Design for a Multiprocessing Heap with On-board Reference Counting

by

David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 163 — Revised

Design for a Multiprocessing Heap
with On-board Reference Counting

by

David S. Wise

July, 1985

This material is based on work supported by the National Science Foundation under grant number DCR 84-05241.

To appear in *The Proceedings of the IFIP Int'l. Conference on Functional Programming Languages and Computer Architecture* in Nancy, France, September 16–19, 1985.

Design for a Multiprocessing Heap with On-board Reference Counting

David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405-4101 / USA

Section 0. Introduction

A project to design a pair of memory chips with a modicum of intelligence on board is described. Together, the two allow simple fabrication of a small memory bank, a heap of binary (LISP-like) nodes that offers the following features:

- 64-bit nodes;
- two pointer fields per node of up to 29 bits each;
- reference counts implicitly maintained on writes;
- 2 bits per node for marking (uncounted) circular references [9];
- 4 bits per node for conditional-store testing at the memory;
- provision for processor-driven, recounting garbage collection.

Many banks like this can be assembled to provide a large heap, which may share the same address space with conventional memory. This distinguished portion of memory responds quickly to additional commands, like "Give me the address of an available node (in your bank)," which are ignored by others.

Such memories may be used in uniprocessors or, more importantly, in multiprocessors where the locality of reference counting avoids the inter-processor synchronization necessary for garbage collection. This kind of architecture is envisioned by the Daisy project at Indiana University.

The remainder of this paper is organized in seven parts. The first discusses a particular multiprocessor environment that motivates this design. The second deals with the history of storage management, with emphasis on the relationship between garbage collection and reference counting. The third questions the course of that history, with its current emphasis on garbage collection. Then the major features and modes of operation of this memory are described. The fifth and sixth sections describe details of the Control Chip and of the Memory Chip. The last section reviews the design progress and offers some conclusions.

Section 1. Motivation

The work reported here is motivated by an architecture designed for multiprogramming [8,10], specifically the Daisy applicative programming language [18, 28]. The advantages of applicative programming and heap memories for multiprocessing have been discussed elsewhere [7, 28], but difficulties of heap management have not been confronted.

In order to enjoy the benefits of multiprocessing it will be necessary to keep more than, say, ten processors uniformly busy. Otherwise, the (usually logarithmic) overhead of process dispatch/recovery would consume any local acceleration in computation speed, nullifying the effort, as compared to uniprocessing which incurs no such costs. And ten processors can consume the available heap very quickly, indeed!

One requirement of heap multiprocessing is that available space necessarily must be distributed so that processors can obtain new nodes from many sources [17]. Otherwise they will queue at the single source of available nodes, and this multiprocessor will run only as fast as that uniprocessor allocator.

Synchronization-avoidance suggests that conventional garbage collection be eschewed as much as possible. Operating on the philosophy of determining garbage "by the process of elimination," it requires some synchronization among all *heap* processors in order to work. It would be far more desirable to determine that a node has become available without communication among processors—without lots of processor/processor interlocks.

Reference counting, in contrast, involves inherently local operations and message passing (which need not involve the processes that manipulate the heap, as we shall see.) Even if it cannot recover certain circular structures, it still can postpone synchronization-intensive garbage collection when a hybrid storage manager is used.

Section 2. History

A consequence of the separation of the memory from the processor in a Von Neumann machine (Babbage's store from his mill) has been that memory conceptually contains no intelligence; all it does is fetch and store. With this perspective, hardware engineers have been very successful over the years in squeezing more and more memory into less and less volume. While there have been proposals for unifying mill and store to form pieces of larger systems (dating from Von Neumann's cellular automata [25]), few multiprocessors have become serious products because of the increasingly efficient designs of fast monolithic memories.

A few efforts to build intelligence into the store have been quite successful. Associative memory is an example. The most familiar example is hierarchical memory, wherein information migrates away from the processor according to the infrequency of its use. In various designs this shows up as cache, paging, virtual memory, or backing store which are commonly implemented by interposing address-logic (that causes surreptitious migration of data among storage media) between the processor and a conventional memory.

