One-bit Counts between Unique and Sticky*
Technical Report 516
July 1998

David J. Roth and David S. Wise
Indiana University Bloomington, Indiana 47405–4101 USA
Email: `droth, dswise@cs.indiana.edu`

Abstract

Stoye's one-bit reference tagging scheme can be extended to local counts of two or more via two strategies. The first, suited to pure register transactions, is a cache of referents to two shared references. The analog of Deutsch's and Bobrow's multiple-reference table, this cache is sufficient to manage small counts across successive assignment statements. Thus, accurate reference counts above one can be tracked for short intervals, like those bridging one function's environment to its successor's.

The second, motivated by runtime stacks that duplicate references, avoids counting any references from the stack. It requires a local pointer-inversion protocol in the mutator, but one still local to the referent and the stack frame. Thus, an accurate reference count of one can be maintained regardless of references from the recursion stack.

CCS categories and Subject Descriptors:
D.4.2 [**Storage Management**]: Allocation/Deallocation strategies;
E.2 [**Data Storage Representations**]: Linked representations.
General Term: Algorithms.
Additional Key Words and Phrases: multiple reference bit, MRB, garbage collection, storage management.

1 Introduction.

While garbage collection is a certain and often (unto itself) a local strategy for storage management, it can detract from the local behavior of the mutator by moving data apart. In contrast, simple recycling by the mutator often sustains greater data locality, by providing nearly in-place reuse. That is, off-line collection that scatters data at its convenience can hobble the mutator simply by undoing the locality of data affinities. Especially in layered memories, inexpensive but local recycling can improve the overall performance of the mutator.

Garbage recycling is here distinguished from *garbage collection.* The classic meaning of the latter term is wholesale assembly of garbage into a new available-space pool,[1] to be reused over time in future. The friendly metaphor of ecological recycling implies a more local reuse of freed space, enhancing locality. Locality of reference is important to speed cached and paged memories [8], so compact structure and its reuse *in situ* is desirable.

A familiar example of compaction from recycling is use of a sequential stack of frames for implementing nested function calls; popping the stack recycles contiguous space for immediate reuse as environments terminate whenever a function returns. In contrast, a purely linked stack—albeit convenient for returning closures of higher-order functions—might accumulate lots of garbage for later collection, and could scatter references to nested environments throughout memory. However, foreknowledge that a particular environment is uniquely referenced allows it to be reused immediately or recycled presently.

A motivating example is the handling of those frames in tail recursion. Tail recursion occurs when the return value of a function results from a recursive call to the function, itself; in that case, the frame can be reused in place, rather than stacked. It is essential to a programming style in which several kinds of `while`-loops are better expressed as tail recursions. Applying the techniques in this paper to frames in JAVA, for instance, could reveal that a returning environment is uniquely referenced and, therefore, could safely be reused. This paper shows how purely local register transactions could be used to reveal to a JAVA run-time engine that such a reuse is safe.

2 One register for available space.

Available space is commonly structured either as a linked list of free nodes or as a compacted region of contiguous memory. Sustaining such a nonlocal structure, the usual result of a garbage collection, generates too much memory traffic for recycling. If locality is to be ensured, recycling must be implemented in registers and cache.

The algorithms below allow a processor to recycle only a few nodes at a time and they, also, must be tracked locally. One strategy is to dedicate one register, `avail`, to remembering the address of a single free node. Alternatively, that register might be an index to the top of a small, cache-resident stack of such addresses. If no nodes are *known* to be free, then it contains 0; if a node is to be allocated then, the node must be allocated from the usual memory

*Research reported herein was sponsored, in part, by the National Science Foundation under a grant numbered CCR 9711269.

©1998 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or `permissions@acm.org`.

To appear in *Proc. 1998 Intl. Symp. on Memory Management.*

[1] We use Knuth's original terminology that distinguishes reference counting from garbage collection [16, p. 413]. Cohen [6] later merged both concepts under the latter term, and the generalization is now widespread [25, 14].

pool. If a node is released when this pool is already full then some node must be abandoned to some following garbage collection.

```
const int availSize = 16;
register int availCount=0;
Object* knownFree[availSize];
Object* p;

/*Code snippet for    "p = new Object()" */
p = (availCount>0) ? knownFree[--availCount]
                   : new Object();

/*Code snippet for    "delete(p)" */
knownFree[availCount] = p;
if (availCount < availSize) availCount++;
```

Such snippets will be implied by `new` and `delete` below.

Thus, either releasing a node for recycling, or reallocating a recycled node requires no main-memory references at all; either operation is local to the register and cache. Finally, it is possible to create more than one `avail` structure—say, one for each type of high-traffic object.

3 Storage management.

The focus of attention in reference-counting storage management [7] is on nodes that are uniquely referenced. Those are the ones that can soon be reallocated, and Clark and Green [5] showed that there are lots of them.

