Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages

R. Kent Dybvig, David Eby, and Carl Bruggeman

{*dyb, deby, bruggema*}*@cs.indiana.edu*

Indiana University Computer Science Department

Technical Report #400

March 1994

Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically-Typed Languages

R. Kent Dybvig, David Eby, and Carl Bruggeman

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
812/855-3608
{*dyb, deby, bruggema*}*@cs.indiana.edu*

Indiana University Computer Science Department
Technical Report #400

March 1994

Abstract

This paper describes a storage management system that is flexible and efficient. The representation of run-time tags yields fast allocation, type testing, and field extraction, and the memory model reduces virtual memory paging during garbage collection. The storage management system coexists gracefully with other languages' run-time systems, facilitating the use of multiple languages within a single program. No special support from the operating system or virtual memory manager is required beyond the ability to obtain additional memory on demand. The system incorporates a generational garbage collector with a tunable number of dynamically-resizable generations. The collector handles large objects efficiently, supports collection of incrementally-compiled code, supports weak pairs, and allows stacks to contain nonpointer data. The system's hybrid type representation employs typed pointers and typed objects for tagging individual objects, and BIBOP typing for classifying objects according to coarser-grained type characteristics mostly of concern only to the collector. A segmented memory model is used, but segmentation is handled transparently outside of the collector so that nearly all types of objects are allocated inline from a single linear allocation area using a single allocation pointer. The storage management system has been implemented and is in use as part of a high-performance production implementation of Scheme.

1 Introduction

Nearly all implementations of Scheme, Common Lisp, Smalltalk, ML, and similar high-level languages provide automatic reclamation of storage occupied by dynamically allocated objects that have become inaccessible. Implementations of these languages intended for production use must provide the best possible speed and use as little physical memory as possible; yet they must support objects of arbitrary size, dynamically resize the heap as needed, and coexist with other languages' run-time systems within the same process image. If the implementation is to be portable, it must also be able to cope with different operating systems and the ways in which they divide up the address space. In particular, a portable implementation must not expect memory to be parceled out in a strictly linear fashion and must be prepared to separate dynamically-loaded executable code from data. This paper describes a storage management system with a generational copying garbage collector designed to meet these requirements.

Many copying-collector-based storage management systems employ a "flat" memory model in which the heap is assumed to be a contiguous array of memory locations. If a nongenerational collector is used, the heap is subdivided into two equal-sized spaces; otherwise, the heap is divided into three or more spaces. Heap allocation using this model can be as fast as stack allocation [1], since all objects can be allocated from a single allocation pointer kept in a register. Sophisticated versions of this model have been described that allow the spaces to be resized during collection and which, unlike most copying collectors, avoid touching all of memory on each collection cycle [2, 18].

In many ways, the flat memory model is the simplest and most efficient model to use for automatic storage management. The flat memory model, however, does not cope well with holes in the memory image caused by allocation requests from other languages' run-time systems and operating systems that do not parcel out memory in a strictly linear fashion. The flat memory model also complicates separation of read-write data from incrementally compiled or dynamically loaded executable code, as required by most modern architectures. It also places additional burdens on the collector when type tags are stored with pointers to objects rather than with the objects themselves: special tags must be added to objects when the objects are copied by the collector so that the objects can be swept correctly.

The system described in this paper, therefore, makes use of the *Big Bag of Pages* (BIBOP) typing mechanism, which views memory as a group of equal-sized segments with an associated type table mapping segment numbers to types [20]. BIBOP typing has fallen into disfavor in recent years because newer type systems provide faster allocation and type manipulation and because of percevied problems with the handling of large objects. We have found, however, that these problems can be eliminated through the use of a hybrid typing system.

In our system, fast primary typing is provided by associating tags with pointers to objects (for most common types) or with the objects themselves. Secondary typing, or *metatyping*, is provided by the BIBOP type table. Metatypes allow the system to mark segments according to certain important characteristics. In particular, segments are marked as "holes" if they are contained within the heap but are not logically part of the heap, and they are marked "executable" if they contain executable code. Segments are also marked according to how the objects within them must be handled by the garbage collector, *i.e.*, objects containing no pointers to be swept, objects containing only pointers, and objects such as stacks which require customized sweeping are all kept in different kinds of segments. The generation to which each segment belongs is also recorded as part of a segment's metatype. Immutable objects can be separated from mutable objects to improve virtual memory behavior and to reduce the number of areas for which the generational garbage collector must look for pointers from older to younger generations.

In order to keep objects segregated, BIBOP requires the use of multiple allocation pointers. This would seem to prevent us from using the fast allocation strategy possible with the flat memory model. Our system delays drawing most metatype distinctions, however, until after an object survives its first collection so that new objects may be allocated into a single allocation area[1]. Thus, we can use a single allocation pointer for nearly all types of objects.

