Research Demonstration of a Hardware Reference-Counting Heap*†
Technical Report 401, Revised

David S. Wise,‡ Brian Heck,‡ Caleb Hess,‡ Willie Hunt,‡ and Eric Ost‡§
Indiana University
Bloomington, Indiana 47405-4101 USA
`dswise@iuvax.cs.indiana.edu`

February 1997

Abstract

A hardware self-managing heap memory (*RCM*) for languages like LISP, SMALLTALK, and JAVA has been designed, built, tested and benchmarked. On every pointer write from the processor, reference-counting transactions are performed in real time within this memory, and garbage cells are reused without processor cycles. A processor allocates new nodes simply by reading from a distinguished location in its address space. The memory hardware also incorporates support for off-line, multiprocessing, mark-sweep garbage collection.

Performance statistics are presented from a partial implementation of SCHEME over five different memory models and two garbage collection strategies, from main memory (no access to RCM) to a fully operational RCM installed on an external bus. The performance of the RCM memory is more than competitive with main memory.

CR categories and Subject Descriptors:
E.2 [Data Storage Representations]: Linked representations; D.4.2 [Storage Management]: Allocation/Deallocation strategies; B.3.2 [Memory Structures]: Design Styles—primary memory, shared memory; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Parallel processors; D.1.1 [Programming Techniques]: Applicative (Functional) Programming Techniques; B.5.1 [Register-transfer-level Implementation]: Design—Memory design.

General Term: Performance.

Additional Key Words and Phrases: Uniprocessor, garbage collection.

*Research reported herein was sponsored, in part, by the National Science Foundation under Grant Numbers DCR 85-21497 and DCR 90-027092.

‡Computer Science Department
§Center for Innovative Computer Applications

1 Introduction

1.1 Technological Maturity Scale

In 1992 Stuart Feldman presented a series of hurdles by which to measure progress in computing [15]. Paralleling the accepted milestones before thermonuclear-fusion power, he specified five milestones for new computer technologies by comparisons with alternative tools:

1. "**Idea/Concept.** An idea has been conceived and, perhaps, published. It sounds good and original but, at most, [only] back-of-the-envelope calculations and trivial (usually paper) examples support it.

2. "**Research Demonstration.** The idea is embodied in a first serious example or running demonstration as proof-of-concept, and its originator still likes it, though it is not useful for actual problems.

3. "**Scientific Breakeven.** The technique is developed sufficiently to be used on some real problems, and is shown to be better than some common technique or tool. Experience is not wide, and is probably controversial.

4. "**Engineering Breakeven.** Not only does the technique or tool do something, but it [is useful] on full-scale problems in concert with other tools.

5. "**Financial Breakeven.** The concept is *known* to be the technique of choice in a significant domain, and that it is profitable to use it in general software practice [and] in concert with other techniques [14]."

In that context, this paper reports a research demonstration of hardware support for reference-counting and garbage collection as part of computer memory. Related work [21] supports scientific breakeven, as well. The concept was published twelve years ago [36].

The path from idea/concept to scientific breakeven is usually a story and, indeed, parts of this paper read like a narrative. However, the writing of stories like this—even unsuccessful ones—is essential to advancing the complex symbiosis between hardware and software. Without it we are denied the ability to whipsaw at their interface, reaching beyond improvements to one whose support can be foreseen in the financial marketplace. Innovative steps must be taken without a timetable for financial breakeven.

1.2 Reference-Counting Memory

We report the design, construction, and testing of a specialized memory that provides system-level heap management in real time, providing atomic transactions that allow multiported access and multiprocessing collection [20] without repeated synchronization. The memory system is installed on a NeXT cube, and experiments compare it here to conventional collection on that machine: both stop/copy and mark/sweep. It has been used to support a purely Applicative File System [21] and it has served as a test for rapid-prototyping of new insights into compile-time management [39].

Storage management is the broad problem of recovering unused space from the heap without explicit instructions from the programmer. It includes both *reference counting, garbage collection,* and hybrids of the two, like that reported here. Although the second term is often used to include all of storage management [7, 25], we choose the original taxonomy [26, p. 412] because we find it more descriptive: reference counting is a better analog of curbside recycling than of trash collection.

Because of the central role played by heap management, it has now become a problem worthy of hardware support. Indeed, others have implemented such hardware for uniprocessing or proposed it for multiprocessing [3, 18, 2, 28, 31, 33]. Although reference counting has been used successfully for managing secondary storage, garbage collection is still thought to be far more suitable for primary storage, where hardware support will be the most effective. As shown here, however, hardware is quite effective in leveling the reputed disadvantages of reference counting.

Every pointer written causes an increment to the new referent, and every pointer erased causes a decrement, as well; thus, the common pointer overwrite causes both. Although the principal strategy is on-line reference counting, implemented purely in memory [36], RCM also includes support for mark/sweep garbage collection [34] that is both fast and parallel, but off-line. In a multiprocessing implementation all transactions (increments and decrements, as well as marking) would be dispatched asynchronously to be performed locally at memory.

A balanced multiprocessor computes fast; therefore, a multiprocessing heap-*mutator* [18] generates garbage at a tremendous rate, and a balanced *collector* requires parallelism to keep apace. Conventional garbage collection, however, assembles global knowledge ("Nothing points here"); thus, any multiprocessor realization requires much synchronization, which is not a constraint on uniprocessors where garbage collection is thought to be perfected [1, 11]

The asynchronous, atomic transactions of reference counting [8] make it the heap manager of choice for a multiprocessor or multitasking system, It is commonly used to recover available sectors from a shared disk, where a traversing collector is only rarely used (*e.g.*UNIX's `fsck`.) In contrast, garbage collection has become the heap-management algorithm of choice on uniprocessors, where its amortized performance, measured in processor-memory transactions. undercuts that of processor-driven reference counting. This paper shows how moving such counting from processor to memory inverts this conventional wisdom.

1.3 Onward

The remainder of the paper is in six parts. The second section reviews the philosophy and the design of reference counting in hardware. The third is a description of the hardware support for off-line garbage collection. The fourth is a brief chronology of the project and its limitations. Next a brief description of the SCHEME software and test problems. The sixth section presents the results of our testing; the final section reviews the future, including some features that have been implemented but, as yet, not exercised.

2 Reference Counting in Hardware

2.1 Why Reference Counting?

There are three reasons why *reference counting* [8] was studied as a primary strategy for memory management, in spite of prevailing practice using either off-line or on-line garbage collection [7, 2, 3, 29, 22, 30].

First, the same properties of reference counting that make it the usual strategy for managing disk sectors also become particularly important in multiprocessing. The three transactions (*v.i.*) that here implement reference counting are all finite and *non-global.* That is, each is performed locally and asynchronously *at* memory without interference from other transactions; each is an atomic action. The various processes are free to continue without synchronizing with storage management.

Second, reference-counting allows the heap's address space to be much smaller than a collected one. Nodes are recycled in real time and existing space is used more densely—even to 100% capacity—without sacrificing any global performance; locality increases.

Finally, reference counting can, indeed, handle many useful data structures, specifically including certain cyclic ones. Conventional wisdom, that it breaks on cycles, stems from overstatements in both textbooks and research references. While arbitrary circular structures are well known to break reference counting, they are often overused; even classic examples are better implemented without any cycles [26, §2.2.4]. Moreover, that "wisdom" ignores strategies for circularities that sustain reference counting in useful situations [9, 5, 17, 39]. Baker [3, 4], for instance, implies that it consumes both address space and processor time, neither of which is used here.