If the compile- or run-time system can detect the release of those last references in real time, then those nodes can be recycled on the spot. And—important in real-time systems—such notice results from purely local information, and also sustains locality by allowing for reuse in the same context. Applications that are real-time or asynchronously parallel need the locality available from on-line reference counting.

In hybrid systems garbage collection remains important to the algorithms below, where reference counting and garbage collection complement each other in several ways. Garbage collection is used behind reference counting after cyclic structures [19, 26, 28] leak away space, or after too-small counters get stuck [5, 9]. Similarly, any counting that postpones garbage collection at the cost of a few local transactions makes collection more suitable to real-time and parallel performance; it improves the amortized cost of memory-cycles-per-allocated-node by increasing the number of nodes reallocated without additional memory cycles and by reusing memory locally.

4 Reference assignment.

Two sorts of reference assignment are common in any system, and might be distinguished here. The first is a temporary assignment to ephemeral pointers, whose contents are unimportant to the integrity of any data structures. They constitute another type of reference—one that the programmer explicitly declares to be redundant and, therefore, exempt from the attention of the storage manager. These can be recognized as Thornton's threads [22, 16, 29] and have also been called "special" or "dead" pointers.[2] Inamura *et al.* [13] use such dead links as the second reference necessary to logic programming. Assignment to the user's dead pointers is not treated further, but they are used by our compiler, described in Section 9.

The second type of assignment is to pointer variables, "live" links that hold data structures together and that, therefore, must be counted. This exposition proceeds in the context of reference counts of limited range. For simplicity, consider a nontrivial range (say `1..255`) with a `stickyCount` (say, `256`) where counts stick, thereafter to be neither incremented nor decremented by on-line counting. A referent with a stuck count is "nailed down" because it is permanent, with respect to reference counting anyway. The `stickyCount` value can be viewed as $\top$ of a flat, semantic domain of reference counts; it means that the reference counting scheme has broken down on this particular node; the number of references here might be anything, including zero or one.

In low-level languages like C, pointer assignment, `p=q`, is a simple integer assignment. In C++, however, overloading the assignment operator for references to objects gets complicated [10]; one must deal with the former referent of `p` as it is dereferenced, as well as the either new or newly shared referent, `q`.

Focusing only on reference counting, C's pointer assignment is usually expanded to something like

```
Object *p, *q;

/* snippet for   "p=q;" */
if      (q->refct != stickyCount) (q->refct)++;
if (p->refct == 1) delete(p);
else if (p->refct != stickyCount) (p->refct)--;
p=q;
```

These three steps proceed in just this order to allow evaluation of values for `q` that depend on the dying `p`, and to correctly handle assignments whose semantics is "p=p" where node `p` is uniquely referenced [17, p. 372 *e.g.*]. That is the count of greatest interest, and it is the focus of this paper.

The right side of the assignment, `q` below, has already been evaluated. Since the addresses, `&p` and `&q`, must already be in the processor, and since we will be focused on very low reference counts, we can prefix a test for `(&p == &q)` at little cost, and then decrement before the increment.

```
/* snippet for   "p=q;" */
if (&p == &q) /*Cancel the assignment*/ ;
else {
  if (p->refct == 1) delete(p);
  else
    if (p->refct != stickyCount) (p->refct)--;
  p=q;
  if   (q->refct != stickyCount) (q->refct)++;
}
```

This form's awkward, intermediate state has a count lower than normal, rather than higher than normal as before. With constrained counts this choice is more desirable because it prevents counts from sticking prematurely.

In the case that `q` becomes a brand new referent, born with reference count already initialized to 1, the first conditions should be omitted. Moreover, if `p` is a UNIQUE reference then that node can be reused in place rather than deleting and reallocating it.

```
/* snippet for   "p = new Object();" */
if (p->refct != 1){
  if (p->refct != stickyCount) (p->refct)--;
  p = new Object();
}
```

The whole point of counting is to reuse space *in situ* like this! Other references inside the reused node will be handled later as they, themselves, are subsequently initialized/assigned [24, 28].

5 One-bit reference counting.

The smallest possible reference count is a single bit [27, 20], also called the multiple-reference bit (MRB) [4, 13]. Of course,

[2] They form a proper subset of "weak" pointers which are not necessarily redundant to live ones and which, if used, demand special attention from garbage collectors.

