Reprinted from *Proceedings of the 16th Annual International Computer Software and Applications Conference*, Chicago, Sep. 1992, pp. 136-142.

Distributed EZ

Alvaro E. Campos and David R. Hanson
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Abstract

EZ is a system that integrates traditional operating systems and programming languages into a very high-level, persistent, string processing language. This paper describes the design and initial implementation of a distributed memory manager that distributes EZ's virtual address space transparently among a network of homogeneous computers. The design adapts the techniques used in recent implementations of shared virtual memory for use in EZ's persistent environment. Unlike most implementations of shared virtual memory, control information is distributed and migrates. This memory manager works in concert with a distributed mark-and-sweep garbage collector, which is also concurrent and real-time. This collector trades time for space and minimal disruption of mutators, which reduces communication costs.

1 Introduction

A system that integrates language and operating system concepts into a single system offers a different perspective on some time-honored features of traditional operating systems. Traditional file systems are an example; there is no reason why files cannot be regarded simply as persistent values bound to variables. Once this "traditional barrier" is breached, it is natural to view all objects — records, arrays, procedures — as persistent values, which leads to one language that supports both programming and manipulating the environment.

EZ is a language-based, exploratory programming environment that supports this view [9, 15]. It accomplishes this integration by encapsulating system services as features in a very high-level language, and by providing a large, persistent virtual address space.

Past research on *EZ* focused mainly on language *design* — finding suitable linguistic encapsulations for system services. This approach has worked well for some services, e.g., file systems and processes, but other services strain this approach. Tough examples include interactive devices and distributed systems.

This paper reports on the different, but complementary, approach used to accommodate distribution in *EZ*. Instead of language design, this approach focuses on language *implementation* — finding suitable techniques for distributing the *EZ* virtual address space transparently among a network of homogeneous computers.

2 The EZ system

EZ is a high-level string processing language with a persistent memory that permits it to double as a programming environment. Services provided by traditional operating systems are cast as *EZ* language features. Values exist indefinitely or until changed, so *EZ*'s strings and associative tables subsume traditional file and directory services. Associative tables are also used for procedure activations [9] and for threads, i.e., lightweight processes, and low-level synchronization [15].

References [9] and [15] describe the syntax and semantics of *EZ* in detail. Briefly, *EZ* is derived from Icon [12] and its predecessors. It shares many of their attributes, such as run-time flexibility, typed values and untyped variables, heterogeneous structures, automatic type conversions, and mechanisms for run-time scope control. *EZ*'s built-in types include numerics, strings, procedures, and associative tables. *EZ* has the usual control constructs, which are driven by the absence or presence of values. Everything in *EZ*, including program code, resides in a single, persistent, 32-bit virtual address space.

EZ is not the only system that offers persistence. APL systems have always operated in a persistent workspace, and some Smalltalk and Lisp systems offer similar facilities [19, 22, 28]. Others have added persistent data types and procedures to Pascal-like languages [3], some languages have mechanisms for transmitting arbitrary values between programs and hence long-term storage [16], and most object-oriented languages have facilities for saving some objects on disk for later retrieval [20]. Persistence is also an important aspect of some programming environments for traditional languages [30] and for maintaining programming environment databases [29]. The key difference between these systems and *EZ* is that most offer persistence as a separate facility and usually restrict it to data. Persistence pervades *EZ* and applies to both data and active objects, like procedure activations and processes.

As described in Reference [15], *EZ* has an interpreter-based implementation similar in detail to other very high-level languages [11]. The virtual address space resides on disk and is managed by a virtual memory manager that caches pages in memory. Currently, all memory management is done by software. The interface between the interpreter and memory manager consists of two functions: `GetPage(`*a, mode*`)` returns a "handle" to the page that encompasses address a, and `PutPage(`*h, dirty*`)` releases the page given by handle h. *mode* indicates access mode and is either `read` or `write`, and *dirty* indicates whether or not a writable page was actually modified. Handles are simply the memory address of the in-cache copy of the page. Flushing the cache saves the system state.