Another is the concept of a self-managing heap memory, most familiar in LISP and data-base-management systems, where unused nodes automatically migrate to Available Space. We study this concept here. Heap oriented languages like LISP, SNOBOL, Scheme [22] and Daisy [18], must include some sort of storage manager or *collector*, commonly a garbage collector. Garbage collection identifies available nodes by elimination; it identifies those nodes that are being used and condemns the others. Originally, such inference required a *quiet* heap (off-line storage recovery) unsuitable for real-time applications. More recently algorithms that work with a *noisy* heap have been developed for on-line [1] or dual-processor [*e.g.* 11] collection; some have been implemented in commercial hardware (*e.g.* by Symbolics [20] or Texas Instruments).

Both quiet and noisy heap algorithms, however, require strong interaction between the collector and the user of the heap—commonly called the *mutator*—lest heap dynamics interfere with the logic of elimination. In the quiet case, off-line collection is alternated with on-line mutation on a uniprocessor, and in the noisy case a mutator process is strongly choreographed (*i.e.* delicately programmed to synchronize) with a collector process using sensitive inter-process communication (*e.g.* 'coloring') on each node in the heap. Neither generalizes to larger-scale multiprocessing.

There have been constrained proposals for multiprocessing collectors. Hewitt and Lieberman [19] propose one that depends on a genetic address ordering remarkably absent in my experience with lazily constructed lists. Hudak and Keller [13] suggest one based on task queues with high message traffic.

State-of-the-art storage managers, as used in Smalltalk systems [15, 6, 24] are hybrids of garbage collection and reference counting. Due to Collins [4], reference counting requires that each node in the heap maintain a count of active references to it; when this count falls to zero, the node becomes available. It is able to reveal and to recover the plethora of nodes that are lightly shared [3] and that have short lifetimes. Contrary to apocryphal tenets repeated by Baker [1], interesting circular structures *can* be handled by a referencing-counting collector [9, 2, 14], and neither extra address space nor extra processor cycles need be required to maintain accurate counts. The point is argued below.

In a hybrid system, moreover, garbage collection and reference counting work together symbiotically. Reference counts cheaply recover the vast amounts [3] of ephemeral [20], lightly or uniquely [23] used space, thus postponing the next garbage collection. Bits already dedicated to counts at each node obviate the need for special 'mark' or 'forward' bits there, to be used exclusively during garbage collection. The certainty of eventual garbage collection allows smaller, but equally effective, reference count fields. Even if an intermediate count had ascended to some artificial, non-decrementable "infinity," eventually it might be accurately recounted to a lower value during traversal by garbage collection [29, 5, 27]. Any accumulation of dereferenced circular structures would be recycled then, too.

Section 3. Why not reference counting?

From the perspective of uniprocessing, it is hard to argue about the virtues of software implementations of storage collectors. Garbage collection has the advantage of accomplishing a lot for the memory cycles invested, because it occurs relatively rarely and each node in use is visited only a few times then. In contrast, reference counting (in software) can require a sixfold increase in the time for *a simple memory write instruction*, while increasing the address space required [1]. The cost-per-node of software garbage collection, thus, compares with the cost of *one* pointer assignment under software reference counting.

With software reference counting, a memory write requires that the count of the newly written pointer first be incremented (2 memory cycles when the arithmetic is done in the processor), the obsolete pointer fetched and its count decremented (3 memory cycles), and finally the new contents stored (1 memory cycle). More work remains when a decremented count reaches zero! Also, each memory cell must be expanded to include a reference count, consuming precious address space.

The software situation has given rise to the following reasoning: "Reference counting in software is slow; software garbage collection is fast; algorithms can be accelerated by hardware implementations; therefore, the fastest storage manager is a hardware garbage collector."

But why couldn't reference counts, like garbage collection, be implemented in hardware? Two ideas occur immediately: first, place the count field in separate memory *at the same address* as the node counted. Second, move the simple increment/decrement operations from the processor to the memory. There, a write instruction requires the dispatch of an increment, fetch of the former contents, dispatch of the decrement, and finally the write. No additional address space is consumed!