The segmented memory model would also seem to complicate the handling of large objects, but this is not the case. Objects larger than one segment in length are simply allowed to span segment boundaries. Since metatype information is stored in a separate table rather than within the segments themselves, the handling of large objects creates no particular problems. In fact, the segmented model allows us to avoid copying large objects during garbage collection; doing so involves little more than changing the segment's metatype information. Since our segments are relatively small (currently 4K bytes), objects need not be extremely large to benefit from this optimization.

The storage management system described in this paper has been implemented and is in use as part of *Chez Scheme*, a high-performance implementation of Scheme [8]. Although hybrid typing mechanisms and segmented memory models are not entirely new, we believe we are the first to implement a segmented memory model using an extensive set of metatypes with fast inline allocation. Also, many of the purposes for which we use metatypes appear to be unique, including separating code from data and separating mutable from immutable data. Our mechanisms for collecting stacks and code objects, which both contain a mix of pointer and nonpointer data, are also new[2] (see Section 5.3). We also describe a variant of card marking [19, 25] for remembering pointers from older generations to newer ones. This variant allows us to perform the marking inline while maintaining a record of the youngest generation involved, reducing card sweeping overhead when more than two generations are used. Many of the other features and derived benefits we discuss, *e.g.*, how our storage manager coexists with other languages' run-time systems, are likely not new but either have not been well documented in the literature or have not previously been implemented.

The remainder of the paper is organized as follows. Section 2 discusses background information on run-time object representation and tags. Section 3 presents the basic framework of our storage management system. Section 4 describes some of the benefits derived from the segmented memory model. Section 5 describes how the system exploits metatype distinctions. Finally, Section 6 summarizes the paper and discusses related work.

2 Type Representation

There are several common mechanisms for dynamic allocation and typing of objects [21]. The simplest representation is that provided by *typed objects*, in which the type tag is stored explicitly within the object.

The pointer to the object carries no information other than the object's location. All objects are stored indirectly, even those which have a potentially compact representation such as integers or characters. To determine an object's type, a memory reference is required to extract the tag

[1] Code objects and weak pairs are distinguished immediately; see Section 3.

[2] Our handling of stack frames is somewhat similar to tagless garbage collection methods [3].

from the object. Typed objects can easily accomodate a large number of tags as the object can simply be made larger to increase the size of the tag.

With *typed pointers* the type tag is included within the pointer to the object. The type tag is typically stored in either the high or the low bits of the pointer, with the rest of the pointer representing the actual address of the object. Storing the tag in the high bits usually restricts the available address space, whereas storing the tag in the low bits usually forces objects to be aligned on multiple byte boundaries. In either case, determining the type of an object requires extracting the tag from the pointer.

The typed pointer representation allows faster type checks than with typed objects, as the pointer need not be dereferenced to extract the tag. If the object is reached without going through the pointer, however, its type cannot be determined since the tag is stored apart from the object. This happens, for example, during the sweep phase of a breadth-first copying garbage collector [10]. One solution to this problem is to require the garbage collector to insert type descriptors as it copies objects, while their types are still available.

Some types of objects, *e.g.*, fixnums and characters, can be encoded in a small number of bits. With typed pointers, these *immediate* objects may be stored within the pointer in place of the address bits. This eliminates the storage overhead and indirect access for this common form of value.

When some object alignment is enforced, some of the low bits of an object's address carry no useful information. For example, if objects are allocated only on even eight-byte boundaries, the three lower bits of any object's address will always be zero. This is a convenient place to store the type tag of a typed pointer. With this representation, masking is usually not required to access the object. Because of the tag, the pointer will always be a known constant away from the object's true address, so the pointer may simply be adjusted before loading or storing the object contents. By using the known constant to adjust the field offset for the field being referenced, the access may be done with little or no extra cost [18, Chapter 3].

To use typed pointers with the tag in the high bits, address extraction is more complicated for most platforms. Some architectures have a word size larger than the address size and ignore the upper bits for addressing purposes. The tag may be safely stored in these bits without affecting the address. On most modern platforms, however, the word size matches the address size, so that a high-bit tag will involve the most significant bits of the address. The pointer cannot simply be adjusted by some constant to compensate for the tag unless the address space is restricted only to the lower (non-tag) address bits. For such architectures, the address must be stored in the lower bits and shifted up to allow full use of the address space.

On the other hand, these bits could be used as part of the address. In this case, the address space is divided into several distinct type regions, with objects of the same type being allocated into the same region [13]. This separation removes the need for type descriptors during garbage collection as object type may be determined by address. Each object type is assigned a type region with its own allocation pointer. As object type depends on enforcement of these divisions, the region boundaries may not change at run time, even for programs with allocation skewed toward some particular types. Object allocation will be more "sparse" than in the other approaches, negatively impacting virtual memory managment on some architectures and operating systems. Immediate values may be encoded as for typed pointers, but as these values correspond to virtual addresses, portions of the address space will be made unavailable.