Unlike the clients of memory managers in operating systems, the *EZ* interpreter is a "trusted" client and is therefore given complete ownership of pages requested via `GetPage`. Other interpreter threads can request the same page. As in classical readers/writers applications, multiple `read` requests are granted, but `write` requests are given exclusive access. The interpreter itself is written to avoid deadlock, but it is possible to write *EZ* processes that deadlock just as it is possible to write programs with threads in Mach [5] that deadlock, for example.

3 Distribution

The *EZ* address space is distributed transparently among a set of homogeneous processors by replacing the memory manager by one that is based on the recent implementations of shared virtual memory [18, 21, 25]. The shared virtual memory manager attempts to collect recently accessed pages at the processor that accesses them. In this design, identical copies of the new memory manager run at each processor, and the entire virtual address space is distributed among the secondary storage devices of the individual processors. Each manager caches the pages that its interpreter clients are using, and these pages are accessed via `GetPage` as described above. The managers hide all distribution details. Managers replicate pages to permit multiple readers, but permit only one writer in order to maintain coherence.

3.1 Coherence

Since `GetPage` yields ownership of a page to the client interpreter, the managers cannot use invalidation techniques [18] to maintain coherence. Page ownership migrates to the manager that needs to grant write access to the page. At any time, each page is owned by exactly one manager, and that owner is responsible for maintaining the disk copy of the page. Consequently, pages migrate among local disks.

Each cached page is known to a manager as *invalid*, *shared*, or *unique*. Invalid pages are those owned by other managers for which the cached copy is known to be out of date. A new copy must be fetched from its owner the next time access is requested. Shared pages are those for which the cached copy might be up to date and other managers might also have shared copies. Read requests for shared pages can be granted if the local copy is known to be valid or after the owner confirms the copy's validity. Unique pages are those that are owned by the local manager and for which there are no valid copies elsewhere. Read and write requests for these pages can be granted without communicating with other managers.

As suggested above, a manager maintains a page cache and an ownership table, which lists all of the pages owned by the manager and their disk addresses. For each cached page, the cache entry holds a pointer to the local copy of the page, the page status (invalid, shared, or unique), a reference count, a queue of pending access requests, an ownership hint, a valid copies hint, and a dirty bit. The last two entries apply only to locally owned pages.

For remote pages (i.e., pages owned by another manager), the reference count is the number of outstanding read accesses that have been granted in response to `GetPage`s from local interpreter threads. For owned (and hence local) pages, this count is the number of local read or write accesses that have been granted plus the number of remote managers that have granted read access to their copies of the page (which they obtain on the first access request). `PutPage`s decrement the count. For remote pages, a message is sent to the owner when the count reaches 0, and the owner then decrements its counter.

Write requests are queued until the count becomes 0. Once granted, the count becomes 1. For remote pages, write requests cause the local manager to ask the page's owner to transfer ownership to the local manager, which occurs when the owner's count becomes 0. Once transferred, the write request is granted as above.

The ownership hint identifies owned pages and gives the probable identities of their owners. The owner's identity accompanies the copies of remote pages when they are fetched. Responses always include the owner's identity, which updates this field, if necessary.

The valid copies hint is a bitmap that identifies other managers to which valid copies of the page have been sent. It is used to avoid sending copies of pages unnecessarily.

Pages with a 0 reference count and no pending requests may appear on one of four LRU lists, which are consulted for page replacement. When the manager must replace a cache entry, it uses the first entry on the list of *invalid* pages. If this list is empty, it uses the first entry on the list of remote *shared* pages. Doing so obligates the local manager to fetch a new copy of the replaced page, if it is accessed again.

If both of these lists are empty, the first entry on the list of *unique* pages is used. Selecting a page from this list obligates the manager to reread it, if it is accessed again, but there is no network cost associated with this choice.

If all else fails, the first entry on the list of owned *shared* pages is used. There are valid copies of these pages elsewhere, but the manager must "forget" about these copies because it is about to reuse the valid copies hint in the cache entry. Consequently, subsequent read requests from remote managers will cause the page to be resent, perhaps unnecessarily.