Borrowing an idea from virtual memory, let us arrange a processor/memory interface so that these four steps are dispatched not from the processor, but from dedicated circuitry at or beyond the memory gateway. The processor need only dispatch a pointer write (as before) and no processor cycles are lost. The increment and fetch can proceed in parallel, as can the decrement and store. Therefore, the pointer write instruction need take time that is the larger of one processor or two memory cycles (really one read-modify-write), instead of six processor cycles!

What happens when a count is decremented to zero? In the case of software reference counting, the processor is responsible for updating the available space list, consuming more processor cycles. But now memory becomes responsible for maintaining that list, and updating may be postponed to occur in parallel with the next memory cycle. Available space may be linked using the same field which was (until the decrement) the reference count associated with that node. With memory maintaining that list however, we introduce a new processor/memory instruction: "Give me the address of an available node."

(The contents of any released node remain unchanged while it is on an "available space list." An idea due to Weizenbaum [26] is implicit here in the eventual reinitialization of an available node's contents—after its reallocation. As the stale contents are overwritten, the archaicly counted references are destroyed. Reference-count decrements, though deferred, remain effective.)

Finally, we observe that reference counts decentralize memory management for a multiprocessor heap. In contrast to garbage collection, which requires implicit communication among all nodes in use, maintenance of reference counts is a thoroughly local exercise, only the nodes directly involved in a pointer transaction are affected. Moreover, these collector transactions may be reduced to operations so simple (*i.e.* two-cycle write) that the mutators need not be significantly slowed. A heap-oriented system containing many memory banks and many processors may run at full capacity for longer periods without using the synchronization, and (perhaps) uniprocessing required for garbage collection.

Section 4. Features

The design provides the following features:

- 64-bit (8-byte) nodes;
- two pointer fields per node of up to 29 bits each;
- 1 bits per pointer for marking (uncounted) circular references [9];
- 2 bits per pointer for conditional-store testing at the memory [8, 10]
- double cycle 32-bit write with automatic reference-counting;
- single cycle allocation of a "new" cell from available space;
- query on census of local Available Space;

•query whether a particular node is uniquely referenced;

•enter/exit garbage-collection mode which enables/disables

  single cycle 32-bit read and unconditional write;

  single cycle increment of reference counts (in lieu of marking);

  2 pointer reversal write-instructions for in-place traversal [21].

•Zeroing of all counts on entry to garbage collection mode.

•Reconstruction of available space list from zero counts on exit.

Present design in CMOS provides for a bank of 2048 32-bit words residing on one chip; two such chips, plus a control chip, make up a bank of 2047 nodes in the heap, and can be packaged in a single 64-pin DIP carrier. (See Figure 1.)

The control chip contains a hidden 12-bit field at each address that is used for storage management. One bit is reserved for use only in garbage collection [19, 24]. The other eleven have either of two uses: to maintain a linked list of available nodes or, in allocated nodes, to sustain an 11-bit reference count. Invisible to the processor, this field responds implicitly to pointer writes, incrementing (the new referent) once and decrementing (the displaced referent) once with each write instruction; thus, the need for two-cycle writes. (Eleven bits are necessary to provide a chain of 2047 available nodes, but not quite sufficient to sustain accurate local counts when complex structures are forced into a single bank [2].) It also maintains the available space list consistently with zeroed counts and requests for "new" nodes.

Several such memory banks may be ganged on an address bus (with different prefix addresses selecting only one at a time) or they may be distributed at one end of a processors/memories switch that allows simultaneous paths from several processors to several banks. The former possibility makes this design attractive for the existing uniprocessor market, but it is the latter arrangement, well suited to multiprocessing, that motivates this design.