The BIBOP typing mechanism [20] also employs a representation in which the object's type is determined by the high bits of its address. Rather than encoding a type tag, however, these bits represent an index into a table where the type is stored. Memory is divided into segments of equal size, each with an entry in the type table, and all objects allocated in a segment are of the same type. The high bits of an object's address thus represent the number of the segment on which it resides, and the low bits the offset from the segment base. The table lends flexibility to the type system. Unlike the fixed boundaries of the high tagged typed pointer mechanism, type regions are dynamically resizable as segments may be mapped to different purposes as needed by simply adjusting the table entries. Space need not be reserved for a type until an object of that type is first allocated, so unused types place no storage burden on the system. On the other hand, type checks require an extra memory reference since they must access the type table.

Both BIBOP and high tagged typed pointers require separate allocation pointers for each object type. In general, a register cannot be dedicated to each type. A few of the common cases may be kept in registers, but the rest must be stored in memory.

Typed objects and low tagged typed pointers do not require this separation. Objects of different type may be allocated in succession, so only a single allocation pointer is needed. By keeping this in a dedicated register, allocation can be made very quick [1].

An excellent survey of automatic storage reclamation techniques for dynamically allocated objects is provided by Wilson [24].

3 Storage Management Framework

Our storage management system employs a hybrid run-time tagging mechanism that incorporates elements of typed pointers, typed objects, and BIBOP typing (see Figure 1). Low tagged typed pointers are used for primary typing. This permits fast typing and compact representations both for a small number of commonly used object types, such as pairs and symbols, and for commonly used immediate values, such as fixnums and characters. All objects are allocated on eight-byte boundaries allowing the low three bits to be used as a type tag. Low-tagged pointers incur little or no run-time overhead for allocation, integer arithmetic, and field dereference, and type testing involves only a mask and compare [21].

Using a larger tag (and a correspondingly coarser alignment boundary) would result in increased storage fragmentation to maintain the object alignment restriction. Three tag bits, however, do not suffice to encode all Scheme object types. All types not representable as primary types are represented as typed objects using one of the primary type tags as an escape code. The resulting hybrid type system is extensible, and it allows fast compact type representations for the most common object types. Some type checking overhead for variable-length typed objects can be