3.2 Communication

Communication is based on reliable byte streams between managers, and it is assumed that the communication subsystem preserves message order. Request messages have a *type*, which identifies the request, a *sender*, which identifies the initiating manager and its processor, and an *address*, which is the virtual address of interest in the request. Acknowledgment messages have a similar form and an optional *page*, which is a copy of the page itself. In acknowlegments, *sender* identifies the owner of the page (which might soon change), and *type* indicates whether or not an optional page follows.

The sender helps managers differentiate between local and remote requests. The addresses in remote requests always specify pages; local requests mirror the semantics of `GetPage` and can specify any virtual address and the page containing that address is returned.

Calls to `GetPage` and `PutPage` generate four message types. `GetPage` generates *read* and *write* requests. `PutPage` generates *clean* requests, which indicate that the client is finished with the page and has not modified it, and *dirty* requests, which indicate that the client is finished with the page and modified it.

Most requests are for pages that the local manager owns, and handling these is straightforward. Requests for pages that the local manager does *not* own require communication with other managers, specifically the owner of the requested page. There are seven message types involved in this communication, which are refinements of the four types described above.

`GetPage` for `read` access is granted immediately if the local manager knows that its copy of the page is valid, e.g., another local interpreter is reading the page. If the local manager has a copy of the page that it believes is valid, a *read* request is sent to the owner. The owner simply grants access, but includes a copy of the page if it is possible that the requester's copy is invalid. Finally, if the local manager does not have a valid copy of the page, it sends a *copy* request to the page's owner who grants the request by sending a copy of the page. When a read-only page is returned via `PutPage`, the local manager decrements its reference count on the page and, when the count reaches 0, sends the owner a *decrement* request, which is the remote equivalent of *clean*.

Calls to `GetPage` that request `write` access cause page ownership to transfer to the requester. The local manager sends the owner a *write* or *fetch* request depending on whether or not it may hold a valid copy of the page. The owner grants the request, including a copy of the page, if necessary, and marks its copy of the page as *invalid* since it is no longer the owner and its copy of the page is now obsolete.

If a requesting manager does not know the owner of the desired page (e.g., because it does not hold a valid copy of the page), it broadcasts equivalents of the message types described above to all managers. Only the actual owner responds to these messages, which identifies the owner for subsequent requests and reduces broadcast traffic. Messages directed to a manager that is no longer the owner of the desired page are turned into broadcast messages by the recipient or simply forwarded if the recipient has an ownership hint for the page.

3.3 Implementation

The implementation of the memory manager including the communication package consists of about 1,500 lines of ANSI C. The thread package is another 350 lines. The interpreter and run-time system consist of 5,000 lines, and the compiler is about 1,800 lines.

The memory manager operates on pages upon which *EZ*'s internal values and data structures are implemented. Most pages hold only one value or internal data structure. For example, strings are implemented by lists of pages, arrays are implemented as pointers to pages of additional pointers to pages of values, which is similar to the allocation of file space on UNIX [27], and the hash tables for associative tables fit in a page. These kinds of techniques collaborate to minimize internal fragmentation and gratuitous inter-page references.

Each cached page is served by a thread, which processes requests from a per-page FIFO service queue. When the queue becomes empty, the thread terminates. Requests that cannot be satisfied immediately because the page is busy are queued and serviced when previous requests complete. Using one thread per page serializes requests and simplifies programming.

Other threads accept requests from remote managers and append them to the service queue of the appropriate page. These threads also initiate per-page threads as necessary.

4 Garbage collection

Earlier versions of *EZ* used an off-line garbage collector to reclaim inaccessible pages in the disk representation of the virtual address space. This approach is fine for a prototype and perhaps adequate for a non-distributed, "single-user" *EZ* system that is subject to frequent idle periods, but an off-line approach is unsuitable for a distributed system.

Distributed *EZ* will use a distributed garbage collector that works in concert with the shared virtual memory manager described above. It is a distributed mark-and-sweep collector [4, 6, 24], and it is concurrent and real-time. Technically, algorithms based on reference counting [10, 17] are more efficient, but require additional data for every pointer that might refer to a page on another processor, or additional synchronization between subsets of the processors at each reference. Besides, these algorithms cannot handle cycles, which makes them unsuitable for *EZ* where cycles abound.