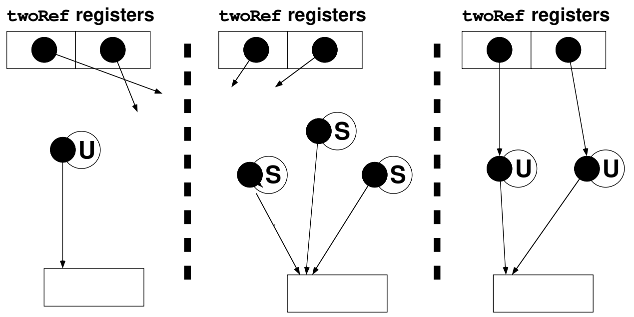

Figure 1: One-bit tag for reference counting: UNIQUE, STICKY, and cached count of two.

such a count distinguishes between only two values: UNIQUE and STICKY, or 1 and $\top$.

The original proposal for 1-bit reference counts [27] located the count at the referenced node. A much better strategy [20] is to place this bit as a tag on each reference to the node—illustrated in the first two sketches of Figure 1— rather than at the node, itself. Although this needs another MRB bit in every reference, it saves the random fetch whenever that count is inspected. The reference count on any node (*e.g.* referenced by `p` above) is known as soon as it becomes *accessible*, allowing it to be recycled without even touching it.

Moreover, a recopying garbage collector [26] can easily correct STICKY tags back to UNIQUE. The idea is that each newly "marked" node remembers the source of its first (original) reference along with its UNIQUE forwarding address. Any second reference updates the forwarding address to be STICKY and also overwrites it at the original source.

The symbiosis between collection and reference counting is significantly enhanced here. With such low mutator overhead, immediate reuse of released nodes becomes practical, postponing garbage collection. Contrariwise, with so low a STICKY count the restoration of minimal UNIQUE counts during collection has even greater impact, amplifying collector effectiveness by enabling further recovery by the mutator sometime in the future.

6 Reference assignment with one-bit counting.

Storage recovery now occurs behind the scenes in the mutator, at a very low level in the run-time environment. For this reason, a C-like programming style was chosen for exposition in this paper. However tempting be the objects of C++ or JAVA, both languages carry an understood run-time burden (including a storage manager for the latter) that is not intended here.

The reference-count tag is positioned in Figure 2 as the low-order (units) bit in any reference. (With nodes at least two bytes long, this bit is not part of their addressing scheme, anyway.) The default 0 value of that bit is chosen as the unique count; since all counts are born unique and (we hope) most will remain so until death, their tags often remain 0 throughout their lifetimes. Moreover, the function `stickIt` in Figure 2 is idempotent. The definitions there establish the protocol; type casts are used to assure, for instance, that all references to `Null` are STICKY, so that `stickIt` is also idempotent on `Null`.[3]

[3]The reader can ignore the distinction between `null` and `Null` on the first reading. Alternatively, an implementor might reverse the 0/1 definitions of `Unique` and `Sticky`, to force `Null` to coincide with a local `null`. Then the MRB tag must be reset to UNIQUE for every `new` node.

```
const int Unique = 0;
const int Sticky = 1;
Object* Null = (Object*) ((long)null | (long)Sticky);

inline static void stickIt(Object** q){
  if (isUnique(*q))
    *q = (Object*) ((long)(*q) | Sticky);
}

inline static int  refCt (Object* p){
  return (int)p &1;
}

inline static Object* thePointer(Object* p){
   return (Object*) ( (long)p & ~(long)Sticky );
}
inline static int isUnique(Object* p){
   return (refCt(p)==Unique);
}
inline static int isSticky(Object* p){
   return (refCt(p)==Sticky);
}
```

Figure 2: Declarations to support tags as one-bit reference counters.

With a one-bit count the assignment statement statement `p=q` collapses. First, the `else` clause decrementing `(p->refct)` becomes irrelevant. Furthermore, the late conditional, "`if (q->refct != stickyCount)`," above, simplifies to "`if (q->refct == 1)`" because 1 is the only count that isn't STICKY. And

```
if ( isUnique(q) ) stickIt(&q);
```

reduces to an unconditional "`stickIt(&q);`"

```
/* snippet for   "p=q;" */
if (&p == &q) /*Cancel the assignment*/ ;
else {
  if (isUnique(p)) delete(p);
  stickIt(&q);
  p=q;
}
```

Compared to the memory traffic of the unembellished assignment, this snippet has one more access that reads `p` before overwriting it, and maybe another to stick `q`.

The code for "`p = new Object()`" also simplifies. The only necessary step is allocation when the old `p` is STICKY (or `Object` varies in size).

```
/* snippet for   "p=new Object();" */
if ( !isUnique(p) ) p = new Object();
```

As above, this assignment incurs an additional memory prefetch on `p`, but here it is likely to be offset by memory cycles saved in initializing the "new" object. Since a reallocated UNIQUE node, which is likely to be already cache-resident from recent use, is reused in place, we save the memory cycle to touch a fresh node away in remote memory. In some cases, that initialization may even be redundant and so elided.