2.2 A heap node

The prototype provides nodes that are either atomic or binary, as the skeletal C++ declaration below illustrates. It models the cons nodes of SCHEME and LISP, but also suffices for aggregates of arbitrary size via naturally corresponding binary trees [26, §2.3.2] or, better, complete trees [26, §2.3.4.5]. The latter are used to implement matrix decomposition in Section 5.2, below. These generalizations are needed by SMALLTALK and C++-style languages, whose efficient implementations likely require a four-pointer node; *cf.* Section 6.

```
typedef word char[4];                         //A 32-bit word at address 4k.
typedef unsigned int bool;       const bool false=0, true=1;
enum Type {atomic, binaryNode};
typedef struct Node{
        union { struct {struct Node *left, *right;} twoLinks;
                double                              atom;
        } data;
        bool mark, liveLeft, liveRight;
        Type type;
        unsigned int RefCt;
} RCMnode;

const leftRight = 4;
      //Bit 2 distinguishes addresses of double words from those of words.
inline double *selectLft(word *ptr) {return ((long)ptr) & (! leftRight);};
inline bool    isRight  (word *ptr) {return ( (int)ptr  &    leftRight);};
```

2.3 Operations at a Node of RCM

Every write of a pointer to *reference-counting memory* (*RCM*) becomes a *read-modify-write* there. That is, a new pointer is overwritten at a memory location only by removal of a former reference in that same location during the same memory cycle. Dynamic RAM provides this operation at little cost beyond a conventional write. Although readily available on stock RAM chips, it is lost on commodity SIMMs where lines for data-in and data-out are wired together; wider SIMMs could do it.

To write a pointer, the address and data are dispatched from a processor to remote memory where the following algorithm is executed uninterruptibly (as a remote procedure local to `destination`, a word address):

```
void store(RCMnode *pointer, word *destination) {
        RCMnode *leftDest = selectLft(destination);
        if (leftDest->type == binaryNode)
               DISPATCH incrementCount(selectLft(pointer));
        if (isRight(destination)){
            if (leftDest->liveRight)
               DISPATCH decrementCount(leftDest->data.right); //to *destination
            (   leftDest->liveRight) = (leftDest->type == binaryNode);
            leftDest->data.right     = pointer;       //*destination = pointer;
        } else {
            if (leftDest->liveLeft )
               DISPATCH decrementCount(leftDest->data.left ); //to *destination
            (   leftDest->liveLeft ) = (leftDest->type == binaryNode);
            leftDest->data.left      = pointer;       //*destination = pointer;
        }
}
```

This code bends the C++ language a bit, reflecting our *ad hoc* hardware; for instance, `DISPATCH`s are signaled to other, asynchronous circuitry. The four steps occur in parallel, subject to two constraints: the increment is dispatched to the "back-door" bus before the decrement[1], and the former content at the destination is read before it is overwritten with the new pointer. Both the fetches from *destination* and its tags, and the stores there occur during the same memory (read-modify-write) cycle. The sequentiality of these three steps in C++ code satisfies the constraints on a uniprocessor, but we intend them to be nearly simultaneous in hardware.

Reference-counting transactions can be interleaved with similar ones, dispatched from other memories to the same destination, as long as increments/decrements arrive at their target as some interleaving of the orders in which they were dispatched. A unique, non-caching path between any source-destination pair, as on a bus or a banyan net [19] meets this constraint.

Consistency of RCM memory is assured because memory overwrites occur as atomic operations locally at each RCM. Writing multiprocessors need not wait for completion after dispatching a new pointer-write to an RCM address because that pointer must already be alive and counted (at its referent) before the dispatch. Any write causes an eventual increment at the newly shared referent. Decrementing the count there would require overwriting of one of its extant, enlivening references, but the *last* decrement (to zero) cannot occur before this new increment arrives without violating the order restriction, above.

At the destination address, both increments

```
const sticky = 16777215;         // 2**24 -1 but can be far smaller---e.g. 255.
void incrementCount(RCMnode *ptr){ if (ptr->RefCt < Sticky) ptr->RefCt++ ; }
```

and decrements occur as atomic transactions; the "sticky" count provides that the counter can be smaller than the worst-case requirement [6].

```
void decrement(RCMnode *ptr){
     if ( (ptr->RefCt >= Sticky) || (bool)(-- ptr->RefCt) )  {;}
     else FREE(ptr);
}
```

The instruction `FREE(ptr)` above indicates return of `ptr` to the available-space structure on board RCM (at memory.)

The `increment` and `decrement` operations are serialized at their target address, but either can be handled simultaneously with a read or write there, because read/write uses circuits distinct from increment/decrement. *Each*

[1] It is necessary to dispatch increment before decrement to handle correctly a reflexive assignment, `ptr=ptr` when `ptr` is a unique reference.

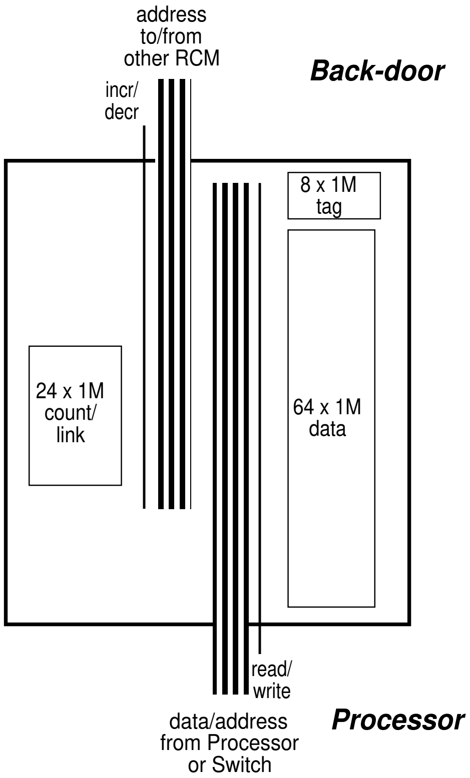

Figure 1: Layout of Reference-Counting Memory.

of these three operations requires only finite time. A node typically returns to available space still containing live, yet-counted pointers [35].

2.4 The physical implementation

Figure 1 illustrates the hardware configuration of RCM as implemented. Every address of RCM has both data memory and reference-count memory. The data memory handles conventional reads and writes from the processor(s), delivered via address and data busses and and strobed just as on ordinary RAM. The counting memory handles increments and decrements as they arrive, resulting from earlier writes or from *ad hoc* commands. *Ad hoc* queries and commands are implemented as reads and writes at a few distinguished locations within each RCM's address space, much like the control for a bus-resident device.

Each of the two memories has its own bus and ports: a data port to the various processors, and also (narrower) ports for real-time dispatch and receipt of increments/decrements to and from other modules of RCM. Since a single data transaction (write) might generate two dispatches, the dispatches must be handled at twice the speed of a data transaction. Fortunately, the bandwidth of an increment/decrement (address + action bit) is less than half that of any data transfer (address + data). Processing of the doubled dispatches from RCM-pointer writes, moreover, is also absorbed by interleaved transactions that do not involve reference-counts: *e.g.* RCM reads, RCM writes or overwrites of "dead" data, and RAM or idle-memory cycles.

Data in RCM is tagged as either *live* or *dead* [39] at memory. Atomic information and immediate references, like

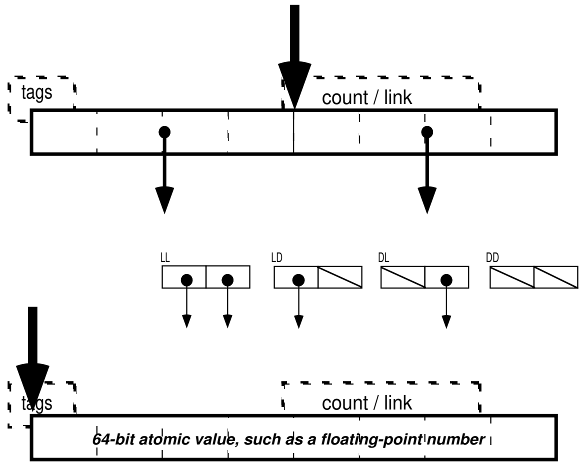

Figure 2: Memory at one RCM node.

