Static and Dynamic Partitioning of Pointers as Links and Threads Technical Report 437

David S. Wise and Joshua Walgenbach
Computer Science Department, Indiana University
Bloomington, Indiana 47405–4101 USA
`dswise,jwalgenb@cs.indiana.edu`

Abstract

Identifying some pointers as invisible threads, for the purposes of storage management, is a generalization from several widely used programming conventions, like threaded trees. The necessary invariant is that nodes that are accessible (without threads) emit threads only to other accessible nodes. Dynamic tagging or static typing of threads ameliorates storage recycling both in functional and imperative languages.

We have seen the distinction between threads and links sharpen both hardware- and software-supported storage management in SCHEME, and also in C. Certainly, therefore, implementations of languages that already have abstract management and concrete typing, should detect and use this as a new static type.

Categories and subject descriptors:
D.3.3 [Programming Languages]: Language Constructs and Features—data types and structures, dynamic storage management, abstract data types; E.2 [Data Storage Representations]: Linked representations; B.3.2 [Memory Structures]: Design Styles—primary memory.
General Term: Languages.
Additional Key Words and Phrases: storage management, reference counting, garbage collection, tags.

1 Introduction

All active references or pointers originate from "roots" in the programming environment; common roots are the register file and recursion stack. Storage management preserves linked structures that are accessible from roots, where the garbage-collecting traversal begins.

Definition 1 *A pointer is a* link *if it is essential to the integrity of a linked structure.*

Informally, a linked structures must be rooted and spanned by links.

Definition 2 *A pointer or reference is a* thread *[12] if it is optional in a linked structure, in the sense that it can be inferred from a traversal from the roots that follows links exclusively.*

On first reading of any program, one may assume that all pointers/references are links; threads can be introduced later. Usually there are several ways to establish a partitioning between links and threads. Of course, it is best to choose a simple one; for instance, links originate from roots and extend homogeneously to the more remote nodes of a structure. Some fields in each record can be statically typed to "link" and others to "thread." Often, however, a dynamic attribute is used with run-time tags present to distinguish the kind of pointer.

Whether or not threads are identified, however, the semantics of a program must remain the same. Our purpose in identifying them is to unburden the storage manager from dealing with them. Because threads are redundant, it can ignore them and the performance of the program improves without changing its result. That is, link/thread can be a static type like lazy/strict (w.r.t. evaluation of a function's argument) or sticky/unique (w.r.t. reference counting); we might conservatively presume that an unknown type is the first of each pair but, whenever we can discover the second, we can compile for better run-time performance.

Examples of threads are already familiar to the reader. The REAR pointer to the end of a singly-linked queue should statically be treated as a thread. Threaded trees [12], in contrast, are an example of a dynamic link/thread distinction, because a pointer requires a run-time tag to imply its meaning. Another familiar thread is the "reverse" pointer paired to every "forward" link as an edge in a doubly-linked list. Others include *weak pointers* [15] in many LISP and SCHEME implementations.

The new principle for implementors of programming systems and for low-level programmers is (Invariant 1) that threads point only to nodes accessible exclusively via links.

Programmers should recognize dynamic threading more effectively than they do now; compilers should better recognize static threading [17]. Their use accelerates production code (sometimes called the *mutator* in systems with automatic storage management) by short-circuiting redundant reallocation, and also they can accelerate space recovery (the *collector* when one exists). This observation certainly applies to systems with automatic collection, like LISP, SCHEME, ML, HASKELL, and SMALLTALK, but we found it to be already useful under languages like C and C++ where space recovery is "manual."

The payoff for sustaining the live/dead distinction on pointers is that the work for the collector is considerably reduced, and the locality of the mutator is similarly increased. The exact impact depends on the kind of storage manager used. (This paper takes the perspective that reference counting is distinguished from garbage collection [24, 8, 12, p. 412], rather than one of its techniques [6].) For instance, a garbage collection can ignore all threads, saving the time to traverse them. Under reference counting, some counts will be one tick lower, saving both their increments and decrements, and often avoiding troublesome cycles. By focusing only on links, we foresee further improvements from compile-time space analysis that uses techniques like linear logic and monads. In all these cases the mutator runs more often, and can run more locally whenever nodes are discovered to be uniquely referenced, enabling *in situ* side-effects instead of allocating more space. The payoff on cache-based architecture is fewer cache misses and far greater speed..

Improved performance is not obtained without a task for the compiler or a burden on the programmer. Unconstrained use of threads leads directly to errors from dangling-references. Removal of the last link to a node renders unstable any remaining threads there; the node can be recycled unpredictably, transforming any dangling threads into a dangling reference (to some new incarnation at that address.) Treating this last thread as a link would have precluded recycling, and avoided a *nasty* error.

The ultimate resolution of such dangling threads is to require the compiler, rather than the programmer, to enforce Invariant 1, below. The compiler must be able either to infer statically or to provide run-time code verifying that certain pointers are threads, ensuring that they are manipulated in a manner that sustains the invariant. That is, the compiler should be able to learn which pointers are of "thread" type, and then to validate their consistent use.