Johnson [17] argues two virtues of separation of processors from memory-managing storage on either sides of a store/forward (*e.g.* a banyan) switch. First, a processor which is rapidly building some data structure may be supplied in real time with nodes uniformly distributed from from all memory banks; the switch is configured as a *new sink,* each node of which sustains the address of one available node in anticipation of a "New" request. Thus, locality of reference—so desirable in virtual memories—is intentionally destroyed. Second, a data structure so distributed is far less likely to cause interprocessor contention for paths to memory. Without locality of reference all memory access patterns are randomized; two processors would be most unlikely to clash in temporary contention when traversing different parts of the same structure.

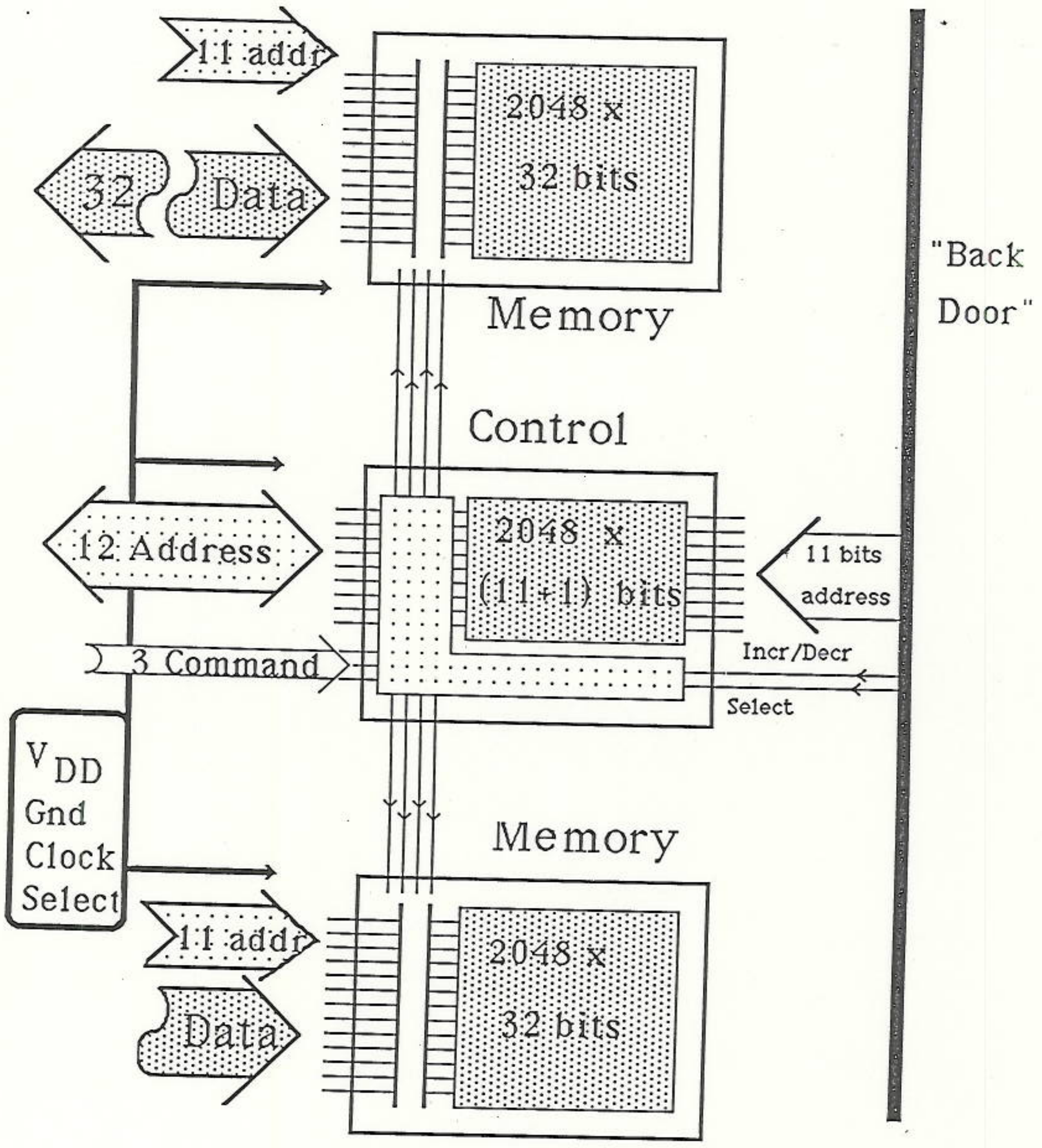

Figure 1. Memory Bank

In either bus or switch configuration, each bank requires a "back door" path to the others (respectively, a bus or a switch) for propagating increment/decrement requests across banks. Such requests occur at twice the rate of writes—an increment and a decrement for each pointer assignment, though not all need be propagated off-bank if references are local. Indicated by a port at the right of Figure 1, this "back door" port is one-way and contains only address lines, select, and one control line (increment/decrement; it might be replaced by synchronizing selection on a two-phase clock.) Thus, a path independent of that for data transfer is necessary among all memory banks *and* from processors to memories (*e.g.* for issuing decrements releasing pointers from registers.)

If used in a multiprocessor system, this memory provides a simple synchronization primitive [7], that acts like a localized "test-and-set" instruction, but without the originating processor's waiting. Nicely interfacing with reference-counting, the double-cycle write instruction is used first to read the previous contents of the destination word; if the selected bit is already set, the overwrite is canceled. Otherwise, the new data is stored on the second cycle.

A *garbage collection* mode is provided to accelerate write instructions and to inhibit implicit reference count operations. During garbage collection reference counts are incremented (from zero), without decrementing. If left in this mode permanently, the package can act like a conventional memory of 4096 32-bit words for traditional (*e.g.* FORTRAN) use.

The choice of 2047 nodes is dictated by current technology. Current design requires nine control pins (described below), 32 data pins, and $2n + 1$ address lines for a memory of 2^n 64-bit nodes. Of these, n address lines are used to receive increments/decrements from other memories (or processors), and $n + 1$ address lines are necessary to communicate an order from a processor to either half of a binary node.