NULL, are dead; pointers are tagged live. When a dead (atomic) datum is written to RCM, no increment is dispatched, and when a dead datum is overwritten there, no decrement is issued. Figure 2 illustrates how each node appears to the programmer, with L tagging a live datum and D tagging a dead one. Atomic nodes are distinguished from binary nodes at allocation, where their types are established and so tagged.

2.5 Available space

Each bank of RCM memory maintains its own available space. Whenever a processor needs a NEW node, it reads from a distinguished memory location there to obtain the address of the new node. Reading from one location allocates a node as an atom; from another, a binary node as coded above. Other distinguished locations are used as control and data registers for the RCM device. For example, writing an address to either of two distinguished addresses dispatches an increment or, respectively, a decrement to the count of the addressed node.

A special garbage-collection mode, during which the interpretation of reads and writes to RCM is *completely* redefined, supports a Deutsch-Schorr-Waite mark-phase [26, 34]. It provides multiprocessor marking in place and in a single processor cycle; it requires no interprocessor synchronization and only a few cycles for each node traversed. This support in hardware for hybrid garbage collection is also new. State-of-the-art storage managers used in Smalltalk systems [23, 10], as well as many secondary-storage managers, are also hybrids of garbage collection and reference counting, but this device is the first to provide hardware support for both.

3 In-place Garbage Collection

This prototype provides several new features not included in the idea/concept paper [36], including hardware support for Deutsch-Schorr-Waite collection [34]. It has two features that render this algorithm competitive with recopying collection, even on this uniprocessor and, more importantly, that also provide for asynchronous, multiprocessing collection. The experiments below, however, can only demonstrate the speed of uniprocessor collection, which turned out to be faster than recopying.

First, the sweep phase is performed on-line in memory after marking is complete, and the sweeper runs even after the mutator has already resumed. Secondly and more importantly, RCM performs all the pointer rotations associated with marking as atomic operations at memory. It fits the live/dead tagging described above, and complements reference counting to form a truly hybrid heap manager.

When garbage collection becomes necessary—when an AVAILable space list is exhausted (in spite of reference counting)—all mutators must synchronize and enter collection together. At that time, all banks of RCM are switched to "GC mode". Arbitrarily many collectors then traverse all active nodes without any synchronization, marking them. After marking, all collectors synchronize and leave GC mode, whereupon the mutators may resume as before. The fact of leaving GC mode initiates a hardware sweeper in each bank of RCM that reconstructs the AVAILable space locally in time overlapped with the mutators.

Not only does RCM admit parallel collection, but also it recomputes all reference counts to perfect accuracy. Like recopying collection, it runs in just the time necessary for traversing and counting active pointers. Each mark is an increment to a recomputed reference count, correcting counts inflated by cycles or, with more compact counts, unsticking shrunken "sticky" counts [9]. Thus, garbage collection improves the performance of reference counting, just as reference counting symbiotically postpones the need for collection.

Aside from initiating the hardware sweeper as GC mode is terminated, that mode redefines the meaning of "read" and "write," so that RCM cannot even be cached as write-through memory while in that mode. There are actually *two* read functions, called *first read* and *later read*, plus a *special write* operation that rotates the node. The two reads are distinguished by the (four-byte) word address, even or odd, by which they access the eight-byte node.

"First read" is an atomic operation that responds differently when a node is read the first time after entering GC mode. On the second and later issues of "first read" to an address, it returns only NIL but increments the reference count. The very first time such a read is applied to a node, the node is marked, its reference count is reset to one, and the content of its first live field is returned. If no fields are alive, then a NIL is returned. To a (parallel) collector, such a NIL means that there is no infrastructure beneath the node to traverse.

If, however, "first read" returns a non-NIL pointer, then that infrastructure must be traversed recursively. To avoid a separate stack and its bookkeeping, a "special write" is used to replace that pointer with the stack/pointer, with a rotation of the node implicit at memory. On one or two later visits, another live pointer can be read/written with "later read," and ultimately the stack is read out and the last live pointer written back, restoring the node to its original configuration [34].

The effect of these special reads and write is to allow traversal of a node with two live pointers in six memory cycles, with one live pointer in four memory cycles, and with both pointers dead (*e.g.* atomic) in just one cycle. Each (later) shared reference to a node costs an additional cycle, but no synchronization beyond that implicit in the path to memory. Collecting a node in less than six cycles compares favorably with recopying performance (although RCM must yet spend another cycle initially to reallocate it.)

The measurements below show that this hardware collector does, indeed, beat non-generational stop-and-copy.

4 Constraints on the Prototype

An undercurrent in this project was an effort to develop memory hardware without hacking the kernel, and without rebuilding a programming environment; the former constraint was met, but the latter was not. At such a low level the

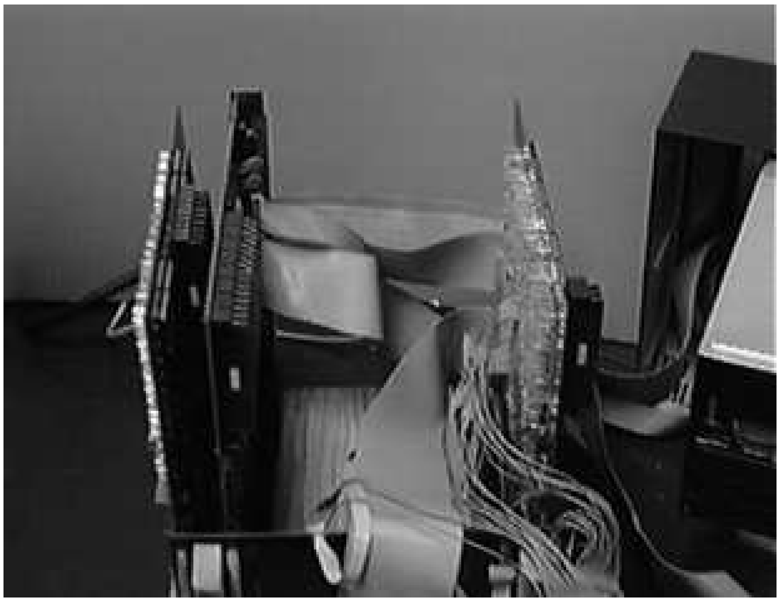

Figure 3: Reference-Counting Memory installed on NextBus.

project was vulnerable to upheavals from manufacturers' revisions to the MACH operating system, and to difficulties with the chosen language implementation. In the end, we developed our own rudimentary SCHEME environment that did establish RCM performance with sound statistics, but which was not complete because of its stack protocol that precluded continuations.

The project had three original motivations: exercise of rapid hardware prototyping from another project, refinement of our digital design tools [24], and support for applicative programming research. While substantially successful on all three goals, the three did erect additional constraints around the design. Later on, the RCM became essential to the Applicative File System [21], toward which the development of the ultimate SCHEME software was directed.

The rapid prototyping effort dictated that we use available hardware and software as much as possible. Thus, prototype RCM is implemented as a "device" resident on the bus, running at bus speed rather than at the speed of main memory, in order to avoid hacking on host hardware.