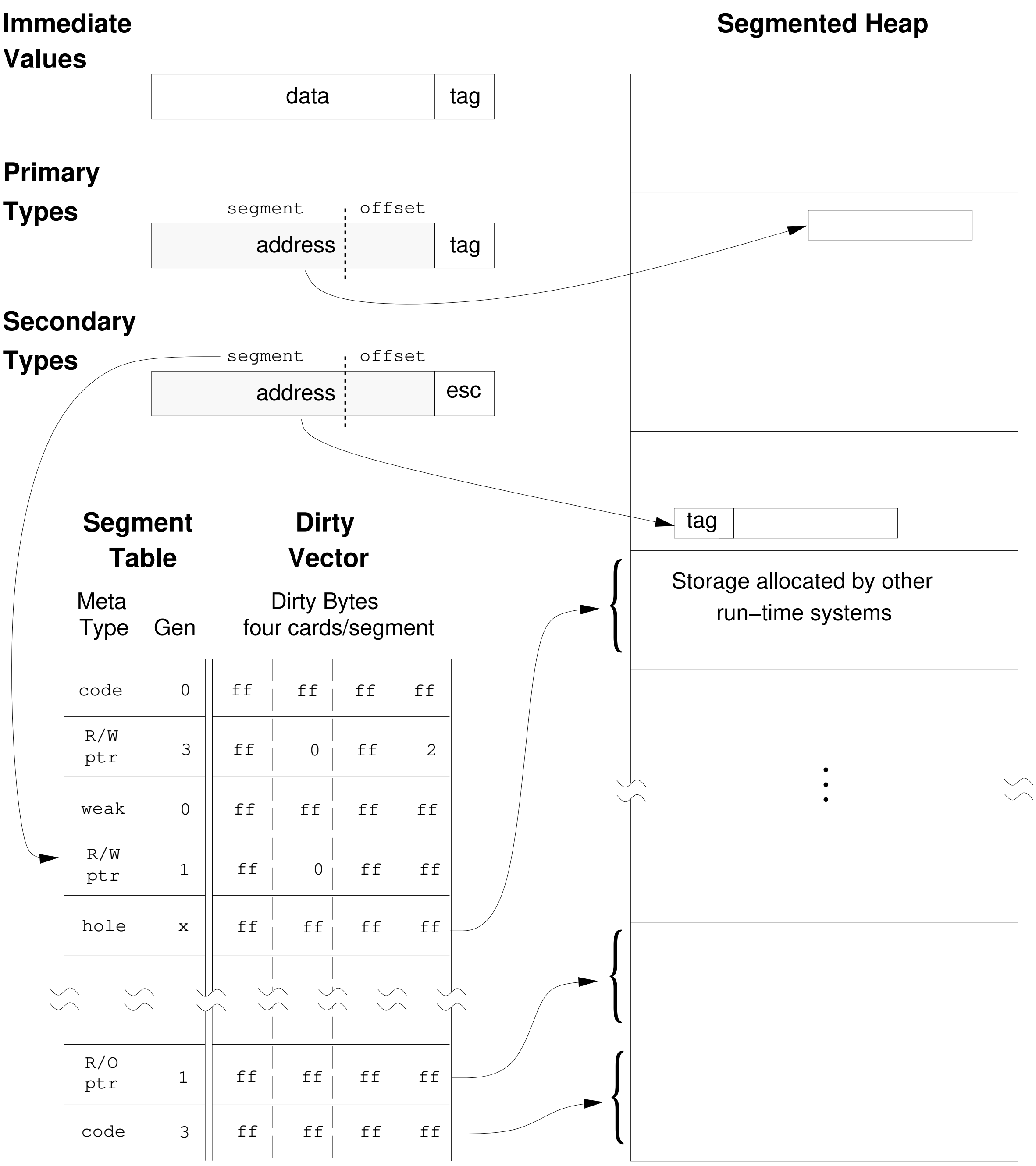

Figure 1. This figure summarizes the key features of our storage management system. The primary type mechanism uses typed pointers. The lower 3 bits of each immediate value or pointer are reserved for type tags (all objects are aligned on 8-byte boundaries). One of the primary types is reserved as an escape to a secondary typed object mechanism. The address field of each pointer is logically divided into a segment number and offset. The segment number indexes both the segment table and dirty vector; each entry corresponds to a 4K heap segment. For the purpose of younger object reference tracking, each segment is logically divided into four 1K cards and each card is associated with a dirty byte in the appropriate word of the dirty vector. Dirty bytes are initialized to a fixed value (the maximum generation number) so that the dirty vector can be swept a word at a time. A dirty byte is changed to zero when an assignment occurs and reset by the collector to reflect the lowest generation number of any object actually pointed to from the card. The metatype field contains three subfields (not shown): a large-object flag, an old-space flag used during collection, and a general metatype tag.

hidden by combining the length (which often must be loaded anyway for range checking) and type tags using techniques similar to typed-pointers.

BIBOP typing is used to associate metatype information, including generation numbers (and dirty bytes; see Section 5.1), with groups of objects on a per-segment basis. Metatype information is recorded in a segment table. This allows objects to be segregated by characteristics, rather than by types. This in turn allows the storage management system to specialize its treatment of objects for performance, behavior, handling, or other reasons.

BIBOP-style segmentation requires that a separate allocation pointer be used for each metatype. Placement into specific metatype areas is delayed for most objects, however, until the objects reach their first collection, at which point the collector decides to which metatype each object belongs. New objects may therefore be allocated into a single allocation area. The new allocation area is simply a set of one or more contiguous segments set aside for this purpose by setting their metatypes to "new" in the segment table.

New allocation is performed inline using a single allocation pointer, held in a machine register when possible, pointing to the next available location. Allocation then requires only a register move and increment for each group of objects allocated, as well as a check for overflow of the allocation area. This check is made by comparing the allocation pointer with an "end-of-allocation" pointer, also held in a machine register when possible. The check need not be made for every allocation operation, but instead may be made once for each entry (or reentry) into a procedure. Where sufficient operating system support is available and trapping overhead is not too great, the first page beyond the new allocation area could be marked read-only so that attempts to write into it result in a trap [1]. In order to use this trick, the compiler must assure that at least one word per page in the object is written before the allocation operation is assumed to have succeeded. Because explicit checks have not been a performance problem in practice, our system currently uses explicit checks regardless of the operating environment.

Two types of objects, weak pairs and code objects, are not allocated into the same new allocation area as other objects. Code objects are kept separate to avoid intermixing read-write data with instructions, as required by some architectures and operating systems. Weak pairs are separated from normal pairs, since they must be handled differently by the garbage collector (see Section 5.4). Since most programs do not allocate many weak pairs or code objects, the additional overhead required to extract their allocation pointers from memory is not a serious problem.

As noted above, the garbage collector performs metatype separation for objects of other types. Any newly allocated object found reachable by the collector is copied into a segment of the appropriate metatype. Since the information required to determine the appropriate metatype (usually, just the object's primary type) is generally needed as well to copy the object, this responsibility adds little overhead to the collector.

Since metatypes are dynamically resized, they are unlikely to be held in contiguous memory segments. More likely, they will be spread across the heap. To aid the garbage collector, a pointer at the end of each segment links it to the next logical segment in the metatype. In addition, a sweep pointer and a pointer to the first segment of each metatype—along with the allocation pointer already mentioned—are kept for each metatype within each generation.

Some fragmentation is possible. When the allocation of an object into a metatype requires more space than is available, the metatype must be extended. Allocation continues in the extension, wasting any remaining space. To offset the effects of fragmentation, a small segment size is used (4K bytes in the current implementation). This does not limit the maximum size of an object since objects may span segment boundaries, but it does lower the number of bytes which may be lost to fragmentation. In practice, fragmentation is not a problem.