It is left to the reader to translate a similar snippet in which a *parameterized* constructor is invoked on the "dying" node `p`, precluding `p`'s early release. A copying construction, for instance, would appear as "`p=new Object(p)`," and depends on Node `p` remaining intact to retrieve its contents. Frequently, as in such a copy, the assignment statement simply distributes over the extant instance variables of the UNIQUE `p`, and the construction may elide recopying of `p`'s instance variables, which arrive already initialized.

```
/* snippet for   "p=q;" */
if (&p != &q) {

  if (hasTwoCount(&p)) {        /*Decrement p's count if it is 2, leaving p    */
      twoRefA = twoRefB = null; /* uniquely referenced by removing from cache, */
  }
  else if (isUnique(p)) delete(p); /* but maybe p was unique. */

  if (isUnique(q)) {                /* q has reference count of either 1 or 2. */
      stickIt(twoRefA);             /*  In either case, the two-counts stick.  */
      stickIt(twoRefB);
      if (hasTwoCount(&q)){                       /*If q does have count of 2,  */
          twoRefA = twoRefB = null;               /*  then raise its count to 3;*/
      }
      else {                                      /*Otherwise, q has count of 1 */
          twoRefA = &q;                           /*  rising to 2               */
          twoRefB = &p;                           /*  with this assignment.     */
      }
  }                                   /*Otherwise q was, and remains, sticky. */

  p=q;
}
```

Figure 3: Code snippet for reference assignment p=q.

7 The problem of "two."

If UNIQUE references are never touched, then the snippets above suffice. In reality, however, structures are later manipulated by shared references. The only practical use that sustains purely UNIQUE counts is construction of pure trees, followed by uncounted sharing, followed by dereferencing. Aside from some clever ways to avoid counting the shared references, mentioned below, general application of this technique seems hobbled. The example of a tail-recursion, introduced in Section 1, is useful here; shared references to the unique referents of a frame will exist momentarily before it is overwritten in place. If UNIQUE counts can be sustained across such a frame copy, then its simplification to *in situ* frame update becomes tractable.

With the snippets above, however, it is impossible to extract arguments involved in a tail recursion *and* to sustain all their counts at UNIQUE. A frequent difficulty is that a count rises to two momentarily while the new environment is being built, and then can no longer fall back to UNIQUE as the current environment is immediately abandoned. The first result of this paper is an inexpensive way to handle counts that rise to two and then fall back to one *really soon*. Without such a short-term transition, one-bit reference counting is hobbled.

Blind use of snippets that count references will merely nail down any structure at the moment it is first shared. Of what practical use, then, are one-bit counts? There have many been many efforts at several levels to answer this question. A common way to avoid premature nailing is to share a node via a dead pointer.

Deutsch and Bobrow [9] use a run-time table to track only nodes with counts above one. Nodes with momentary counts of two pass through their Multiple Reference Table (MRT). The first proposal for one-bit reference counting [27] mentions a similar table, pointing to *nodes* whose count "ought to be two," but this doesn't work where the UNIQUE count is a remote tag on two different references.

Barth [2], Bloss [3], and others have focused on compile-time analysis to determine the point that storage can be released without run-time counting. Barth, in effect, moves Deutsch-Bobrow counting back to the compiler. Suzuki [21] generalized the assignment statement to handle multiple assignments and, in particular, the ubiquitous pointer rotations [17, pp. 8–9 *e.g.*] that change no reference counts, in aggregate.

Efforts to use linear logic [11] on this problem led to the introduction of monads [23]. This approach is, in effect, a constraint on programming style, and a bit more drastic than what is wanted [1], even in functional languages. CLEAN [18] is not so restrictive, but depends on the programmer for advice. Jones and Le Métayer [15] offer an abstract semantics to expose many instances of unique reference as a type.

Deutch's and Bobrow's remains the most representative of methodical strategies that work uniformly for all assignment statements. Most of the other techniques try to restrain the growth of counts unnecessarily. Theirs simply treats UNIQUE as the most frequently occurring count, and invests extra effort in tracking counts that rise above one, in the hope that many will soon either fall back to UNIQUE or drift away STICKY.

8 The two-register reference cache.

The solution to sustaining a *few* accurate counts of two is to dedicate a pair of fast registers toward monitoring the two different references to a shared node, as illustrated in the right of Figure 1. Call such registers, collectively, the *cache* of references. Any cached reference is still tagged UNIQUE, but its reference from the cache overrides the tag, indicating that it really has a reference count of two.

In the following declaration, the two references in such a cache are labeled `twoRefA` and `twoRefB`. They should be implemented in a processor's registers, in primary cache.

```
register Object **twoRefA=null, **twoRefB=null;

inline static int hasTwoCount(Object** r){
  if (r==twoRefA) return true;
  if (r==twoRefB) return true;
  return false;
}
```

p=q where p is / q is	UNIQUE	Two count	STICKY
UNIQUE	p released; cache contents STICKY; p, q now 2 in cache.	p released. q nailed to 3 and cache flushed.	p released; cache untouched.
Two count	Old p released to UNIQUE; p, q to cache, count 2.	p==q necessarily; No change.	p released to UNIQUE; cache emptied.
STICKY	Cache contents to STICKY; p, q now 2, in cache.	q nailed to 3 and cache emptied.	Cache untouched.