We chose the NEXTBUS because we had familiarity with Motorola 680x0 architecture and two machines: MACINTOSH and NeXT, that use it. The availability of NeXT's NEXTBUS Interface Chip (NBIC) and timely delivery of it and prototype boards swung us to NeXT as a testbed. Moreover, both the lab and the manufacturer were receptive to new ventures on the new product.

Standard 8-bit SIMMs are the prototype memory, and so initially contained one million nodes, each of ninety-six bits: twelve SIMMs (Figure 1). The user sees an address space of only eight megabytes (Figure 2). The remainder remains hidden except to engineers who can access all of RCM as RAM via a special addressing mode, which was used in the Heap on-a-bus and RAM on-a-bus tests, below. Figure 3 shows the two wire-wrapped boards that implement RCM on either end of the bus, with the SIMMs visible on each: nine on one and three on the other.

Some constraints turned out worse than anticipated. Figure 4 illustrates the physical location of RCM within the NeXT computer. Our original sketch of RCM-prototype timing provided a 240 nanosecond cycle time. This approximated the understood bus-contention time for read-modify-write and the NEXTBUS. After allowance for a

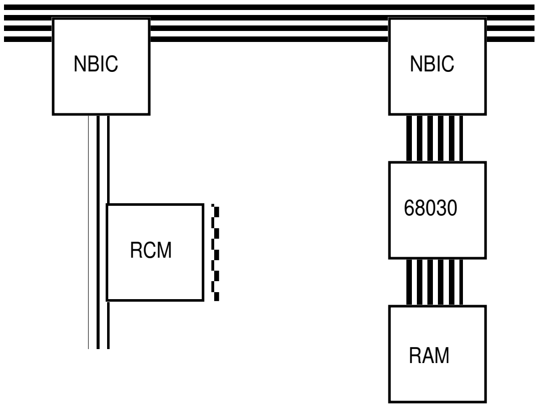

Figure 4: RCM on NEXTBUS.

subtlety of the NBIC—that three NEXTBUS cycles are needed to seize a second bus "within" each device—the sketch suddenly rose to 960 ns., overwhelming both the minimum needed for read-modify-write on SIMMs (row-address, followed by double-strobe of column-address) and the 300 ns. read-followed-by-write that was actually implemented. In fact, the timings were much faster because of another subtlety, described later with the results. These surprises arose from the exercise of hard implementation; simulation would never have exposed them.

Software has lots of references from the recursion stack to RCM, each one of which requires *ad hoc* increments and decrements to sustain counts with pushes/pops of the stack. To provide an alternative as fast as the stack, we expanded a portion of RCM, originally a RAM-like register file that issues increments and decrements but is otherwise physically isolated from the pool of heap nodes. The enlarged serial file is called pacRAM, for "pointers always counted" RAM. The tested version has two megabytes of pacRAM and six of RCM heap (3/4 megaNode).

Rooting pointers in the stack get deleted at run time as the recursion stack, residing in PacRAM, is overwritten. Such deletion, however, only occurs on a subsequent stack-push, rather than immediately on a stack pop, as intuition suggests. It is ironic that a stack pop is so passive with respect to storage management, just as it is with memory contents.

The hardware was built in eight months, on schedule, and the pacRAM design modification was made four months later. Minor wiring changes were made later (one wiring error, one counter expanded, and error-traps set), but the original design remained unchanged except for the pacRAM expansion.

5 Description of the Experimental Software

5.1 Hooks in the operating system

Multiprocessor use of RCM requires that caching not interfere with the order that writes are dispatched from the processor to memory; write-through caching suffices if writes from cache occur in the same order as the writes from the processor. Otherwise, caching must be disabled. (This requirement is more fodder for the debate over RISC vs. cache-coherency on multiprocessors.) This protocol suffices to prevent a node from being prematurely released by a write from one processor while a pending increment is asynchronously delayed within the cache of another. Therefore, RCM is accessed as a private device without caching, system traps, or interrupt protocols. Transparent user access to bus-resident memory is a surprising exercise; no kernel hacking was necessary!

The MACH operating system allows a user's program to have dynamically loaded kernel servers. These provide that a bit in SCHEME's process-control block indicates to the scheduler that the state of the transparent-translation register should be preserved across each context swap. That provision and a newer version of the slot driver from NeXT allowed user software the privilege of transparent access to devices on the NEXTBUS. This, of course, would not have been a problem for non-virtual systems like the 68020 Macintosh, but it would not have been possible on some UNIX systems.

5.2 The SCHEME Compiler and its Five Memories

The initial goal in testing RCM hardware is not to race it against other hardware (because the NEXTBUS delay is unfair), but to compare its own RAM-heap speed to its RCM-heap speed, factoring out the bus delay. Therefore, a series of problems was designed and refined to provide extrapolation of the performance of RCM as if built in VLSI technology and installed close to the processor like dynamic RAM. These experiments cannot indicate its multiprocessor performance or reveal contention that might result from intensive `increments` and `decrements`, because the bus cycles are long enough to absorb all RCM-internal cycles.

Figure 5 illustrates the five versions of the partial SCHEME implementation that we used. It is quite a good compiler on a minimal language, omitting bignums, floats, and continuations, for example. All are identical, except for where the recursion stack resides, where the heap of binary nodes resides, and how it is managed. The memory available is either cached RAM, or uncached RCM memory on the NEXTBUS. The latter is slower and in no cases is it cached. The host is a NeXT cube running under MACH 3.0 with a 68030 processor and 16 megabytes of RAM, exclusive of RCM.

- **Stock RAM:** This is a small SCHEME compiler that has its stack, heap, and code all resident in the NeXT's stock RAM. Any vectors are embedded within the heap, composed of two semispaces of three megabytes each. It uses a fast stop-and-copy garbage collector. NeXT's stock read cycle is 520 ns. for sixteen bytes, but the first four become available within 160–200 ns.

- **Heap on-a-bus:** In order to establish the slowdown from the geography of RCM's remote location on the bus, the compiler was altered to use RCM memory for its random-access heap only. RCM is used in its engineering mode, storing all binary `cons` cells in two three-megabyte semispaces there, with all specialized RCM transactions suppressed. Vectors remain in RAM, and remain static in all these tests. The purpose of this version is to establish the impact of the slower memory on the NEXTBUS. Because of size restrictions in RCM, only three-megabyte semispaces are used.

- **RAM on-a-bus:** This is identical to Heap on-a-bus, except the stack is also moved out to RCM, accessed as RAM but coincident with pacRAM. We move these objects out in two steps to separate the impact of heap transactions from that of stack transactions.