This paper has six sections, including this introduction. Section 2 gives definitions and Invariant 1. The next section reviews historical examples of the concept, and is followed by Section 4 on a logic for the invariant and anticipates its formal type. Section 5 describes our motivation and results, the motivation for the general observation, and the final section offers conclusions and a challenge for compiler-writers.

2 Definitions and Invariant

Definition 3 $T = \{live, dead\}$ *is the set of* tags.

"Tag," is used as in threading of binary trees [12]. Links are live, and threads are dead.

Definition 4 *Let the set of active* nodes *in the heap be N; each one is perceived as a record with a small set of fields, each identified from a finite set L of* labels. *The edges or* pointers *in our linked structure comprise the mappings from a function in $N \times L \to N$, a digraph with labeled edges.*

Of course, the function defining pointers changes with every step that changes linking.

Definition 5 *The* source *of pointer $\langle m, l\rangle \mapsto n$ is m. Its* label *is l; its* destination *is n.*

Definition 6 *Node m is said to* emit *any pointer of which it is a source. Node n is said to* absorb *a pointer of which it is a destination.*

Definition 7 *The set of pointers in any structure is partitioned into a set of* links *and a set of* threads.

There are two ways to achieve this partitioning. One is to use a static function in $L \to T$ to tag every label; this corresponds to static typing on pointers that does not change even if a pointer does; see Section 4. Another is to extend the pointer function to one in $N \times L \to N \times T$ so every pointer carries a tag; this corresponds to dynamic tagging of an attribute at run time.

Definition 8 [7] *The* reference count *of a node is the number of links absorbed by it.*

Definition 9 [27] *A link is* unique *when its destination has a reference count known to be one. Otherwise, it is* sticky.

The term "sticky" is borrowed from the convention of fitting a static infinity into the range of reference counts [5], whence neither increments nor decrements change it. (But a full garbage collection might [24, 27].) A reference count can be both one and sticky, after an imperfect counting protocol loses the precise count on a node and no longer "knows" it.

Convention *Roots emit only links.*

Definition 10 *A node is* accessible *if it absorbs a link from a root or an accessible node.*

Invariant 1 *Accessible nodes can emit threads only to other accessible nodes.*

There is no requirement that threads be introduced at all, but if Invariant 1 can be assured, then run-time performance will be enhanced by them. Two more verbs are useful:

Definition 11 *Changing a tag from "live" to "dead"* kills *the associated pointer; changing a tag from "dead" to "live"* resurrects *it.*

Maintenance of Invariant 1 is the obligation of a programmer who would use threads, or of a compiler that would kill links. Although it seems to be burdensome, it can often be sustained by simple coding practice or by stronger, verifiable constraints.

3 Historical Examples

3.1 Static Links and Threads

The REAR pointer to the end of a singly-linked queue should be treated as a thread. Then FRONT and all its internal links likely remain unique. A corollary to this convention, that directly yields superior code, is that REAR of an empty queue must be undefined: $\perp$ [11], rather than something meaningful [12, pp. 256–257] [20, p. 29] [13, pp. 79–80], because there is no accessible node to absorb a thread.

In doubly-linked lists where forward pointers are links, we can treat reverse pointers as threads.

Definition 12 *Reverse pointers in a doubly linked list are called* counterpointers.

In the instance that the list is not circular (when both ends point to NIL), the links form a singly linked list, referenced by a link FRONT and the counterpointing threads form a singly linked list in the reverse direction. beginning from the REAR thread.

In all cases of a linear list, the links form a simple, singly linked list whose space might easily be recycled, either by hand, by elementary reference counting, or by "unique" typing at compile time.

The use of reference counting in early list-processing systems [7, 23] reveals other kinds of pointers long treated as threads. The READERS of SLIP, which used reference counting well [24], contained references that were never counted

[23, Fig. 3]. From a modern perspective, the READERS implemented a crude recursion stack of threads. Weizenbaum's convention worked because the extra references were necessarily redundant, satisfying Invariant 1. Alternatively, we might view their destinations as "nailed down" by other pointers.

Section 5 describes how the recognition of thread–as–type was used implicitly in a C program to simplify storage management tremendously. Using a hardware device that manages the heap in real time, purely local transactions provided storage management directly from the mutator, without its processor accessing more RAM .

3.2 Dynamic Links and Threads

Singly-linked circular lists can be recovered by real-time reference counting when the enclosing link can be treated as a thread [9]. However, there must be a run-time tag on the pointer field to distinguish the enclosing thread from an ordinary link. (The tag can be manifested as a relative address if the list were otherwise stored in genetic order—at monotonically increasing addresses.)

If a doubly-linked list is circular, then the closing forward link must also be also treated as a thread—like the circular list's—but it must also be tagged; all counterpointers are threads and, so, need not be. The presence of header nodes [23, 12, §2.2.5] alters these conventions only slightly.