When further segments are needed the heap is extended through an operating system request, *e.g.*, the Unix `sbrk` call. The segment table representation must be able to handle such extensions. There are several options. A fixed-size table could be used, allowing no more than some fixed maximum number of heap segments. Such a table would need to be excessively large to handle extreme cases—negatively impacting small applications. Limiting its size, however, would disallow use of the entire address space. A more flexible option is to include each segment's metatype information as a record or tag in the segment itself. Then the table will extend naturally whenever a segment is added. This approach, however, does not allow objects to span segment boundaries without seriously complicating the mechanism. Worse yet, the locality of type records will be significantly worse than with a compact, separate table. To scan even a small portion of the segment table, potentially many pages must be brought into main memory and the cache. A full scan of the segment table would touch every segment in the heap—for a relatively small amount of information.

To get both good locality and flexibility, a resizable table is maintained instead. A small number of segments are set aside to hold the segment table at system startup and more segments added as the heap outgrows the table. These segments must be contiguous to allow fast table indexing based on an object's address, so table relocation is sometimes necessary during table resizing if the next contiguous segment is not available.

4 Flexibility of the Memory Layout

By employing a segmented memory layout, the storage management system gains a great deal of flexibility. Total heap size is limited only by available virtual memory, as are the sizes of individual metatypes (including generations). The storage management system can dynamically resize metatypes by adding or removing segments as needed, and, if a metatype is unused, it requires no space. Because objects may span segment boundaries, there is also no inherent limit on object size.

The ability to dynamically resize metatypes is especially effective when combined with a copying garbage collector. With a conventional copying collector, the heap is divided into two semispaces, with one used for allocation and the other reserved for garbage collection. The collector copies all reachable objects from the first semispace to the second, then switches their roles. As the virtual memory pages of the reserved semispace are unused until garbage collection, however, they are all less likely to be in physical memory than any of the pages of the allocation space. This leads to a cyclic page access pattern which can have devastating effects on platforms with virtual memory. If there are N pages in each semispace, $2N$ pages will be used each complete allocation/collection cycle.

With a segmented memory layout, however, it is not necessary to reserve memory for the new space in advance. As objects are copied, the corresponding metatypes are dynamically resized as needed. So, if the reachable objects cover a total of M pages, only $M + N$ pages are required during each allocation/collection cycle. If collection is successful, M will tend to be much smaller than N and many fewer pages will be required than in the basic model, although fragmentation for unused portions of some segments may raise the total somewhat. More importantly, once collection is over, the segments just vacated can be recycled immediately for new allocation while they still reside in physical memory. These effects are also noted by Hudson [17].

Allocation by other run-time systems is easily handled through metatyping. If, during heap extension, the memory returned by the operating system is not contiguous with the rest of the heap, the intervening memory must be avoided as it may used by subprograms written in other languages running within the same process image. These segments are marked as metatype "hole" in the segment table and are not touched by the collector or used for allocation. These segments fragment the heap somewhat, but they do not otherwise affect the system.

Objects may be allocated directly into any metatype at any time. Consequently, objects may easily be allocated directly into older generations. This feature is exploited by our I/O subsystem, which must sometimes enlarge the buffer associated with a string output port (a string-based output file). Rather than allocate the new buffer in the youngest generation, it is allocated in the same generation as the port. This allows us to pretend that the port contains no mutable pointers so that we can avoid remembering the potential older to younger generation pointer.

The organization of the metatype system allows the use of any number of metatypes with minimal changes to the run-time system. This greatly facilitates experimentation, since new metatypes can be added or existing ones altered (along with their handling routines) without significantly affecting the rest of the system. For example, the generational garbage collector supports an arbitrary number of generations. More generations may be added by simply increasing the number of valid values for the generation field of the segment table, and modifying certain garbage collection routines to recognize and use these new values.

5 Metatyping Applications

The segmented memory model allows us to separate objects by whatever characteristics we choose. Although some metatypes map to individual object types, a one-to-one correspondence between metatypes and object types is not needed in many cases. For example, the collector can make use of the segregation of objects that contain pointers from those that do not, but the collector would have little use for a separation of bignums from ratnums. Some metatype information, such as generation numbers, does not correspond to object type in any way. A segment's generation number is orthogonal to other metatype information, so each generation may encompass segments of many different metatypes.

In the remainder of this section we describe some of the applications for which metatypes are used in the current implementation.

5.1 Card marking

The card marking [19, 25] mechanism for tracking pointers from older generations to newer generations fits well with the BIBOP heap organization. The dirty marks used to record that an assignment has occurred to an object within a card are simply another form of metatype information to be tracked for each segment.