- **Nailed RCM:** As an intermediate step toward reference-counting SCHEME, we next enabled the mark-sweep collector on RCM without yet enabling reference counting. Since mark/sweep runs in place, only one "semispace" is needed in RCM; the hardware provides a six megabyte heap, but we twiddled it (at minor cost to

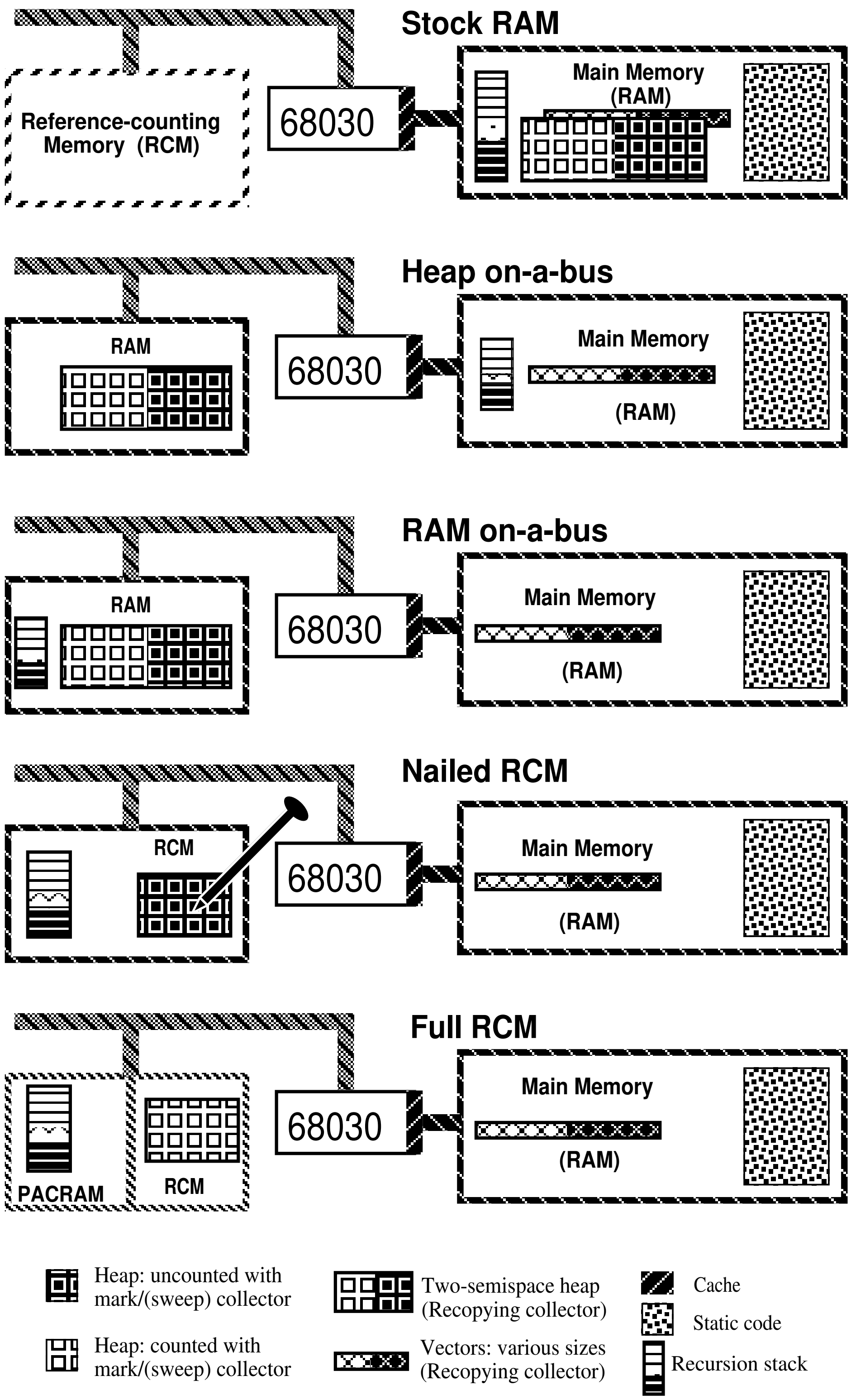

Figure 5: Five versions of SCHEME using different memory.

System	Number of GCs	RAM bytes recovered	RCM bytes recovered	Collector (seconds)	Mutator (seconds)	Total (seconds)
Stock RAM	30	47,597,608	0	520.1	482	1002
Heap on-a-bus	30	1992	47,602,796	544.4	516	1061
RAM on-a-bus	30	1992	47,602,796	559.0	683	1242
Nailed RCM	30	1992	47,602,928	190.3	708	898
Full RCM	0	0	47,679,856	0	710	710

Table 1: Experimental results: Balanced-Tree Insertion

collection timings) to be only three megabytes. New nodes are allocated in RCM, but with their reference counts initialized too-high-by-one; NEW reads from the "wrong" register. Each count must also be artificially incremented during every garbage collection, similarly, at the expense of an extra memory cycle for each node in use. The excessive count has the effect of "nailing down" every node, so that even though the reference-count machinery is operating, no count ever reaches zero. This version exercises the garbage-collection machinery, and exhibits the impact of support for the mark-phase and the on-line hardware sweeper, all performed by memory-resident hardware.

- **Full RCM:** Finally, the SCHEME implementation maintains accurate reference counts. The difference between this and "Nailed RCM" is one bit of object code, so NEW reads from the proper register (and the renailing cycle, omitted from garbage collection. It was not exercised in these tests.)

6 Tests and Results

We tested two programs that use trees and dags heavily, that create and release lots of intermediate results, that solve well known problems, and that consume tens of minutes of run time.

6.1 Performance on Balanced-Tree Insertion Problem

The first problem builds familiar binary-search trees. Appendix A presents our code for a SCHEME function that inserts a key–information pair into an initially empty, AVL or balanced tree [27, §6.2.3]. It is purely functional, returning the resulting tree from each insertion without requiring side effects to its arguments. The first test is to apply it in a tail-recursion to insert 75,000 different random numbers into an initially empty tree.

Because all keys and information are immediate data (Keys are small ints), each interior node in the balanced tree is a list of four cons-boxes. Since each box occupies eight bytes (Figure 3), the space required for the final tree is 2.4 megaBytes. With an average depth of 15.5, however, recopying the spine of each tree to a typical insertion point reallocates about 500 bytes—more when a "balancing act" is necessary. The results appear in Table 1.

Even though the first four systems, that collect garbage, all do so exactly 30 times, the first recovers less because another 1992 bytes of static vectors share its heap; the others can use that space to allocate and to recover even more nodes. The third, "Nailed RCM" system collects garbage slightly earlier than the two "on-a-bus" systems, because three nodes are pre-allocated by hardware; therefore, its recovery counts are slightly different. Full RCM counts bytes recovered in real time, up to the last transaction; all the other systems' figures are slightly lower because they derive from complete collections, not including the abandoned space yet to be collected at termination.

Both the mutator and collector times increase as the heap and stack are moved out to the NEXTBUS, although not as much as had been feared. Indeed, moving only the heap to the bus has comparatively little impact, perhaps because the (slow) reads there balance with the (NBIC-buffered) writes. The collector, likewise, exhibits this balance even when the recursion stack is moved there (because it does little stacking). When the mutator's stack is moved out to the bus, however, we observe a marked slowdown, likely because of heavier reading from stack frames. In aggregate, there is a 41% mutator slowdown as both heap and stack are moved out to RCM's address space.

Nailed RCM performs extremely well, recovering about the same space in two-thirds less time than RAM on-a-bus needed (even including the extra cycle to reset each nail.) The balanced tree is an ideal structure for this collector, because so many nodes have two dead pointers. Full RCM runs even faster, avoiding all garbage collection. Its mutator time, therefore, is its total time; and its time of 710 seconds extrapolates (if RCM were local memory) to 501 seconds on Stock RAM, which would halve the performance, including collections, that we observed there.

6.2 Exact-arithmetic Quadtree-Matrix Inversion

The second test program is exact-arithmetic matrix inversion, chosen because the problem is familiar and non-trivial, and because the purely applicative algorithm [37, 16]—though unfamiliar and not compiled well—*must* perform *in place* if it is ever to compete with popular alternatives. The results appear as Table 2 and Figure 6.

The problem is to compute from an integer matrix, A, both $d = \det A$ and (when $d \neq 0$) another matrix $A' = dA^{-1}$.

The quadtree representation of matrices [38] offers a uniform representation of both sparse and full matrices as directed, acyclic graphs (*dags*). Briefly, a matrix is either homogeneously zero (NIL), or a non-zero 1×1 scalar (that integer), or it is a quadruple of four equally-ordered submatrices. NIL is used to pad a matrix so that its order seems to be a power of two, and to fill sparse matrices internally. Its properties as additive identity and multiplicative annihilator unifies the algorithms for sparse and dense matrices.