Table 1: The effects of the p=q snippet on the reference cache.

The snippet in Figure 3 gives the code for general implementation of the assignment statement p=q. Although lengthy, *the new operations are internal to processor and cache,* except for two possibly remote stickIts and inspecting p before overwriting it.

If the cache is not null then it points to two different references, both tagged UNIQUE that, in turn, refer to the same node; its reference count is, thereby, two.

The interesting case is that q was really UNIQUE and so is inserted with p into the cache. On the assumption that &p$\neq$&q (that the assignment statement is not a trivial one), Table 1 gives a summary of the effect of the assignment snippet.

As before, assignment of a reference to a newly allocated node is a special case. Now, however, &p may have to be deleted from the cache. Actual allocation of a new node only occurs when p is not UNIQUE before the assignment. Otherwise, Node *p is reused in place, with future advantage from its locality.

```
/* snippet for   "p=new Object();" */

switch (0) { default:
      /* C idiom for a block with a break.*/
  if ( hasTwoCount(&p) ) twoRefA=twoRefB=null;
  else if ( isUnique(p) ) break;
  p = new Object();
}
```

9 Freeing the stack from reference counting.

The discussion above suffices for languages that use registers as local variables and that use displays to access frames deeper in the stack. Another strategy uses the stack more liberally, duplicating—rather than accessing—those non-local variables that are not subject to side effects. Thereby, they create multiple references to what may be a UNIQUEly referenced node throughout nested function calls. For this case we develop here a protocol that *counts no references from the stack*; all stack references are treated as "dead."

Registers have no machine addresses; in contrast, elements in the stack not only have addresses, but also those addresses are implicitly tagged (by their continuity to the sequential memory containing the recursion stack. With a linked stack, they might be explicitly tagged.) This allows link inversions, creating a *knot* of indirection whenever a unique reference is being shared (repeatedly even) from the stack.

Our stack size is doubled for arguments that are references; each argument is represented by its *value*, and possibly a pointer to the memory address whence it came—its *source.* In a typed system, doubling of the whole stack can be avoided for everything but references to the counted type. In particular, immediate values (integers, booleans, characters, etc.) need not have a SOURCE field.

In our implementation, the bottom elements for every frame, the return pointer and a closure pointer, are already paired in this way. Figure 4 illustrates how the former is treated as a SOURCE and the latter is treated as a VALUE.

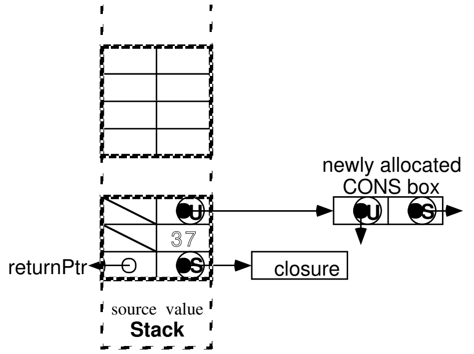

Figure 4: A frame on the stack, with return, closure, and two unshared arguments.

Our implementation was built over an implementation of a limited SCHEME that has little compile-time type checking; run-time tags determine what is and what is not a reference. Constrained by the simplicity of our compiler, we provided two stack entries for every parameter and every local variable. Figure 4 also illustrates that the SOURCE is often NIL, in this case for the immediate integer 37, and also for a newly allocated CONS box that is referenced nowhere else yet. (Figure 9 hints about how it receives its first memory reference.)

Frames are portions of the stack enclosed in heavier dashed boxes; the bottom of every frame contains the return pointer and the saved closure. Loading a stack frame is considered first, followed later by a description of its necessary unloading. The space above the top of the stack is, therefore, considered to have been "emptied" and all values are meaningless.

First, consider Figure 5. Two values are about to be loaded into the new frame, indicated by question marks. Immediate values are easily loaded; Figure 4 shows that their SOURCEs are empty. Similarly, loading a reference that is already STICKY (from Figure 5) generates a STICKY VALUE and an empty SOURCE in Figure 6.

Loading a UNIQUE reference is more delicate. The step from Figure 5 to Figure 7 illustrates how a *knot* is tied in the pointer structure. The source of that reference is stacked along with the UNIQUE reference, itself. The heap-resident pointer is, importantly, replaced with a referent back to this lowest entry in the stack where the UNIQUE pointer now "resides" close to the processor.