Our system uses bytes rather than bits to store the dirty marks for two reasons. First, it is typically faster to write a byte than to set a single bit [9, Chapter 6]. Second, because the garbage collector supports more than two generations, it is useful to know for each card the youngest generation to which a pointer exists, rather than simply whether a pointer to some younger generation exists. Without this information, the collector would have to sweep every dirty card on each collection, even if the card contains no pointers into the generations being collected.

This refined information is derived by the collector whenever it sweeps a dirty card; assignments simply write a zero byte to the dirty vector, denoting the possibility that a pointer exists to the youngest generation. Because assignments need only extract a card index from the address to which the assignment is made, and write a byte to the dirty vector (which may be held in a machine register) at this index, assignment operations need not be performed out-of-line and are not unduly penalized.

In our system dirty marks are kept in a table separate from the segment information so that each group of four bytes is 32-bit word aligned, allowing the collector to sweep the dirty vector a word at a time. A byte set to the maximum possible generation number, $\texttt{ff}_{16}$, denotes a totally clean card, so the bit pattern $\texttt{ffffffff}_{16}$ denotes four totally clean cards. (See Figure 1.)

5.2 Large object handling

Ungar notes that substantial gains may be made by treating large objects specially [22, 23]. In his system, objects are reachable only through their headers. By keeping large objects in a separate *large object area*, but leaving the headers in the same area as the small objects, copying costs for large objects can be largely eliminated (though objects which may contain pointers must still be scanned). The large object area is managed as a free list.

Hudson [17] also stores large objects in an area separate from the main heap, but does not introduce the indirection required to go through an object header for Ungar's mechanism. Instead, the generation for each object is recorded with the object itself.

In both of these mechanisms, large objects are managed differently from other objects. In our system, large objects merely span segment boundaries; this causes no particular difficulties and no other special treatment is required. On the other hand, the segmented memory provides a simple mechanism for avoiding the copying of large objects. Each large object is copied at most once, when it is first collected, and placed alone into a group of contiguous segments. The first of these segments is marked "large" in the metatype field of the segment table. When the collector encounters an object on a segment marked "large" during a subsequent collection, it merely marks the object as having been forwarded by eliminating the old bit and updating its generation number in the segment table, and places the object on a queue for later sweeping (if necessary).

For newly allocated objects that do not fit within the current new allocation area, the "large" determination is made immediately, rather than at the first collection, in which case the object need never be copied. This will always be the case for objects larger than the initial new allocation area.

5.3 Sweeping based on metatypes

As noted in the introduction, metatyping can be used to distinguish objects that contain pointers, like pairs and vectors, from those that do not contain pointers, like strings and bignums. For objects that do not contain pointers, the collector can avoid unnecessary sweeping if such objects are kept separate from those that can contain pointers.

As pointers from older to younger objects must be recorded to allow the generational garbage collector to locate portions of the older generation to sweep, it is useful to further divide the pointer-containing objects into those which are read-only and those which may both be written as well as read. Since younger object references can be caused only by assignment operations, tracking is necessary only for writable pointer-containing segments. With a card marking system such as the one described in Section 5.1, this allows the collector to completely avoid touching objects that cannot contain pointers to younger objects when scanning dirty cards.

Code objects are given their own metatype because they contain binary data (instructions) as well as embedded pointers. These pointers must be swept by the garbage collector. So that the collector can them, the location of each pointer in the code is recorded in a relocation table associated with the code object. An entry in the relocation table specifies where a pointer is stored (the code offset), the offset from the start of the pointer to the actual address stored there (the item offset), and how the pointer is stored (the addressing mode). A pointer to the code object stored in the relocation table header is not updated to point to the new code object until after the pointers embedded in the code object have been swept; this is used during collection to compute the displacement from the old location of the object to its new location so that any PC-relative displacements can be recomputed.

Stack objects are also given their own metatype. Stack objects may contain nonpointer data such as floating point values and possibly "holes" (caused by "dead" variables or introduced for alignment) as well as live pointers. A live-pointer mask is associated with each frame by placing it behind the return point in the instruction stream (see Figure 2). The collector uses this mask to determine which frame locations need to be swept. Return addresses found in each frame represent untagged pointers into the middle of code objects; these return addresses must be treated specially as well. The code object offset is stored with the live-pointer mask behind the return point so that the code object can be relocated and the updated return address computed. Stacks are "walked" by the collector using frame size information also stored behind the return point [16].