The algorithm used to exercise RCM is an exact-arithmetic LU decomposition and inversion algorithm [37], followed by a back-multiply (AA') to the matrix (dI). Full and undulant-block pivoting are used. That is, any $2^p \times 2^p$ subtree (submatrix) might be eliminated in a single elimination step; pivoting attempts to eliminate a large block with a small determinant.

One matrix has been selected from the Harwell-Boeing data set [12] of sparse matrices. We chose CAN62 because it is small, integer, and non-singular. Its decomposition consumes many nodes without generating bignums that would skew measurements of heap use. It is a 62×62 (patterned) symmetric matrix with 140 nonzero elements—all ones, with determinant 117. Inversion of this matrix generates massive intermediate structures that are completely RCM-resident in the RCM tests.

For all the garbage generated, this particular problem never uses much heap at any one time (an average of 63 kiloBytes measured at garbage collection), so that the collection load is atypically small. Therefore, we loaded the heap with static linear lists of length 0, 100,000, 200,000, 300,000 (8-byte nodes) to constrain the garbage collector with more typical loads, in addition to all the traffic from inversion.

Each of these tests is generated by a single run, but we have reproduced some of these repeatedly and consistently,

System	Load (bytes)	**Number of GCs**	**RAM bytes recovered**	**RCM bytes recovered**	**Collector** (seconds)	**Mutator** (seconds)	**Total** (seconds)
Stock RAM	0	9	26,955,428	0	14.0	430	444
Stock RAM	.8M	12	26,399,800	0	115.5	435	550
Stock RAM	1.6M	20	27,934,824	0	355.5	440	796
Stock RAM	2.4M	47	28,111,692	0	1218.4	444	1663
Heap on-a-bus	0	9	444	27,023,120	12.5	467	480
Heap on-a-bus	.8M	12	448	26,510,892	112.6	472	585
Heap on-a-bus	1.6M	20	464	28,125,700	349.0	478	827
Heap on-a-bus	2.4M	46	520	27,925,732	1175.0	483	1658
RAM on-a-bus	0	9	444	27,023,120	13.0	644	657
RAM on-a-bus	.8M	12	448	26,510,892	113.2	621	734
RAM on-a-bus	1.6M	20	464	28,125,700	350.8	628	979
RAM on-a-bus	2.4M	46	520	27,925,732	1178.9	634	1813
Nailed RCM	0	9	444	27,715,104	7.2	661	669
Nailed RCM	.8M	12	448	27,332,736	58.0	677	735
Nailed RCM	1.6M	19	460	28,138,384	164.8	695	860
Nailed RCM	2.4M	41	508	27,950,328	528.7	738	1267
Full RCM	0	0	0	28,207,208	0	639	639
Full RCM	.8M	0	0	28,207,208	0	647	647
Full RCM	1.6M	0	0	28,207,208	0	655	655
Full RCM	2.4M	0	0	28,207,208	0	663	663

Table 2: Experimental results: Integer Matrix-Inversion.

with the total time varying by no more than five seconds. Within the accuracy sought, these times are sufficiently precise for our claims. All mutator and total times except for Stock RAM can have 3 seconds subtracted for RCM initialization, but this time is considered a negligible constant and is included this data. The Full RCM tests have been repeated nearly sixty times without any space leaking to garbage collection.

- The number and effectiveness of garbage collection is consistent (with one anomaly). The number of GCs is the same for Stock RAM, Heap on-a-bus, RAM on-a-bus, and (almost) Nailed RCM. That Stock RAM does one more collection than the others for 2.4M loading can be explained by the fact that both vectors and nodes are in the same heap and they are separated elsewhere.

- The number of bytes recovered is consistent with expectations. Again Stock RAM has an effectively smaller heap because vectors are included there, and so it recovers slightly less space except where squeezed into an extra collection. When the heap is moved out to RCM, the count of RAM-bytes recovered becomes negligible—due only to abandoned vectors—whose growth is too small to come from the inversion algorithm. Heap on-a-bus and RAM on-a-bus perform identically, as they should. Nailed RCM performs similarly, recovering slightly more space.

- The garbage collection times are essentially the same for the first three systems: Stock RAM, Heap on-a-bus, and RAM on-a-bus; they grow with the loading, as expected. When available heap becomes cramped, time for stop-and-copy collection soars. When the hardware support for mark/sweep collection is enabled that time drops precipitously.

- The mutator time does not grow much (8%) from Stock RAM to Heap on-a-bus, but it grows 33% from Heap on-a-bus to RAM on-a-bus. This cost reflects not only the distance from the processor to the RCM memory, but also the loss of caching for the stack; not all of the 33% would be recovered even if RCM were resident (with RAM) on the CPU board. Nevertheless, the total times for Nailed RCM recover much of this difference because of its more efficient collector.

- Full RCM performs very nicely, indeed. Garbage collection is eliminated not only for the single run measured here, but also for at least 59 successive runs—until that repetitive test was killed. Therefore, no vectors were recovered either; probably none were generated after startup. Full RCM is seen to recycle the appropriate number of bytes at run time, slightly more than the garbage collection statistics because the GC tests do not account for garbage recoverable at termination of the program.

- Full RCM has no GC time, and so its mutator timings are its total times. And its total time is far less than all the other systems (except the simplest case). Compared to Stock RAM, Full RCM slows down because the mutator cannot cache its stack and must write all entries through NEXTBUS, but it speeds up by eliminating garbage collection.

- Full RCM exhibits the flat performance profile that one expects from a program designed for real-time and for parallel performance. Our test jig cannot yet verify the latter performance, but these numbers do indicate that it is available. On multiprocessor systems this architecture requires synchronization only just before and just after garbage collection, if one were ever necessary. Except for that, it interleaves among processors just like conventional RAM.