Likewise, copying collectors are also more efficient and can be made both concurrent and real-time [1]. Most designed for distributed address spaces are not concurrent or real-time, and some require special hardware to be efficient [13] or impose restrictions on inter-processor pointers, such as double indirection [23]. More importantly, objects in *EZ*'s virtual address space cannot move in order to simplify the implementation of the persistent address space.

For systems with large, persistent address spaces, efficiency is less important than concurrency; a collector must not pause and must cope with network and node failures, even at the cost of collector efficiency. Indeed, all that is required is that the collector replenish the supply of free pages fast enough so that applications rarely have to wait to allocate a new page, and that it eventually collects all inaccessible pages.

An identical copy of the collector runs forever on each processor (technically, there is a collector for each memory manager). As usual, a collector marks all pages that hold accessible objects starting from a few system "root" objects. It also marks pages referenced from within objects on marked pages, which may cause some inaccessible pages to be marked. Indeed, the collector is conservative: it marks a superset of the accessible pages and collects only a subset of the inaccessible ones [6]. It repeats the collection continuously, so it eventually reclaims all inaccessible pages.

Each collector processes only the pages owned by its cooperating memory manager. After marking the accessible owned pages, a collector exchanges information about inter-processor references with the collectors on the other processors. This information feeds another marking cycle that expands the local collector's set of accessible pages. This activity continues until no collector can expand its accessible page set.

Concurrent collectors need the cooperation of mutators when updating references. In particular, when a reference is updated, either the old target or the new target must be marked atomically. Failure to do so may cause the collector to reclaim either the old or new target erroneously. Most collectors mark the new target to avoid hanging on to the page holding the old target unnecessarily. *EZ*'s collector, however, marks the old target for two reasons. First, marking the new target requires direct mutator assistance. Second, marking the old target permits the collector to use virtual memory hardware to mark pages referenced by a page before it is modified [2]. At the beginning of a collection, all owned pages are set to read only. The first write to a page causes a page fault, and pages referenced within the faulted page are marked before the fault handler approves write access to the page. Marking the referents of a page *before* it is updated is equivalent to marking the old targets of updates. Similar comments apply to calls to `GetPage` for write access, but virtual memory hardware is unnecessary.

At any time during marking, each page is marked with either white, grey, or black. Stop-and-collect, uniprocessor collectors that are not concurrent use only two colors; the third, *grey*, labels pages that have been marked but not yet traversed for pointers.

At the beginning of each collection, locally owned pages have the same color, say, white. After marking, pages will be colored either white or black, and white pages can be reclaimed. As suggested above, some black pages may really be inaccessible, but they will be reclaimed during a subsequent collection. Pseudocode for the collection algorithm is shown in Figure 1. Initially, *retain* and *gather* are black and white, respectively. The colors reverse roles in subsequent collections. The *owned* and *free pages* sets and the per-page data are shared by the collector and the memory manager with appropriate locking.

Collection begins by coloring locally free pages black so that the memory manager can allocate them without help from the collector. The access for owned pages is set to `readonly` so that the referents for these pages will be marked if they are modified, as described above. Writes to these pages are caught at faults or calls to `GetPage` and the page requested is scanned before write access is granted. This action is the only interaction between the mutator and the collector. As shown in Figure 1, scanning a page shades its referents, colors the page black, and unprotects it. The referents of the local roots are colored by *Shade*, which colors owned white pages grey or adds remote pages to m, which is empty initially.

Marking then begins to cycle. In each cycle, locally owned grey pages are scanned. Scanning a page may yield another grey page, but this activity ends eventually. Once all grey owned pages are scanned, all pages reachable from local roots are colored black, and m is the set of all remote pages that should have been colored grey.