Any traversal that would use the original reference must be forwarded to this lowest stack entry to find it. More importantly, if any process would write into that heap location, then this forwarding of the former contents would provide for the NILling out of the

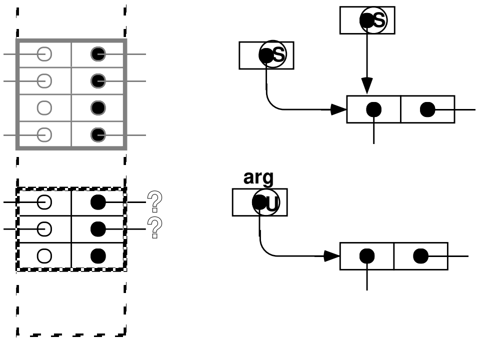

Figure 5: Ready to load a reference into a frame, the first time.

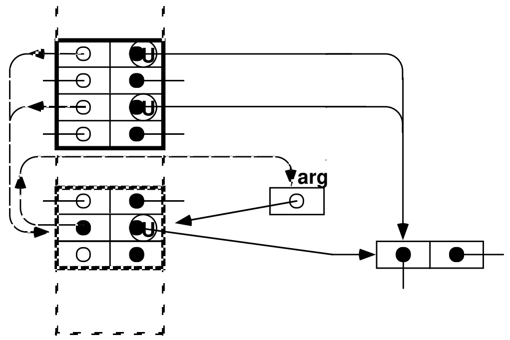

Figure 8: Copying that reference deeper into the stack.

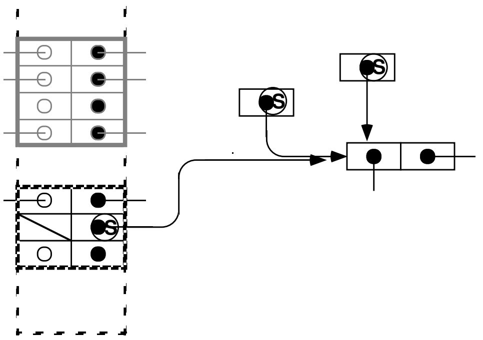

Figure 6: A sticky reference as a shared argument.

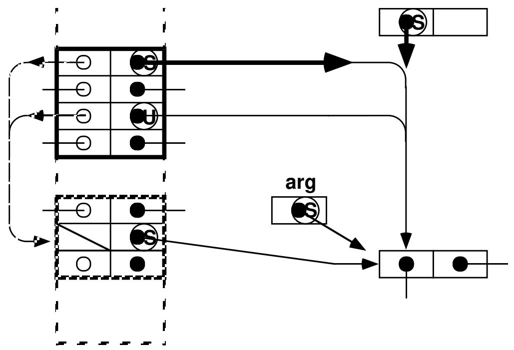

Figure 9: After storing the topmost, unique reference to memory.

SOURCE entry at this stack address, rendering the VALUE used, but not counted (*cf.* Figure 4).

Importantly, this UNIQUE reference might be passed to nested frames higher in the stack without additional reference counting. Figure 8 illustrates how these nested references copy the reference as is, but direct their SOURCE within the stack to the lowest, knotted entry where it first appeared. They may read and traverse that UNIQUE VALUE freely, but they may not store it into the heap without checking the status of the knot.

If the UNIQUE reference is ever written a second time to the heap (heavy pointers in Figure 9) then the knotted stack entry, as well as the original SOURCE reference, becomes STICKY. Moreover, the SOURCE there becomes NIL so that the referent now is indistinguishable from one that had been STICKY from the beginning.

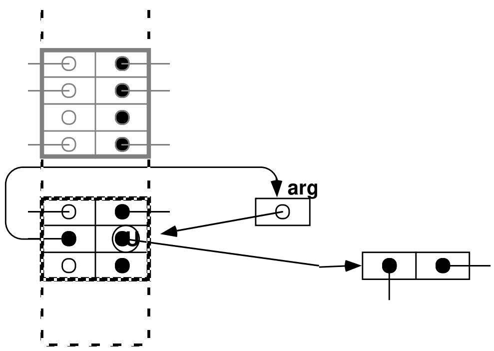

Figure 7: The unique reference loaded, and "knotted."

Alternatively, the nested references remain UNIQUE until their frames are popped. Still, every reference parameter must be inspected as the stack is popped to untie any remaining knots. As long as the SOURCE refers within the stack, nothing else need be done; no space is yet released. (Such a pop appears like a transition from Figure 8 to Figure 7.)

If a SOURCE field is NIL in a popped frame, then the associated reference is an honest one; if it is STICKY then it can be ignored, but if it is UNIQUE then it really is the *only* reference and that node can be recycled immediately. And if a SOURCE field isn't NIL and doesn't point down the stack, then the knot must be untied. (Such a pop would appear like a transition from Figure 8 to Figure 7.)