Moreover, memory requires two chips with 2^{5+n} bits stored on each, one for each half of the 64-bit node. This constraint suggests that n be odd, so that these chips be squarish. Current technology provides 64-pin DIPs and RAM bits of area 400-600 μm^2, so both constraints indicate that $n = 11$. Thus, an ordinary memory chip is similar to a $64K$ RAM.

This memory requires more control lines than conventional read-write/select control. There are four ordinary lines for power, ground, clock, and select. Instead of a single line to determine read/write, three lines provide for eight instructions. Of these, two provide for fail-safe switching between *heap mode* and *collection mode*. In either mode the remaining six operations may be specialized.

In heap mode, there will be three flavors of write instructions, a read instruction, a NEW instruction, that requests the address of an available node, a test whether a particular address is uniquely referenced, and a query for the census of local Available Space. Of these, only the first four use all address and data pins, so that the last three may be overloaded as a single instruction, leaving one code to spare.

Three bits in either half of the node are distinguished. Let us perceive these as low-order, so that they would be masked off pointers to 8-byte nodes in a byte addressed memory. Two of these are used to implement the *sting* conditional store instruction [7, 16] on separate bits. One is used to indicate an uncounted circular reference [9].

The three write instructions are, then, the unconditional write, and the conditional write on either of the two distinguished bits. The third bit does not affect direct ("front door") control; if set during a write instruction, it inhibits the ordinary reference counting mechanism [9], as described in Section 6.

Collection mode is intended to provide for a mark/sweep collector. All the 12-bit fields are initialized on entry into garbage collection. During collection eleven of these bits are used as a glorified *mark* to recount references to each node, while the twelfth is used to guide the in-place traversal [19]. Additional write instructions are tailored to count accesses and to perform pointer-reversals (tree inversions), conditional on those eleven marks or the twelfth tag.

The sweep phase of garbage collection may be handled in either of two ways. Current design has the command to leave collection mode initiate a hardwired sweep phase, rebuilding the linked available space list from all nodes unmarked (with zero counts) before returning to heap mode. Thus, the exit from collection mode will be quite slow.