"Nailing down" a node is a dynamic trick that uses a static protocol. The term means that the programmer is certain that it is referenced at least once (by the "nail.") Precise reference counting on several intervening reference transactions becomes redundant, so long as all of them are abandoned before the distinguished reference is "pulled." In this context, the nail is perceived as the link, with all intervening references created as threads. No explicit tags are necessary because a static partitioning exists only for the span of code while the nail is "in."

There are several ways to thread a tree [12, §2.3.1–2]: for example, inorder [19] or level-order [12, p. 350] successors. Tarjan introduces several threadings to explain his palm trees [21]; his tree arcs are links, but his fronds, reverse fronds, and cross-links are all threads. Where these threads are overloaded in a field that could alternatively contain a link, they must be treated as dynamic.

Several LISP and SCHEME implementations provide *weak pointers*. These are pointers installed by the programmer as a convenience, flagged so not to be traversed by the garbage collector. In MACLISP these appeared first in un-garbage-collected arrays [16, pp. 79–80] whose content were threads, but with a caution instead of Invariant 1. Later incarnations [15, §2.2.2] had the garbage collector replace them with NIL (or equivalent). While this protocol enforces Invariant 1, it renders a pointer only mostly dead [10], because it causes work for the collector and because a NIL weak pointer now confuses the empty list with lost denotation that disappeared in an intervening collection.

3.3 More Examples are Solicited

The examples cited here are by no means intended to be complete. The authors seek other (especially classic) examples of extant use of link/thread typing, even if only implicit in design or validation of an algorithm. We conjecture that many programmers use these ideas subliminally, enabling them to manage heaps well without algorithmic storage management.

4 Programming Logic and Types

This section presents a programming logic in the form of preconditions for imperative programming that preserves Invariant 1, and anticipates a type theory for strongly typed languages.

Invariant 2 *Invariant 1 can be restated: if node m is accessible and $\langle m, l\rangle \mapsto n$ is a pointer, then either it is a link or n is accessible.*

This formulation of the invariant translates into four cases for imperative programming that are itemized below: resurrection and killing, and two cases for pointer assignment to a live/dead pointer variable or field.

4.1. Dynamic tag changes:

   (a) Resurrection is always safe. Enlivening cannot violate Invariant 2.

   (b) Killing is safe only if the pointer is not a unique reference. If the pointer were unique, killing it would render its destination inaccessible.

4.2. Pointer assignment $P \leftarrow Q$, is analyzed according to the cases for P before assignment. The delicate case occurs when P is not unique. The tag of Q is copied along with its pointer.

   (a) P a thread: Like resurrection, the assignment $P \leftarrow Q$ cannot violate Invariant 2.

   (b) P a link: If P is non-unique, then assignment is safe. If P is unique then the assignment is safe only if all nodes in P's structure that are uniquely accessible (that is, its prefix that is about to become inaccessible), absorb threads only from those same prefix nodes.

Points 4.1a and 4.2a are simple; losing a thread does not threaten the invariant. Point 4.1b is also easy to understand.

The interesting case is Point 4.2b, which has an important subtlety when Q is also a link and shares part of P's structure, as frequently happens while deleting nodes. In that case and at that point, the structure shared by both P and Q is not uniquely referenced, and so the critical nodes are those accessible from P but not Q (or any other link). This case is very useful, for instance, when deleting a node from the middle of a singly linked list. (*v.i.* Figure 5.)

On entering a new block that declares a local pointer variable, for the purposes of this analysis the uninitialized pointer should be treated as a thread. Similarly, any variable that is released on block termination can be treated as if NIL were assigned to it there. That is, it loses its role as a root. A particularly interesting case arises when unique pointers are returned as values from a function to the calling environment.

Formal typing depends on our ability to identify unique references automatically, which work is in progress. The live/dead domain, T, has only two points; "live" is $\bot$, "dead" is $\top$. If any pointer declaration in a program can be validated as $\top$ and maintain the invariants, then reference counters can ignore it and garbage collectors need never traverse it.

A type checker needs help getting started, because no explicit constants, either atoms or functions, locally distinguish $\perp$ from $\top$. Either the programmer must identify links as seeds, or an eager compiler might optimistically conjecture the threads. (Fat chance!) In either case, the checker must validate these invariants.

5 Experience

The perspective above is a result of experience with a design for hardware that was built [28] (in prototype without initialization of its live/dead tag) and subsequently used in serendipitous ways that were not anticipated at its design [26]. The short story is that tagging of threads proved *most* useful at both development time and at run time; code written with this perspective was more succinct and reliable, and would (with the designed hardware tagging) run faster.

5.1 Reference-Counting Hardware

The hardware, briefly described below, understands and acts on dynamic live/dead tags at run time. That experience led to our vision of static link/thread typing at compile time.

The reference-counting memory (RCM) was designed in 1984 [26] and built in 1989 [28] as an experiment in rapid prototyping, in digital design derivation, and in memory support for multiprocessing. The uniprocessing hardware currently supports an elementary SCHEME compiler and direct manipulation through languages like C or C++. The latter programming style is reported here.

