represents a final reference partition. The Tracer looks for segments which are white for the trace and calls the owning pool to reclaim the space occupied by remaining white objects within.

It's up to the pool to decide whether to return the reclaimed space to its own free list, or to the arena.

7. FUTURE DIRECTIONS

The Memory Management Group at Harlequin was whittled away to nothing as Harlequin slid into financial trouble. Parts of the system are incomplete or have unclear status. A large amount of design documentation exists, but it is fairly disorganized and incomplete. We would like to organize all this information to make the MPS a more useful resource.

The MPS was designed around many abstractions that make it very adaptable, but it is not very well packaged and is unlikely to work in new applications without some modification. We would

like to improve the MPS to make it easier to apply without modification.

The MPS is currently commercially licensed to Global Graphics Software Limited for use in the Harlequin RIP®, and to Configura Sverige AB for use in their Configura® business system. We are seeking further licensees and consultancy.

8. AVAILABILITY

The MPS project tree is available on the web at `<http://www.ravenbrook.com/project/mps/>`. It includes all of the non-confidential source code and design documentation. This is a mirror of the tree in Ravenbrook's configuration management repository, so it will continue to reflect the development of the MPS.

9. RELATED WORK

The MPS resembles the Customisable Memory Management (CMM) framework for C++ [3] and shares some of its design goals. The MPS was not designed for C++, but as a memory manager for dynamic language run-time systems. In fact, it was specifically designed not to require C++.

The Memory Management Reference Bibliography contains all the papers that we collected during the project, and can be found on the web at `<http://www.memorymanagement.org/bib/>`.

10. CONCLUSIONS

During our stay at Harlequin we were often frustrated by confidentiality. We were not able to reveal ideas and techniques which we believed were both innovative and useful. The MPS contains many such ideas – the results of hard work by many people (see 11). Now, at last, we can reveal almost all.

The MPS is a highly portable, robust, extensible, and flexible system for memory management, based on very powerful abstractions. It contains many more useful concepts and abstractions not covered in this paper.

We hope that the ideas, techniques, and code of the MPS will be useful. We also hope that companies will license the MPS or engage us to extend and develop it further.

11. ACKNOWLEDGEMENTS

The authors would like to thank all the members of the Memory Management Group for their contributions to the design and implementation of the MPS.

The members of the Memory Management Group were:

Name	Period of membership
Nick Barnes	1995-08/1997-11
Richard Brooksby	1994-02/1997-11
Nick "Sheep" Dalton	1997-04/1998-07
Lars Hansen	1998-06/1998-09
David Jones	1994-10/1999-06
Richard Kistruck	1998-02/1999-06
Tony Mann	1998-02/2000-02
Gavin Matthews	1996-08/1998-11
David Moore	1995-04/1996-04
Pekka P. Pirinen	1997-06/2001-04
Richard Tucker	1996-10/1999-09
P. Tucker Withington	1994-02/1998-11

12. REFERENCES

[1] American National Standards Institute. *American National Standard for Information Systems: Programming Language C*, Dec. 1989.

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] G. Attardi, T. Flagella, and P. Iglio. A customisable memory management framework for C++. *Software Practice and Experience*, 28(11):1143–1183, Nov. 1998.

[4] N. Barnes. MPS Format Protocol. MPS Project Documentation, November 2001. `http://www.ravenbrook.com/project/mps/master/mmdoc/protocol/mps/format/`.

[5] J. F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, Oct. 1989. Sources available in ftp://gatekeeper.dec.com/pub/DEC/CCgc.

[6] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[8] R. Brooksby. MM/EP-Core Requirements. MPS Project Documentation (Confidential), November 1995.

[9] R. Brooksby. Allocation Buffers and Allocation Points. MPS Project Documentation, September 1996. `http://www.ravenbrook.com/project/mps/master/mminfo/design/mps/buffer/`.

[10] R. Brooksby. Dylan Requirements. MPS Project Documentation, October 1996. `http://www.ravenbrook.com/project/mps/master/mminfo/req/dylan/`.

[11] R. Brooksby. The architecture of the MPS. MPS Project Documentation, January 1997. `http://www.ravenbrook.com/project/mps/master/mminfo/design/mps/arch/`.

[12] CMMI Product Development Team. CMMI^{SM} for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SE/SW, V1.02) (Staged Representation). Technical report, Software Engineering Institute, 2000.

[13] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[14] R. K. Dybvig, C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 207–216, Albuquerque, NM, June 1993. ACM Press.

[15] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1995.

[16] H. Goguen, R. Brooksby, and R. M. Burstall. Memory management: An abstract formulation of incremental tracing. In *Types for Proofs and Programs, International Workshop TYPES'99*, pages 148–161. Springer, 2000.

[17] B. Hayes. Finalization of the collector interface. In Y. Bekkers and J. Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, Stanford

University, USA, 16–18 Sept. 1992. Springer-Verlag.

[18] D. A. Moon. Garbage collection in a large LISP system. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, Aug. 1984. ACM Press.

[19] P. P. Pirinen. Barrier techniques for incremental tracing. In R. Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 20–25, Vancouver, Oct. 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[20] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.