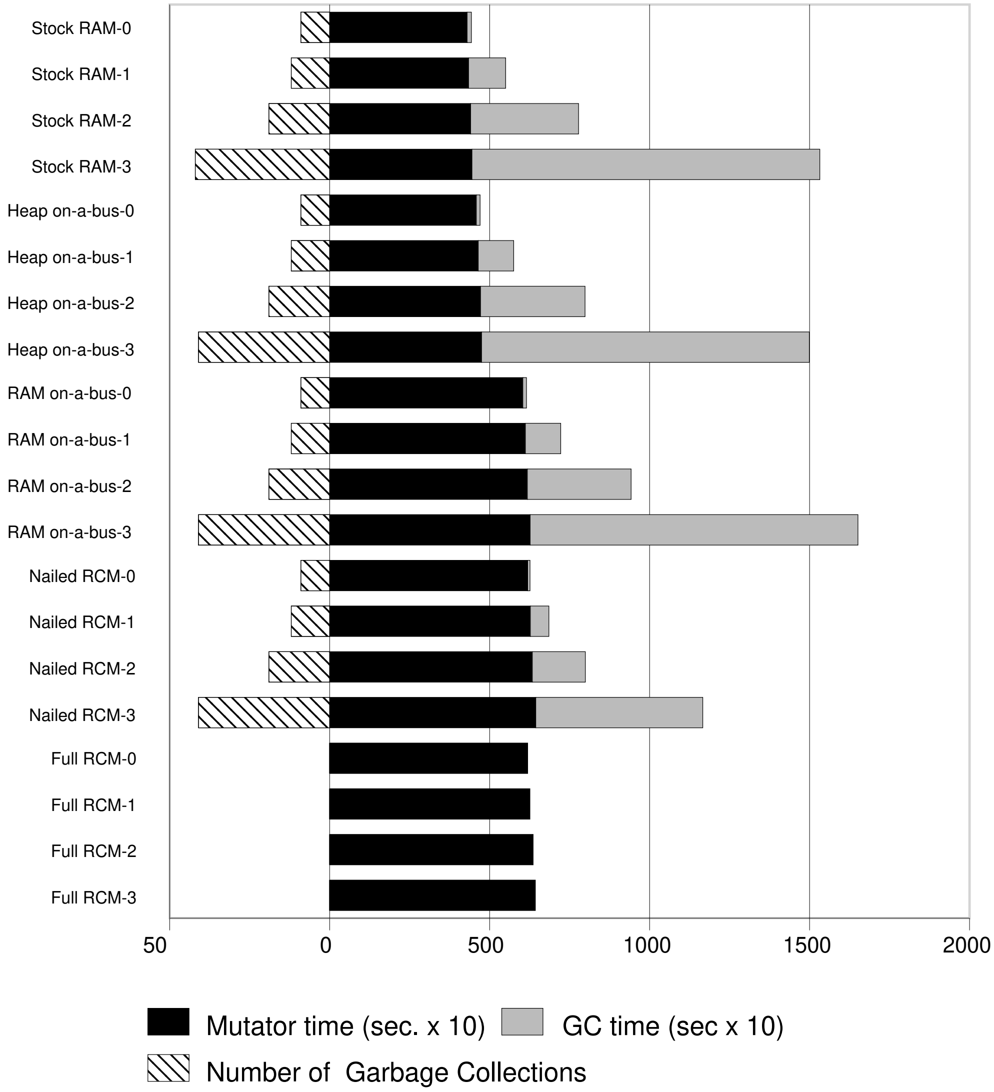

Figure 6: Experimental results.

- Full RCM (as well as Nailed RCM) actually has twice the capacity than that tested. It has a 0.75 megaNode heap, of which only 0.375 was available to these tests. If anything, the effort in tying down the other 0.375 megaNode increased the collector timings somewhat, but only on Nailed RCM because Full RCM never required garbage collection.

7 Conclusions and Future Directions

7.1 Parallel heaps

If parallel processing is to become cost-effective, then it must apply both to large computations, where the overhead of process-scheduling can be amortized, and to new problems so the cost for software revision can be justified. However, exactly this class of large, parallel programs is where current garbage-collection technology fades [11]. With storage management essential to modern programming languages, like Smalltalk, ML, Lisp, Haskell, and Java, we must either abandon languages that depend on automatic storage management and cast our parallel programs in the likes of C, or find strategies for managing dynamic storage on parallel processors. A third, most distasteful, alternative is to abandon multiprocessing entirely for innovative projects that exploit such languages.

7.2 Technological Maturity

These experiments constitute the first research demonstration of hardware support for reference counting. They also offer the first research demonstration of such support for mark/sweep collection. Although these are yet uniprocessor results, together, they contribute to a research demonstration of hardware support for multiprocessing storage management.

Furthermore, we can now recognize the Applicative File System [21] as a demonstration of scientific breakeven of reference counting in hardware. RCM allowed management of both disk and main memory under a homogeneous storage-management strategy: on-line reference counting as the primary algorithm, backed up by an expensive, off-line garbage-collector. That melding of reference counting in RAM with existing reference counting on disk would have been impossible in real time without the RCM hardware. Indeed, it was critical to the success of that project. Some might not accept it as scientific breakeven, however, because in-house experience may be too narrow.

In a broader sense, all these results contribute to scientific breakeven (over software) of hardware support for storage management, where extramural research demonstrations already exist [30, 31, 33].

7.3 Future Work

Advocates and implementors of functional programming and heap memory should take care to separate their assumptions from C++'s, from ours, and from an ideal. Many presume a low-level continuation-passing style, which makes it easy to return functions as results to outer environments. That presumption can place an unusual demand on heap-management, especially where programs do not need it and might do better to exploit a hardware stack, just as C always does. Some compilers of applicative languages manage to eliminate heap-resident continuations for algorithms that could also be expressed in C [13]. Such a compilation would be necessary for a fair comparison to C, and is suggested by our experience with this design.

The present implementation is a board-level prototype installed as an I/O device on a bus, but the underlying design supports VLSI implementation [36] like ordinary dynamic RAM. The prototype is now installed in a uniprocessor, but the design is targeted for several banks of RCM within a multiprocessor. Moreover, the software now available is only a prototype compiler, so there is much opportunity for improvement there too. All three enhancements are necessary before engineering breakeven to be demonstrated.

There are a few features designed into RCM that have not been tested yet. One is on-line multiprocessor reference-counting; a related one is off-line multiprocessor garbage-collection. Another is an unimplemented facility to store a dead pointer (or thread) in a single cycle; originally intended to recover circular structures, it has since demonstrated itself useful for reducing counts in other contexts, as well [39], and must be included in any revision.

We foresee many RCMs built to a standard SIMM interface and installed on a stock multiprocessor. The biggest difference from the one described here is that AVAILable space will be constantly swept by hardware, obviating the need for so large a count/link field. therefore, the hidden memory (Figure 1) will be cut in half. Moreover, we would also provide four-pointer nodes, motivated by a need within system software for more efficient, randomly-referenced records like SCHEME's lexical frames.

This research shows that simple hardware at memory accelerates both reference-counting and garbage collection in a manner consistent with large-scale parallelism. The cost in hardware redesign, in fact, is small; processors are unaffected, the processor–memory path is unchanged, and some simple circuitry is needed at the perimeter of each memory bank. And that memory still remains available as dynamic RAM for conventional access. The worst difficulty arises from sharing of cached data, a problem already familiar under parallelism.

Acknowledgements: Thank You to MACH's designers, for providing preservation of transparent-translation tables, to NeXT for most generously providing us NBICs and prototype boards. to Bob Wehrmiester for early help on hardware, to Peter Beckman for preparing the matrix data, and to Esen Tuna for loaning us a compiler [13]. Thanks also to anonymous referees for helpful comments.

References

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

[2] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *Proc. SIGPLAN '88 Conf. Programming Language Design and Implementation, ACM SIGPLAN Notices* **23**, 7 (July 1988), 11–20.

[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. Preface to H. G. Baker, Jr. (ed.) *Memory Management. Lecture Notes in Computer Science* **986**, Berlin, Springer (1995), v–viii.

[5] D. G. Bobrow. Managing reentrant structures using reference counts. *ACM Trans. Prog. Lang. Sys./* **2**, 3 (July 1980), 269–273.

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

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

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

[9] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, real-time garbage collector. *Comm. ACM* **19**, 9 (September 1976), 522–526.

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the SMALLTALK-80 system. *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 297–302.

[11] A. Diwan, D. Tarditi, & E. Moss. Memory-system performance of programs with intensive heap allocation. *ACM Trans. Comput. Sys.* **13**, 3 (August 1995), 244–273.

[12] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software* **15**, 1 (March 1989), 1–14.

[13] R. K. Dybvig. *Three Implementation Models for* SCHEME, Ph.D. dissertation, Univ. of North Carolina at Chapel Hill (April 1987).

[14] S. I. Feldman. Technological Maturity Scale. Personal communication (1991).

[15] S. I. Feldman. Technological maturity and the history of UNIX. Keynote address, USENIX Summer 1992 Technical Conference, San Antonio, TX (June 10, 1992).

[16] J. Frens & D. S. Wise. Matrix inversion Using quadtrees implemented in GOFER. Technical Rept. 433, Computer Science Dept., Indiana Univ. (May 1995).