At its core, RCM is a heap of 8-byte nodes, each of which is either a homogeneously atomic datum (like a floating-point number) or two 4-byte pointer fields. A write to the former, atomic type does not invoke reference counting, but writing to the latter, binary node does. (Any pointer cache must be treated as write-through.) When a node is allocate, by reading its address from either of two distinguished addresses, a hidden tag on the node is set, distinguishing between these two types.

Each 4-byte field also carries a hidden, hard-wired live/dead tag from T in hardware, associated with its current content. That tag is both used and reset as each field is written. The tag on the old content, being destroyed, determines whether a decrement must be dispatched to that address (if it were a link). The type of the node (if it is a binary node) determines whether the field is to be tagged as live and, inseparably, whether an increment is dispatched to that address.

The three tags, just described, ride with a node throughout its lifetime and back through AVAILable space as the node is recycled. Although they reside at the address of the node, they do not consume the address space usually associated with main memory; "hidden" memory contains the tags and the reference counts at the same address. (This illusion contrasts with Baker's assertion [3] that reference counting consumed both address space and processor cycles.) Although the type is reset at allocation, the content and live/dead tags remain meaningful until a write instruction changes both them and the visible content of the node.

That is, when a binary node is first allocated to receive two links, it may yet contain dead bit patterns from its former incarnation (*e.g.* as an atom.) Contrariwise, when an atom is first allocated, it might yet contain live, counted references to archaic structure that only becomes collectible as its content is overwritten, as those live references dispatch decrements, and as their reference counts reach zero. This is a hardwired revision of an algorithm due to Weizenbaum [23, p. 527].

RCM's design also contains an unimplemented provision that a pointer can be tagged (in its units bit that should be 0 with word addressing) as a thread. It that bit were 1 as a pointer were written into a binary node, not only would the usual increment be cancelled but also that field would be tagged as dead—as if it were in an atomic node. Intended to provide circular references [27], that protocol is simulated in this work by three write instructions that tell RCM

- to write the pointer as if it were live;
- to decrement the reference count of its content, just incremented;
- to reset the tag on that field to be dead, cancelling the future decrement.

Therefore, it now requires two more control instructions to simulate the single write, as designed for multiprocessing. Speculation that this work triples the timing, however, is inappropriate because the implementation (on a NU-bus with NEXT's controller chip) muddles such analyses [28, §2].

As mentioned, we discovered that this simulated instruction proved to be far more useful than merely to build simple circular lists. Indeed, our experience implementing a sample data base showed that it handled many kinds of cycles and reduced reference counts, even in acyclic structures.

5.2 Skippy-list example

As a demonstration of RCM and of the impact of the proper use of threads there, a very simple example was derived from the model of skip lists. Its main purpose here is to characterize the impact on storage management of distinguishing links from threads, and to lay a foundation for understanding the tables in the next subsection.

A skip list [18] is a search structure generalized from a sorted linear list, with additional pointers woven into it that allow its search algorithm to take long strides. As defined, it is an acyclic structure. An immediate observation here is that all the "additional pointers" should be statically typed as threads, so that all its linear links are unique. The exercise is to allocate a list of length 1000, 2000, 5000, or 10,000, and then to release it merely by overwriting the unique reference to its root.

A simple program was written in C using RCM that represents a node in a level-1 skip list as two RCM nodes. One RCM node links the spine of the skip list. The other RCM node contains the search key, and every-other one has an additional pointer. In order to generate a cyclic structure, however, that additional pointer is corrupted to point to a *random* sibling. We call it a *skippy list* here to suggest that it is so very like a skip list. A skippy list is very simple, highly circular, and fairly useless.

Four tests were run, three times each, and the average counts are presented in Table 1. Details of the tests are described below. The columns are headed by the length of the skippy list, with the last column extrapolating a rough ratio of the counts in that row to the length of the list. Since each node in the skippy list is represented by two RCM nodes, each with two fields, two reads and four writes are required just to allocate and initially fill it.

Test	Count	1k nodes	2k nodes	5k nodes	10k nodes	Ratio
Real RCM	Reads	9499	18999	47499	94999	4.75
	Writes	12001	24001	60001	120001	6
Ideal RCM	Reads	9499	18999	47499	94999	4.75
	Writes	7500	15000	37500	75000	3.75
No threads	Reads	12998	25998	64998	129998	6.5
	Writes	10495	20999	52500	104999	5.5
No cleaning	Reads	9499	18999	47499	94999	4.75
	Writes	7500	15000	37500	75000	3.75
	GC nodes	1988	3996	9999	19997	2

Table 1. Counts of memory hits for a skippy-list example.