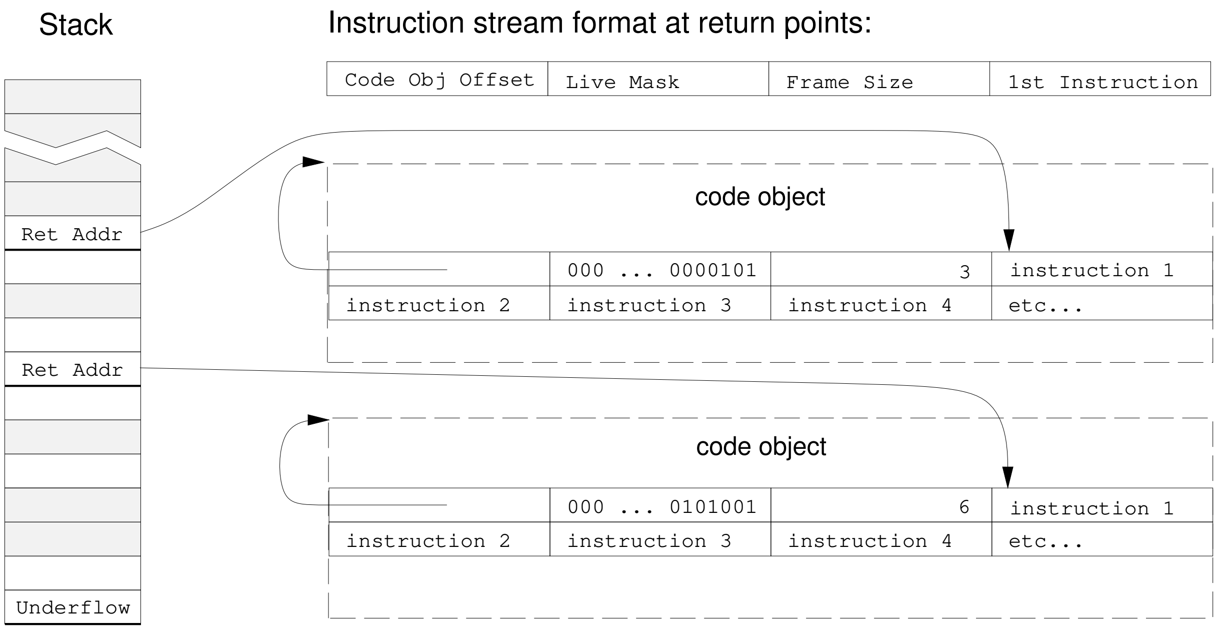

Figure 2. Our compiler stores three words in the code stream behind each return point: the frame size, the live-pointer mask, and the code object offset. These fields allow the collector to "walk" the stack without the need for an explicit frame link placed in the stack dynamically by the procedure call code; to sweep only those stack locations that contain live pointers; and to locate and forward the code objects into which the return addresses point. For large frames the live-pointer mask is a pointer to a variable-length heap-allocated mask.

The live-pointer mask is similar to the type descriptors required to support tagless garbage collection [3].

5.4 Weak pairs

Weak pairs are identical to normal pairs except that they have a weak `car` field pointer. A weak pointer to an object is treated like a normal pointer so long as nonweak pointers to the object exist. If only weak pointers to an object exist, however, the pointers are "broken" by the garbage collector and the object is released. As a result, an object that is not accessible except by way of weak pointers is ultimately discarded. In this case, a known value, such as false, is left in the `car` field to indicate the loss.

Like code objects, weak pairs are segregated from other objects when they are allocated because they require special treatment by the garbage collector [11]. They are identical in every other way to normal pairs. If this were not the case, *i.e.*, if we had chosen to use a different representation for weak pairs, all pair manipulation primitives would have to operate on both representations, resulting in undue overhead and complexity.

6 Conclusions and Related Work

In this paper we describe the design and implementation of a high-performance storage management system that is portable and that provides a flexible metatyping mechanism. This storage management system is used by *Chez Scheme* [8] on a variety of architectures under several operating systems with widely varying capabilities.

Use of the segmented memory model and BIBOP metatyping results in no overhead outside of the garbage collector. Scheme code compiled for the segmented model is identical to that which would be produced for a flat memory model, with fast inline allocation and fast type testing. The collector incurs some overhead for maintaining the segment table and a slightly higher per-object copying cost because of the need to maintain multiple allocation pointers. This cost can be offset somewhat by dedicating registers to hold the allocation pointers of the most common metatypes during collection. Some fragmentation within segments is possible, although as we noted earlier this is not a problem in practice. The memory required to hold the segment table is minimal: only a few bytes per (4K byte) segment.

Added overhead in the collector is also offset by several optimizations not possible in a flat heap structure. The garbage collector avoids unnecessary sweeping by separating objects that might contain pointers from those that cannot and also by separating mutable from immutable objects. Executable code and data stored in the heap are placed on different pages, reducing cache flushing costs when code objects are created or relocated. Copying costs during garbage collection are eliminated for large objects without the need to treat them specially when they are allocated.

The segmented memory model also yields better virtual memory behavior than the standard flat n-space memory model, as vacated pages are reused immediately after garbage collection. Appel [2] and Shaw [18, Chapter 6] have independently described a generation-based collector for a flat memory model with the same property.