Tail calls have been implemented but must also follow the protocols just outlined; all popped frames must be inspected. A pseudo-frame can be created on top of the stack, and then copied down over a lower frame. However, the SOURCE pointers in the new frame must be checked to see if they point into the portion of the stack about to be popped, and if they do then the base (knot) structure must be elevated into that new frame. As this is done, the knotted, heap references into the stack must be redirected to the proper address within the pseudo-frame's destination lower in the stack. Then all intervening frames must be scanned for knots to untie or for UNIQUE structures to recover.

10 Experimental results

The protocol of Section 9 has been implemented an tested on a compiler for a pure subset of SCHEME. Without provision for con-

Test: 1-bit Test: Cheney	Pointers per Semispace	Number of collections	Pointers collected	Time in Mutator	Time in Collector
AVL (side effects)	162k	1	33,700	3.28	0.07
AVL (side effects)	162k	7	375,000	0.56	0.28
AVL (pure)	162k	19	1,384,000	4.73	0.74
AVL (pure)	162k	30	2,232,000	0.76	1.10
AVL (pure)	256k	8	1,278,000	4.68	0.31
AVL (pure)	256k	12	2,134,000	0.81	0.41
AVL (pure)	358k	5	1,366,000	4.78	0.22
AVL (pure)	358k	8	2,220,000	0.84	0.30
Quicksort	184k	13	511,000	1.90	3.07
Quicksort	184k	16	628,000	0.33	2.80

Table 2: Performance of the stacking protocol.

tinuations, all frames are stack resident. The target machine is a DEC Alpha, but the code became quite tangled and we do not consider the timings an accurate reflection of the technique. In particular, we attribute much of the remarkable slowdown in the mutator much to poor code that we know how to improve. Nevertheless, a sixfold mutator slowdown is difficult to accept.

Our implementation is only a proof of concept; it is hardly as efficient as it could be. But it does demonstrate how to avoid many shared references that might be introduced by the compiler. The doubling of the stack, because of its purely local use, does not seem to be a problem. The requirement to reread frames as they are popped is problematic.

Two codes for AVL-tree insertion and one for Quicksort were compiled and run, with and without runtime recycling using the one-bit reference counter. The AVL problem was to insert 16,000 nodes into an initially empty balanced binary-search tree. Only one recovered node could be remembered at a time; others were cast adrift (Section 2.) One version of the code was purely recursive; one used side effects to reuse tree structure already allocated. The Quicksort example is a purely recursive Quicksort of a list of 16,000 random integers.

Table 2 presents the results. All entries are paired, with the first entry using the one-bit reference count and the stack protocol of the previous section. The second entry is from a simple Cheney two-semispace collector. For the measures in this table, a "Pointer" is half of a CONS box.

With capacity for recovering only one node at a time, we nearly reproduce the 30%–40% recovery of others [20, 4, 13]. Certainly there are more nodes to be recovered via the limited queue of Section 2.

The collector has the additional burden of restoring accurate one-bit counts. Still, in the AVL tests, the real-time recovery more than compensated for the time *in* the slower collection. Even this payoff is not seen in the Quicksort example.

Still, the mutator slowdown is daunting, but these are only preliminary results. The current implementation scans every SOURCE field as the stack is popped; if the closure were type-specific many of those could be skipped.

11 Conclusions.

The snippets of Section 8 show how a methodical treatment of assignment statements with one-bit reference counts might handle temporarily higher counts without excessive memory traffic. The effective count rises to two and still falls to UNIQUE by using a "multiple-reference cache" of the two references, or by excising all counts from the recursion stack. With the references in registers or in hard, primary cache the processor should be hardly slowed.

The technique is a twist on Deutsch's and Bobrow's multiple-reference table, that holds a reference to each multiply referenced *node*. In contrast, it calls for a pointer to each of the multiple *references*, instead. These addresses will be already available from earlier or immediately recent assignment statements, and so should already be local to the processor.

These common examples demonstrate ample advantage from tracking only momentary counts of two from fast registers. It is useful not only for discovering that results of a function may be momentarily shared references to one of its arguments, but also for noticing that the structure of an activation record is a momentary sharing of one about to be released in a tail recursion.

So, this reference cache has been shown to handle transitional counts of two, sufficient for cascading assignments and for the motivating example of tail recursion, even with several arguments of different types. The cost on modern RISC processors is minimal: two registers for sharing UNIQUE references, plus one to track a locally available node. The new operations in a pointer assignment to `p` are local to three registers, except for a prefetch of uncached `p`, and `stickIts` on uncached references to nonUNIQUE nodes.