Four tests compare performance under different management modes. The first is RCM as built, which requires extra writes to reset the dead tag and count on any thread; also, an extra RCM write is required to nail successively newer roots to the skippy list. The second set of counts is identical, but adjusts for those resets, which ought to have been done while writing the pointer. The third test does not use threading as a node is allocated, but must traverse the skippy list to reset them later—as the list is released. The last test neglects this traversal, allowing the highly cyclic list to confound the reference-count machinery and leaving almost all nodes to C's garbage collector (:-).

The results show that we really should have built threading initialization into the RCM hardware. The cost to reset the dead tags is visible, two writes per node, but not as bad as the traversal to erase the cycles at release, which costs 70% more probing. And, finally, a mutator runs faster when it can completely ignore storage management, provided here by RCM.

5.3 Relational Object Database

The Relational Object Database (ROD) is a system for creating, manipulating, and searching objects and relations between objects. It was originally conceptualized for checking dependencies between program constructs. For example, a function f can be related in ROD to another function g: "f uses g." Once it is known that "uses" has an inverse and stated to the database that "f uses g," ROD infers the inverse relationship of "uses" and also makes it a known fact that "g is-used-by f." ROD infers such symmetry and answers questions by searching its database, a dynamic structure like any typical database.

ROD was originally written in C, using linked lists and arrays for data structures, at New Mexico State University in Fall 1993 and used there in later semesters [14], In Spring 1995, ROD was converted to run using the Reference-Counting Memory at Indiana University, as an experiment to excise explicit memory management from the ROD system, replacing it with RCM's hardware support. In a sense this was to be a test of the generality of RCM because ROD had been designed independently of it.

The basic List structures of ROD follow:

5.3.1. A global doubly-linked list of *types*. Every object is classified as exactly one type, such as function or procedure.

5.3.2. A global doubly-linked list of *relations* containing the name and a reference to its inverse relation (in this list) if an inverse is known.

5.3.3. A global doubly-linked list of *objects* that are being manipulated, each containing the name of the object, a pointer to its type (in List 5.3.1), and a doubly-linked list described as List 5.3.4. Each object occupies two RCM nodes in the tests, but the space is static.

5.3.4. A doubly-linked list hung off each object in List 5.3.3 representing *relationships* to other objects contained in List 5.3.3. Each contains a pointer to a relation (in List 5.3.2) and a pointer to the object-being-related-to (in List 5.3.3). If appropriate, it also contains a pointer to the corresponding inverse relationship in the list of relationships that is hung off the object-being-related-to (in List 5.3.3). Each relationship occupies three RCM nodes in the tests, forty relationships per object, and this space is recycled.

Figures 1–6 show an example that includes the last two of these structures.

- The first discovery in this exercise was that doubly linked lists are, indeed, easy for RCM to manage if all counterpointers are threads. This convention is illustrated by the solid pointers paired with inverted, dashed pointers in Figure 1. The convention is used in implementing all four kinds of lists. As illustrated in Figure 1, this casts any list as singly-linked with respect to links, and Lists 5.3.3 and 5.3.4, of objects and their relationships, become a simple tree (ignoring, for a moment, the pointers to inverse relationships.)

- A second insight is that a queue (in this case the extant queue of objects, doubly linked as above) should also have its REAR pointer as a thread. This convention suffices to reduce a redundant reference counts, but is not necessary to recover a cycle. It was a clue that the live/dead tag could be useful beyond circular structures. As sketched in Figure 1, it makes the queue everywhere uniquely referenced.

- The interesting discovery is that *both* pointers to inverse relationships, which must necessarily be symmetric, should also be threads; their manipulation is explained in some detail below. That is, the pair of relationships, that is a single relationship and its inverse hanging from two different objects, contain a trivial cycle that need not be counted. If these two pointers were live then the RCM exercise would have failed to recover just those nodes. This last step, moreover, completes the picture of the object list as a tree (with respect to live nodes.) It makes it simple, for instance, to destroy all objects *and* the relationships on them in one cycle.

The most interesting operation on the ROD data structures, deletion of an element from the list of relationships (List 5.3.4), is now reviewed as an illustrative example of how the system works, and of how it uses dynamic live/dead tags. While deleting a relationship node, its inverse relationship—if there is one—must be deleted at the same time. Using live and threads, the relationship structures can be arranged so that all the nodes can be deleted as a chain reaction with the deletion of one node. Figures 1–6 show the sequence of event in a node deletion.

Figure 1.: Solid arrows are links and the dotted arrows are threads. The pointer O refers to the structure containing both the HEAD and TAIL of the *object* queue [25]. The nodes S_1 and S_3 are relationships that are inverses of each other, relating objects O_1 and O_3. That is, "O_1 uses O_3" and "O_3 is-used-by O_1." P is a reference to the relationship node to be deleted, S_1. The only node in Figure 1 with a reference count greater than one is S_1; P emits its second reference.

Figure 2.: S_1 is removed from the relationship list hanging from O_1. [Points 4.2a, and 4.2b without uniquely accessible nodes.] Its reference count drops to one. The reference count on S_2 increases because it is now reference from both S_1 and O_1.