A possible extension of this work is to allow the compiler to use the metatyping mechanism to customize collector behavior at a much finer granularity in order support tagless garbage collection. For statically typed languages, like ML, type tags are required only for collection. Appel [3] has suggested associating a record describing the types of the frame elements with each activation record. Goldberg [14, 15] subsequently noted that the compiler could generate code the collector invokes directly rather than a record which the collector must interpret. In either case, type information is determined only during the tracing phase of a collection. In order to use a nonrecursive breadth-first collector the type must be associated with the object during collection, although, as Appel [3] notes, the types can be stored separately from the objects, swept in parallel, and discarded after the collection.

Our metatyping mechanism provides an alternative solution. Objects of the same type can be segregated when they survive their first collection and assigned a metatype that enables the collector to collect such objects. Because ML allows a program to define an unlimited number of types and because there may only be a few objects of a given type, this approach may result in significant fragmentation. Fragmentation can be reduced, however, by using a hybrid approach; the most common types can use the metatyping mechanism to avoid tagging overhead while less common types can be placed with their tags into a shared metatype. The distinction between common and uncommon types might be made dynamically based on actual object counts.

In his book, *Compiling with Continuations*, Appel [4, page 211] makes a similar observation. He notes that with BIBOP typing for older generations, type tags can be removed from objects that survive their first garbage collection and associated with segments instead. Although he does not make the observation that this idea can be applied to tagless garbage collection, the generalization is fairly obvious. He does not mention how he would handle fragmentation caused by the proliferation of types. This fragmentation could be especially serious with the large segment size he suggests (64K bytes).

Like our storage management system, the language-independent garbage-collection toolkit described by Hudson [17] divides memory up into segments (called *blocks*). They do so in order to reduce memory demand (compared to a standard semi-space collector) and to support a generation-based collector with a varying number of generations of varying size. Large objects, however, are placed in area separate from the main heap and are managed entirely differently from other objects. They mention the possibility of using a table constructed by the compiler to map the live frame locations for the collector that is similar to the live-pointer mask mechanism that we have implemented and described here. They do not mention using metatypes for segregating objects based on characteristics, and they make no commitments with respect to typing or allocation since their system is intended to be language-independent.

Bartlett's "mostly copying" collector [5] uses a segmented memory layout similar to ours in order to mark groups of objects pointed to by an ambiguous root set (whose values cannot be modified) as "forwarded" so that they need not be copied. Later work [6] associates a generation number with each segment as well. In addition, Bartlett's Scheme->C implementation [7] uses metatyping to separate pairs from variable length objects, in order to simplify sweeping, and to mark segments as "continued" when an object spans a segment boundary so that the collector can find the beginning of the object. Although his system supports fast inline allocation, doing so is trivial since only two allocation pointers are required to handle the two metatypes his system supports.

The page-based incremental collector described by Ellis [12] also uses a "boundary crossing" table to enable the collector to find the beginning of an object that crosses page boundaries. This can be seen as a simple use of metatyping.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25, June 1987.

[2] Andrew W. Appel. Simple garbage collection and fast allocation. Dept. of Computer Science CS-TR-143-88, Princeton University, Princeton, NJ 08544, March 1988.

[3] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2):153–162, June 1989.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA, 94301, February 1988.

[6] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA, 94301, October 1989.

[7] Joel F. Bartlett. SCHEME->C a portable Scheme-to-C compiler. WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA, 94301, January 1989.

[8] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme.*

[9] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language.* PhD thesis, Stanford University, 1992.

[10] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[11] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Design and Implementation*, pages 207–216. SIGPLAN, ACM, June 1993.

[12] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multi-processors. Systems Research Center 25, Digital Equipment Corp., February 1988.

[13] Scott E. Fahlman and David B. McDonald. Design considerations for CMU common lisp. In Peter Lee, editor, *Advanced Language Implementation*, chapter 6, pages 137–156. The MIT Press, 1991.

[14] Benjamin Goldberg. Tag-free garbage collection for stongly typed programming languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176. SIGPLAN, ACM, June 1991.

[15] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 53–65. SIGPLAN, ACM, June 1992.

[16] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Design and Implementation*, pages 66–77. SIGPLAN, ACM, June 1990.

[17] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Dept. of Computer and Information Science COINS Technical Report 91-47, University of Massachusetts, Amherst, Object Oriented Systems Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, September 1991.

[18] Robert A. Shaw. *Emperical Analysis of A Lisp System.* PhD thesis, Stanford University, February 1988.

[19] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. Thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science Department, Cambridge, MA., September 1988.

[20] Guy Steele, Jr. Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, September 1977.

[21] P.A. Steenkiste. Tags and run-time type checking. In Peter Lee, editor, *Advanced Language Implementation*, chapter 1, pages 3–24. The MIT Press, 1991.

[22] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88 Conference Proceedings*, pages 1–17. SIGPLAN, ACM, September 1988.

[23] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[24] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management: IWMM 92 proceedings*, pages 1–42. INRIA in cooperation with ACM SIGPLAN, Springer-Verlag, September 1992.

[25] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *OOPSLA '89 Conference Proceedings*, pages 23–35. ACM, October 1989.