Alternatively, configuring the counts as an associative memory would obviate the need for linking available space. Collection mode could be terminated in one cycle, with "New" requests handled by an associative memory search for zeroed counts. In this case the count fields could be much smaller than the eleven bits suggested above [3, 5].

Two additional control lines are associated with the "back door" (See Figure 1), the extra n address lines that receive increments/decrements from other memory banks. These are a bank-select and the increment/decrement indicator.

Thus, the $4+3+2=9$ control pins have been accounted for. Of these, the least conventional are the middle three, because a processor must be able to transmit 3 bits of command along with 12 bits of address and 32 of data at each bank of memory.

How can these extra two control bits be transmitted from conventional processors? If the conditional-write instructions are not to be used in a particular application, then the extra two bits may coincide with the aforementioned low-order sting bits of the data, those bits that address bytes within a node. Alternatively (and less desirably) two high-order address or data bits could be sacrificed to control, reducing effective address space to 2^{30} bytes.

Section 5. Control Chip

The control chip is one of three elements forming the bank of 2047 nodes or 16,384 bytes, and it contains none of this visible memory. It has three parts: a memory of 2048 twelve-bit "fields," and control circuitry for both heap mode and collection mode. Since most tasks are done simultaneously with others, it is difficult to describe it completely; details of its design are not yet firm.

In "heap mode," these twelve-bit fields serve either of two purposes, depending upon whether that node is available or not. If it is available, then it is linked onto an Available Space List, which is terminated by the 0000_8 link. (If associative memory is implemented, it might have content of 4000_8 indicating that it should be on that list.) If it is in use then it will have there a reference count of ∞ or between 1 and 2046, respectively denoted by 4001_8 to 7776_8. Denoted by 7777_8, ∞ is a "sticky" reference count that can be neither incremented nor decremented.

The zero terminator for the Available Space List precludes the use of all 2048 addresses as heap cells; Word 0 of each bank is not accessible in the heap because its address cannot be linked within the Available Space List, and because 'allocation' of that address indicates exhaustion of available space. Thus, there are only 2047 nodes in each bank. When used as conventional memory (in collection mode) this constraint need not be observed, because there is indeed memory at address 0; there are then 16,384 bytes in a bank.

Only the address and control lines reach this chip; it need not "see" all of the data lines between memory and processor. In terms of the pin counts of the previous section, 9 control lines and $11 + 12$ address lines reach this chip. In addition, four lines, not connected to the external world, run from this chip to either of the two memory chips in the DIP. The only connections necessary to the data bus are for responding to special processor queries like that for the address of a new node, for the census of available space, or testing whether a node is uniquely referenced. The responses to all these queries might be offered on the eleven address lines (on the twelfth address line in the last case), but that arrangement would require that a processor be able to *read* address lines. To avoid this unconventional protocol, the twelve data lines must be tied to twelve address lines during these instructions; the tie might occur either at the processor or, more likely, here in the memory which requires this splice.

In both operating modes, the control chip receives the commands from outside the DIP, interprets and forwards them to either memory chip for action. For instance, in heap mode this chip responds to requests for new memory addresses; it will remove a node (without altering its 64-bit content [26]) from its internal list of available space, set its reference count to one, and deliver the address on the address lines to a requesting processor ("front door"). When no nodes are available it responds with the zero address, the address of the unusable 2048th node. It answers queries regarding population of available nodes and unitary reference counts of specific nodes in the same way. It is also this chip that responds to requests to increment and decrement reference counts from the other memory banks (back door), again without disturbing memory content.

When memory activity is necessary, the control chip enables either of the two memory chips to connect to the 32-bit data lines, giving them a three bit instruction telling what is to be done. In heap mode these are commands to read (single cycle) or one of three write commands (double cycle): unconditional and conditional (*sting* [7,10]) on either of two bits. In collection mode these are a read or write (each single cycle), or two memory/data-bus swaps (double cycle). All together, six different commands are possible, although not all are used in either mode.