Figure 3.: The pointer from S_1 to S_3 is resurrected. [Point 4.1a.] S_3's reference count rises to two.

Figure 4.: S_3 is removed from the relationship list hanging from O_3, and its reference count drops to one. [Points 4.2a, and 4.2b without uniquely accessible nodes.] The reference count on S_4 increases because it is now referenced from S_3 and O_3. P now points to a linked structure whose uniquely referenced prefix, up to the sharing at S_4, comprises the nodes to be removed.

Figure 5.: The reference from P is lost. [as if $P \leftarrow$ NIL; Point 4.2b without any incoming threads.] The reference count on S_1 automatically drops to zero and can be recycled. When—in due course—S_1 is overwritten, the reference count on S_3 will also drop to zero and it will be recycled. No further traversal is necessary on RCM.

Figure 6.: The ROD data structures after deletion. All reference counts have decremented to reflect accessible references.

The following test was run to exercise RCM and to reveal the impact of proper threading. The data structures above were build up serially for 500, 1000, 2000, and 2500 objects, each participating in 40 relationships with its peers. Then the relationships were removed in a random order, ideally releasing all their space.

The order that the structure is built is immaterial, since new nodes are inserted at the front of List 5.3.4. Removing them in random order causes that list to be traversed repeatedly, searching for the relationship to be removed. So the exercise becomes read-intensive.

The results appear in Table 2, read like Table 1. Again, the experiments were run three times and the ratios are remarkably stable. There are far more reads than writes, but writes will have cost more because of their dirty-bit and write-through requirements.

Installing hardware support for writing threads to RCM would save about six writes per node, arising from the three threads there; this is about 5.5% of the memory cycles. Alternatively, the relationships can be traversed and returned by *ad hoc* unthreading code ("no threads"), but more reads and one more write to each node are needed; in fact, the cost without threads is quite nominal because RCM only returns a very few nodes at a time here. Finally, the test, run without returning any nodes at all, obtains the timings for the idealized RCM, but leaves behind all the space for garbage collection.

6 Conclusions

The point of this paper is to identify the domain T of links/threads as an attribute or type to be used by the programmer or the compiler to improve the efficiency and efficacy of run-time storage management. An important symbiosis exists between static typing by T, and identification of unique references. Killing pointers will decrease the aggregate reference count in a system, making it easier to discover uniquely referenced nodes. On the other hand, accurate identification of unique references may help to maintain Invariant 1, through more effective elimination of inaccessible pointers from the verification problem.

The experiments show that run-time use of threads, both by hardware and by the programmer, improves storage management. The number compares favorably with the best collectors without requiring extra memory for recopying. The resulting savings can be significant even for C programs. Integrating threads as a static type at compile-time remains as future work.

Uniquely referenced nodes should be identified at compile time by a type system or at run time by tagging. Then they can be reused through purely local transactions (while resident in cache) or recycled with less resources, even, than idealized collection [1]. Baker offers a good example of this behavior [4]. Distinguishing threads from links has the effect of reducing reference counts, making unique references more frequent and more frequently useful.

Generation-scavenging, however, will suffer from violations on the genetic order of addresses that arise from this sort of *in-situ* reuse. When measuring the locality of generation scavenging, its advocates should better address locality lost in the mutator whenever it avoids reuse in order to support the collector. In contrast, multiprocessor architectures demands such reuse: both to preserve locality in the mutators and to avoid synchronizations among the collectors. We offer static threading and compile-time space analysis as tools necessary to realize the dream that Functional Programming will yet become a *lingua franca* for parallel processing.

The original purpose of this work was an exercise to test the generality of the hardwired implementation of reference-counting memory. In extending this device to handle the typically circular structures from data-base systems, we discovered that the distinction of link/thread was far more useful, and far more important than was realized when it was designed—and not built into our prototype. And RCM's effectiveness, as described in Section 5.1, was considerably improved by enforcing the invariants, above. Since it is likely that low-level programmers now subliminally use such a type to manage storage manually, it becomes important for automated managers (even garbage collectors) that would strive