[17] 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.

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

[19] 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.

[20] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Sys./* **7**, 4 (October 1985), 501–538.

[21] B. Heck and D. S. Wise. Implementation of an applicative file system. In Y. Bekkers and J. Cohen (eds.), *Memory Management. Lecture Notes in Computer Science* **637**, Berlin, Springer (1992), 248–263.

[22] 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.

[23] 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.

[24] S. D. Johnson. B. Bose & S. D. Johnson. DDD-FM9001: Derivation of a verified microprocessor; an exercise in integrating verification with formal derivation. In G. Milne & L. Pierre (eds.), *Proc. IFIP Conf. on Correct Hardware Design and Verification Methods. Lecture Notes in Computer Science* **683**, Berlin, Springer (1993), 191–202.

[25] R. Jones & R. Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management,* New York, John Wiley & Sons (1996).

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

[27] D. E. Knuth. *The Art of Computer Programming* **III,** *Sorting and Searching*, Reading, MA, Addison-Wesley, (1973).

[28] B. Lang, C. Queinnec, J. Piquer. Garbage collecting the world. *Conf. Rec. 19th ACM Symp. on Principles of Programming Languages* (1992), 39–50.

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

[30] D. Moon. Garbage collection in a large LISP system. *Conf. Rec. 1984 ACM Symp. on Lisp and Functional Programming* (1982), 235–246.

[31] K. Nilsen. Progress in hardware-assisted real-time garbage collection. In H. G. Baker (ed.), *Memory Management. Lecture Notes in Computer Science* **986**, Berlin, Springer (1995), 355–379.

[32] J. Rees and W. Clinger (Eds.) Revised^3 report on the algorithmic language SCHEME. *ACM SIGPLAN Notices* **21**, 12 (December 1986), 37–79.

[33] W. Schmidt & K. Nilsen. Performance of a hardware-assisted real-time garbage collector. *ASPLOS-VI Proc.: 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM SIGPLAN Notices* **29**, 11 (November 1994), 76–85.

[34] 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.

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

[36] 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.

[37] D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comput. Programming* (to appear).

[38] D. S. Wise and J. Franco. Costs of quadtree representation of non-dense matrices. *J. Parallel Distrib. Comput.* **9**, 3 (July 1990), 282-296.

[39] D. S. Wise & J. Walgenbach. Static and dynamic partitioning of pointers as links and threads. *Proc. 1996 Intl. Conf. on Functional Programming, ACM SIGPLAN Notices* **31** , 6 (June 1996), 42–49.

A Functional SCHEME code for balanced-tree insertion

```
;Balanced (AVL) Tree Copy/Insertion algorithm
(define balancedInsert          ;Parameters are (key info tree).
          ; returns a pair (grew? newTree) ,
          ; where   grew?  is a Boolean value, true iff height increases,
          ;   and  newTree  is the resulting balanced (AVL) tree.
     ;A balanced tree is either empty () or
     ;  it is the structure (left balanceFactor right key . info)
     ;where  balanceFactor is -1, 0, or +1, the difference in the heights
     ;            left and right subtrees:  heavy-left/balanced/or heavy-right;
     ;       key.info is ("dotted") key and information.
     ;       left and righte are each balanced subtrees:
     ;                  the heights of the two subtrees differ by at most one;
     ;                  all of left's keys precede key and all of right's follow
     ;                      (keys are sorted to be ordered by inorder of tree).
  (let( (less? <)       ;Any total ordering on keys will do here.
        (-1+ (lambda (n) (- n 1)))
        (+1+ (lambda (n) (+ n 1)))
        (newTree (lambda (balanceFactor key:info left right)
                   `(,left ,balanceFactor ,right . ,key:info) ))
        (getLeft    car)
        (getRight   caddr)
        (getBalance cadr)
        (getKey     cadddr)
        (getInfo    cddddr)
        (getKeyInfo cdddr) )
      (lambda (key info tree)
        (if (null? tree) `(#t ,(newTree 0 (cons key info) tree tree))
            (let( (rootKey (getKey tree))
                  (rootKeyInfo (getKeyInfo tree)) )
              (cond
                ((eqv? key rootKey)
                 `(#f ,(newTree (getBalance tree) (cons key info)
                                (getLeft tree) (getRight tree) )) )
```

```
((less? key rootKey)  ; Insertion goes into left subtree.
 (let( (grew?-newLeft (balancedInsert key info (getLeft tree)))
       (oldBalance (getBalance tree)) )
   (cond
     ((not (car grew?-newLeft))    ;Did the left tree not grow?
      `(#f ,(newTree oldBalance rootKeyInfo
                     (cadr grew?-newLeft) (getRight tree) )) )
     ((not (negative? oldBalance)) ;Will a new balancefactor do?
      `(,(zero? oldBalance)
        ,(newTree (-1+ oldBalance) rootKeyInfo
                  (cadr grew?-newLeft) (getRight tree) )) )
     (else                  ;Balancing act necessary---lefthand.
       `(#f             ;Both balancing acts yield ungrown tree.
          ,(let( (left (cadr grew?-newLeft)))
             (let( (alpha (getLeft left))
                   (beta (getRight left))
                   (leftKeyInfo (getKeyInfo left))
                   (leftBal (getBalance left)) )
               (if (negative? leftBal)  ;simple balancing act.
                 (newTree 0 leftKeyInfo
                    alpha
                    (newTree 0 rootKeyInfo beta (getRight tree))
                    )
                 ;else left son is heavy to right:
                 (let( (gamma (getLeft beta))
                       (delta (getRight beta))
                       (betaKeyInfo (getKeyInfo beta))
                       (betaBalance (getBalance beta)) )
                   (newTree 0 betaKeyInfo
                        (newTree
                          (if (positive? betaBalance) -1 0)
                          leftKeyInfo alpha gamma)
                        (newTree
                          (if (negative? betaBalance) +1 0)
                          rootKeyInfo delta (getRight tree) )))
                 ))))) )))


(else                 ; Insertion goes into right subtree.
 (let((grew?-newRight (balancedInsert key info (getRight tree)))
      (oldBalance (getBalance tree)) )
   (cond
     ((not (car grew?-newRight))   ;Did the right tree not grow?
      `(#f ,(newTree oldBalance rootKeyInfo
                     (getLeft tree) (cadr grew?-newRight) )) )
     ((not (positive? oldBalance)) ;Will a new balancefactor do?
      `(,(zero? oldBalance)
        ,(newTree (+1+ oldBalance) rootKeyInfo
                  (getLeft tree) (cadr grew?-newRight) )) )
     (else                 ;Balancing act necessary---righthand.
       `(#f          ;Both balancing acts yield balanced tree.
          ,(let( (right (cadr grew?-newRight)))
             (let( (beta  (getLeft right))
                   (alpha (getRight right))
                   (rightKeyInfo (getKeyInfo right))
                   (rightBal (getBalance right)) )
               (if (positive? rightBal)  ;simple balancing act.
                 (newTree 0 rightKeyInfo
                     (newTree 0 rootKeyInfo (getLeft tree) beta)
                     alpha)
                 ;else right son is heavy to left:
                 (let( (delta (getLeft beta))
                       (gamma (getRight beta))
                       (betaKeyInfo (getKeyInfo beta))
                       (betaBalance (getBalance beta)) )
                   (newTree 0 betaKeyInfo
                            (newTree
                              (if (positive? betaBalance) -1 0)
                              rootKeyInfo (getLeft tree) delta)
                            (newTree
                              (if (negative? betaBalance) +1 0)
                              rightKeyInfo gamma alpha)))
                 ))))) ))) )))) ))
```