Upon entry to (off-line) garbage collection mode, all twelve-bit fields are initialized to this 4000_8 pattern; if the node is never touched it will remain there after collection to be linked or scavenged into available space. As a node is accessed thereafter, this count is incremented (until, as above, it might stick at $\infty \cong 7777_8$). Thus, eleven low-order bits together act as a mark bit.

The garbage collector uses the high-order bit to direct its binary traversal without altering the low-order eleven bits containing the count. Two pointer-reversal writes encapsulate the critical steps in the in-place traversal algorithm. In one, the processor presents the address of the current 'stack' [19] on the data line as it presents the address of any accessible (probed) node on the address lines. If the node is already marked (has already been counted at least once), then the count at that address is simply incremented; if not, the value on the data lines are swapped with one pointer in that address, its count incremented to one, and its tag flipped. Thus, another node is stacked. The other pointer-reversal write is used to pop the stack and does not increment counts. In that instruction the value on the data bus is swapped into either half of the node (depending on its tag), and the tag might be flipped (depending on its value). The net effect is popping the stack.

Several processors may use these two instructions to mark the variously rooted structures *without mutual interference.* Because pointer-reversal occurs *at memory* simultaneously with mark and tag tests, each node may be stacked only once, but every probe to it will be counted. The alternate address bus (back door) is ignored during collection.

Section 6. Memory Chip

As the control chip is the difficult part of this design, so the memory chip is the elegant part. Two identical chips are required; most of their area is composed of conventional memory cells. The interesting difference from conventional memory is two 32-bit data registers at the periphery. The two chips correspond to the CAR and the CDR fields of ordinary LISP nodes. They have been designed as 2048×32-bit memories, to provide 64-bit nodes.

On a typical write instruction (heap mode), the instruction is decoded by the control chip, which enables and instructs either of the memory chips. Simultaneously, the new pointer arrives at that chip on the first memory cycle and is latched in an *outer* 32-bit register on the edge of the memory chip.. An increment message will already have been dispatched to the alternate address line (back door) of the DIP to which those 32 bits point. Simultaneously, the extant content at the destination of the write instruction is fetched to an *inner* 32-bit register adjacent to the other register.

During the second cycle, the contents of the two registers are swapped, the new contents are written into the destination, and a decrement message is sent from the outer data register on this chip to the "back door" associated with the former contents of that memory. In the case where the former and new pointers coincide, the net effect of the transaction is a null operation (unless the pointer's reference count initially was 1022_8.) This case is important, because it shows that the decrement message must *follow* the increment to the referent (so that a count of 1 goes to 2, and back to 1; instead of to 0, whence it is recycled before it can get back to 1.)

The conditional-write instruction and manipulation of circular references involves simple twists to the control described above. In the case of either of two "sting instructions" [7, 16], it is necessary that the entire write be canceled when the respective memory bit is *already* set to 1 as the write is attempted. In many ways it is similar to a "test and set" (although the processor does not wait) or the "add to memory" [12].

To implement the "sting," one of the two distinguished bits is extracted from the inner register, which contains extant contents of the addressed field. If that bit is "on," then the register-swap described just above is inhibited. The write completes in the ordinary manner: the extant content is returned to memory (without changing its count), and the "new" content is shipped off as a "decrement" message to arrive on the heels of the already dispatched increment, nullifying it.

The idea behind reference counting of circular structures is that certain distinguished circular links are not counted [9]. It remains a burden on the programmer that such cycles be created all at once, that no reference into the cycle outlive the reference by which it gained access there, and that these distinguished links be easily detectable. A bit in each pointer is dedicated to distinguishing such "cycle closing" links.

If such a bit is present in the new contents being written to a pointer field in memory, then the usual increment message will not have been sent (or perhaps redirected to local address 0). When such a bit is found in the obsolete pointer register (which would generate a decrement message), then that message is similarly canceled (or diverted).

The same swapping mechanism will be used to provide a single instruction to invert pointer chains during collection mode. The collector will access a node by addressing it, while offering a chain address on the data line. The control chip will (increment the reference count, marking the node, and) chain the stack in the appropriate way. It will use the same two registers on one (and later the other) of the two chips to perform the pointer swap that is the critical step in the classic Deutsch-Schorr-Waite Algorithm [21].