The techniques of Section 9 were developed because of the restrictions of the available test engine. The SCHEME compiler available for modification depended heavily on shadowing arguments with redundant, but frame-local, pointers. This was particularly troublesome for arguments static during tail calls, all of which were momentarily shadowed; they would not even be mentioned throughout a corresponding iteration. In an implementation without this restriction, the technique of Section 8 would have sufficed for any argument that changed. Such shadowing is important, however, to languages with first-class functions; that is, closures must shadow shared free variables. So we were doubly motivated to test the Section 9 ideas.

The technique for removing all counted references from the stack was successful in design but appears too expensive in practice unless much more can be done, perhaps in hardware, to relieve the mutator from stack operations to support the collector. The necessary operations are bounded in time, and will be local in cache if the stack remains compressed.

References

[1] H. G. Baker. Lively linear lisp—'Look Ma, no garbage!' *SIGPLAN Not.* **27**, 8 (August 1992), 89–98.

[2] J. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM* **20**, 7 (July 1977), 513–518.

[3] A. Bloss & P. Hudak. Variations on strictness analysis. *Conf. Rec. 1986 ACM Symp. on Lisp and Functional Programming*, 132–142.

[4] T. Chikayama & Y. Kimura. Multiple reference management in flat GHC. In J.–L. Lassez (ed.), *Logic Programming, Proc. 4th Intl. Conf.* **1**. Cambridge, MA: M.I.T. Press (1987), 276–293.

[5] D. W. Clark & C. C. Green. A note on shared list structure in LISP. *Inf. Process. Lett.* **7**, 6 (October 1978), 312–315.

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

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

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, & T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM* **39**, 11 (November 1996), 78–85.

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

[10] R. Gillam. The anatomy of the assignment operator. *C++ Report* **9**, 10 (November-December 1997), 15–23.

[11] J.–Y. Girard. Linear logic. *Theor. Comput. Sci.* **50** (1987), 1–102.

[12] D. Gries. An exercise in proving programs correct. *Commun. ACM* **20**, 12 (December 1977), 921–930.

[13] Y. Inamura, N. Ichiyoshi, K. Rokusawa, & K. Nakajima. Optimization techniques using the MRB and their evaluation on the Multi–PSI/V2. In E. L. Lusk & R. A. Overbeek, *Logic Programming, Proc. of North American Conf. 1989* **2**, Cambridge, MA: M.I.T. Press (1989), 907–921.

[14] R. Jones & R. Lins. *Garbage Collection.* Chichester: Wiley (1996).

[15] S. B. Jones & D. Le Métayer. Compile-time garbage collection by sharing analysis. *Functional Programming and Computer Architecture,* New York: ACM (1989) 54–74.

[16] D. E. Knuth *The Art of Computer Programming* **I,** *Fundamental Algorithms* (3rd ed.), Reading MA: Addison-Wesley (1997).

[17] H. R. Lewis & L. Dennenberg. *Data Structures & Their Algorithms.* New York: HarperCollins (1991).

[18] R. Plasmiejer & M. van Eekelen. *Functional Programming and Parallel Graph Rewriting,* Workingham, UK: Addison-Wesley (1993), §8.5.

[19] M. G. Sobel. *A Practical Guide to the UNIX System* (3rd ed.) Redwood City, CA: Benjamin/Cummings (1995), 609–611.

[20] W. R. Stoye, T. J. W. Clarke, & A. C. Norman. Some practical methods for rapid combinator reduction. *Conf. Rec. 1984 ACM Symp. on Lisp and Functional Programming*, 159–166.

[21] N. Suzuki. Analysis of pointer 'rotation.' *Commun. ACM* **25**, 5 (May 1982), 330–335.

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

[23] D. N. Turner & P. Wadler. Once upon a type. *Conf. Rec. of FPCA '95: Functional Programming Languages and Computer Architecture,* New York: ACM Press (1995), 1–11.

[24] J. Weizenbaum. Symmetric list processor. *Commun. ACM* **6**, 9 (September 1963), 524–554.

[25] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers & J. Cohen (eds.) *Memory Management, LNCS* **637**, Berlin: Springer (1992), 1–42.

[26] D. S. Wise. Stop-and-copy and one-bit reference counting. *Inf. Process. Lett.* **46**, 5 (July 1993), 243–249. Also ftp://ftp.cs.indiana.edu/pub/techreports/TR360.ps.Z

[27] D. S. Wise & D. P. Friedman. The one-bit reference count. *BIT* **17**, 3 (September 1977), 351–359.

[28] D. S. Wise, C. Hess, W. Hunt, & E. Ost. Research demonstration of a hardware reference-counting heap. *Lisp Symb. Comp.* **10**, 2 (July 1997), 159–181.

[29] D. S. Wise & J. Walgenbach. Static and dynamic partitioning of pointers as links and threads. *Proc. 1996 ACM SIGPLAN Intl. Conf. on Functional Programming, SIGPLAN Not.* **31**, 6 (June 1996), 42–49.