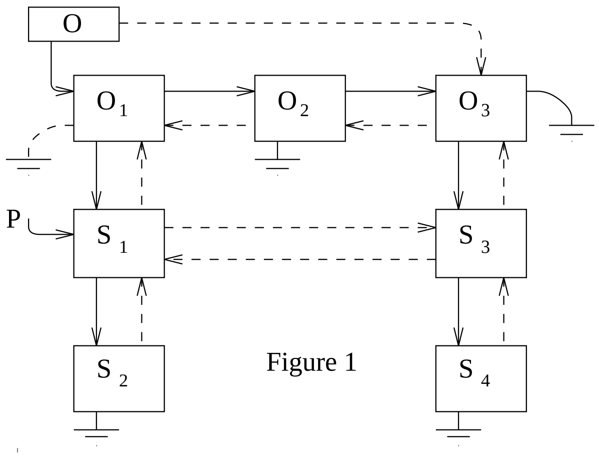

Figure 1

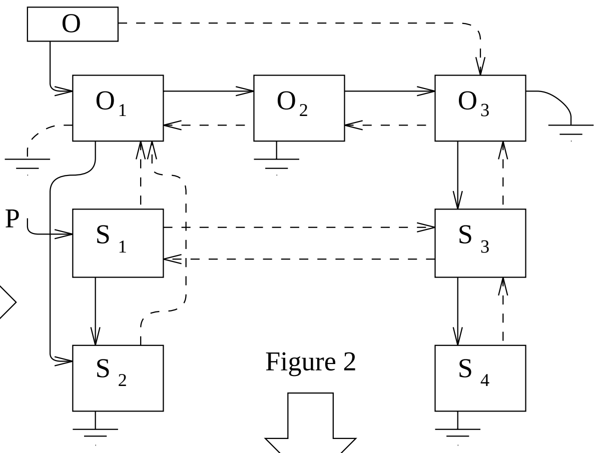

Figure 2

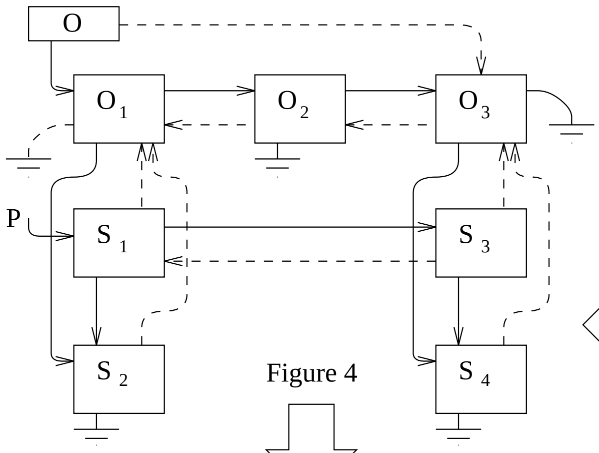

Figure 4

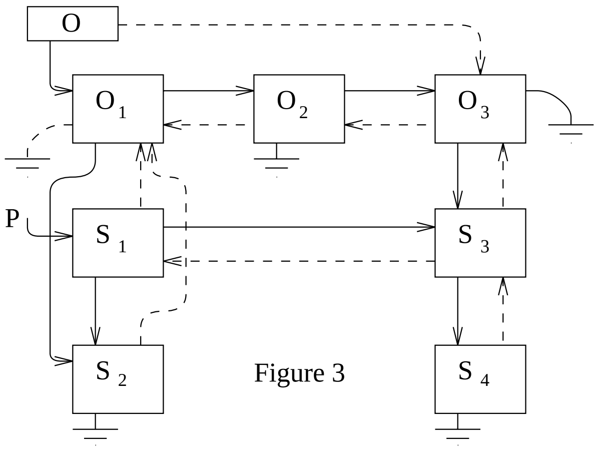

Figure 3

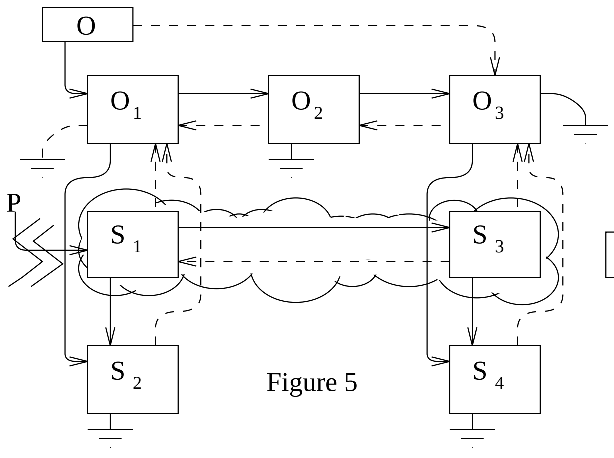

Figure 5

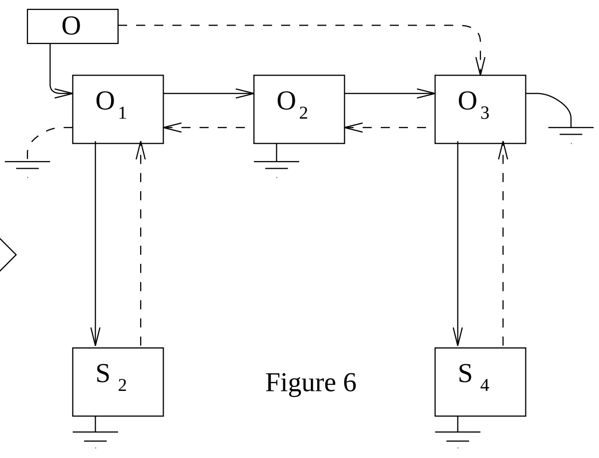

Figure 6

Number of objects		500	1000	2000	2500	
Number of relationships		20,000	40,000	80,000	100,000	Ratio
Real RCM	Reads	2,051,511	4,105,761	8,214,261	10,268,511	102.68
	Writes	329,131	658,381	1,316,881	1,646,131	16.46
Ideal RCM	Reads	2,051,511	4,105,761	8,214,261	10,268,511	102.68
	Writes	209,711	419,461	838,961	1,048,711	10.48
No threads	Reads	2,120,982	4,244,982	8,492,982	10,616,982	106.17
	Writes	229,382	458,882	917,882	1,147,382	11.47
No cleaning	Reads	2,051,511	4,105,761	8,214,261	10,268,511	102.68
	Writes	209,711	419,461	838,961	1,048,711	10.48
	GC nodes	60,000	120,000	240,000	300,000	3