Section 7. Progress and Conclusions The design of the memory chip, described in the preceding section, has been pursued as far as a partial layout. Its floor plan and layout are firm and well understood. The control chip has a functional specification, and a spare command code remains that might be used in further development.

While "pin out" has been considered here, there has been no enumeration of the pads—specifically the test pads–that would be necessary to support production. The control chip offers a particular challenge in this respect, because its memory is not visible from any external pins. One must provide additional pads to allow tractable testing of such internal circuits.

Finally, Hughes [14] has recently proposed the use of Tarjan's linear-time cycle detection algorithm to allow reference counting to recover small, circular structures. It might be possible to provide such a facility on the control chip when the cycle is localized to reside entirely within the local memory. While there is area and an instruction available for such a facility, and while it is consistent with Bobrow's recommendations [2], its time overhead might undo the constant-time behavior already provided.

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. D.P. Friedman and D.S. Wise. An approach to fair applicative multiprogramming. In G. Kahn (ed.), *Semantics of Concurrent Computation*, Berlin, Springer (1979), 203-250.

11. D. Gries. An exercise in proving programs correct. *Comm. ACM* **20**, 12 (December, 1977) 921-930.

12. A. Gottlieb, R. Girshman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer—Designing an MIMD shared memory parallel computer. *IEEE Trans. Computers* **C-32**, 2 (February, 1983), 175-189.

13. P. Hudak and R.M. Keller. Garbage collection and task deletion in distributed applicative processing systems. *Conf. Rec. 1982 ACM Symp. on LISP and Functional Programming* (1982), 168-178.

14. R.J.M. Hughes. Reference counting with circular structures in virtual memory applicative systems. Programming Research Group, Oxford (1984).

15. D.H.H. Ingalls. The Smalltalk-76 programming system: design and implementation. *Conf. Rec. 5th ACM Symp. on Principles of Programming Languages* (1978), 9-15.

16. S.D.Johnson. Connection networks for output-driven list multiprocessing. Tech. Rept. 114, Computer Science Dept., Indiana University (October, 1981).

17. S.D. Johnson. Storage allocation for list multiprocessing. Tech. Rept. 168, Computer Science Dept., Indiana University (March, 1985).

18. A.T. Kohlstaedt. Daisy 1.0 reference manual. Tech. Rept. 119, Computer Science Dept., Indiana University (November, 1981).

19. H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Comm. ACM* **26**, 6 (June, 1983), 419-429.

20. D. Moon. Garbage collection in a large LISP system. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming* (1982), 235-246.

21. H. Schorr and W.M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM* **10**, 8 (August, 1967), 501-506.

22. G.L. Steele, Jr. and G.J. Sussman. The revised report on SCHEME: a dialect of LISP. MIT A.I. Memo 452 (January, 1978).

23. W.R. Stoye, T.J.W. Clarke, and A.C. Norman. Some practical methods for rapid combinator reduction. *Conf. Rec. 1984 ACM Symp. on LISP and Functional Programming* (1982), 159-166.

24. N. Suzuki and M. Terada. Creating efficient systems for object-oriented languages. *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 290-296.

25. J. Von Neumann. *Theory of Self-Reproducing Automata* (Edited and compiled by A. W. Burks), Urbana, Univ. of Illinois Press (1966).

26. J. Weizenbaum. Symmetric list processor. *Comm. ACM* **6**, 9 (December, 1963), 524-544.

27. D.S. Wise. Morris's garbage compaction algorithm restores reference counts. *ACM Trans. Prog. Lang. & Systems* **1**, 1 (July, 1979), 115-120.

28. D.S. Wise. The applicative style of programming. *Abacus* **2**, 2 (Winter, 1985), 20-32.

29. D.S. Wise and D.P. Friedman. The one-bit reference count. *Nordisk. Tidskr. Informationsbehandling (BIT)* **17**, 3 (September, 1977), 351-359.