A subset of m is broadcast to other processors. This subset is the set of pages that have not been announced by previous broadcasts. *cycle* records the size of this subset, and m is added to M, which accumulates remote pages announced by any processor. The collector then consumes similar messages from the other collectors, accumulates their sizes in *cycle*, and shades the pages mentioned in the messages. This shading colors owned white pages grey or adds remote pages to M. These messages serve not only to communicate the remote page references between collectors, but also to synchronize them.

As collection progresses, each collector's M becomes larger until $m - M$ becomes empty, i.e., until all grey pages everywhere have been colored black. At that point, all owned white pages are added to the set of free pages, the roles of white and black are reversed and collection begins anew.

If a page migrates during collection, it is scanned *before* it is sent to its new owner, if necessary. The memory manager triggers this scan by simulating a local write, which causes the page to be scanned, as described above. The new owner colors the page black.

```
retain ← black
gather ← white
do forever
    M ← m ← ∅
    for every p ∈ free pages do Color(p) ← retain
    for every p ∈ owned do Access(p) ← readonly
    for every reference r in the local roots do
        Shade(Page(r), m)
    do
        while ∃p ∈ owned ∧ Color(p) = grey do
            Scan(p, m)
        cycle ← |m − M|
        broadcast m − M
        M ← M ∪ m
        for every other processor P do
            receive message k from P
            cycle ← cycle + |k|
            for every p ∈ k do Shade(p, M)
        m ← ∅
    while cycle > 0
    for every p ∈ owned do
        if Color(p) = gather then
            free pages ← free pages ∪ {p}
    gather, retain ← retain, gather

Shade(p, s):
    if p ∈ owned then
        if Color(p) = gather then Color(p) ← grey
    else s ← s ∪ {p}

Scan(p, s):
    for every reference r in p do Shade(Page(r), s)
    Color(p) ← retain
    Access(p) ← read/write
```

Figure 1: Garbage Collection Algorithm.

Space efficiency and minimal disruption of mutators are more important than time efficiency of the algorithm itself. There are reasonably efficient representations for all of the data structures used in the algorithm. Marks are kept in a private bitmap, 2 bits per page. Page sets for k, m, and M are represented as lists of arrays of page numbers or page ranges, and most page ranges can fit in 32 bits as a 23-bit page number and a 9-bit spread or as two 16-bit page numbers. The memory manager already maintains *owned* as a bitmap, and a subset of this bitmap identifies owned `grey` pages. Finally, *free pages* is a list of available pages represented by a list of arrays of page ranges stored in disk blocks as in UNIX [27]. It is accessed as a LIFO list, so the first disk block is often in the memory manager's cache.

5 Discussion

The memory manager and its associated communications layer have been integrated into the *EZ* system, and implementation of the garbage collector is underway. These changes also necessitated basic changes in the *EZ* virtual machine in order to ensure atomicity and consistency.

Previously, many primitives that accessed shared memory (as opposed to per-thread memory) accepted pointers into the cache as operands because the previous version of `GetPage` was atomic and uninterruptible. To avoid deadlock, these kinds of primitives had to be decomposed and re-cast in terms of three operations on associative tables: membership testing, insertion, and deletion. These operations are atomic with respect to the tables on which they operate, but can be interrupted by other, unrelated operations.

Performance measurements will undoubtedly induce modifications to the design and to the current implementation. For example, even though relatively few shared pages are actually modified, *EZ*'s adherence to strict consistency may lead to thrashing, which might be attacked by selective use of release consistency within the interpreter, or by instituting a minimum ownership time [8], which would give owners time to complete several modification operations before a page changed owner because of a write request.

Other than the interface via `GetPage` and `PutPage`, there is little in the virtual memory system that is specific to *EZ*. These techniques can be applied to other distributed persistent languages and environments.

EZ's 32-bit virtual address space is too small, especially in light of the impending availability of 64-bit processors. The techniques described in this paper can accommodate such large address spaces, but other techniques might have advantages. Pointer "swizzling" [31] could be used for a non-distributed *EZ*, but it is unclear how to adapt this technique to distributed systems.