Table 2. Counts of memory hits for ROD.

for high performance to recognize it and to use it well.

7 Acknowledgements

We thank Henry Baker for critical help in tracking down origins of the term "weak pointer," and referees for suggesting clarification between static and dynamic threads. High recognition for Caleb Hess and Willie Hunt who build the RCM prototype!

References

[1] A. W. Appel. Garbage collection can be faster than stack allocation. *Inform. Proc. Ltrs.* **25**, 4 (June 1987), 275–279.

[2] A. W. Appel & Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *J. Funct. Programming* (to appear).

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

[4] H. G. Baker, Jr. Lively linear LISP—'Look Ma, no garbage.' *ACM SIGPLAN Notices* **27**, 8 (August 1992), 89-98.

[5] D. W. Clark and C. C. Green. A note on shared structure in LISP. *Inform. Proc. Ltrs.* **7**, 6 (October 1978), 312–314.

[6] J. Cohen. Garbage collection of linked data structures. *Comput. Surveys* **13**, 3 (September 1981), 341–367.

[7] G. E. Collins. A method for overlapping and erasure of lists. *Comm. ACM* **3**, 12 (December 1960), 655–657.

[8] L. P. Deutsch & D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Comm. ACM* **19**, 9 (September 1976), 522–526.

[9] D. P. Friedman and D. S. Wise. Reference counting can manage the circular environments of mutual recursion. *Inform. Proc. Ltrs.* **8**, 1 (January 1979), 41–44.

[10] W. Goldman. *The Princess Bride,* (screenplay). Nelson Entertainment & Twentieth-Century Fox Film Corp. (1987), Miracle Max scene.

[11] E. Horowitz & S. Sahni. *Fundamentals of Data Structures in* PASCAL, Rockville, Md, Computer Science Press (1983).

[12] D. E. Knuth. *The Art of Computer Programming,* **I,** *Fundamental Algorithms* (2nd ed.), Reading, MA, Addison–Wesley, (1973.)

[13] H. R. Lewis & L. Dennenberg. *Data Structures & their Algorithms,* New York, HarperCollins (1991).

[14] D. Liles, P. Mamnami, R. Sinclair, J. Walgenbach, & S. Williams. ROD User's Guide. Class notes for Software Development, Computer Science Dept., New Mexico State Univ. (Spring 1994).

[15] J. S. Miller. *MultiScheme: a Parallel Processing System Based on MIT Scheme,* Ph.D. dissertation, Mass. Institute of Tech. (1987).

[16] Moon, David A. *MACLISP Reference Manual*, Project MAC, Mass. Institute of Tech. (April 1974).

[17] Y. Park & B. Goldberg. Static analysis for optimizing reference counting. *Info. Proc. Lett.* **55**, 4 (August 1995), 229–234.

[18] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. ACM* **33**, 6 (June 1990), 668–676.

[19] A. J. Perlis & C. Thornton. Symbol manipulation by threaded lists. *Comm. ACM* **3**, 4 (April 1960), 195–204.

[20] T. Standish. *Data Structure Techniques,* Reading, MA, Addison–Wesley ((1980).

[21] R. Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.* **3**, 1 (March 1974), 62–89.

[22] D. N. Turner & P. Wadler. Once upon a type. *Conference on Functional Programming and Computer Architecture.* New York, ACM Press (1995), 1–11.

[23] J. Weizenbaum. Symmetric list processor. *Comm. ACM* **6**, 9 (September 1963), 524–544.

[24] J. Weizenbaum. More on the reference counter method of erasing list structures. *Comm. ACM* **7**, 1 (January 1964), 38.

[25] D. S. Wise. Referencing lists by an edge. *Comm. ACM* **19**, 6 (June 1976), 338–342.

[26] D. S. Wise. Design for a multiprocessing heap with on-board reference counting. In J.–P. Jouannaud (ed.), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201,** Berlin, Springer (1985), 289–304.

[27] D. S. Wise. Stop-and-copy and one-bit reference counting. *Inform. Proc. Ltrs.* **46**, 5 (July 1993), 243–249.

[28] D. S. Wise, B. Heck, C. Hess, W. Hunt, and E. Ost. Uniprocessor performance of reference-counting hardware heap. Technical Report 401, Computer Science Dept., Indiana Univ. (June 1994).