Another alternative design under consideration accommodates multiple virtual address spaces anywhere in a network and achieves distribution by inter-address space references. This approach can be viewed as the complement of the distributed memory manager approach. Here, the original memory manager remains nearly untouched, but the language, interpreter and run-time system are modified.

References to other address spaces are made through inter-address space pointers. These pointers are functionally equivalent to capabilities used in some distributed systems [26]. They contain an address space identifier and an address within that address space. Capability-like rights could be added to restrict the set of legal operations.

The attraction of this approach is that address spaces could be encapsulated as tables much like strings encapsulate files. The cost, however, is that the interpreter must take special action in order to access these tables. Efficient implementation techniques for this kind of dereferencing have been used in other high-level language systems [11], in heterogeneous systems [7], and for implementing implicit synchronization [14]. Pages would not migrate in this approach, which simplifies memory management and increases reliability. Remote dereferencing translates into essentially remote procedure calls to `GetPage` and `PutPage`.

Reliability and fault tolerance for distributed persistent systems like *EZ* remains an important area for future work. Currently, distributed *EZ* uses timeouts to recover from network and machine failures. These timeouts cause the memory manager to terminate the requesting *EZ* process. This somewhat unsatisfying approach works for user-level processes, but is unacceptable for system-level processes, like the garbage collector, and other mechanisms are under investigation.

References

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 23(7):11–20, July 1988.

[2] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, Apr. 1991.

[3] M. P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, Oct. 1985.

[4] L. Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages (LNCS 259)*, pages 75–93. Springer-Verlag, Berlin, 1987.

[5] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1988.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, Nov. 1978.

[7] R. B. Esick IV. *The Cross-Architecture Procedure Call*. PhD thesis, The University of Illinois at Urbana-Champaign, Urbana, IL, 1987.

[8] B. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *ACM Symposium on Operating Systems Principles*, pages 211–223, Dec. 1989.

[9] C. W. Fraser and D. R. Hanson. High-level language facilities for low-level services. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 217–224, New Orleans, LA, Jan. 1985.

[10] B. Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):313–321, July 1989.

[11] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, NJ, 1986.

[12] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[13] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *ACM Conference on LISP and Functional Programming*, pages 9–17, Austin, TX, Aug. 1984.

[14] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[15] D. R. Hanson and M. Kobayashi. *EZ* processes. In *Proceedings of the International Conference on Computer Languages*, pages 90–97, New Orleans, LA, Mar. 1990.

[16] M. Herlihy and B. H. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.

[17] C.-W. Lermen and D. Maurer. A protocol for distributed reference counting. In *ACM Conference on LISP and Functional Programming*, pages 343–350, Cambridge, MA, Aug. 1986.

[18] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[19] C. Low. A shared, persistent object store. In S. Gjessing and K. Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 390–410, Berlin, Aug. 1988. Springer-Verlag.

[20] B. Meyer. *Eiffel: The Language*. Prentice Hall International, London, 1992.

[21] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, Aug. 1991.

[22] A. Paepcke. PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 374–389, Berlin, Aug. 1988. Springer-Verlag.

[23] M. Rudalics. Distributed copying garbage collection. In *ACM Conference on LISP and Functional Programming*, pages 364–372, Cambridge, MA, Aug. 1986.

[24] M. Schelvis and E. Bledoeg. The implementation of a distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, *ECOOP'88: European Conference on Object-Oriented Programming (LNCS 322)*, pages 212–232, Berlin, Aug. 1988. Springer-Verlag.

[25] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[26] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Dec. 1990.

[27] K. Thompson. UNIX implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.

[28] S. R. Vegdahl. Moving structures between Smalltalk images. *OOPSLA'86 Conference Proceedings, SIGPLAN Notices*, 21(11):466–471, Nov. 1986.

[29] D. Wiebe. A distributed repository for immutable persistent objects. *OOPSLA'86 Conference Proceedings, SIGPLAN Notices*, 21(11):453–465, Nov. 1986.

[30] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An experiment in persistent typed object management. *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments, SIGPLAN Notices*, 24(2):130–142, Feb. 1989.

[31] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *Computer Architecture News*, 19(4):6–13, June 1991.